A Scalable P2P Platform for the Knowledge Grid

Hai Zhuge, *Senior Member*, *IEEE*, Xiaoping Sun, Jie Liu, Erlin Yao, and Xue Chen

Abstract—The Knowledge Grid needs to operate with a scalable platform to provide large-scale intelligent services. A key function of such a platform is to efficiently support various complex queries in a dynamic large-scale network environment. This paper proposes a platform to support index-based path queries by incorporating a semantic overlay with an underlying structured P2P network that provides object location and management services. Various distributed indexing structures can be dynamically formed by publishing semantic objects as indexing nodes. Queries are forwarded along the chains of semantic object pointers to search for objects. We investigate the deployment of a scalable distributed trie index for broadcast queries on key strings, propose a decentralized load balancing method for solving the problem of uneven load distribution incurred by heterogeneity of loads and node capacities and by the distributed trie index, and give an approach for improving the availability of the semantic overlay and its trie index. Experiments demonstrate the scalability of the proposed platform.

Index Terms—Peer-to-peer, semantic overlay, knowledge grid, path query, distributed trie index, load balancing, replication.

1 Introduction

IMAGINE (Integrated Multidisciplinary Autonomous Global Innovation Networking Environment) is a scalable Knowledge Grid environment for effectively sharing and managing semantic-rich data and knowledge for geographically dispersed cooperative scientific research [39], [41]. The scalability, availability, and semantic-based operations are three major challenges of implementing the environment. Building a semantic overlay on a P2P network is a way to obtain both the semantic-based operations and scalability, but this encounters the following three main issues:

1. Provide architectural extensibility for different types of complex queries.
2. Obtain scaling performance of queries.
3. Improve the utilization and the availability of the semantic overlay.

Approaching above targets cause conflicts in many aspects of dynamic and large-scale networks. So, we need to consider a reasonable trade-off so that an acceptable scalability of the whole system can be achieved, rather than in a certain aspect.

The initial studies of P2P networks mainly focus on efficient resources locating in dynamic and large-scale environments. Unstructured P2P networks adopt flooding or random walk methods on randomly connected networks (e.g., [24]). To solve the scaling problem of flooding search, DHT (Distributed Hash Table) is used to build structured overlays of various topologies (e.g., [7], [21], and [37]), where each node is responsible for a range of IDs allocated by DHT mechanisms, and messages are routed along deterministic paths to the target nodes. Most of those P2P systems are limited to data or file sharing services. Since DHT overlays encounter difficulties in supporting complex queries, many specific P2P overlay topologies and routing methods are proposed to support particular types of query (e.g., [1], [4], and [9]). There still lacks open and synthetic architectural solutions for applications with various types of query.

This paper presents IMAGINE-P2P, a scalable platform that supports path queries (queries are forwarded along paths of an index to find the matched keys) by publishing nodes of an index on a structured P2P overlay network. The platform has an object overlay that uses the Chord DHT overlay to manage objects. Above the object overlay is a semantic overlay where objects are published with semantic relations of neighboring nodes of an index to support path queries. The platform deploys a distributed trie index on the semantic overlay to support wildcard and broadcast queries on key strings. To scale the trie index in the distributed environment, we design a compressed pruned trie that can avoid moving existing keys when inserting new keys. It can also reduce the average search hops. The trie index has a short search path that is independent of the network size and the number of keys, which make it scalable in dynamic environments. To improve the system utilization, we adopt a decentralized load-balancing method that does not rely on any global load information. Additionally, to improve the index availability, a replication method is used on the semantic overlay and on the distributed trie index. The scalability of the platform is verified by experiments.

2 Related Work

Early P2P systems such as Napster (www.napster.com) used centralized directory servers for resource discovery. The second-generation P2P systems like Gnutella (www.gnutella.com) adopt flooding search methods in unstructured overlays. DHT overlays aim at providing scalable lookup functions on structured P2P networks (e.g., [27], [30], [37]). Each node maintains $O(\log N)$ or $O(1)$ neighbor nodes, and the lookup procedure can locate

• *The authors are with the China Knowledge Grid Research Group, Key Lab of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, 100080, PO Box 2704-28, Beijing, China. E-mail: {zhuge, sunxp, lj, alin.yao, chenxue}@kg.ict.ac.cn.*

Manuscript received 5 Apr. 2005; accepted 6 June 2005; published online 19 Oct. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0132-0405.

remote objects in $O(\log\ N)$ hops [3]. Some distributed data sharing/storage systems are built on DHT overlays (e.g., [7], [21], and [27]). Hybrid P2P systems use supernodes to balance search efficiency, robustness, and safety in large-scale networks (e.g., [36]). High-level architectures and applications on P2P networks are attracting more and more attention. JXTA aims at providing a P2P platform with standardized resource lookup functions and communication protocols [22].

To improve the efficiency and availability of P2P systems, much work has been done on load-balancing and replication methods. In [24], a replication method is used to improve search efficiency on an unstructured P2P network. In [12], a decentralized information monitoring system is built on a DHT overlay, and a relaxed addressing schema is implemented to efficiently deal with the heterogeneity of the nodes. In [26], a dynamic load-balancing method is used to transfer data from highly loaded nodes to lightly loaded nodes based on global load distribution information. It is further reduced to a centralized method on DHT overlay networks in [13]. In [10], an online load-balancing method is proposed to balance the storage of range-partitioned data by adjusting ranges among neighboring nodes based on local neighboring load information as well as global load information that can be obtained through a skip graph structure. Publishing functions and topology features can be leveraged to manage copies (e.g., [7], [12], and [37]). In [21] and [27], erasure code methods are used to create copies and maintain consistency of replica. In these methods, copy placement depends on publishing functions and topologies of their overlay networks.

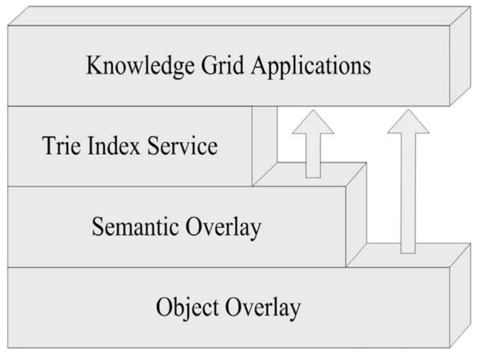
Many P2P systems adopt various indexing schemas to improve search efficiency and to support more complex queries. In [5], a route index is built on an unstructured P2P overlay to facilitate queries. The index is created and maintained based on data from neighbor nodes. In [35], local indices are built to help selecting the most promising neighbor to forward a query, thus improving search efficiency. In [1], [9], and [19], overlay topologies are manipulated to support specific types of queries. Building an overlay network with a specific indexing topology can efficiently support certain types of queries such as range query. However, it is difficult to adapt such an indexing topology to other kinds of queries. In [11], [17], and [28], specific indexing services are deployed on P2P overlays. These systems rely on specific overlays and indexing topologies, thus, they still cannot be extended to support different query requirements. In [14], a key string is decomposed into multiple subkeys to support complex queries on existing DHT networks. In a certain sense, it is similar to the trie index with the granularity of subkeys larger than that of the trie index and search is based on a multicast method. Litwin et al. [2] introduces a scalable distributed data structure (SDDS) LH* that uses hash function to allocate keys among distributed servers in a stably connected network. RP^* deploys a B^+-tree index in a local area network where tree indexes are replicated, and each physical node handles a range of keys [23]. It uses a broadcasting method to process data operations such as inserting or deleting keys. B-tree-based distributed indexes can efficiently balance indexing node. These indexes do not consider heterogeneity of data object load and physical node capacity. Kröll and Widmayer [20] analyze the scalability of distributed random tree (DRT) index in dynamic environment. In a DRT, replicas of subtrees are used to relieve bottleneck of entry nodes and a lazy update method is used to maintain replications. In [6], a distributed B^+-tree index is built on a ring-like P2P overlay where each peer maintains a relatively independent $\mathrm{sub\text{-}B}^+\text{-tree}$ index. A stabilization process is used to maintain consistency among local $\mathrm{sub\text{-}B}^+\text{-tree}$ indexes. In a dynamic and large-scale environment, such a stabilization process is difficult to maintain and optimize. P-Grid uses a distributed binary prefix tree to build a structured P2P overlay [1]. Although the randomized exchange method can form a balanced binary tree, P-Grid needs to solve skewed data distribution. Ganesan et al. [10] present a P2P network that supports range queries by arranging peers on a ring network with ordered data partitions allocated for peers. And, a skip graph is used to efficiently locate peers for a range of data. Physical nodes and data objects should have comparable names or IDs when using skip graph to route queries, which limit the key types of data objects on a P2P overlay. In DHT overlays, two ID spaces are unified by hash functions and, thus, any types of keys can be applied.

Some overlay networks are built directly on the relationships among physical nodes or cluster semantically close data among those neighboring physical nodes (e.g., [4], [31], and [38]). Semantic proximity of nodes or data objects on an overlay network can help improve search efficiency and reduce network load. However, clustering physical nodes or data objects by a certain type of semantic proximity requires the adjustment of the topology or the location mapping of objects.

