FI MU

Faculty of Informatics
Masaryk University Brno

LOBS: Load Balancing for Similarity Peer-to-Peer Structures

by

David Novák
Pavel Zezula

FI MU Report Series **FIMU-RS-2007-04**

 June 2007

Publications in the FI MU Report Series are in general accessible via WWW:

`http://www.fi.muni.cz/reports/`

Further information can be obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**

LOBS: Load Balancing for Similarity Peer-to-Peer Structures

David Novák
Masaryk University, Brno, Czech Republic
xnovak8@fi.muni.cz

Pavel Zezula
Masaryk University, Brno, Czech Republic
zezula@fi.muni.cz

June 29, 2007

Abstract

The real-life experience with the similarity search shows that this task is both difficult and very expensive in terms of processing time. The peer-to-peer structures seem to be a suitable solution for content-based retrieval in huge data collections. In these systems, the computational load generated by a query traffic is highly skewed which degrades the searching performance. Since no current load-balancing techniques are designed for this task, we propose LOBS – a novel and general system for load-balancing in peer-to-peer structures with time-consuming searching. LOBS is based on the following principles: measuring the computational load of the peers, separation of the logical and the physical level of the system, and detailed analysis of the load source to exploit either data relocation or data replication.

This report contains detailed description of the fundamentals and specific algorithms of LOBS, a theoretical analysis of its behaviour, and results of extensive experiments we conducted using a prototype implementation of LOBS. We tested LOBS with the peer-to-peer structure *M-Chord* having a various number of peers. We used a real-life dataset and query sets of various distributions. The results show that LOBS is able to cope with any query-distribution and that it improves both the utilization of resources and the system performance of query processing. The costs of balancing are reasonable compared to the level of imbalance and are very

small if the system has time to adapt to a query-traffic. The behaviour of LOBS is independent of the size of the network.

1 Similarity Peer-to-Peer Structures

The volumes of digital data being produced by the mankind are growing rapidly. According to IDC company[1], the increase is mainly due to digitization of photography, video, music and other formerly analogous data types which are now accessible to a high number of people and are often produced automatically. This data can hardly be efficiently managed, processed and shared by a single computer unit – some form of distributed processing is crucial.

Fortunately, the growing data production goes hand in hand with the amount and performance of computational resources available. Therefore, the peer-to-peer (P2P) structures [2] seem to be a promising, cheap and self-organizing alternative to systems running on clusters or grids. The P2P structures have potential to provide both an extensive data storage and a strong distributed searching engine.

The query paradigms of the P2P structures has undergo an evolution. The Distributed Hash Tables (DHTs) [21, 20] with a simple key-object location have been followed by structures designed for (multi-dimensional) interval queries [8, 4, 13, 9] and, recently, also for similarity (content-based) search [5, 11, 19]. One of the issues that has to be considered in all types of P2P structures is balancing of the load among the participating peers.

The load may be defined in various ways. Majority of balancing techniques [1, 12, 10, 3, 15, 14] focus on balancing of the data volume stored by individual peers, which is a suitable strategy for simple query paradigms. These techniques consider data insertions and deletions and react to imbalances caused by data volume modifications. Some balancing mechanisms [22] consider (or may theoretically consider [12, 14]) the number of query accesses as the relevant load measure. They react to the varying query traffic and count a query hitting a peer as a single unit of the load.

In this report, we focus on P2P structures for the similarity search, which typically requires a time-consuming local search at the peers involved in the query processing [6]. The actual time of the query processing is varying and depends on the particular query, on size of the local data, on quality of the local index, and on other fluctuating and

[1] International Data Corporation, http://www.idc.com

hardly predictable variables. Therefore, the load cannot be measured as a simple number of accesses and it is reasonable to consider the computational load of the peers and try to keep it balanced. We would also like to take into account impact of the load-balancing to performance of the query processing.

We can see a logical parallel between this goal and the classical problems of task-assignment or job-routing [17] solved in parallel and distributed systems. The settings differ in the instruments that we have to influence the load. In the P2P environment, the "jobs" are data-dependent. The placement of the jobs can be influenced only through the data: we can either relocate or replicate it. Please, note that it does not mean that keeping the data volume balanced makes the processing load balanced.

Experiments on P2P structures for similarity search show that even a query traffic with a uniform query distribution is hardly predictable and does not apply the load to the system uniformly. Moreover, the real-life traffic tends to have Zipfian distribution, which makes the situation even worse – example of such a distribution of the processing load can be observed in Figure 1. This waste of resources and degradation of system performance call for improvement.

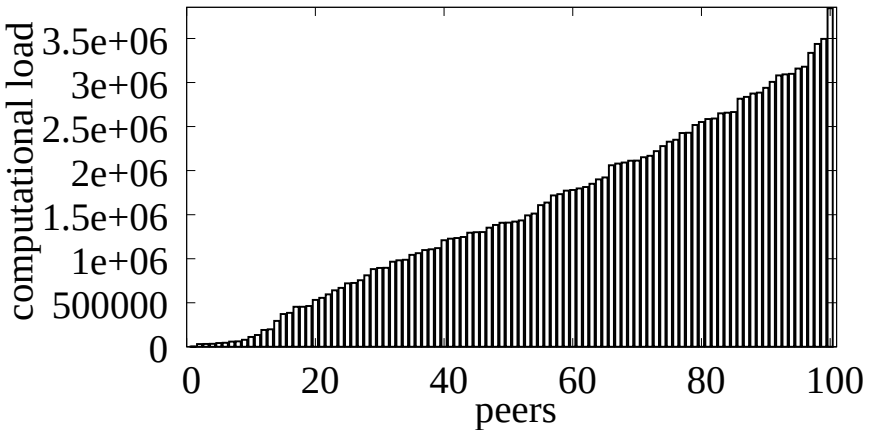

Figure 1: Example of processing load distribution without load balancing.

Another trouble with the P2P structures for similarity search (especially with those [6] based on the metric space model of data [23]) is often a nonlinear topology of the network – data cannot be simply shifted from a node to its neighbour. This reduces the options of the load-balancing process since the action of "data shift" is very natural and often used by current balancing techniques.

Problem Formulation

The goal of this work is to propose a load-balancing mechanism for structured peer-to-peer networks which provide a time-consuming similarity search. The mechanism should

- consider the processing costs of the search requests evaluation,
- take into account structures with general topology (e.g. trees),
- be fully decentralized.

The quality of the balancing mechanism should be evaluated on an existing P2P structure with a reasonable dataset and a real-life query traffic. Besides the distribution of the load in the system, the evaluations should consider the influence of the balancing to performance of query-processing from the user's point of view.

Report Structure

Section 2 maps current achievements in the area of load-balancing in structured P2P networks. It draws the conclusion that current balancing techniques are not suitable for time-consuming query paradigms. In Section 3, we introduce LOBS – Load Balancing for P2P Similarity Structures. Namely, we discuss the fundamental ideas of LOBS, the specific load measures used by LOBS, the operations available for load balancing, and the actual balancing strategies. Section 4 contains a detailed analysis of the behaviour of LOBS – both theoretical and based on a series of experiments on a prototype implementation of LOBS. The paper concludes by Section 5 with directions of future work.

2 Related Work

The Distributed Hash Tables [21, 20], as the first and the simplest P2P structures, primary focus on keeping the data distribution fair. They can solve this problem by hashing the key-space to the navigation-space using a pseudo-random hash function with a uniform distribution. This method is not applicable to more complex search paradigms, such as interval search, since it destroys the locality property of the data.

Majority of recent works in the area of P2P load balancing [1, 12, 10, 3, 15, 14] is motivated by the need of efficient processing of interval queries. The load-function

considered by these techniques is either the *amount of data* stored by individual peers or the *number of accesses* per peer. This is legitimate since the data I/O-costs are the most important aspect to be considered for the interval queries. These balancing strategies expect linearly-sorted and range-partitioned data with the option of shifting part of the data from a node to its *neighbor* and with the option of splitting a node into two half-loaded nodes.

Another way to balance the access-load is purely by replication of the high-loaded logical nodes to less loaded peers. This concept is adopted, e.g., by the HotRoD system [22] – a DHT-based architecture for interval queries.

None of the currently available techniques focus on balancing for query-paradigms that require time-consuming and variable local processing. The strategies also assume sortable domain and linear architecture of the structures.

3 The LOBS Balancing Framework

In this section, we describe LOBS – a general framework for load balancing of peer-to-peer structured networks which is designed for systems providing similarity search or other time-consuming query-paradigm.

3.1 Principles

The LOBS framework assumes the following general model of a peer-to-peer structured network. The system is formed by a set of *nodes* $N = \{n_1, \ldots, n_k\}$ and each node consists of the local *storage* (data with a search mechanism) and the *routing structure* (links to other nodes plus the navigation strategy):

$$\forall n \in N : n = \langle data_n; R_n \rangle, \quad R_n \subseteq N \text{ is a set of links to other nodes.}$$

As discussed in Section 1, LOBS focuses on balancing of the *computational load* in the system. We can see two general ways to reduce a load of a node:

- Relocate part of the data elsewhere in order to decrease the processing costs of the queries processed locally and to, eventually, decrease number of queries hitting the node (data relocation).
- Replicate the node in order to spread the query load between the replicas (data replication).

Data Relocation

A basic data-relocation operation used by balancing techniques for structures with linear topologies and sorted data domains involves shifting part of the data to a neighbor. Due to the lack of total ordering, a general P2P structure for similarity search does not provide such an operation but it supports node splitting thereby – creating a new node to carry approximately one half of the node's data.

This leads us to the following concept: Let us separate the physical peers from the logical nodes and allow the peers to carry more than one node. The load is naturally measured per peer. Then, we can split a node that caused an overload to a less-loaded peer. Also, this opens up the possibility of using *any* peer with a light load to relieve an overloaded peer. Another good motivation for letting having several nodes at a peer is *replication* – besides regular nodes, a peer can carry one or more replicas of other nodes (see below).

More formally, there is a set of physical peers $P = \{p_1, \ldots, p_m\}$ and each node is hosted by one physical peer:

$$h : N \longmapsto P$$

where h is a total mapping. Figure 2 depicts the schema described. The set of nodes hosted by a peer p ($h^{-1}(p)$) is also denoted as $nodes(p)$.