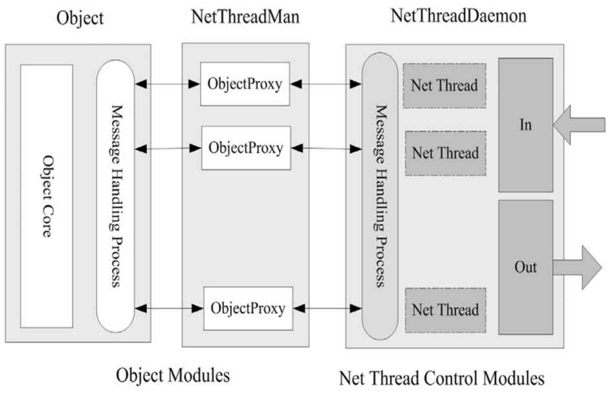
3 Design Rationale

Many current DHT-based P2P overlays use ID comparison to determine message routing based on local routing indexes. Our analysis shows that route indexes of N physical nodes on a comparison-based DHT overlay should connect node IDs to form a linear ordered ID space, which features the lower bound of $O(N\ \log_2\ N)$ comparisons in construction (shown in the Appendix which can be found on the Computer Society Digital Library at http://www.computer.org/tkde/archives.htm). Thus, the deterministic mapping between data objects and physical nodes organizes the IDs of data objects in a linear-ordered space on the network. In comparison-based DHT overlays such as Chord [30], Tapestry [37], and PAST [27], one comparison denotes one hop of routing a message from one node to another so as to build route indexes by numerically comparing IDs. Although Tapestry-like overlays do not explicitly build a linear ordered ID space, the route indexes inherently follow the same feature of a linear ordered ID space for message routing because IDs are digitally compared. Many construction and maintenance processes take $O(N^2)$ comparisons since they are the same as an insert sorting process. For example, in a Chord overlay, inserting a node will traverse the whole ID space to find its right position. Although finger tables can help locate the position at a faster speed, a stabilization process traversing the whole network is necessary to adjust finger tables after inserting new nodes.

On a linear ordered ID space formed by IDs of physical nodes, searching an ID can achieve $O(\log_k\ N)$ hops by adding each node $(k-1)\ \log_k\ N$ indices on the IDs space. Chord uses finger tables to form a $\log_2\ N$-ary indexing tree for each node. Tapestry also uses tree-like index to facilitate search. Some P2P overlays achieve $\mathrm{O}(\log_k\ N)$ lookup hops with $\mathrm{O}(\log_k\ N)$ neighbors on networks with specific topologies such as de Bruijin graph [16] and butterfly



(a)



(b)

Fig. 1. Architecture of IMAGINE-P2P. (a) Layered architecture of IMAGINE-P2P and (b) object and net thread structures.

graph [34]. Those routing methods are not based on ID comparison (many of them use bits shift) and impose strong limits to the node number and the topology. The comparison-based structured P2P networks are preferred in dynamic environments due to their simplicity and robustness. Among them, the ring-like overlay is the most direct and easy one to build and maintain because its topology immediately reflects the inherent property of routing indexes of comparison-based DHT overlays. Chord overlay is such a kind of DHT overlay that has a ring topology with proved correctness of stabilization [30]. This is why we select it as the underlying overlay.

Path queries, such as exact queries, wild-card queries, or range queries, can be forwarded along paths in tree-like route indexes on a structured P2P overlay. However, those indexes are based on hashed IDs but not on data values. Thus, only queries on IDs can be facilitated. One can devise an indexing schema to enable path queries on data values by using data values instead of their hashed IDs to build route tables of physical nodes. There should be a deterministic mapping between data values and node physical addresses (e.g., [10] and [15]). However, different types of queries may require different index topologies. To incorporate such a heterogeneity into our design, we abstract an index as a directed labeled graph where vertexes indicate indexing nodes and directed edges denote semantic relations between two neighboring indexing nodes. We publish the directed labeled graph on the P2P overlay to form a distributed index that can effectively and efficiently support path queries. This is inevitably at the cost of search hops, messages, and storage. Our system provides methods for scaling the index with the change of the network size and key number by restraining increases in search hops, storage, and message cost.

Improving system utilization is a decisive factor in scaling the performance of the platform. On structured P2P networks, DHT functions do not consider the heterogeneity of data object load and capacities of physical nodes. Building a distributed index can generate extra load on the object overlay. Moreover, a skewed data distribution on indexing imposes heavy load on a few indexing node. This case is more serious in an index with fewer internal indexing nodes. So, we adopt a decentralized load-balancing method that directly acts on data objects to achieve a high utilization of physical nodes by incorporating both data object load and node capacity distributions.

The path availability of an index published on the semantic overlay is crucial for path queries. In a dynamic environment, the underlying object overlay network is not always reliable. We apply a replication method to the semantic overlay to improve the availability of semantic objects. Path information is stored on semantic objects as replica to improve the availability of index paths.

4 The Architecture of IMAGINE-P2P

4.1 Overview

The scalability of a platform lies in many folds. On architecture, it indicates how well a platform can support the growth of function and service components. Providing standard interfaces and isolating logical services from physical platform are two approaches to scale out an architecture. In an IMAGINE-P2P platform, the object overlay is the underlying infrastructure that provides standard interfaces for accessing the physical P2P network. Upon it, each layer encapsulates specific services that can be used by its upper-level services (Fig. 1a). In the object overlay, an object is the entity that encapsulates both data and functions to perform certain services. IMAGINE-P2P implements the basic object that supports Chord DHT lookup functions, event-driven message processing, and net thread management (Fig. 1b). Applications can implement specific data and computing services by extending functions of the basic objects. The namespace defines the accessible domain of objects identified by their IDs and the namespace ID. Net threads are used to support coordination and consistency management of objects.

Two types of net threads, weak and strong, can run on the object overlay. In a weak net thread $wnt_i(o_1, o_2, \ldots, o_n)$, each object can be concurrently involved in many threads of either type. In a strong net thread $snt_i(o_1, o_2, \ldots, o_n)$, an object cannot be invoked by any other strong thread until it is released and object o_i can be run only after o_{i-1} finishes. To simplify the consistency maintenance, two phases are used to start a strong net thread from its initial object o_1. Before it starts running, o_1 sends the object list $o_2, \ldots, o_n$ to o_2. If o_2 grants the access, it will pass the list to o_3, etc., until to o_n. Only when o_n grants the access, can the net thread start running. If any object denies the access, the net thread aborts. Exclusively continued use of an object by a strong net thread can cause deadlocks. We use an edge-chasing algorithm to detect deadlock [29]. To recover from a

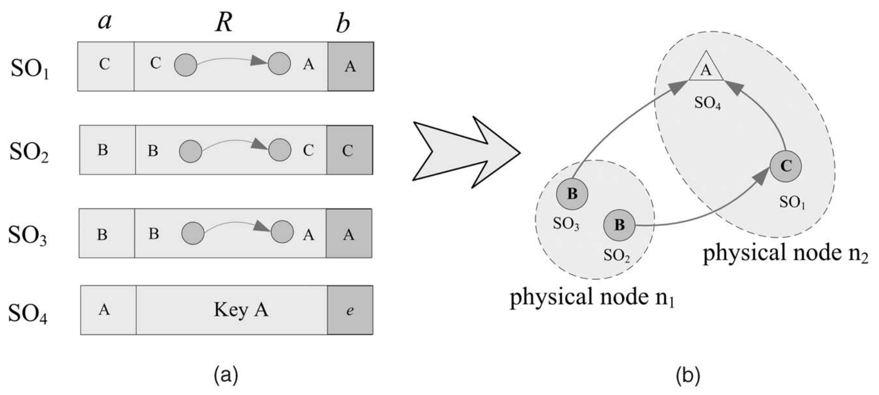


Fig. 2. Semantic objects of a distributed index on the semantic overlay. (a) Three semantic objects using a as the key to publish themselves. (b) A distributed idex on the semantic overlay formed by three semantic objects that are hosted by two physical nodes published by DHT functions.

deadlock, only one thread is granted the access to the object by thread priorities, lengths of net threads, and requesting object IDs. Complex concurrency control can be implemented based on weak net threads.

4.2 The Semantic Overlay

Semantic objects are published on the object overlay to package indexing nodes of an index. A semantic object is a triple $SO = (a, R, b)$, where R is the directed semantic relation between neighboring indexing nodes a and b. The semantic overlay consists of many such relation graphs formed by semantic objects. A semantic path

$$sp(a_1R_1a_2R_2 \ldots a_{n-1}R_{n-1}a_n)$$

on the semantic overlay follows a series of semantic relations $R_i (i = 1, 2, \ldots, n-1)$ from an indexing node a_1 to a_n. Path queries can be forwarded along a semantic path to get the answer. Fig. 2 shows an index example on the semantic overlay. A path query on key A sending from node n_1 can reach node n_2 by following SO_2 and SO_1. It can also directly follow SO_3 to find the key. Different indexing schemas can coexist on the semantic overlay by using different namespaces. Tree-like indexing data structures can be deployed on the semantic overlay as graphs.

To publish a semantic object $SO = (a, R, b)$ onto the object overlay, either a or b, or both can be used as the keys by the DHT function. Thus, it can have two copies in the overlay and the consistency between copies should be maintained by applications. A semantic path

$$sp(a_1R_1a_2R_2 \ldots a_{n-1}R_{n-1}a_n)$$

is decomposed into $n-1$ semantic objects, $SO_1(a_1, R_1, a_2)$, $SO_2(a_2, R_2, a_3), \ldots,$ and $SO_{n-1}(a_{n-1}, R_{n-1}, a_n)$. If it is to be uniquely represented by semantic objects, path keys are used to decompose $sp(a_1R_1a_2R_2 \ldots a_{n-1}R_{n-1}a_n)$ into $SO_1(a_1, R_1, a_2)$, $SO_2(a_1a_2, R_2, a_3), \ldots,$ and

$$SO_{n-1}(a_1a_2 \ldots a_{n-1}, R_{n-1}, a_n).$$

A path key contains a path from a starting indexing node to an ending indexing node. Removing a semantic path carries out in the same way as it is published. A strong net thread is responsible for the removal.

To locate a single object $SO(a, R, b)$, either a or b can be used as the key by the DHT lookup function on the object overlay. A path query $q = a_1R_1a_2R_2 \ldots a_{n-1}R_{n-1}a_n$ looks for a set of semantic objects of a semantic path

$$sp(a_1R_1a_2R_2 \ldots a_{n-1}R_{n-1}a_n).$$

It is decomposed to $n-1$ subqueries that are sequentially processed, $q_1 = a_1R_1a_2 \ldots a_n$, $q_2 = a_1a_2R_2a_3 \ldots a_n$, and

$$q_{n-1} = a_1a_2a_3 \ldots R_{n-1}a_n.$$

q_1 is first sent to the physical node that holds $SO_1(a_1, R_1, a_2)$. If q_1 is matched, q_2 is forwarded to the next physical node for $SO_2(a_2, R_2, a_3)$ or $SO_2(a_1a_2, R_2, a_3)$ when using path keys, and so on until the last subquery q_{n-1} is matched. Accessing the first semantic object $SO_1(a_1, R_1, a_2)$ requires $O(\log_2 N)$ hops through the DHT function of the object overlay (N is the number of physical nodes). The rest hops can follow the cached addresses of pointers of semantic objects. Only if a cached address fails, is the DHT lookup function used to determine the new address of the object. Thus, when all the pointers of semantic objects are correct, the total hops can be $O(\log_2 N + L)$, where L is the length of a semantic path that a query follows. In the worst case where all pointers are lost, there will be $O(L * \log_2 N)$ hops. Although the total length of hops is longer than $O(L)$ and $O(\log_2 N)$, we trade it against the extensibility and flexibility of the architecture to support various complex path queries.

Table 1 lists a set of basic queries on the semantic overlay. x represents any key of indexing node, a and b are specific keys of indexing nodes, $*$ indicates any relation, R^* denotes a semantic path formed by a relation R, aR^* represents all the possible semantic paths starting from a and following a relation R, TTL(Time-to-Live) is a predefined search hop count, the depth is the length from the starting node along a semantic path, and $\sim a$ is a delimiter that excludes a during a search. Basic queries can be combined to support more complex queries.

4.3 Distributed Trie Index on the Semantic Overlay

As the first step, we deploy a trie index on the semantic overlay. A trie index is a tree-like index that supports path queries such as wildcard queries and range queries on key strings. In a trie index, a key string is a permutation of a set of m attributes $(a_1, a_2, \ldots, a_m)$ that take on values from a finite attribute set (such as English character set). Two keys that have the same k initial attributes in the same order share the k prefix. A trie can be viewed as an m-ary tree where keys are arranged on leaf nodes. Keys with the same

TABLE 1
Basic Queries on the Semantic Overlay

Query q	Published Keys	Match patterns	Return results if find	Search hops
a R b	(a), (b), (a, b)	By a SO(a,R,b)	A SO(a,R,b)	1
a R^* b	(a), (a, b)	By a sp(aRx_1R…x_nRb)	sp(aRx_1R…x_nRb)	n hops/TTL
a ∗ b	(a), (b), (a, b)	By a SO(a,R,b) with any R	All the existing SO(a,R,b)	1
aR^*	(a), (a, b)	By any sp(aRx_1…Rx_n)	Each sp(aRx_1…Rx_n)	Depth/TTL
$a_1R_1a_2$… $R_{n-1}a_n$	(a), (b), (a, b)	By sp($a_1R_1a_2$… $R_{n-1}a_n$)	sp($a_1R_1a_2$… $R_{n-1}a_n$)	n hops
$a_1R_1a_2$… $R_{n-1}a_nR^*$	(a), (a, b)	By any sp($a_1R_1a_2$… $R_{n-1}a_n$ R^*)	any sp($a_1R_1a_2$… $R_{n-1}a_n$ R^*)	Depth/ TTL
Rb	(b), (a, b)	By a SO(x,R,b)	any SO(x,R,b)	1
R^*b	(b), (a, b)	By any sp(xR^*b)	Each sp(xR^*b)	Depth/TTL

k prefix share the same path of k indexing nodes from the root to leaves [18]. Searching a key string on a trie index starts from the root and follows the attributes that meet the query along a trie path until to a leaf node. A trie path $tp(a_1a_2\dots a_me)$ is a unique path from the root trie node to an internal trie node a_m, where e is an ending tag of a trie path. Each key $K = a_1a_2\dots a_m\dots a_n$ has a unique trie path $tp(a_1a_2\dots a_me)$ $(m \leq n)$.

Publishing a key $K = a_1a_2\dots a_n$ is a searching/publishing process of the trie path $tp(a_1a_2\dots a_me)$ $(m \leq n)$ of K: Searching along the trie path of the key, if there is no such trie path, publish it; if there exists a $tp_1(a_1a_2\dots a_ie)$ $(0 < i < m)$, publish the rest part $tp_2(a_ia_{i+1}\dots a_me)$ of the path; and, finally, the key is attached to the ending indexing node a_m. We use semantic objects to package two connected indexing nodes of a trie path. If a trie node a_{i+1} is a direct child of the trie node a_i on a trie path $tp(a_1a_2\dots a_ia_{i+1}\dots a_me)$ of a trie index, there exits a relation a_iSa_{i+1}. Using the path key mentioned in Section 4.2, a_iSa_{i+1} is represented by a semantic object $SO(a_1a_2\dots a_i, S, a_{i+1})$. Publishing a trie path $tp(a_1a_2\dots a_me)$ is the same as publishing a semantic path $sp(a_1Sa_2S\dots Sa_mSe)$, i.e., decomposing it into m semantic objects, $SO_1(a_1, S, a_2)$, $SO_2(a_1a_2, S, a_3), \dots, SO_{m-1}(a_1a_2\dots a_{m-1}, S, a_m)$, and

$$SO_m(a_1a_2\dots a_{m-1}a_m, S, e).$$

The key K is attached to the last semantic object SO_m. For example, to publish a key $K = back$ in Fig. 3a, a trie path $b \to a \to c \to k$ of K is decomposed to four semantic objects $SO_1(b, S, a)$, $SO_2(ba, S, c)$, $SO_3(bac, S, k)$, and $SO_4(back, S, e)$. Basic queries on the semantic overlay can be applied to the trie index. A trie index can support broadcast queries by issuing a query $q = a_1a_2\dots a_mS^*$. A broadcast query will follow every possible branch from the indexing node a_m. Range queries can be decomposed into a set of broadcast queries with delimiters to search on a portion of a trie index that is in charge of the queried range. This paper focuses on broadcast queries since they consume more messages than range queries.