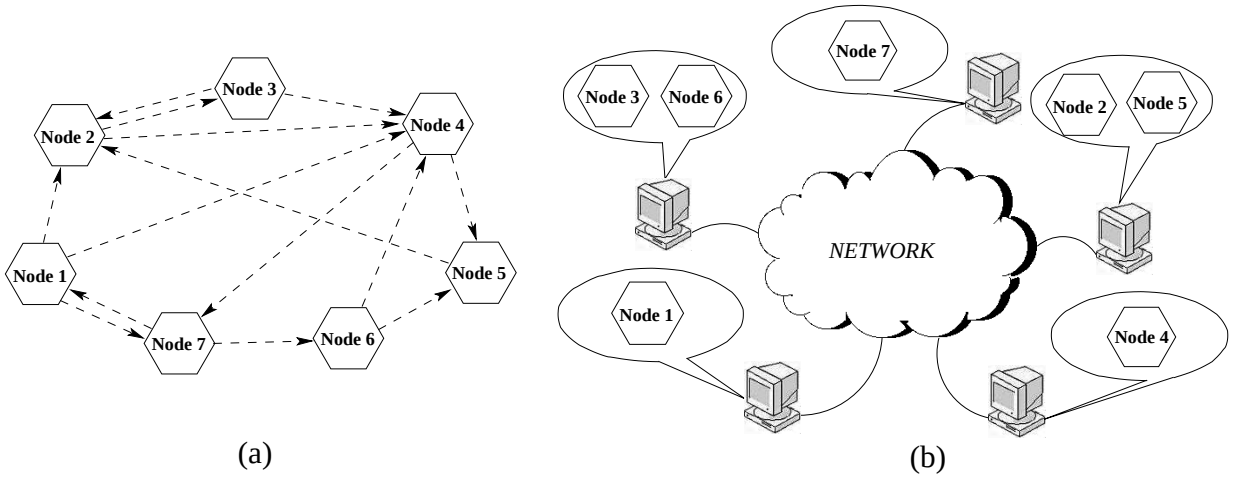

Figure 2: The logical structure (a). Mapping of the nodes to the peers (b).

Data Replication

In general, the replication can be implemented in the following two modes:

1. Every navigation link to node n is, in fact, a link to a set of copies of n. All the copies can be used when forwarding a message to n.

2. Only the master node n is referenced by other nodes and it can forward a message to be processed by one of its replicas. Any updates are first made at the master node and are then spread to the slave-replicas; queries can be answered by any replica.

The first mode can be used only if the specific P2P structure supports replication by itself (it cannot be added by external balancing mechanism). It also requires a master-master replication where updates can be made at any replica and are spread to all others.

In the second mode, the master node does all the navigation work and the usage of replicas increases the routing hop count by one. Therefore, it is not suitable for systems where navigation time is crucial and the local query processing is rather negligible. But it can spread the load in a nice way in the systems which we focus on, i.e. systems where the local data searching is very expensive and the *computational load* is to be balanced. Such a replication can be added to any structure and managed from the outside.

In order to keep LOBS general, we will expect the second type of replication but the first one can be used if provided by the specific P2P structure. Formally, set N contains both regular nodes and *replicas* of other nodes. We define a function *rep* that assigns each node its set of replicas (if any):

$$rep : N \rightarrow 2^N.$$

The node and all its replicas must be hosted by different peers:

$$\forall m, o \in rep(n) \cup \{n\}, m \neq o : h(m) \neq h(o).$$

3.2 Measuring the Load

The LOBS framework is very general and assumes only a little knowledge about the particular P2P structure. The nodes communicate via messages and any node can insert a data item which is navigated to the proper position in the structure and stored. Any node can pose a `Search` request, which is routed to a subset of nodes $X \subseteq N$ and each node $m \in X$ searches its local $data_m$ and returns a partial answer to the query originator. LOBS focuses on balancing the computational costs caused by processing of queries at particular peers.

The general goal of a load balancing is to "keep a fair load distribution among the participating peers". But if we focus on query processing, this fairness policy itself is not the primary motivation – the motivation is to process the queries as efficiently as possible. The LOBS system tries to find a definition of the load in order to maximize profit for the query processing.

In a system where users can issue queries from any peer and the query processing takes a nontrivial amount of time, multiple queries can arrive at a peer simultaneously and they either form a queue or compete for the resources. In either case, the overall time of the `Search` operation processing at a single peer is influenced by the following aspects:

1. evaluation of `Search` at the peer's local data,
2. other queries hitting the peer.

The first aspect can be influenced by adjusting the data volume stored at the peer. Such an action influences the second aspect as well but in an unpredictable way. If a peer is overloaded because of an enormous number of queries hitting it then replication (and thus spreading the load) is a sure way and it is the only solution in some cases.

The philosophy of LOBS is the following: Analyze the overload precisely in terms of the two above-mentioned cases and choose the balancing action appropriate for the specific situation. Therefore, we define two separate load measures – the *processing load caused by a single query* (**single-load**) and the *overall processing load of the queries hitting a peer* (**load**).

Single Query Load

Let us define the costs of processing a `Search` operation at node $n \in N$:

$$cost_n(\texttt{Search}) = \begin{cases} cost(\texttt{Search}(data_n)), & \text{if } \texttt{Search} \text{ is to be processed at } n, \\ 0, & \text{otherwise,} \end{cases}$$

where $cost(\texttt{Search}(data_n))$ is defined as either the CPU time or as an approximation suitable for the specific application, e.g. the I/O costs or the number of distance computations for a costly distance measure in a metric space.

A single `Search` request may hit several nodes hosted by the same peer $p \in P$. Thus, in order to measure the costs of a `Search` operation processing at peer p, we have to sum

up the costs at individual nodes:

$$cost_p(\texttt{Search}) = \sum_{n \in nodes(p)} cost_n(\texttt{Search}).$$

Now we can define the **single-load** per peer as an average over last s search operations $\texttt{Search}_1, \texttt{Search}_2, \ldots, \texttt{Search}_s$ processed at p:

$$\textbf{single-load}(p) = \begin{cases} \frac{1}{s} \sum_{i=1}^{s} cost_p(\texttt{Search}_i), & \text{if } s \text{ queries already processed at } p, \\ \texttt{DontKnow}, & \text{otherwise.} \end{cases}$$

We also measure $\textbf{single-load}(n), \forall n \in N$ in order to have full information when choosing the balancing action. The value is defined accordingly.

Multiple Queries Load

Now, let us define **load** – the main indicator of the computational load in the system. The **load** of a node is measured as the total costs of all search queries that hit the particular node in a given period of time Δt (using principle of *sliding window*). Let $Q_n^{\Delta t}$ be the set of `Search` queries processed at node n in last Δt period of time:

$$\textbf{load}(n) = \begin{cases} \sum_{\texttt{Search} \in Q_n^{\Delta t}} cost_n(\texttt{Search}), & \text{if measured for whole } \Delta t, \\ \texttt{DontKnow}, & \text{otherwise.} \end{cases}$$

The $\textbf{load}(p)$ for a peer $p \in P$ can be defined as a sum of **load** values of its nodes:

$$\textbf{load}(p) = \sum_{n \in nodes(p)} \textbf{load}(n).$$

If any of the summands is `DontKnow` then the whole sum is `DontKnow`. The **load** is the main indicator of the peer's computational load.

Data Load

Finally, the balancing algorithm can utilize information about the size of data stored by nodes and peers – this information is especially useful while there is no query traffic in the system:

$$\begin{aligned} \textbf{data-load}(n) &= size(data_n),\ n \in N, \\ \textbf{data-load}(p) &= \sum_{n \in nodes(p)} \textbf{data-load}(n), p \in P. \end{aligned}$$

3.3 Operations

In this section, we describe the operations that LOBS can exploit for balancing and we discuss influence of these operations on the load. The LOBS system requires only two structure operations to be provided by the particular P2P network: `Split` and `Leave`. This is in order to make LOBS convenient for, e.g. native metric-based structures, e.g. *GHT** or *VPT** [6]. Besides the operations that LOBS expects to be provided by the particular P2P structure, there is a few operations defined by LOBS.

In general, when a new node wants to `Join` a P2P structure, an existing node has to split its data equally (if possible) and send the half of it to the new node. Complementary, a node that leaves the system (in a proper way) merges its data with some existing node. Using LOBS, a new logical node can be created at any peer $p \in P$ (without any outside-initiated `Join`) and similar situation is with the leave operation.

More formally, the following two operations modifying the logical structure of the network are required by LOBS – they are inherently supported by all P2P structures known:

Procedure n`.Split(p)`

RESULT: split node n creating a new node n' at peer p

$N = N \cup \{n'\}$; `// split` $data_n$ `equally, if possible`

$h(n') = p$

Procedure n`.Leave`

RESULT: remove node n from the structure

n`.Unify(r)`, $\forall r \in rep(n)$; `// remove replicas of` n`, if any`

merge $data_n$ with node n`.MergingNode` $\in N$

$N = N \setminus \{n\}$

There are two possibilities of how to handle split of a replicated node. The first is to remove the replicas before splitting and let the balancing mechanism replicate the new nodes if necessary. This may result in a smaller number of replicas but would cause a temporary overload of participating peers. We only mention this as a possible way but we apply the other method in the following.

The second way is to create replicas of the new nodes at the same peers at the original node was replicated. This can be done, again, in two ways – either by splitting the created replicas or by creating the replicas again from the new nodes. The first is convenient for systems with a cheap `Split` operation and expensive network transfers while the other way in the opposite case.

Let us discuss the expected influence of these operations on the load of the participating peers using the following notation.

Notation: Let us denote **load** the load value *before* a balancing action and **load**$'$ the value *after*.

Unlike the data-volume load, influence on the processing load can be hardly calculated precisely. We only estimate influence on the **load** indicator. The **single-load**(p) indicator of a peer $p \in P$ is not a pure sum of **single-load**$(n), \forall n \in \mathit{nodes}(p)$ since not always a single query is processed at all nodes at a peer and that makes a qualified estimation almost impossible.

Speaking of the **load**, we expect that the n.`Split` operation relocates one half of the **load**(n) from a peer to another. More precisely, the estimated influence of n.`Split`(p) is the following (let us denote $h(n) = q$ in the rest of this section):

$$\mathbf{load}'(q) = \mathbf{load}(q) - \mathbf{load}(n)/2,$$
$$\mathbf{load}'(p) = \mathbf{load}(p) + \mathbf{load}(n)/2.$$

These values assume that the processing costs of node n are split equally both in terms of the query processing time and frequency of queries which in not necessarily a precise estimation but the only reasonable, in our opinion.

As for the n.`Leave` operation, the replicas have to be removed before the operation itself. In practice, the balancing mechanism tries not to delete replicated nodes. Let us denote $h(n.\texttt{MergingNode}) = p$. The influence of n.`Leave` is the following:

$$\forall m \in \{n\} \cup \mathit{rep}(n) : \mathbf{load}'(h(m)) = \mathbf{load}(h(m)) - \mathbf{load}(n),$$
$$\mathbf{load}'(p) = \mathbf{load}(p) + (|\mathit{rep}(n)| + 1) \cdot \mathbf{load}(n).$$

The `Replicate` and `Unify` operations are defined by LOBS to manage the replication of nodes. The `Replicate` operation creates a new replica of a given node $n \in N$ at peer $p \in P$. `Unify` is a complementary operation to remove a given replica.