There are mainly two types of trie indexes [18]. A full trie uses all attributes of a key string as the trie path from the root

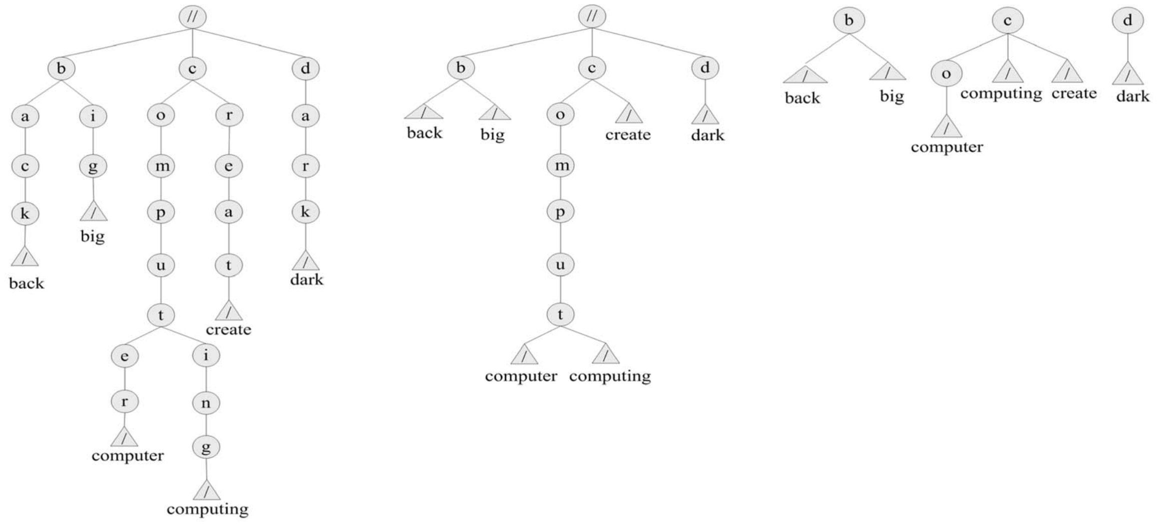


Fig. 3. Trie indexes. (a) A full trie, (b) a pruned trie, and (c) a compressed pruned trie.

to the leaf node of the key (Fig. 3a). A pruned trie only uses the attributes common to the prefixes of two or more key strings to build internal nodes (Fig. 3b). A trie is an efficient indexing structure if there are many keys with limited key string length. Search hops on a trie are proportional to the average depth of the trie (it is also proportional to the average length of key strings). Other tree-like indexes, for example, a B-tree, can be published by wrapping range relations among indexing nodes into semantic objects. However, inserting or deleting keys on a B-tree index requires splitting existing indexing nodes and moving keys. Publishing a full trie index does not require such dynamic adjustments. But, it has much longer search paths. A pruned trie has shorter search hops but needs to move existing keys when inserting new keys. We devise a compressed pruned trie index to avoid moving existing keys. Such an index can also reduce the average search hops (Fig. 3c).

To build a compressed pruned trie index, we use a special semantic object called *key object* to store key strings on a pruned trie index. An existing key object does not have to move when inserting a new key that shares the prefix with it. A key object is defined as $KO(a_1a_2\dots a_j, S, K)$, where key $K = a_1a_2\dots a_j\dots a_n$ and a_j are the leaf trie nodes of the trie path of K. When publishing the key $K = a_1a_2\dots a_j\dots a_n$ from the node holding the semantic object $SO_1(a_1, S, e)$:

1. If there is no $SO(a_1, S, e)$ or $SO(a_1, S, a_2)$, $SO(a_1, S, e)$ is published and the key K is published by $KO(a_1, S, K)$.
2. If there is $SO(a_1, S, e)$ but no $KO(a_1, S, K_1)$, where $K_1 = a_1b_2b_3\dots b_n$ $(b_2 \neq a_2)$, the key K is published by $KO(a_1, S, K)$.
3. If there are already $SO(a_1, S, e)$ and a $KO(a_1, S, K_1)$ that shares some prefixes with K, where $K_1 = a_1a_2\dots a_jb_{j+1}\dots b_m$, $j \geq 2$, and $b_{j+1} \neq a_{j+1}$, $SO(a_1, S, e)$ is changed to $SO(a_1, S, a_2)$ and two objects are published. One is $SO(a_1a_2, S, e)$, the other is $KO(a_1a_2, S, K)$.
4. If there is already a $SO(a_1, S, a_2)$, forward the key K along the trie path $tp(a_1a_2\dots a_me)$ until to $SO(a_1a_2\dots a_m, S, e)$ $(m \leq n)$. If there is no such a $KO(a_1a_2a_3\dots a_m, S, K_2)$ that
$$K_2 = a_1a_2\dots a_ma_{m+1}b_{m+2}\dots b_p,$$
just publish a $KO(a_1a_2a_3\dots a_m, S, K)$. Else change $SO(a_1a_2\dots a_m, S, e)$ to $SO(a_1a_2\dots a_m, S, a_{m+1})$ and publish objects $SO(a_1a_2a_3\dots a_ma_{m+1}, S, e)$ and $KO(a_1a_2a_3\dots a_ma_{m+1}, S, K)$.

In above steps, publishing a SO and a KO with the same prefix can be combined into one publish. The whole process does not move or split any existing indexing nodes or keys but the new keys. When searching on the compressed pruned trie, a physical node first checks if it has KOs that matches the query. If there is no matched KOs, then follow the semantic objects for the next hop on the trie path.

4.4 Optimized Search on the Distributed Trie Index

We further optimize the search procedure to reduce search hops on physical nodes by skipping unnecessary accesses to the same physical node for one query. When a physical node holding $SO_1(a_1a_2\dots a_j, S, a_{j+1})$ matches a subquery $q_j = a_1a_2\dots a_jSa_{j+1}$ of query $q = a_1a_2\dots a_n (1 \leq j < n-1)$, it checks all objects in its local storage to determine the hop of the next subquery. If there is a $SO_2(a_1a_2\dots a_h, S, a_{h+1}) (h > j+1)$ and a_h is the closest attribute to $a_n (h \leq n)$, the subquery $q_h = a_1a_2\dots a_{h+1}Sa_{h+2}$ is forwarded directly to the physical node that holds $SO_3(a_1a_2\dots a_{h+1}, S, a_{h+2})$, rather than forwarding the subquery $q_{j+1} = a_1a_2\dots a_{j+1}Sa_{j+2}$ to $SO_4(a_1a_2\dots a_{j+1}, S, a_{j+2})$. In Fig. 4a, the query q for a key "*abcdef*" starting from the physical node A will visit physical node B for two times, while in Fig. 4b, the query skips the physical node B since it finds the shortcut to physical node C. The optimized search can apply to the full trie index and the pruned trie.

In a compressed pruned trie index, a key object is attached to an internal node of a trie path. Then, internal nodes cannot be skipped. When issuing a subquery, already matched parts of the original query are marked to avoid subsequently repeated accesses to a physical node. Given a subquery $q_1 = a_1Sa_2\dots a_n$, we assume that physical node n_1 holds $SO_1(a_1, S, a_2)$. If there is also a $SO_2(a_1a_2\dots a_j, S, a_{j+1})$ $(j \geq 2)$ at n_1, then n_1 refines the query q_1 to the subquery $q_2 = (a_1a_2Sa_3\dots{}^{\sim}a_ja_{j+1}\dots a_n)$ by marking a_j as a delimiter. After all the local semantic objects and key objects are checked in n_1, the refined query q_2 is sent to the next physical node n_2 that is responsible for the semantic object $SO(a_1a_2, S, a_3)$. When n_2 receives q_2, it will first use the local index to refine the q_2 to $q_3 = (a_1a_2a_3Sa_4\dots{}^{\sim}a_j\dots{}^{\sim}a_m\dots a_n)$. If a_4 is a delimiter, then it is skipped by refining the q_2 to $q_h = (a_1a_2a_3\dots a_hSa_{h+1}\dots{}^{\sim}a_ja_{j+1}\dots{}^{\sim}a_m\dots a_n)$ until a_{h+1} is not a delimiter. Then, q_h is sent to the next physical node that has $SO(a_1a_2a_3\dots a_h, S, a_{h+1})$. Figs. 4c and 4d show the example of such a refined query on a compressed pruned trie. Multiple accesses to the physical node A can be avoided.

Locating a single key using a distributed trie takes $O(\log_2 N + L)$ hops on physical nodes, where N is the physical node number and L is the hops on the trie path leading to that key. Even using optimized search method, it still needs more hops than directly using DHT function to publish and locate a key in $O(\log_2 N)$ hops. However, a broadcast query $q = a_1a_2\dots a_jS^*$ can achieve a high search speed with moderate message cost since queries follow trie paths directly to the physical nodes that hold those matched keys. The speed is the max search depth on a trie index, plus $O(\log_2 N)$ hops to locate the starting trie node. The message cost is the number of internal trie nodes that cover the queried keys. If there is no such an index, the only way to find those matched keys is to browse all the physical nodes using broadcasting method on the network. If sequentially forward a broadcast query in two directions, it will take $N/2$ hops and $N-1$ messages. Using finger tables to broadcast queries can achieve a $O(\log_2 N)$ broadcast depth with $O(N \log_2 N)$ messages. In [8], an optimized broadcasting method can achieve $O(\log_2 N)$ broadcast depth with $N-1$ messages. In this case, the query hops and message cost depend on the number of network nodes and even when the number of returned keys is small, a query has to traverse the entire network, generating large amounts of messages. Generally, search hops and message cost are two contradictory factors. Using the optimized method can reduce the total message number among physical nodes. When a subquery of a broadcast query visits a physical node, it checks the local storage to find all internal trie nodes that match the broadcast query and issue