Procedure n`.Replicate(p)`

RESULT: replicate node n to peer p, $p \neq h(n)$

$N = N \cup \{r\}$, $\mathit{rep}(n) = \mathit{rep}(n) \cup r$; `// create replica r of node n`

$h(r) = p$; `// place the replica at peer p`

Procedure n`.Unify(r)`

RESULT: remove a replica r of node n, $r \in \mathit{rep}(n)$

$N \leftarrow N \setminus \{r\}$; `// delete node r`

Once a node is replicated, the query traffic is uniformly distributed to the replicas. Therefore, when a new replica of n is created by n.`Replicate`(p), loads of peer p and of all actual replicas should be adjusted to share the overall load uniformly. Expressed precisely, (the set $rep(n)$ is considered before the operation):

$$l_- = \frac{1}{|rep(n)| + 2}; \quad l_+ = \frac{|rep(n)| + 1}{|rep(n)| + 2}$$

$$\forall r \in rep(n) \cup \{q\} : \mathbf{load}'(r) = \mathbf{load}(r) - l_- \cdot \mathbf{load}(n),$$

$$\mathbf{load}'(p) = \mathbf{load}(p) + l_+ \cdot \mathbf{load}(n).$$

The load values are modified accordingly after the `Unify` operation.

Having more then one node per peer, we can ease the peer by *migrating* one of the nodes to a peer with lighter load – the last balancing operation is `Migrate`. Implementation of this operation requires cooperation of the specific P2P structure.

Procedure n.`Migrate`(p)

RESULT: migrate node n to peer p, $p \neq h(n)$

$h(n) = p$; `// place` n `to peer` p

`// let other nodes in the system know about the migration`

Let us discuss the way in which migrated node n propagates its new location to the rest of the system. Every node n in a P2P structure has a set of "links" to other nodes (the set was denoted as R_n in Section 3.1). These links define a binary relation L on N "having a link to a node": $\forall n \in N : (\forall m \in R_n : (n, m) \in L)$ (and nothing else is in L).

In some P2P structures, relation L is *symmetric* and every node n knows exactly which nodes have link to n. This works e.g. in tree-based structures *GHT** and *VPT**, in *MCAN*, or in *M-Chord* using Skip Graphs. Then a migrated node n can inform the relevant nodes about its new location at peer p. If the relation is not symmetric (e.g. in Chord-based structures), then the `Migrate` operation is implemented as a consecutive `Leave` and `Join` at peer p.

Migration of a replica is very simple – the replica informs its master node about the new location. Accordingly, master node informs its replicas, if migrated.

The effect of the `Migrate` to the load of the participating peers is the same as for the `Leave` operation.

Figure 3 schematically depicts three of the five operations which can be exploited by LOBS for the load balancing.

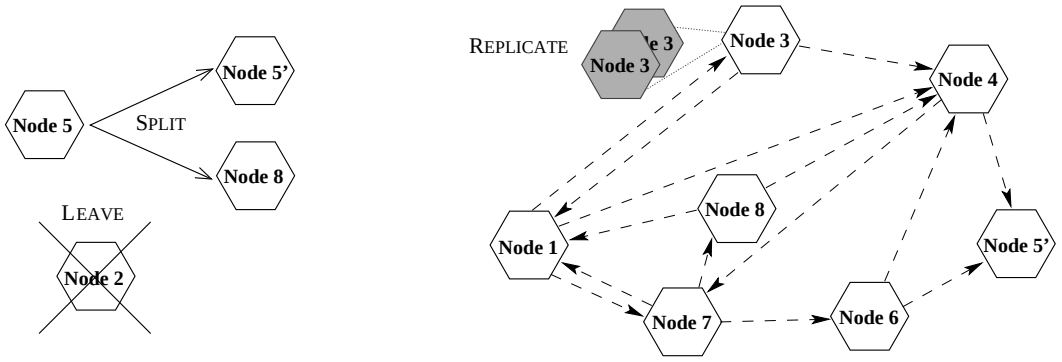

Figure 3: The `Split`, `Leave` and `Replicate` operations.

3.4 Global Knowledge

The P2P systems with nontrivial search paradigm typically exchange quite a high number of messages during the query processing and for the management. The LOBS system utilizes these messages and adds some piece of information to them in order to:

- maintain information about the average load in the system,
- exchange information about the most loaded and the least loaded peers in the system.

To calculate the average, we apply a distributed P2P message-driven algorithm [16] that uses the concept of *gossips*. It enables the peers to maintain an approximation of an average of some characteristic published by each of the peers. The characteristics may develop over the time. This algorithm meets the needs of LOBS and is slightly modified to use the standard messages instead of emitting its own – additional gossiping messages are sent only when there is no traffic for given time period. The `DontKnow` values are ignored by the algorithm.

Each peer maintains the following values: **avg-load**, **avg-single-load** and **avg-data-load** as the actual estimations of the overall averages of **load**,**single-load** and **data-load**, respectively. Each peer can then compare its own load values with the averages and eventually decide to perform a balancing action.

Each peer also maintains (approximated) information about the *least loaded* and the *most loaded* peers in the system. The following tuples are exchanged and kept up-to-date for each peer p:

$$\textbf{load-info}(\mathrm{p}) = \langle \mathrm{p}, \textbf{load}(\mathrm{p}), \textbf{single-load}(\mathrm{p}), \textbf{data-load}(\mathrm{p}), \mathit{Time} \rangle$$

where *Time* is the time stamp of the particular piece of information. Each peer maintains lists `MostLoadedPeers` and `LeastLoadedPeers` each of which contains a fixed number of these **load-info** tuples. The lists are appended to each message sent and are kept up-to-date by merging with the arriving lists. Peer $p \in P$ always updates and sends the **load-info**(p) tuple.

The `MostLoadedPeers` and `LeastLoadedPeers` lists are sorted according to the **load** values decreasingly and increasingly, respectively. The tuples which have **load** = `DontKnow` are excluded from the lists. Before a peer from either list is used for load balancing, it is contacted in order to update its load values. This is an alternative to finding suitable balancing partners by random sampling of known peers. Both techniques can be combined.

3.5 Load Balancing

Let us recall that, in compliance with the peer-to-peer paradigm, we try to construct an autonomous decision-making module present at each peer. The process of load-balancing can exploit a large variety of instruments described in the previous sections: creating several logical nodes at a peer, replication, two load measures based on processing costs, continuously updated global averages, information about peers with the highest and the least load in the system. The work process of the balancing module can be depicted by the simple schema in Figure 4.

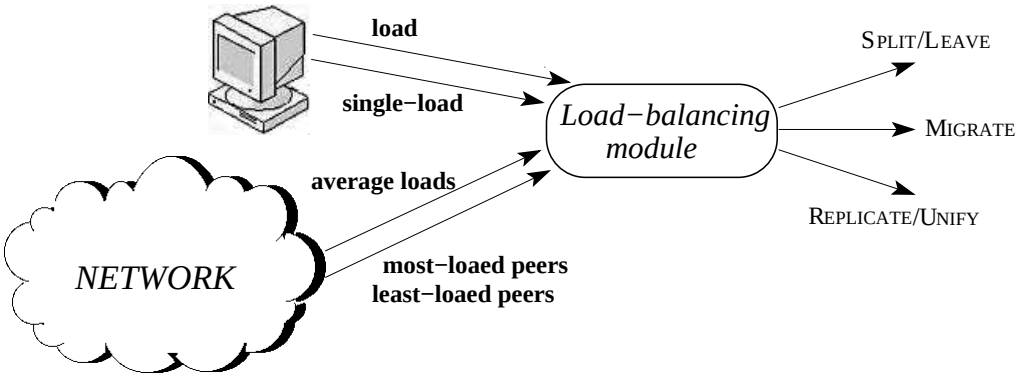

Figure 4: Work schema of the load-balancing module.

Both the inputs and possible outputs of balancing process are quite rich and, therefore, there is a high number of ways to define the specific load-balancing strategies and

algorithms. Before finding a balancing strategy, let us in detail formulate the goals we want to reach.

Goals of the Load Balancing

In general, we can see the following two ways of measuring the impact of the load balancing:

1. Monitor the specified load measures of individual peers – measure the fairness of the load distribution.

2. Observe influence of the balancing process on performance of the system – measure the practical impact.

While the first type of observation predicates of the quality of the balancing process, the second type gives evidence of the whole approach including the choice of the load measures.

We consider both types of the quality evaluation. The first type is represented by monitoring the ratio between the current maximal and minimal load of the peers. The balancing process tries to keep this *imbalance ratio* under a certain constant. This is quite a standard way [12] of measuring the fairness of the load distribution.

The practical impact of the balancing is measured by:

- the query throughput of the system – number of queries processed by the system within a fixed period of time,

- average response time of a single query,

- volume of work done by individual peers within a period of time and its distribution in the system.

It is important to measure also the costs of the load balancing. The data transfers increase the load of the network and also the local reorganizations are not for free. The `Split` operations can be more expensive in some systems [5] than in others [19, 11] and, in general, replication requires only minor inner reorganization. This requires a system-dependent cost model that could be taken into account by the balancing module and that would be used to evaluate the costs of the load balancing. In the presented version, we only measure the costs as the *data volume sent over network during a balancing action*.

The LOBS system also takes into account the following aspects:

- The computational load caused by the query traffic may be a highly fluctuant measure in comparison with the data load. Therefore, the load should be measured within a longer time period or as an average (see Section 3.2).
- The load is compared to a global average computed by a gossip-based algorithm, which could have a delay if a sudden change or a fluctuation of the traffic occur. Therefore, the overload should be rechecked before a balancing action is performed.
- The opportunity of having multiple nodes at a peer could lead to a dangerous nodes-explosion. Therefore, the LOBS system should in general prefer the `Migrate`, `Leave` and `Unify` operations over `Split` and `Replicate`.

These features lead to a "conservative" and reluctant behaviour of the balancing system and resulting in necessary temporary imbalance in the system. We consider this better than letting LOBS to generate needless or premature balancing actions, which are costly.

Balancing Strategies

In this section, we describe the decision-making procedure in the balancing module of every peer $p \in P$ (see Figure 4). The procedure is executed periodically and it has the following general features:

- The balancing action can affect only peers if their **load** differs from `DontKnow`.
- After every balancing operation that affects peer p, the **single-load**(p) and **load**(p) are set to `DontKnow` by resetting the buffers they are computed from (see Section 3.2). This ensures, that one "overload problem" invokes only one balancing operation.
- The overload is rechecked before performing a balancing action (see the previous Section).

The LOBS system considers the **load** the main indicator of the load since it represents the real computational load of the peer and covers both the complexity and the frequency of the queries hitting the peer. LOBS focuses on keeping the **load** imbalance ratio under the constant 4. The **single-load** is a side-indicator that focuses on the response time of a single query and LOBS tries to keep it under a reasonable limit (twice

an average). In other words, the goal of the LOBS balancing module at peer p is to maintain the following conditions:

$$1/2 \cdot \textbf{avg-load} \leq \textbf{load}(p) \leq 2 \cdot \textbf{avg-load} \quad (1)$$

$$\textbf{single-load}(p) \leq 2 \cdot \textbf{avg-single-load} \quad (2)$$

A load-balancing action initiated by an overloaded peer p typically tries to utilize a peer q with lesser load. LOBS can estimate the load transferred from p to q as described for individual operations in Section 3.3. Function `IsSafe` uses the conditions above to decide whether shifting load from p to q using a given operation wouldn't be source of another imbalance. Term `Operation` stands for `Split`, `Leave`, `Replicate`, `Unify`, or `Migrate`. This check can be considered only in some cases – overloading of q cannot be avoided always.