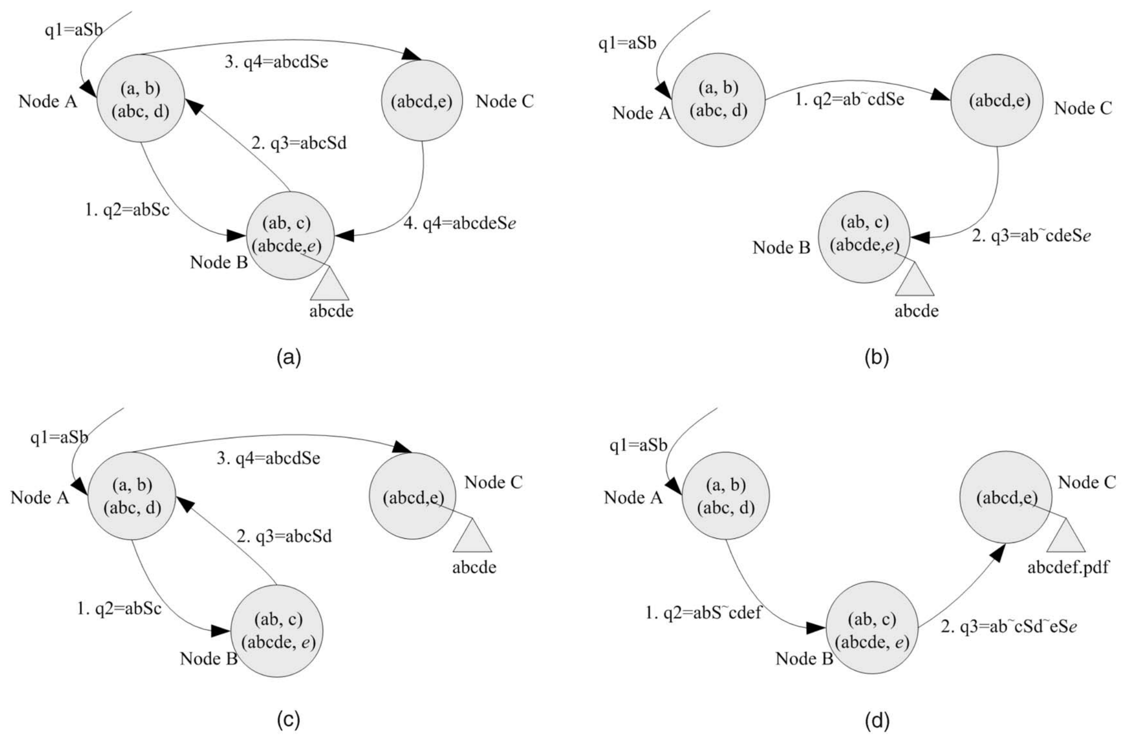


Fig. 4. Optimized queries on distributed trie indexes. (a) Queries access node A and B multiple times on the full trie index and the pruned trie index. (b) Optimized queries on the full trie index and the pruned trie index. (c) Queries access node A multiple times on the compressed pruned trie index. (d) Optimized queries on the compressed pruned trie index.

subqueries along those internal trie nodes. Then, subqueries latterly arriving at this physical node will be discarded and the total messages can be reduced.

5 Performance and Availability Improvement

5.1 Decentralized Load Balancing

To improve the system utilization, we use a decentralized load-balancing method similar to the decentralized load-balancing methods. Decentralized load-balancing methods, such as diffusion and dimension exchange, have been extensively investigated for massively parallel computer architectures with fixed network topologies (e.g., [25], [32], and [33]). In decentralized load-balancing methods, load migration takes place in neighboring nodes based on local load distribution. We take the finger table of a Chord node on the object overlay as its neighbors. Load migration is controlled to avoid overloading neighbors. Objects are moved to neighbor nodes one by one when balancing conditions are met.

In the object overlay ring, n_i denotes a physical node, and n_{i+1} is its nearest successor. N is the total number of the nodes. The last node n_N takes n_1 as its nearest successor. Let L_i^t be the workload of the node n_i at time t, and C_i be the capacity of the node n_i. Let $W = \{w_1, w_2, \ldots, w_m\}$ be m objects distributed on the network with w_j indicating the jth object and its workload. Let $d_i^t = \{w_k | w_k \in W\}$ be the set of objects on the node n_i at time t. Then, $L_i^t = \sum_{w_l \in d_i^t} w_l$ is the workload of the node n_i at time t. We use $S_i = \{n_{i+2^0}, n_{i+2^1}, \ldots, n_{i+2^r}\}$ to denote the set of neighbor nodes of the node n_i, where $r = \log_2 N$. Let $L = \sum_{i=1}^m w_i$ be the total workload of objects on the network and $C = \sum_{i=1}^N C_i$ be the total capacity of nodes on the network. Then, the ideal target distribution of workload follows such a distribution that for each node $n_i (i = 1, 2, \ldots, N)$, $\frac{L_i^t}{C_i} = \frac{L}{C}$. The target distribution is the same as that used in [26]. In the load-balancing process, each node should make three decisions:

1. *Which object should be moved*. To reduce the turbulence of the workload distribution in the load balancing, we select a moveable object with the least workload defined as $w_s = \min\{w_k | w_k \in d_i^t\}$.
2. *When should the object be moved*. If $w_s \in d_i^t$ is the object at n_i to be moved at time t, it is moved when $\frac{L_i^t}{C_i} > \frac{L_j^t}{C_j}$ and $\frac{L_i^t - w_s}{C_i} \leq \frac{L_j^t + w_s}{C_j}$, so that the target node n_j is not loaded more than the source node.
3. *Where should the object be moved*. The neighbor node n_j with $\frac{L_j^t}{C_j} = \min\left\{\frac{L_k^t}{C_k} | n_k \in S_i\right\}$ is selected as the target node.

Nodes on the network simultaneously and periodically collect the load and the capacity information from their neighbor nodes to evaluate load-balancing conditions. We use the future reserved load to decide on load migration. After deciding to move an object, a node first sends a reservation request to the target node. The target will add the load from the requestor to its future reservation load and will inform other nodes of this reservation load rather than the current real load. Object movement can be bounded by setting the max move times for objects so that frequent movement of objects can be avoided.

TABLE 2
Index Properties

Index	Key Number	Key Type	Average Key String Length	Internal Nodes	Average Hops	Moved Keys	Splits
F Trie	13931	*.*	22.84	155158	22.84	0	0
P Trie	13913	*.*	22.84	43686	15.14	5980	43608
CP Trie	13913	*.*	22.84	7513	6.04	0	0
B-tree	13913	*.*	22.84	596	4	7696	592
B$^+$-tree	13913	*.*	22.84	594	4	8850	590
F Trie	2349	*.pdf	55.06	100865	55.06	0	0
P Trie	2349	*.pdf	55.06	12338	17.41	900	12278
CP Trie	2349	*.pdf	55.06	1203	4.85	0	0
B-tree (26)	2349	*.pdf	55.06	137	3	1742	134
B-tree (3)	2349	*.pdf	55.06	1813	9	3608	1804
B$^+$-tree (26)	2349	*.pdf	55.06	142	3	2085	139
B$^+$-tree (3)	2349	*.pdf	55.06	1838	9	5487	1829

5.2 Availability Improvement of the Semantic Overlay and the Distributed Trie Index

To improve the availability of a semantic object $SO(a, R, b)$, we use key a and key b to publish it twice, i.e., $SO(a, R, b)$ and $SO(b, \tilde{}R, a)$. Each can be easily recovered from the other. To maintain consistency between copies, a strong net thread is used to limit an operation to either the primary object or its copy. The distributed trie index on the semantic overlay can inherit the replication method from the semantic overlay. We use path key to further improve the availability of a trie path.

A path key of a semantic object contains the path information of the objects published before it on the same path. A semantic object can be recovered from any latter published semantic object. The recovery process is carried along with a query on a trie path. For a path query $q = a_1 a_2 \ldots a_m$, the first subquery $q_1 = a_1 S a_2 \ldots a_m$ reaches the semantic object $SO_1(a_1, S, a2)$. If SO_1 fails, the subquery q_1 is sent to the possible successors with $SO_2(a_1 a_2, S, a_3)$ and so on until to $SO_m(a_1 a_2 \ldots a_m S, e)$. If none of them responds, the query returns an empty result. If any semantic object receives a query that should have been matched already, its immediate predecessor can be presumed lost and an attempt can be made to recover it. However, when a semantic object with keys is lost, the semantic objects after it on the trie path cannot determine what the keys are.

Since each copy of a semantic object can be recovered from the other, the availability of a semantic object is measured by

$$P_s = \sum_{i=0}^{n-m} \frac{\binom{M}{i}\binom{N-M}{n-i}}{\binom{N}{n}},$$

where $n = 2$, $m = 1$, and there are M unavailable physical nodes in the network with N nodes. Under 10 percent failures of physical nodes, 99.99 percent availability of an object can be achieved. Assuming that all the semantic objects of a trie index have the same availability $p_s \leq 1$, if the replication method is disabled, the availability of a trie path with r semantic objects is $p_{tp} = p_s^r$ and $p_{tp} \leq p_s$. When using path keys, a semantic object can be recovered by those published after it on a trie path. Thus, the availability can be evaluated by

$$p_{pk} = (1-(1-p_s)^{r-1})(1-(1-p_s)^{r-2})\ldots (1-(1-p_s)^{r-(r-1)}),$$

obviously $p_{pk} > p_{tp}$.