Function `IsSafe(Operation, n, p, q)`

RESULT: Returns **true** if given `n.Operation` which shifts load from peer p to peer q would not cause any other imbalance.

evaluate estimated influence of `n.Operation` on loads of p, q

RETURN

$$\textbf{reduced-load}(p) \geq 1/2 \cdot \textbf{avg-load} \quad \wedge$$
$$\textbf{increased-load}(q) \leq 2 \cdot \textbf{avg-load}$$

Algorithm 7 shows the strategy adopted when peer p is overloaded because of the main indicator – the **load**.

In compliance with the previous section, the strategy prefers the `Leave` and `Migrate` operations if there is more then one node at peer p. First, the possibility of deleting an existing node is checked – see procedure `CanLeave`.

If there is only one node at peer p and it is not a replica, either `Split` or `Replicate` operation is performed. The **single-load** serves as a side-indicator saying whether the overload is because of time-consuming processing of individual queries (then `Split`) or because of high query frequency (then `Replicate`).

If there is only a single node at peer p and it is a replica r then no balancing action is taken by p. The master node $m \in N, r \in \mathit{rep}(m)$ must be overloaded as well and the balancing should be initiated at peer $h(m)$.

Algorithm 7: Overloaded peer p: **load**$(p) > 2 \cdot$ **avg-load**

```
// consider only nodes n ∈ nodes(p) with load(n) > 0
IF |nodes(p)| ≥ 2 THEN
    // relocate a node from p
    let n be the n ∈ nodes(p) with the smallest load(n)
    IF CanLeave(p, n) THEN
        Leave(n); // prefer deleting node, if possible
    ELSE
        n.Migrate(q), where q ∈ LeastLoadedPeers such that load(q) < avg-load
ELSE
    IF n ∈ nodes(p) is a replica THEN
        EXIT
    IF single-load(p) > 2 · avg-single-load THEN
        n.Split(q), where q ∈ LeastLoadedPeers such that load(q) < avg-load
    ELSE
        n.Replicate(q), where q ∈ LeastLoadedPeers such that
        load(q) < avg-load
```

Procedure `CanLeave(p, n)`

```
RESULT: Returns true if node n can be deleted from p without making the peer
        hosting n.MergingNode overloaded and p underloaded.
IF n is a replica OR rep(n) ≠ ∅ THEN
    RETURN false
let us denot q a peer h(n.MergingNode)
RETURN IsSafe(Leave, n, p, q)
```

Algorithm 7 utilizes for balancing peer $q \in P$ which have **load**$(q) <$ **avg-load**. Such peer should always exist in the system, if any peer is overloaded. Section 4.1 contains deeper analysis of this algorithm and its influence on the system.

Algorithm 9 is applied when peer p that is underloaded, e.g. **load**$(p) < \frac{1}{2} \cdot$ **avg-load**. In general, it tries to find a heavily loaded peer q and perform some balancing action with q.

The algorithm first tries to find out whether some node can be removed from the system (either by `Unify` or `Leave`) to place more load on p. If not, it finds the most loaded peer known such that **load**$(q) >$ **avg-load**, which should always exist. By an analysis of q, some load is shifted from q to p either by `Migrate`, `Split`, or `Replicate`.

Algorithm 9: Strategy for peer p if $\textbf{load}(p) < \frac{1}{2} \cdot \textbf{avg-load}$

```
// consider only nodes n with load(n) > 0
IF ∃n ∈ nodes(p) that is replicated AND IsSafe(Unify, n, q, p) where q is the most
loaded peer such that q = h(r), r ∈ rep(n) THEN
    n.Unify(q), where q ∈ rep(n)
    EXIT
IF ∃n ∈ N such that h(n.MergingNode) = p AND CanLeave(q, n) THEN
    Leave(n)
    EXIT
let q ∈ MostLoadedPeers such that load(q) > avg-load
IF nodes(q) ≥ 2 THEN
    let n be node from nodes(q) with the smallest load(n)
    n.Migrate(p)
ELSE
    IF single-load(q)/avg-single-load ≥ load(q)/avg-load THEN
        n.Split(p) where n ∈ nodes(q)
    ELSE
        n.Replicate(p) where n ∈ nodes(q)
```

This algorithm cannot make any other peer underloaded and cannot make p overloaded unless the counterpart q were heavily overloaded before the balancing which is in impossible due to the following rule: *If a peer* q *is overloaded, Algorithm 7 has priority over letting Algorithm 9 exploit* q *for balancing of an underloaded peer* p.

Algorithm 10 is an auxiliary algorithm to be used if a peer is overloaded only due to **single-load** (and not according to **load** at the same time). It follows similar policy as Alg. 7 but is applied only if it does not cause any other imbalance.

Algorithms 7, 10 and 9 can be applied only when there is some query traffic in the system – the **avg-load** > 0. Although LOBS does not focus on balancing of other then processing load, it has a simple rule for utilization of the fresh peers joining the network even if there is no traffic. See Algorithm 11 for details on the simple rule.

4 Evaluation of LOBS

In this section, we analyze the performance, impact and costs of the load balancing using LOBS. The evaluation is partly on the theoretical level and partly by evaluation

Algorithm 10: Overloaded peer p: **single-load**$(p) > 2 \cdot$ **avg-single-load**

```
// consider only nodes n ∈ nodes(p) with load(n) > 0
IF |nodes(p)| ≥ 2 THEN
    // relocate a node from p
    let n be the n ∈ nodes(p) with the smallest load(n)
    IF CanLeave(p, n) THEN
        Leave(n); // prefer deleting node, if possible
    ELSE
        n.Migrate(q), where q ∈ LeastLoadedPeers is such peer that
        IsSafe(Migrate, n, p, q)
ELSE
    IF n ∈ nodes(p) is a replica THEN
        EXIT
    n.Split(q), where q ∈ LeastLoadedPeers such that IsSafe(Split, n, p, q).
```

Algorithm 11: Basic data-volume balancing (if there is no query traffic)

```
IF data-load(p) > 2 · avg-data-load AND
∃q ∈ LeastLoadedPeers : data-load(q) = 0 THEN
    IF |nodes(p)| ≥ 2 THEN
        n.Migrate(q), n ∈ nodes(p) with the smallest data-load(n)
    ELSE
        IF n ∈ nodes(p) is not a replica nor it is replicated THEN
            n.Split(q)
```

of a series of experiments conducted on the *M-Chord* structure [19] and real-life data (MPEG7 features from digital images).

4.1 Theoretical Analysis

In Section 3.5, we mentioned two general ways of measuring quality of the load balancing – to monitor the distribution of the defined load measure and to observe influence on the system performance. Theoretical analysis of the impact to the system performance (response time, query throughput, etc.) would be extremely difficult since it is influenced by a high number of application-dependent free variables: specific computational demands of the queries, average number of peers involved in the query process-

ing, etc. In our opinion, the experimental evaluation on the real-life data is of greater value.

Let us analyze impact of LOBS on the distribution of **load**. To make the analysis possible, let us assume the following (we comment on these premises below):

1. global averages are up-to-date;
2. the `LeastLoadedPeers` and `MostLoadedPeers` contain the actual least and most loaded peer, respectively;
3. the balancing actions have exactly the influence on load as estimated in Section 3.3.

Before we formulate a theorem about balancing of the overloaded peers, let us mention a simple lemma and remind a notation.

Lemma 4.1. $\exists p \in P : \mathbf{load}(p) > \mathbf{avg\text{-}load}$ *if and only if* $\exists q \in P : \mathbf{load}(q) < \mathbf{avg\text{-}load}$.

The lemma is obvious (proof straitforwardly from definition of the average) and we will use it in the following.

Notation: Let us denote **load** the load measure *before* a balancing action (or a sequence of actions) and $\mathbf{load}'$ the measure *after*.

Theorem 4.2. *Let us denote* $O \subseteq P$ *the set of all peers* p *such that* $\mathbf{load}(p) > 2 \cdot \mathbf{avg\text{-}load}$*. Balancing according to Algorithm 7 will lead to the state where* $\forall p \in O$*:* $\mathbf{avg\text{-}load} \leq \mathbf{load}'(p) \leq 2 \cdot \mathbf{avg\text{-}load}$ *and* $\forall q \in P \setminus O$ *will hold true:* $\mathbf{load}'(q) \leq 2 \cdot \mathbf{avg\text{-}load}$.

Proof. If $O = \emptyset$ then the conditions are trivially fulfilled. Otherwise Algorithm 7 is executed simultaneously for each peer $p \in O$ and the balancing process continues until $\forall p \in O : \mathbf{load}'(p) \leq 2 \cdot \mathbf{avg\text{-}load}$. Let us denote $U \subseteq P$ the set of all peers q such that $\mathbf{load}(q) < \mathbf{avg\text{-}load}$. According to Lemma 4.1, U is nonempty whenever $O \neq \emptyset$. Let us prove that in each step of the balancing process, triggered by an overloaded peer p, set U either shrinks or remain unchanged (no peer is removed nor added from/to U) but only for a finite number of consecutive steps. Proving this means that the balancing converges to a state in which $\forall p \in O : \mathbf{avg\text{-}load} \leq \mathbf{load}'(p)$ and since U is finite (we cannot remove peers from it ad infinitum), the process is finite, i.e. $\forall q \in P : \mathbf{load}'(q) < 2 \cdot \mathbf{avg\text{-}load}$.

Let us analyze all branches of Algorithm 7 (they cover all possible cases). All the branches, but the case when p hosts only one node and it is a replica, contain a balancing action.

- Operation `Leave`(n) exploits peer h(`n.MergingNode`) which does not have to be from set U and then U would not shrink. Since at most $1/2 \cdot$ **load**(p) is moved from p to q, **avg-load** $\leq$ **load**$'(p)$ and set U cannot grow. Now either **load**$'(p) \leq 2 \cdot$ **avg-load** and balancing process of p ends or the process is repeated but, since *nodes*(p) is a final set, operations `Leave` and `Migrate` cannot run ad infinitum.

The other operations exploit a peer $q \in U$ which must exist (Lemma 4.1) and is in list `LeastLoadedPeers` (Premise 2).

- Operation n.`Migrate`(q) where $n \in$ *nodes*(p) is with the smallest **load**(n). At most $1/2 \cdot$ **load**(p) is moved from p to q, therefore, **avg-load** $\leq$ **load**$'(p)$ and set U cannot grow. Now, if **avg-load** $\leq$ **load**$'(q)$ then U shrinks. Otherwise, either **load**$'(p) \leq 2 \cdot$ **avg-load** and balancing process ends or the process is repeated but, since *nodes*(p) is a final set, operations `Migrate` and `Leave` cannot run ad infinitum.

- Running n.`Split`(q) operation, $1/2 \cdot$ **load**(p) is moved from p to q, thus **avg-load** $\leq$ **load**$'(p)$ and **avg-load** $\leq$ **load**$'(q)$ and set U shrinks.

- Since always a master-node and all its replicas are uniformly loaded by queries, we presume that all replicas have equal **load**. Operation n.`Replicate`(q) decreases **load** of p and of its actual replicas by maximally $1/2 \cdot$ **load**(p) (see Section 3.3) and, thus, set U does not grow. Further, **load**(q) is increased by at least $1/2 \cdot$ **load**(p) (see Section 3.3) and set U shrinks.

If *nodes*(p) $= \{r\}$ and r is a replica then the master node $m \in N, r \in$ *rep*(m) must be overloaded as well ($m \in O$) and balancing is initiated at peer $h(m)$. □

Let us prove an analogous theorem for balancing of underloaded peers. Let us recall the following rule: *If peer* q *is overloaded, Algorithm 7 has priority over letting Algorithm 9 exploit* q *for balancing of an underloaded peer* p.

Theorem 4.3. *If* $\exists p \in P$: **load**(p) $< 1/2 \cdot$ **avg-load** *then balancing according to Algorithm 9 will lead to the state where for all peers* $q \in P$ *which were involved in the balancing process (including* p*):* $1/2 \cdot$ **avg-load** $\leq$ **load**$'(p) \leq 2 \cdot$ **avg-load**.

Proof. Operations `Leave` and `Unify` are performed only if not imbalance is caused (condition `IsSafe` is checked). Operations `Split`, `Replicate` and `Migrate` (of the least

loaded node) exploit a peer $q \in P$: **avg-load** $<$ **load**(q) and, thus, the actions cannot make q underloaded. After one of the operation, peer p is not overloaded unless $2 \cdot$ **avg-load** $<$ **load**(q) which is in contradiction with the rule above.

If **load**$'(p) < 1/2 \cdot$ **avg-load** then the process is repeated. This can happen only if operations `Unify`, `Leave` or `Migrate` were performed. But since sets $rep(n), n \in nodes(p)$ and $nodes(q)$ are finite, operation `Split` or `Replicate` must be applied in future which makes the balancing process finite. □

The balancing according to Algorithms 7 and 9 are run in parallel but since Alg. 7 cannot make any peer underloaded and vice versa, the processes do not influence each other. Of course, the ideal case is when an overloaded peer utilizes a low-loaded one before it becomes underloaded and runs it own balancing action (and vice versa).

Every time a peer happens not to satisfy Condition 1, either Algorithm 7 or 9 are activated and the **load** gets within the limits again. Therefore, the imbalance ratio is, in general, kept under the constant *four*. Every imbalance situation is confirmed before running a balancing action which brings a short-term violation of the constant limit.

The actual process may be a bit damaged by temporal imprecision of global averages and lists of the least and most loaded peers in the system (Premise 1 and 2). The average-calculating algorithm converges swiftly [16] and adapts to the load-fluctuations promptly but it depends on the message traffic in the system as well as the process of exchanging information about the peers' load. The intensity and distribution of messaging highly depends on particular P2P structure and types and frequency of queries. The correctness of the influence estimation of the balancing actions (Premise 3) depends on particular structure as well. Therefore, the real-life behaviour should be evaluated experimentally.

4.2 Experiments Design

The basic experiments scenario is to execute a set of queries in a similarity P2P structure and to measure various characteristics during the processing. The experiments with various parameters are conducted without load balancing and with LOBS under various circumstances and the results are compared.

The prototype implementation of LOBS was integrated into the MESSIF platform [7] – a framework that supports building data structures for similarity search based on metric space and that provides extensive support for designing P2P networks. Linking

LOBS with MESSIF brings two benefits: support for P2P approach, messaging, statistics, etc. and existence of several MESSIF-based implementations of similarity P2P structures – *GHT**, *VPT**, *M-Chord*, and *MCAN* [6]. Actually, load balancing in these structures was the primary motivation for development of LOBS.

In this report, we present experiments with *M-Chord* [19]. This P2P structure maps the metric space into a linear domain and uses the Chord navigation algorithm [21] (or Skip Graphs [4], alternatively) to distribute the data among the participating nodes. A basic similarity query is evaluated by identifying several intervals of the linear domain and navigating the query to nodes that cover the intervals. The `Split` and `Leave` operations are standard operations of Chord (Skip Graphs), the `Migrate` operation can be easily implemented for the Skip Graphs navigation since the relation of "link between two nodes" is symmetric (see Section 3.3). We used the Skip Graphs version for the experiments.

The application domain we focused on was similarity searching in digital images. The testing dataset was formed by MPEG7 features extracted from one million color images. From each image, we extracted these features: Scalable Color, Color Structure, Color Layout, Edge Histogram, and Homogeneous Texture [18]. Each of these features can be compared using a metric function and the distance function used in the experiments is a weighted sum of these metrics. Evaluation of this function is computationally intensive (one distance computation takes a 0.1 ms on standard hardware) and is an ideal candidate for balancing the processing load of the peers.

The experiments were conducted on networks composed of various number of peers – from 50 to 300. The overall size of the dataset was fixed to one million in order to investigate the behaviour of structures with various data volume per peer.

An important question when designing experiments is the set of search queries to be posed into the system. We executed *range queries* [23] which are the basic similarity queries used with the metric-space abstraction. Given a query object q and a radius r, the **Range**(q, r) query returns all objects in the database whose distance from object q is lower than or equal to r. The queries used in the experiments had various radii (from a reasonable interval) in order to model the diversity of a real traffic. The query objects were taken from the following three sets:

uniform a set of query points randomly uniformly selected from the dataset. This distribution is the most natural but it does not reflect the query traffic prevailing in real systems.

sliding window for **load** (Δt)	90 s
number of queries for **single-load** (s)	10
balancing period	10 s
number of rechecks before balancing	3
size of `MostLoadedPeers` and `LeastLoadedPeers`	10

Table 1: LOBS parameters

zipfian a (multi)set of queries that reflects the Zipfian (or power law) distribution (see below). This distribution better reflects the real-life query traffic. It places higher demands on the load balancing.

one-query a single query repeatedly posed into the system. This scenario is not realistic but it could prove the ability of LOBS to cope with the most difficult query distribution.

The Zipfian distribution of a multiset represents, in general, that: the most frequent object occurs approximately twice as often as the second most frequent object, which occurs twice as often as the fourth most frequent object, etc. Since we consider the similarity search, we slightly modify the rule as follows: we do not put the *very same* objects to the query set several times, but we create *clusters* of very similar objects and we consider these objects identical.

The values of other LOBS parameters are summarized in Table 1.

4.3 The Measurements

The LOBS measures the computational load of the peers. In the center of the load definitions (see Section 3.2) are the costs of searching the local data: *cost*(`Search`(*data*)). This measurement is application-dependent and should approximate the time spent by local searching.

Since the experiments are performed on a structure that uses the metric space model of data, we measure the *cost* as *the number of evaluations of the distance function*. This value is a common indicator of the metric structure's efficiency. The CPU costs of other operations are usually practically negligible compared to the distance evaluation time. In this case, this is a fair measurement which is independent of implementation efficiency, but any other *cost* measure is possible (pure CPU time, number of I/O operations, etc.)

As discussed in Sections 3.5 and 4.1, we run the experiments in order 1. to observe distribution of the **load** (imbalance ratio) and 2. to evaluate influence of the balancing on the system performance. More specifically, we monitor and report on the minimal and the maximal load of peers in the system and the development of these values in time. The *imbalance ratio* is calculated from these values. The impact of the balancing on system performance is assessed by the following measurements.

- *Parallel distance computations* is the maximal number of distance evaluations performed in a sequential manner during one query processing. In other words, it is the longest branch of the parallel processing of the query. It is a good approximation of the *response time of a single query*.

- *Overall parallel distance computations* – this value expresses the *total processing time of a set of queries*. It is measured as the maximal number of distance computations performed at a single peer during a simultaneous processing of a set of queries Q:

$$\max_{p \in P} \left\{ \sum_{\texttt{Search} \in Q} cost_p(\texttt{Search}) \right\}.$$

- The *interquery improvement* represents the average number of queries that can be processed by the system simultaneously without slowing the processing down – the *throughput* of the system. We obtain this value as a ratio between sum of the processing times of individual queries from a query set Q divided by the time of simultaneous processing of Q. Expressed formally:

$$I_Q = \frac{\sum_{\texttt{Search} \in Q} \textit{parallel distance computations of } \texttt{Search}}{\textit{overall parallel distance computations of } Q}.$$

- *Histogram of work* represents the volume of work (number of distance computations) done by individual peers during processing a set of queries Q. This provides us with a nice intuitive insight into the work distribution.

We present these values rather then actual response times in time units since our testing environment uses virtualization and maps several peers on one physical CPU. Therefore, the actual improvement achieved by reorganization of the work is not significant since the physical CPUs are usually fully loaded even without the load balancing.

As discussed in Section 3.5, we measure the costs of the load balancing as the *data volume transferred over network during the balancing processes*. We also report the number of individual balancing actions performed.

4.4 Experimental Results

This section provides evaluation and interpretation of the experiments' results. Section 4.4.1 deals with the load distribution during the balancing process while Section 4.4.2 captures the impact of the balancing on the query evaluation performance. Both sections contain results for various numbers of peers and for various query distributions (see Section 4.2).

4.4.1 The Load-Balancing Process

As discussed above, the indicator we use for measuring the load distribution quality is the *imbalance ratio* – ratio of the most loaded and the least loaded peer in the system. We observe the development of these values while the system processes a set of queries. Each experiment run starts in a network with a fair *data distribution*, with one logical node per peer, and with no replicas.

To properly evaluate the influence and costs of LOBS, we first pose the query set to the system with the balancing off and then turn the balancing on and execute the same query set twice again (explained below). The results for 50 peers with the **uniform** query set are in Figure 5.

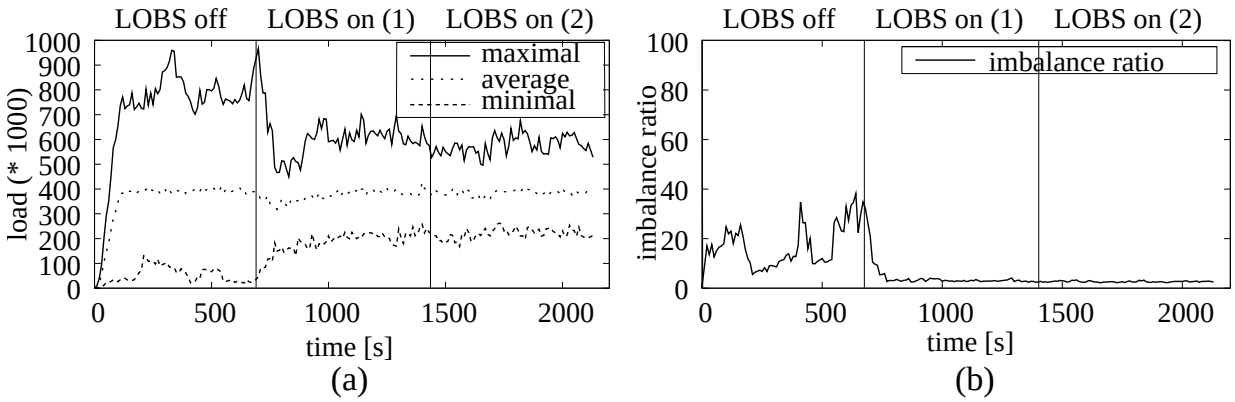

Figure 5: Load development (a), imbalance ratio (b). Query set: **uniform**, 50 peers.