6 Experiment Results

6.1 Distributed Trie Index Evaluation

The efficiency of deploying indexes on the semantic overlay is determined by the factors including storage cost, maintenance cost, search hops, system utilization, and index availability. To evaluate the storage cost and the average search hops of trie indexes, we concern two metrics, the number of internal indexing nodes, and the average search hops on indexes. The number of semantic objects published by path keys is equal to the number of internal trie nodes. The average search hops is the average depth of the trie index. Using optimized search method can reduce the average search hops since internal trie nodes can be skipped. To measure the maintenance cost, we evaluate movements of existing keys when inserting new keys. System utilization and index availability are tested in the following sections.

Table 2 compares basic properties of a full trie (F Trie), a pruned trie (P Trie) and a compressed pruned trie (CP Trie). B-tree and B$^+$-tree are also compared. The average hops of B-tree and B$^+$-tree are referred as the max depth of trees. We use a set of 13,913 disk file names (indicated as *.*) and a set of 2,349 PDF file names (*.pdf files use paper titles as their names) as key strings to be published on indexes. Among three trie indexes, the compressed trie index has the least internal nodes and the shortest average search path. With a different bucket size, the B-tree can have various tree depth and key movement cost. When the bucket size is set to 26, the B-tree and the B$^+$-tree have the least internal indexing nodes and the shortest search paths. Although many keys have to be moved in the B-tree with a large bucket size, the total cost can be still less than that of the

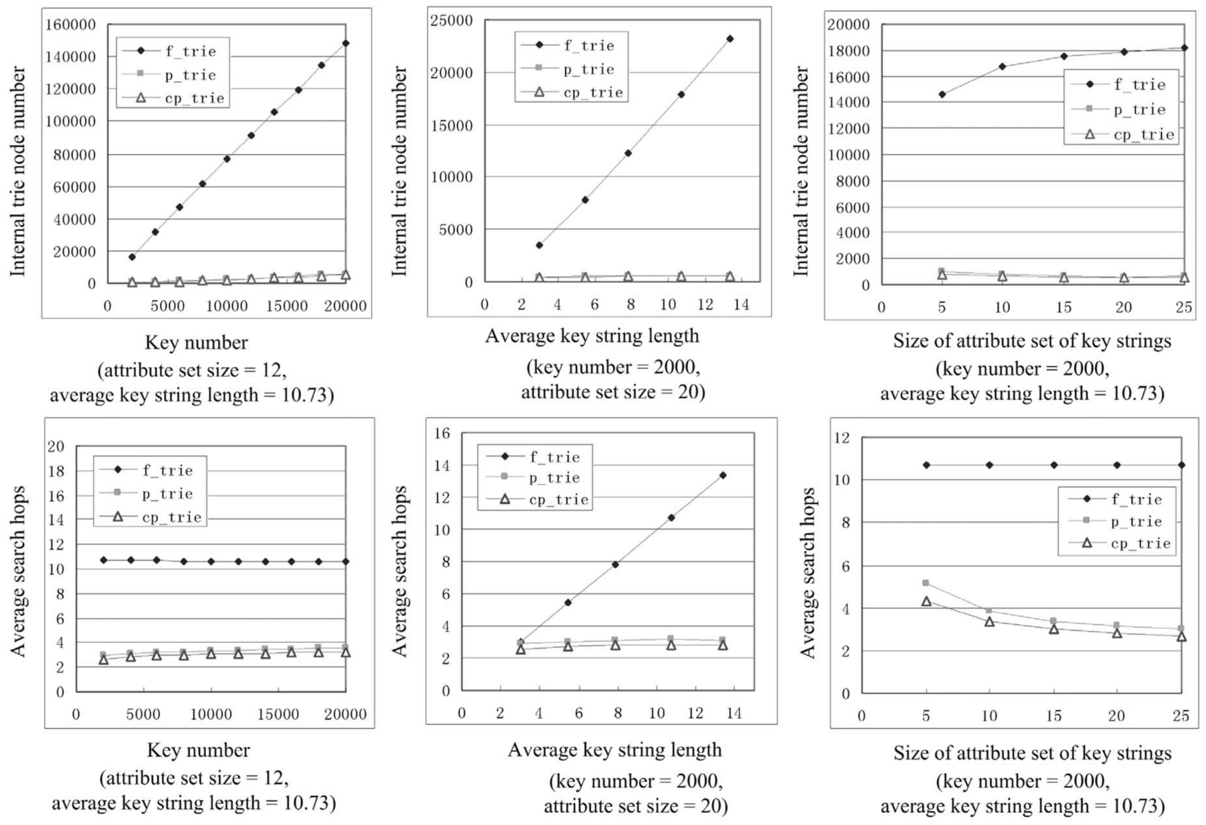


Fig. 5. Internal trie nodes and average search hops on trie indexes with randomly generated keys.

compressed pruned trie index, even counting in key movements. If the bucket size is set to 3, the total construction cost is larger than that of the compressed pruned trie. Deleting keys in a trie index involves only indexing nodes on the trie path of the key. However, deleting keys on a B-tree will take the key movements no less than the cost of inserting keys when merging indexing nodes.

Fig. 5 shows the internal nodes and the average search hops on trie indexes where key strings are randomly generated with the same distribution. Lengths of key strings are uniformly distributed within a max length and attributes are randomly selected from an attribute set including English characters. When the key string distribution is static, the average search hops on the pruned trie and the compressed trie are not much sensitive to the key number, the average key length, and the size of attribute set.

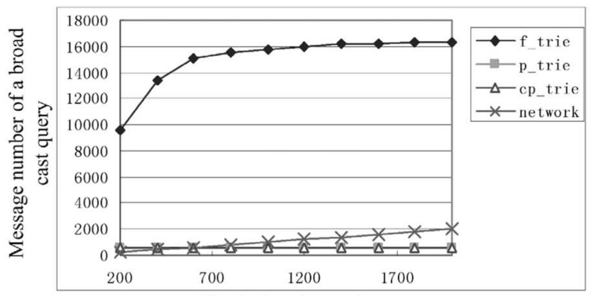
A broadcast query on trie indexes can achieve an efficient query hops while incurring limited message cost. We test the optimized search method on trie indexes to evaluate the effect on reducing messages for a broadcast query that returns all the keys on the network. Although using finger table to broadcast a query on the overlay can achieve log N hops with $N-1$ messages, Fig. 6a shows that the broadcast query for 2,000 keys on the pruned trie and the compressed pruned trie index has less message cost than using finger tables on the Chord ring when the network size increases. When both the key numbers and the network size increases, the broadcast query can still have less message costs than broadcasting directly on the overlay as shown in Fig. 6b. Figs. 6c and 6d show that the optimized search can greatly reduce messages in the full trie index when key number increases. However, the effect worsens when physical node number increases.

Table 3 shows the messages cost of the optimized query for all the keys on trie indexes when using the PDF file names as the keys. The average hops and the total messages of the full trie can be greatly reduced. The effect is not obvious for the pruned trie and the compressed pruned trie because they have less internal trie nodes and shorter trie paths. Using finger tables to broadcast the query on the Chord overlay with 2,000 nodes can take 11 hops and 2,000 messages. In the compressed trie, it only takes 1,176 messages to return the whole keys and the average extra hops on trie indexes is less than 5.

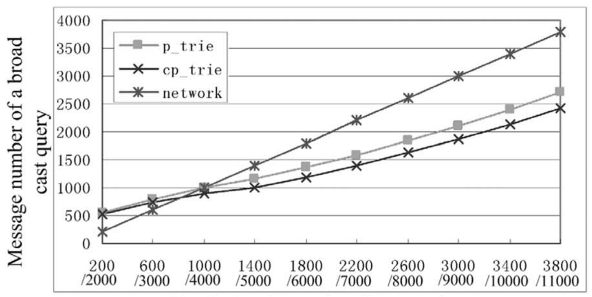
6.2 Load-Balancing Evaluation

We now evaluate the effect of the load-balancing method on improving the system utilization. First, we use randomly generated data object loads and node capacities by different distributions to test the load-balancing method. In the following section, we test the load balancing with objects of distributed trie indexes. An event-driven environment is used to simulate the load balancing in a network with a ring topology. We use the variance of L_i/C_i, $\mathrm{var} = \frac{1}{N-1}\sum_{i=1}^{N}(\frac{L_i}{C_i} - \frac{L}{C})^2$, to evaluate the utilization of a network with N nodes, where L_i indicates the current load of node i and C_i is the capacity of node i. The smaller the variance of L_i/C_i, the better the load balancing. The x-axis indicates the simulation rounds and the y-axis is the variance of the whole network load distribution.