When the balancing is off, the minimal and maximal values of load differ significantly making the imbalance ratio fluctuate between 8 and 40 (all imbalance ratio graphs have maximal value of 100). The LOBS balancing makes the imbalance ratio fall down and stay very close to value 4 which is a desired behaviour. The observable fluctuations of the maximal load are caused by non-homogeneous character of the randomly-generated query set.

The scenario when the balancing is turned abruptly on is not very natural and therefore we pose the same set of queries to the system once more – denoted as "LOBS on (2)". The main difference between results of LOBS on (1) and (2) are the costs of the balancing – see Table 2.

	LOBS on (1)	LOBS on (2)
Number of actions	13	2
`Split`	3	0
`Replicate`	7	1
`Migrate`	2	1
`Leave`	1	0
`Unify`	0	0
Transferred data	262391 (26 %)	43674 (4 %)
Transf. data per query	262	43

Table 2: Load-balancing costs. Query set: **uniform**, 50 peers.

The table summarizes the number of balancing actions performed during query processing and the number of data items transferred by the balancing. We can see that the initial correction of load imbalance after turning the balancing on naturally has higher requirements than a balancing process which has already been running for some time. The very low balancing costs of "LOBS on (2)" prove the property of LOBS to overcome the short-term fluctuations of the load without premature balancing.

The last line of the table illustrates the average number of data objects transferred by LOBS per a processed query. This value can be compared with the average query recall which was about 500 objects. Therefore, the load of the network is increased by factor 1/2 for the non-natural scenario LOBS on (1) and by factor 1/10 in the other scenario.

Figure 6 depicts the same experiment with the Zipfian distribution of the query set. We can see that the imbalance is higher then for the **uniform** query set (which was expected) but LOBS copes with the imbalance swiftly and keeps the loads stable after several initial fluctuations.

Having a higher imbalance to cope with, the costs of the balancing are slightly higher in the first run with LOBS on – see Table 3. On the other hand, a more realistic scenario, represented by LOBS on (2), seem to have even lower costs then for the **uniform** query set. It is caused by a higher stability of the load generated by the **zipfian** set.

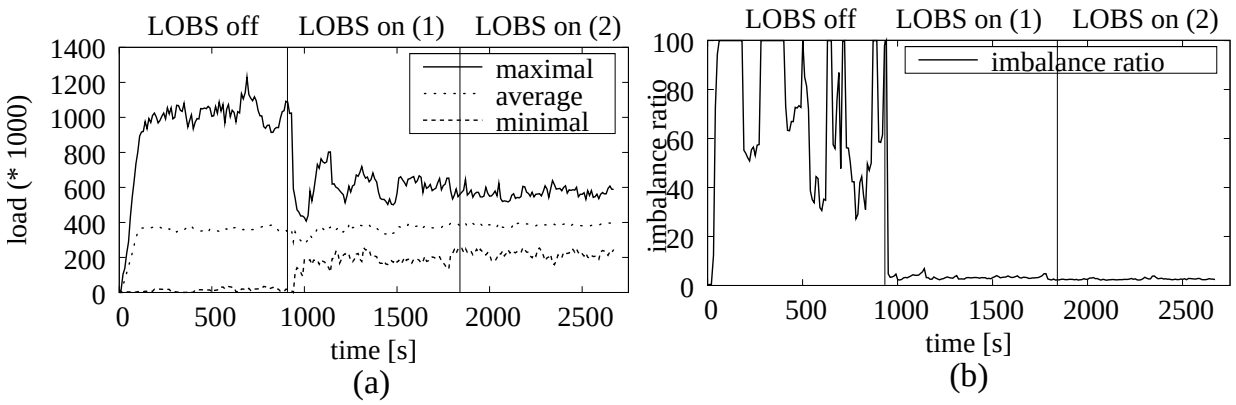

Figure 6: Load development (a), imbalance ratio (b). Query set: **zipfian**, 50 peers.

	LOBS on	LOBS on (2)
Number of actions	20	3
Transferred data	366565 (36 %)	31512 (3 %)
Transf. data per query	366	31

Table 3: Load-balancing costs. Query set: **zipfian**, 50 peers.

The third query set used for testing represents a single query repeatedly posed into the system. Since this query does not hit all the peers in the system, the imbalance ratio is maximal, when the balancing is off – see Figure 7. LOBS keeps the imbalance ratio very stable and on the desired level.

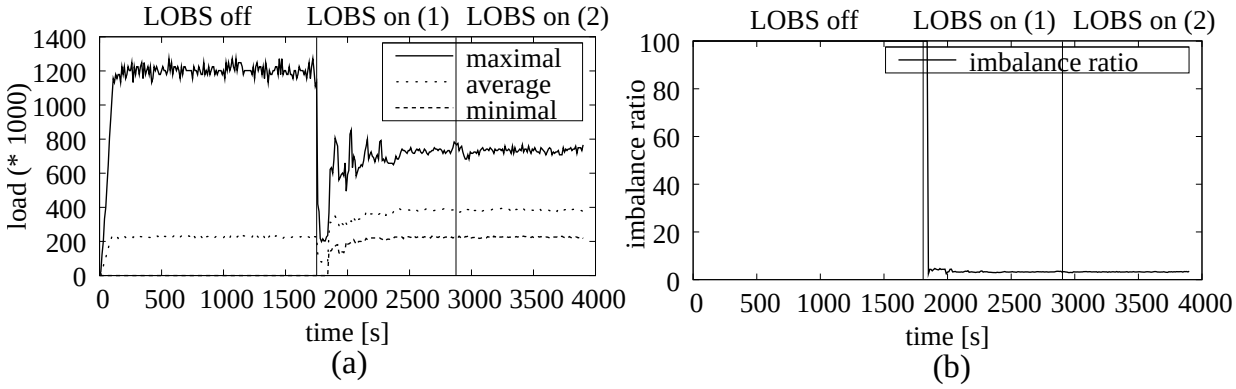

Figure 7: Load development (a), imbalance ratio (b). **One-query** set, 50 peers.

Table 4 shows that costs of the first run with balancing on are quite high but this query set and this scenario of turning balancing on abruptly serve only for testing purposes. On the other hand, the second execution of the same query set requires no balancing at all – this is due to a very stable load generated by the queries.

	LOBS on	LOBS on (2)
Number of actions	38	0
Transferred data	431050 (43 %)	0
Transf. data per query	431	0

Table 4: Load-balancing costs. Query set: **one-query**, 50 peers.

All the results presented so far are for a 50-peers network. Increasing the number of peers, the trends of the load development and imbalance ratio are very similar, therefore, we do not present all the results – only an example in Figure 8. Please, note that the actual load values are proportionally smaller than with a smaller set of peers.

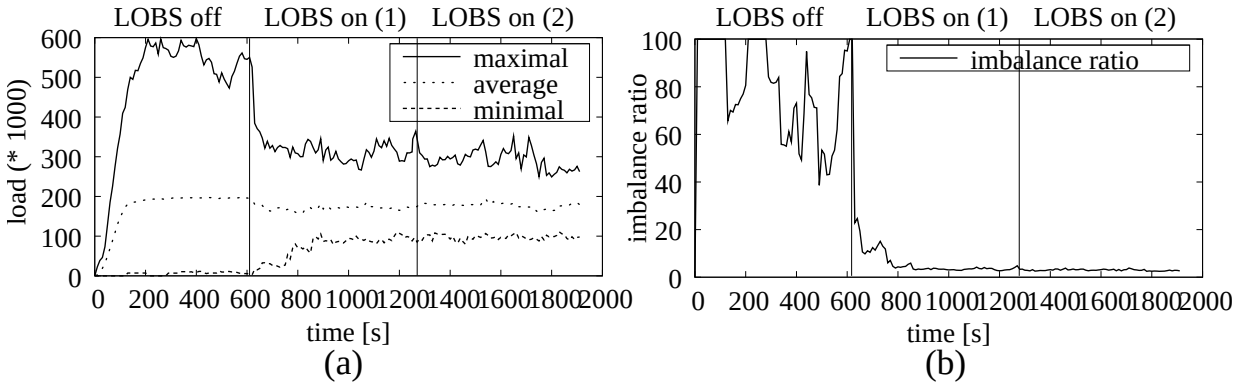

Figure 8: Load development (a), imbalance ratio (b). Query set: **zipfian**, 150 peers.

Let us observe the balancing costs as varying the size of the network. Figure 9 shows the volume of data transferred while balancing for various number of peers and for all query sets (in both phases – with LOBS on (1) and (2)). We can see that the costs for the **zipfian** and **one-query** sets slightly decrease as the network grows. It is caused by a finer partitioning of the dataset and better-targeted balancing actions.

On the other hand, costs for the **uniform** query set slightly grow with the network. The reason is the following: Having a larger network with a finer partitioning of the data space, a smaller percentage of peers is involved in the processing of a single query. Since the query frequency is the same for all network sizes (querying simultaneously from ten peers), the heterogeneity of the randomly generated query sets causes higher fluctuations of the load. These fluctuations require more frequent balancing actions. The lower load stability of larger networks could be improved by increasing the query frequency but we wanted to keep this parameter fixed for all experiments.

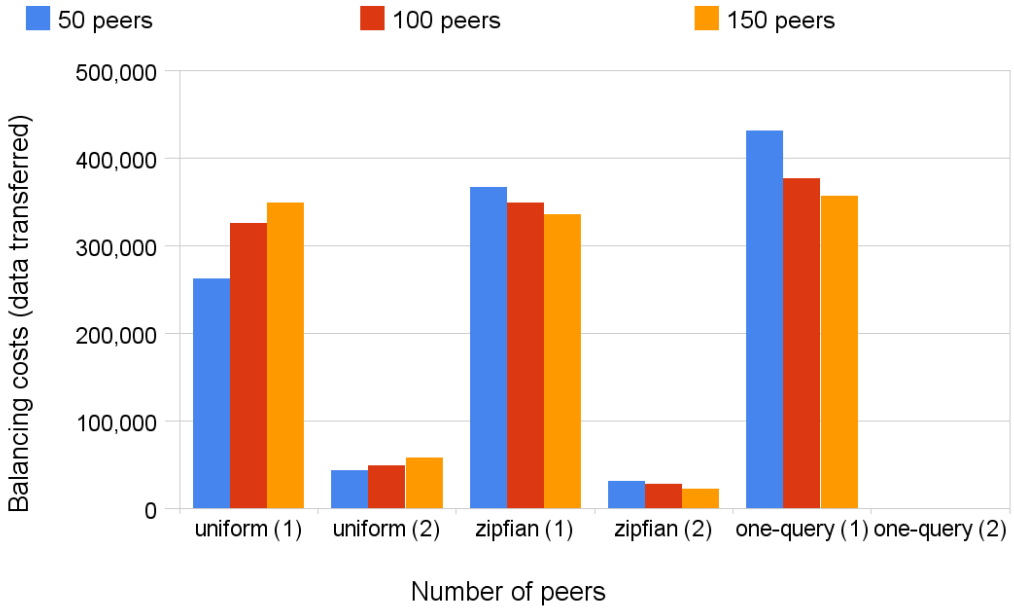

Figure 9: Costs of load-balancing while growing the network.