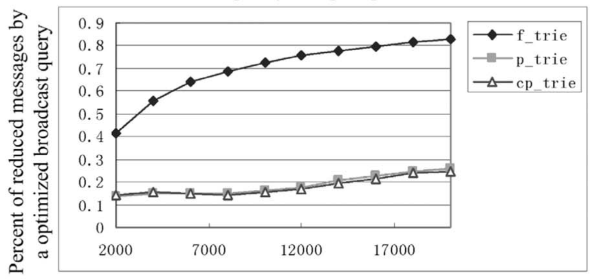
First, we set the load of all the data objects to one unit. The number of objects on each physical node is uniformly



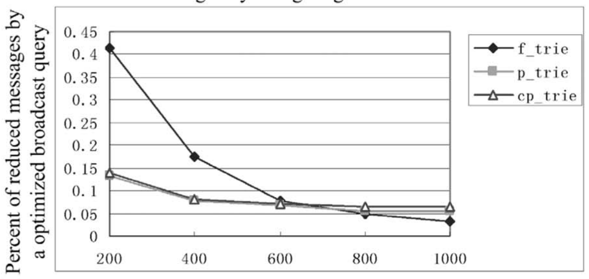
(a) 2000 key strings, attribute set size = 26, average key string length = 10.73



(b) attribute set size = 26, average key string length = 10.73



(c) 200 physical nodes, attribute set size = 26, average key string length = 10.73



(d) 2000 key strings, attribute set size = 26, average key string length = 10.73

Fig. 6. Message cost of a broadcast query using optimized search on trie indexes.

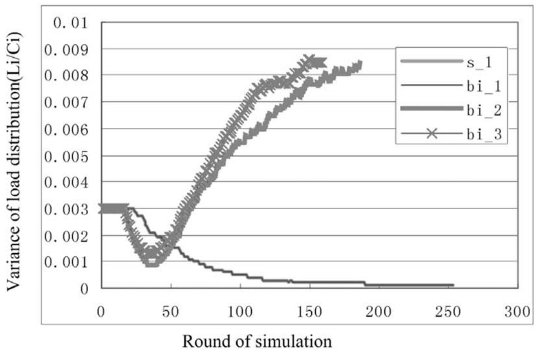
distributed. In Fig. 7a, s_1 shows the load-balancing process when neighbor nodes cover only the successors of a node. As shown by bi_1, balanced load can also be achieved when neighbor nodes cover both predecessors and successors. However, if the successors and the predecessors are separately considered when moving objects in two directions, the loads will not balance as shown by bi_2 and bi_3. The following experiments use only one-direction migration.

Fig. 7b shows the effects with four different workload and node capacity distributions. The node capacities in a_1 and a_2 experiments are uniformly distributed within a range from 10 to 200 unit loads, and those in a_3 and a_4 exponentially distributed with expectation of 50 unit loads. In a_2 and d_3, loads of 856 objects are distributed exponentially with an object load expectation of 10 unit loads. In a_1 and c_4, object loads are distributed uniformly in a range from 1 to 15 unit loads. The load balance can be achieved under different initial load distributions. When the bound of the object move times is set to 2 (a_4_2 in Fig. 7c), the convergence is very close to that of a_4 without an upper bound. In Fig. 7d, *node_in* shows that, when 50 new nodes randomly join in a network with 200 nodes, the load-balancing process can be facilitated. *node_out* shows that, although the load distribution worsens at first when nodes leave the network, the load-balancing method still works but is slowed down. *node_in_out* shows that the joining node can counteract load unbalance incurred by node departures.

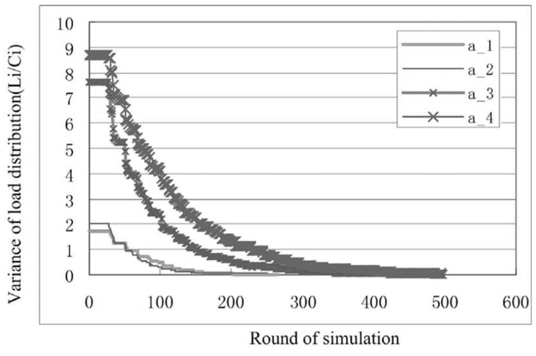
When objects are dynamically published in the load-balancing process, the load inbalance can be restrained. In Fig. 7e, when the load distribution and the capacity distribution are the same as those of a_2, *dyn_no_lb* shows a sharply increasing variance of load distribution when the load balancing is disabled and *dyn_in_lb* shows the restrained increase under the load-balancing process. This also demonstrates that the load-balancing method can be used to publish objects. When an object's location is suggested by the DHT lookup, the object will not immediately be transferred to that location. That location

TABLE 3
An Optimized Search on Trie Indexes with 2,349 PDF File Names as Keys

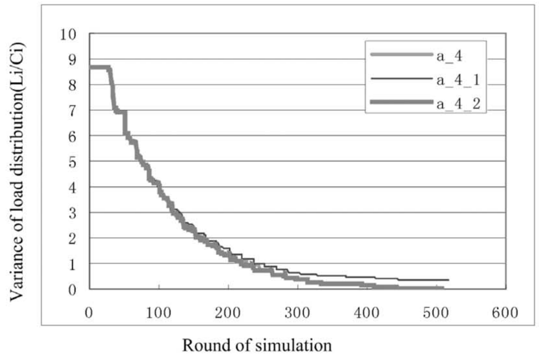
Index	Physical nodes	Messages without optimization	Messages in optimized search	Optimized avg hops	Average hops
F Trie	200	103076	23211	12.67	55.23
F Trie	2000	103076	98179	32.17	55.23
P Trie	200	12482	7986	8.09	17.36
P Trie	2000	12482	12340	13.047	17.36
CP Trie	200	1237	1092	4.89	4.89
CP Trie	2000	1237	1176	4.89	4.89



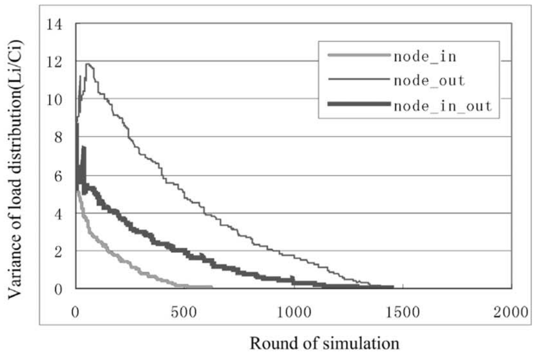
(a)



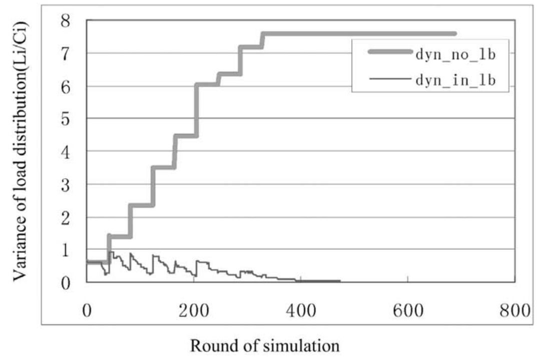
(b)



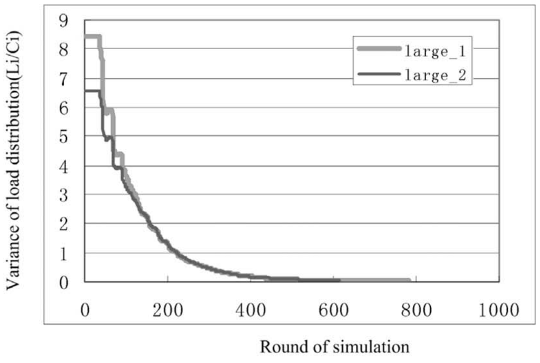
(c)



(d)



(e)



(f)

Fig. 7. Convergence process of the load balancing.

may first be changed by the load-balancing process since the host node can use the reservation to balance the load. By limiting the times an object may move in publishing, the final location will be determined exactly. This process is similar to the relaxed matching method of PeerCQ that moves the object only once to its direct neighbor nodes [12]. Finally, we test the load-balancing method in a 2,000-node network. As shown by *large*_1 and *large*_2 in Fig. 7f, similar convergent process can be achieved under the exponential load distribution and uniform load distribution, respectively.

Table 4 shows the cost of load balancing to achieve a balanced distribution. In most cases, when the variance is reduced by more than 90 percent, the average move times in the load-balancing process,

$$avg\ mov = \frac{total\ move\ times\ of\ all\ objects}{total\ number\ of\ objects},$$

is less than 1.6. The total workload of moved objects is evaluated by $moved\ load = \frac{total\ moved\ workload}{total\ workload\ of\ objects}$. Moving an object one time will incur one more search hop. When moving each object 1.6 times in a_3, it is about an 18 percent increase in the $\log_2 N$ hops that are required to publish an object in a network with 200 nodes. When nodes continually depart, it requires many more movements to become balanced. In a_4_1, we limit the object movements to 1 and the total object movements are less, but with poorer final load distribution. In a_4_2, where object movement is limited to two times, the load can be well balanced with fewer object movements than a_4. In the large network with 2,000 nodes, the average objects movement is still bounded.