4.4.2 The Load-Balancing Impact

The previous section analyzed the process of balancing in terms of load development. Now, let us observe influence of the balancing to the query-processing performance of the system. First, we present the *histogram of work* measured as the number of distance computations performed by individual peers during processing of given query set – Figure 10.

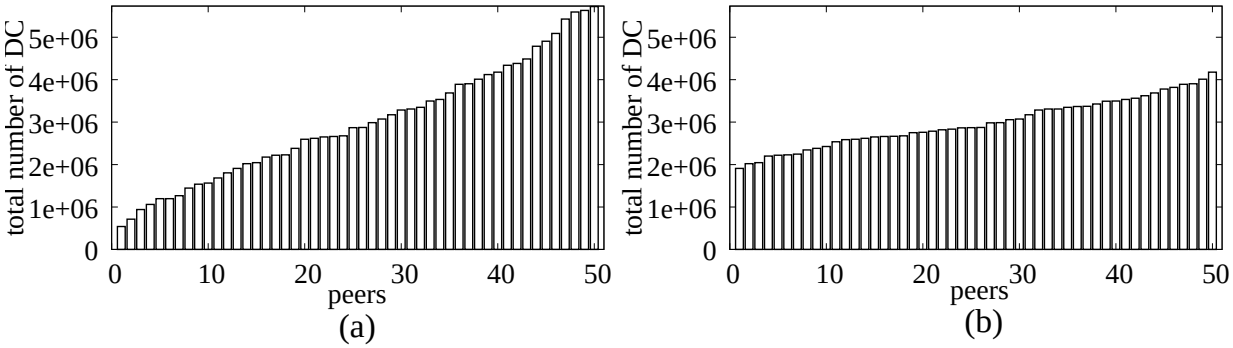

Figure 10: Work histogram without LOBS (a) and with LOBS (b). Query set: **uniform**, 50 peers.

We can see that even with a uniform query set and uniform data distribution within the network, the distribution of work done by individual peers is skewed. LOBS makes the work distribution more fair.

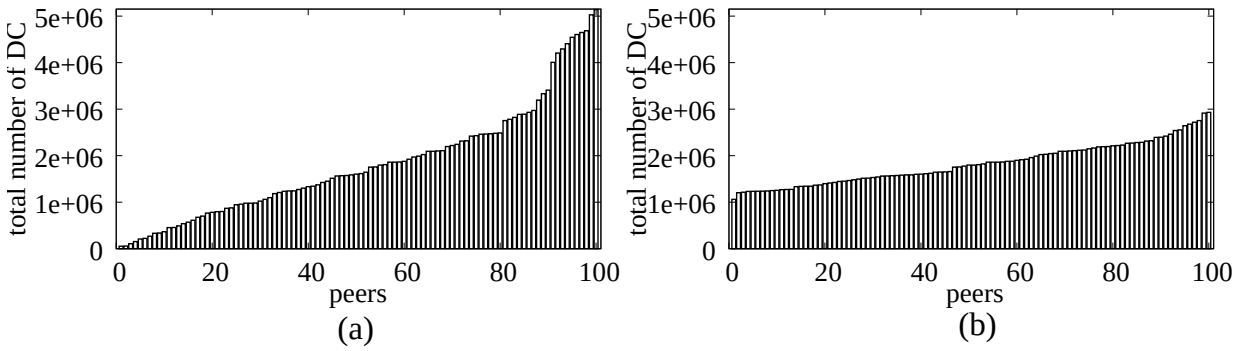

Figure 11: Work histogram without LOBS (a) and with LOBS (b). Query set: **zipfian**, 100 peers.

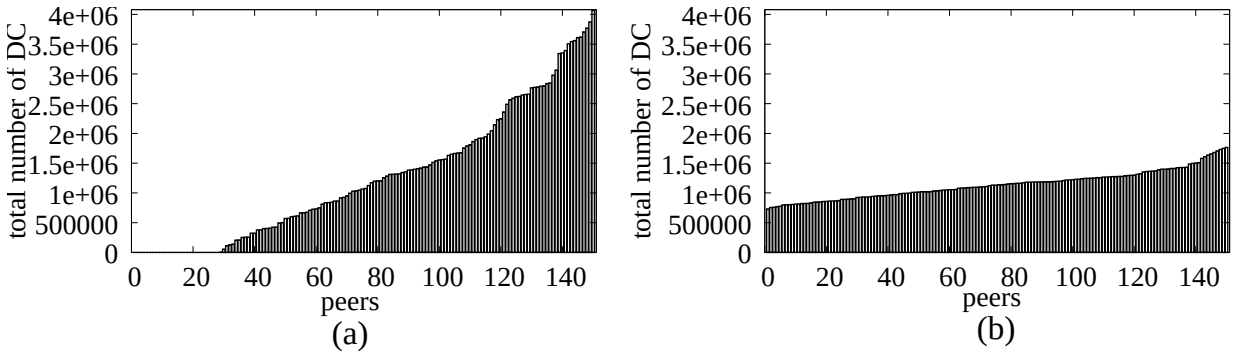

Figure 12: Work histogram without LOBS (a) and with LOBS (b). Query set: **one-query**, 150 peers.

Even a more skewed work histogram can be observed for the **zipfian** query set in Figure 11 (the results are for 100 peers). LOBS managed to balance the work, though. As discussed in the previous section (Table 3), this balancing process is more expensive with comparison to the **uniform** query set.

Figure 12 depicts the histogram of work for the **one-query** set. Naturally, a single query hits only a subset of the peers which can be observed in the part (a) of the figure. LOBS has employed the rest of the peers and made the work histogram almost fully balanced (b).

Finally, we present graphs which give evidence of the throughput and response time of the system with balancing off and on. As discussed in Section 4.3, we measure values *parallel distance computations*, *overall parallel distance computations* for a set of queries and an *interquery improvement*.

Figure 13 depicts the most important indicator of the system throughput – an approximation of the total processing time of a set of queries. We can see that values with

LOBS off are significantly higher for **zipfian** query set and **one-query** set. Turning the LOBS on reduces the values especially significantly for these two query sets – the values with balancing are comparable with the **uniform** query set. Naturally, as the number of peers grows, a set of queries is processed in a shorter time period.

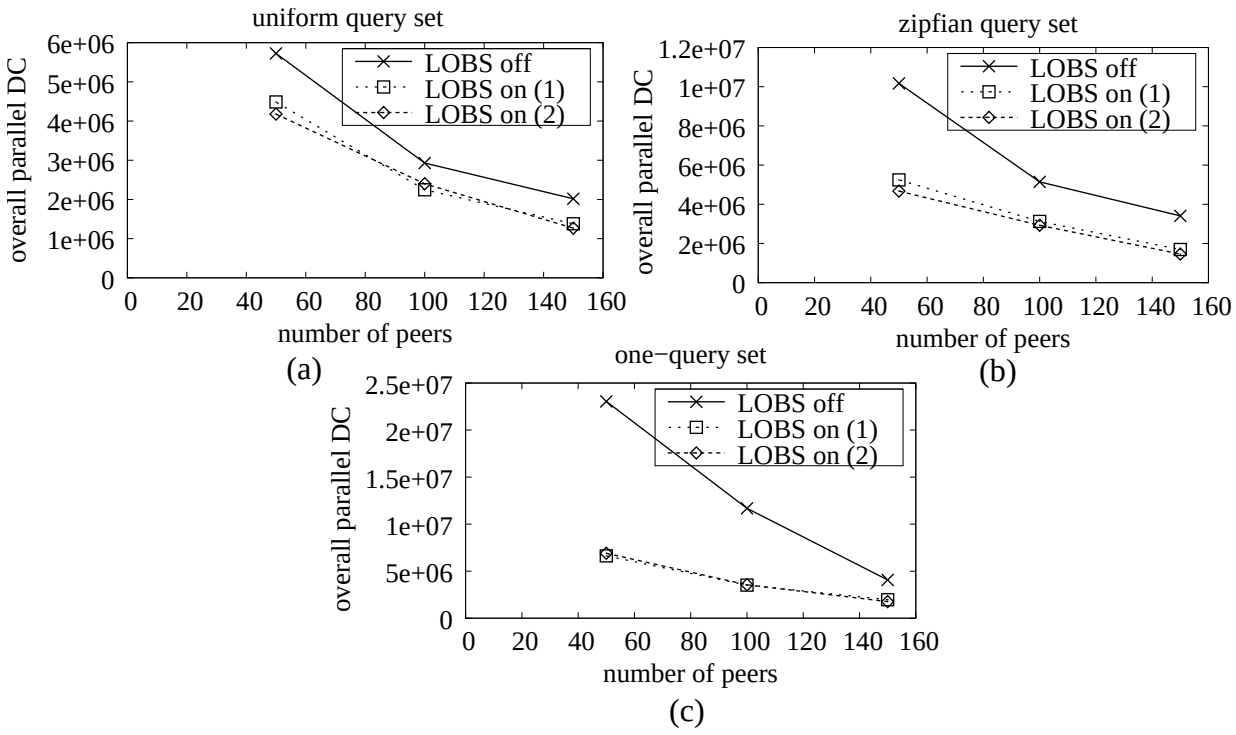

Figure 13: Overall parallel costs. Query sets: **uniform** (a), **zipfian** (b), **one-query** (c).

In general, placing several logical nodes or replicas at one physical peer can increase the response time of a single query, if this query hits more then one node at a peer. Knowing this, we introduced the **single-load** measurement to take into consideration the response time also. Figure 14 shows the *parallel distance computations* as an approximation of a single query response time.

We can see that using LOBS has worsened the response time a little bit for **uniform** query set, improved it for **zipfian** and, expectably, significantly improved it for **one-query**. The value falls with the growing number of peers in the network.

Finally, we present the *interquery improvement ratio* – approximation of an average number of queries that can be processed by the system simultaneously – see Figure 15. We can see that usage of LOBS increases this value for all query sets. Looking at the graph for **one-query** (c), we can see that the ratio without LOBS is equal to *one* – the system cannot process several identical queries in parallel without replication. The increase of the ratio with LOBS seems rather slender but it is caused by a contemporary

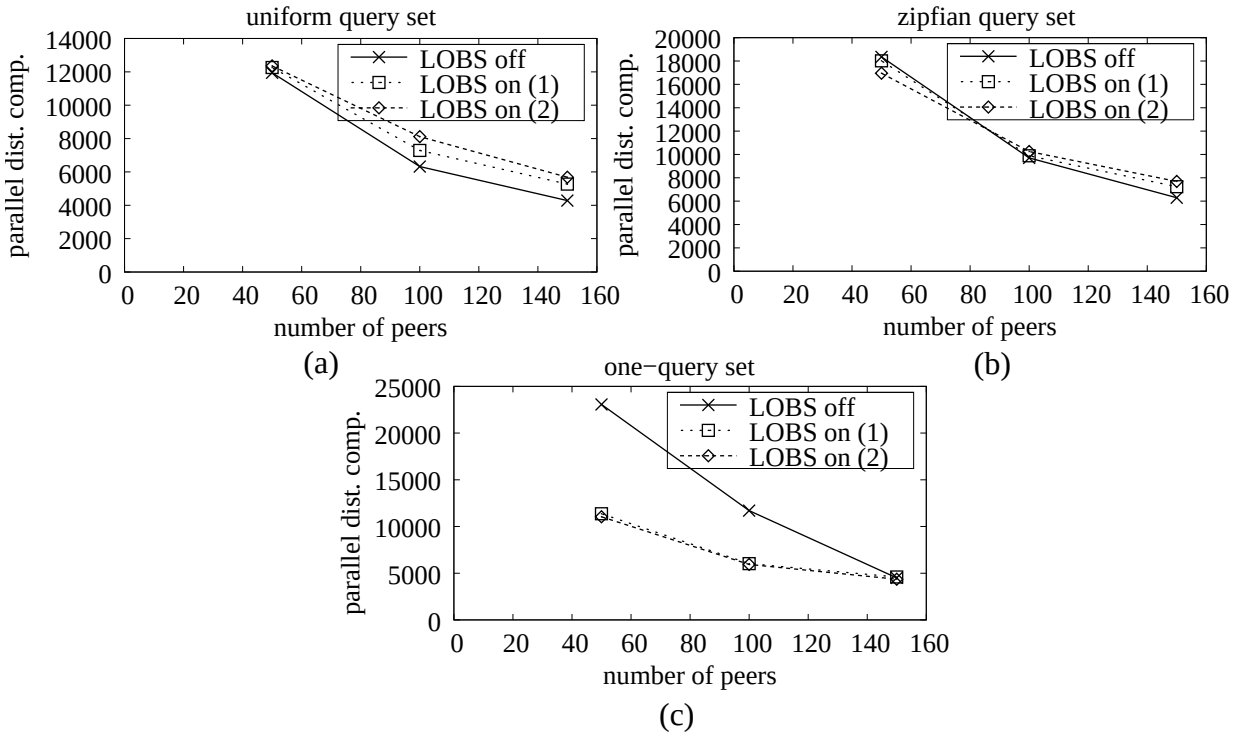

Figure 14: Average parallel costs. Query sets: **uniform** (a), **zipfian** (b), **one-query** (c).

reduction of the response time (see Figure 14 (c)) – the single query response time is in the numerator of the ratio.

5 Conclusions and Future Work

The similarity searching is the focus of attention of many academical and commercial researchers since the real content-based retrieval is both difficult and very expensive in terms of processing time. Especially huge data collections, like "all digital images on the WWW", require massive distributed processing and advanced indexing techniques, should they be efficiently searched using similarity. Recently, there have emerged some attempts [6] to exploit the peer-to-peer paradigm to achieve this goal. Our work is motivated by these systems.

Due to high computational demands of the similarity query paradigm, the load of the individual peers is measured as the "computational load". The distribution of this load is highly skewed even for a uniform query distribution and this fact calls for a load balancing. Existing P2P balancing techniques cannot be used and therefore we propose LOBS – a novel and general system for load-balancing in P2P structures with the following features:

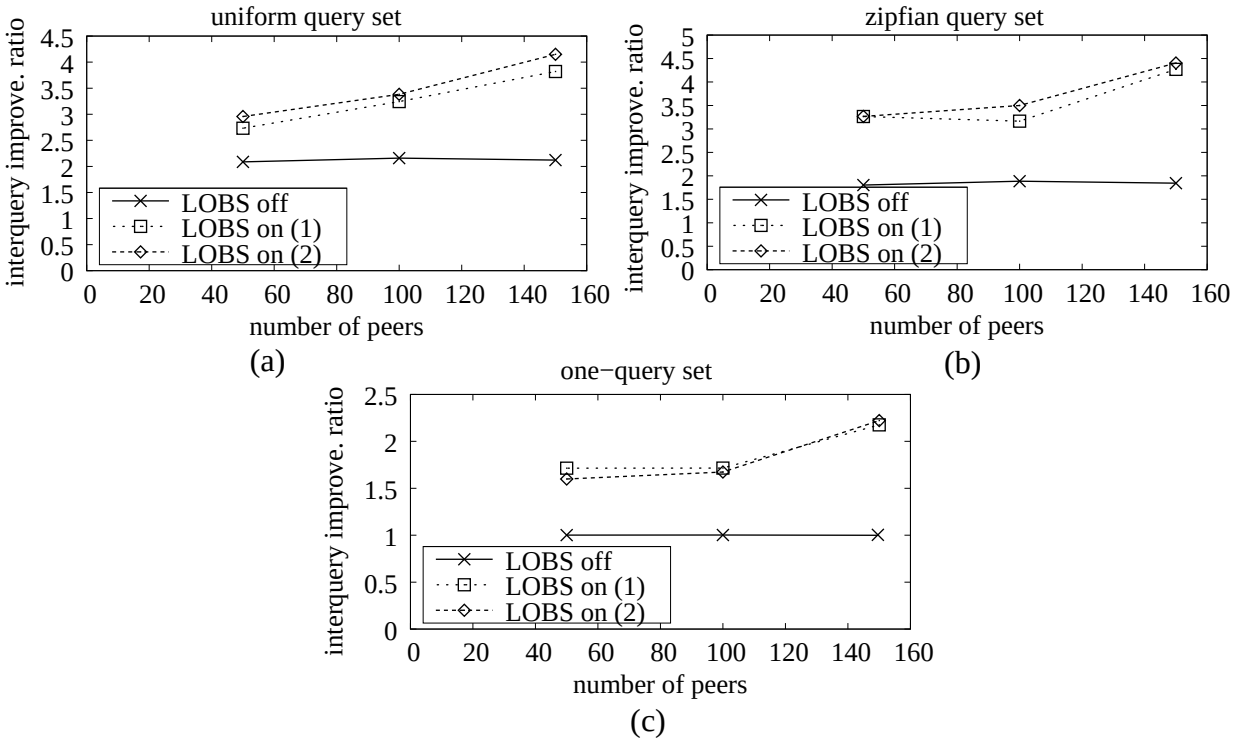

Figure 15: Interquery improvement ratio. Query sets: **uniform** (a), **zipfian** (b), **one-query** (c).

- it considers the computational load of the peers,
- it does not presume a linear and range-partitioned data domain and network architecture,
- separates the logical and physical layers of the system,
- analyzes the source of the load precisely and uses either data relocation or replication,
- the balancing strategies are rather "conservative" and reluctant to ignore temporary load fluctuations and to avoid needless balancing actions.

We have implemented a prototype of LOBS and have executed a number of experiments with system *M-Chord* [19] and a real-life dataset. The results prove that LOBS is able to cope with any query-distribution and that it improves both the utilization of resources and the system performance of query processing. The costs, in terms of data transferred due to the balancing, seem to be very small in a "living system" where the balancing had time to adapt to a query-traffic. If the balancing is turned on abruptly in

a never-balanced system, the balancing actions transfer up to 40 % of the database by means of data replication and relocation. To have a comparison – this generates approximately the traffic caused by processing of 1000 queries. These costs correspond to the level of imbalance observed in the network without balancing.

We believe that this work is a step towards applications that would efficiently manage and search up to hundreds of millions multimedia objects with a heavy query traffic. Our research group plans to build a prototype of such a distributed structure and load balancing will be an important component of the system architecture.

References

[1] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. The Quest for Balancing Peer Load in Structured Peer-to-Peer Systems. Technical report, EPFL, Swiss, 2003.

[2] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[3] James Aspnes, Jonathan Kirsch, and Arvind Krishnamurthy. Load balancing and locality in range-queriable data structures. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 115–124, New York, NY, USA, 2004. ACM Press.

[4] James Aspnes and Gauri Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, 2003.

[5] Michal Batko, Claudio Gennaro, and Pavel Zezula. Similarity grid for searching in metric spaces. *DELOS Workshop: Digital Library Architectures, Lecture Notes in Computer Science*, 3664/2005:25–44, 2005.

[6] Michal Batko, David Novak, Fabrizio Falchi, and Pavel Zezula. On scalability of the similarity search in the world of peers. In *Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 – June 1, 2006*, pages 1–12, New York, NY, USA, 2006. ACM Press.

[7] Michal Batko, David Novak, and Pavel Zezula. Messif: Metric similarity search implementation framework. In C. Thanos and F. Borri, editors, *DELOS Conference 2007: Working Notes, Pisa, 13-14 February 2007*, pages 11–23. Information Society Technologies, 2007.

[8] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[9] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. MAAN: A multi-attribute addressable network for grid information services. In *GRID '03: Proceedings of the Fourth International Workshop on Grid Computing*, pages 184–191, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-Ring: An index structure for peer-to-peer systems. Technical Report TR2004-1946, Cornell University, NY, 2004.

[11] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content-addressable network for similarity search in metric spaces. In *Proceedings of the 3rd International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P 2005), Trondheim, Norway, August 28–29, 2005*, pages 126–137, August 2005.

[12] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. Technical report, Stanford U., 2004.

[13] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: Multi-dimensional queries in P2P systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24, New York, NY, USA, 2004. ACM Press.

[14] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings of the 2004 Conference on Computer Communications, Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2004)*, volume 4, pages 2253–2262, 2004.

[15] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, New York, NY, USA, 2004. ACM Press.

[16] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS '03: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, page 482, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Dejan Milojičić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[18] MPEG-7. Multimedia content description interfaces. part 3: Visual. ISO/IEC 15938-3:2002, 2002.

[19] David Novak and Pavel Zezula. M-Chord: A scalable distributed similarity search structure. In *Proceedings of First International Conference on Scalable Information Systems (INFOSCALE 2006), Hong Kong, May 30 – June 1, 2006*, pages 1–10, New York, NY, USA, 2006. ACM Press.

[20] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), San Diego, California, August 27-31, 2001*, pages 161–172. ACM Press, 2001.

[21] Ion Stoica, Robert Morris, David R. Karger, Frans M. Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM 2001), San Diego, California, August 27-31, 2001*, pages 149–160. ACM Press, 2001.

[22] Peter Triantafillou, Theoni Pitoura, and Nikos Ntarmos. Replication, load balancing and efficient range query processing in DHTs. In *Proceedings of the International Conference on Extending Database Technology (EDBT), Munich, Germany*, 2006.

[23] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer-Verlag, 2006.