TABLE 4
Cost of Load Balancing for Reducing the Variance by 90 Percent

	s_1	bi_1	a_1	a_2	a_3	a_4	a_4_1	a_4_2	dyn_in_lb	node_in	node_out	node_in_out	large_0	large_1
initial var	0.025	0.003	1.71	2.00	7.60	8.69	8.69	8.69	0.64	8.69	8.69	8.69	8.45	6.57
final var	0.001	0.0001	0.028	0.03	0.04	0.03	0.3	0.04	0.03	0.04	0.05	0.03	0.05	0.04
avg mov	0.81	0.62	0.99	0.98	1.61	1.39	0.75	1.14	1.69	1.02	1.34	1.06	1.81	1.56
moved load	0.5	0.55	0.94	0.94	1.25	1.14	0.75	1.05	1.18	0.88	1.02	0.86	1.41	1.3

Chord uses virtual servers to improve the load balance, where each physical node holds more than one virtual server and data objects are mapped by DHT function to virtual servers instead of physical nodes [30]. They proposed that log N virtual servers per physical node can be optimal with high probability when considering only the number of keys. Fig. 8 depicts the load distribution under different number of virtual servers per node when considering object load and physical node capacity. Under the same workload of virtual servers, i.e., the object load distribution on virtual servers is fixed, it is difficult to determine the optimal point when setting different numbers of virtual servers for physical nodes. Our load-balancing method can reduce the variance by more than 90 percent in any workload distribution.

Table 5 shows the convergent speed of the load-balancing process with different network size under the same load and capacity distribution. The simulation rounds for reducing the initial variance by 95 percent is not much sensitive to the network size. So do the average object movements (*avg mv times*) and the moved load (*mv load*). They mainly depend on the ratio of load and capacity (l / c) and the initial variance (*initial var*).

6.3 Load Balancing on the Distributed Trie Index

In this section, we test the load-balancing effect on increase in search hops of distributed trie indexes. In a network with 200 nodes, we publish 856 data objects with papers' names as keys by a full trie index, a pruned trie index, and a compressed pruned trie index, respectively. The data objects are stored with the semantic objects that hold the leaf trie nodes of the keys of those data objects. The load distribution of object loads and the node capacities are the same as that of a_4 in Fig. 7b. The load-balancing process can move all semantic objects or can move only those semantic objects that currently hold data objects. Using trie indexes to publish data objects can obviously incur more serious load unbalance. Fig. 9 shows that the load balancing

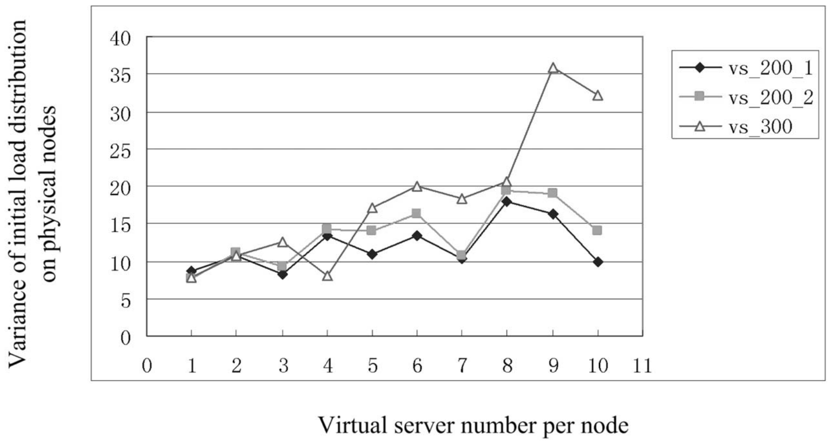


Fig. 8. Load distribution with different numbers of virtual servers per node.

TABLE 5
Convergent Speed for Reducing the Variance by 95 Percent

node number	100	200	300	400	500	600	700	800	900	1000	2000
rounds	114	221	310	261	259	284	294	286	290	252	312
initial var	4.08	7.6	7.9	6.97	7.73	8.12	8.62	8.76	7.82	7.45	8.44
final var	0.21	0.36	0.37	0.34	0.38	0.40	0.43	0.44	0.39	0.37	0.42
avg mv times	0.74	1.20	1.35	1.26	1.28	1.25	1.28	1.26	1.29	1.151	1.26
l/c	0.68	0.68	0.84	0.78	0.77	0.8	0.79	0.78	0.79	0.78	0.76
mv load	0.85	1.11	1.19	1.12	1.14	1.16	1.16	1.14	1.17	1.12	1.18

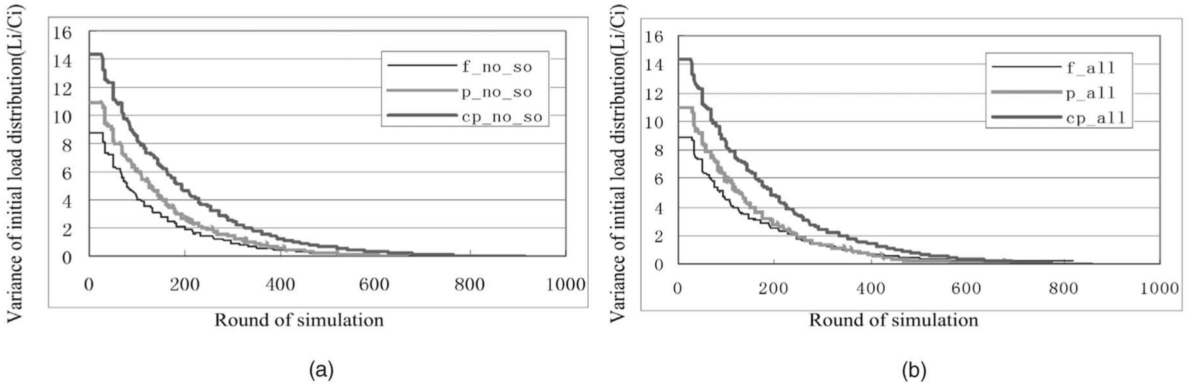


Fig. 9. Load balancing on distributed trie indexes.

can greatly improve the utilization. *f_no_so*, *p_no_so*, and *cp_no_so* indicate the load-balancing processes that only move semantic objects with data objects for the full trie, the pruned trie, and the compressed pruned trie, respectively. *f_all*, *p_all*, and *cp_all* are for the load-balancing processes that move all the semantic objects. Two cases show little difference because the most semantic objects are of small storage cost compared with those holding data objects.

Table 6 shows the object movements and the increased search hops incurred by the load balancing. The average move times (*avg mov* in Table 6) of semantic objects are no more than 2.2 in the most seriously unbalanced situation. Thus, the extra lookup hops incurred by load balancing are still controlled. When balancing only semantic objects that hold data objects, the average search hops (*avg hops*) is evaluated by *avg hops = avg index hops + avg mov*. *avg index hops* is the average hops on trie indexes and *avg mov* is the average object move times in the load-balancing process. The result shows that the increased hops is minor. When balancing all semantic objects, the average search hops is evaluated by *avg hops = (avg index hops * avg mov) + avg index hops*. Balancing all semantic objects can incur much more extra search hops for the pruned trie and the compressed trie.

6.4 Query Delays under the Load Balancing

If each extra hop incurred by the load balancing does not significantly delay a query, the average query latency under load balancing can be reduced when only considering storage consumption of objects. For simplicity, one unit load indicates one storage unit. The time to transfer an object is equal to the load of the object. Assume that a FIFO queue is used to cache the queries for objects in each physical node, then the query's latency on one physical node is its waiting time in the queue, namely, the total processing time of those queries before it. If the sought object is in the queried physical node, the processing time of a query is equal to the time for transferring the object to the requester. If the sought object is moved to a successor by the load-balancing procedure, the query is forwarded to the next node with its waiting time increased by a certain amount of unit load. Then, the total delay of a query on the network is evaluated by the total process time in all the nodes that processed the query. Fig. 10 shows the average latency of queries accessing all the objects at different query issuing rates. Two query distributions are tested and the load balancing can reduce the average query latency.

Fig. 11 indicates the percent of the reduced average query latency by load balancing for four load distributions tested in Fig. 7b. Queries are issued in a uniform distribution on data objects. The average object load in *a*_1 and *a*_4 is 7.55 unit loads, and 8.05 in *a*_2 and *a*_3. In general, the higher the initial variance of load distribution, the lower the reduced latency. As the latency of one hop increases, the reduced average query latency is worsened.

6.5 Availability of the Distributed Trie Index

Finally, we test the availability of trie indexes published on the network with 200 nodes and 2,000 keys by generating one query for each key. The availability is evaluated by the successful rate of all generated queries. When some physical nodes are disabled, the proportion of failed queries depends on the replication strategy. Fig. 12a shows that in a full trie index, the path key replication is good enough (*f_path_replication*) and it is quite close to using both the path key and semantic object replication (*f_so+path_replication*). But, the full trie has poor availability if only taking the semantic object replication (*f_so_replication*). Fig. 12b shows

TABLE 6
Cost of Load Balancing for Reducing the Variance by 90 Percent for Distributed Trie Indexes

	f_no_so	f_all	p_no_so	p_all	cp_no_so	cp_all
initial var	8.80	8.92	10.93	10.96	14.33	14.33
final var	0.016	0.25	0.031	0.039	0.024	0.027
avg mov	1.27	0.13	1.34	1.35	1.35	2.2
moved load	1.13	0.86	1.21	1.04	1.28	1.2
avg_index hops	61.42	61.42	8.57	8.57	4.25	4.25
avg hops	62.69	69.40	9.91	20.14	5.6	13.6

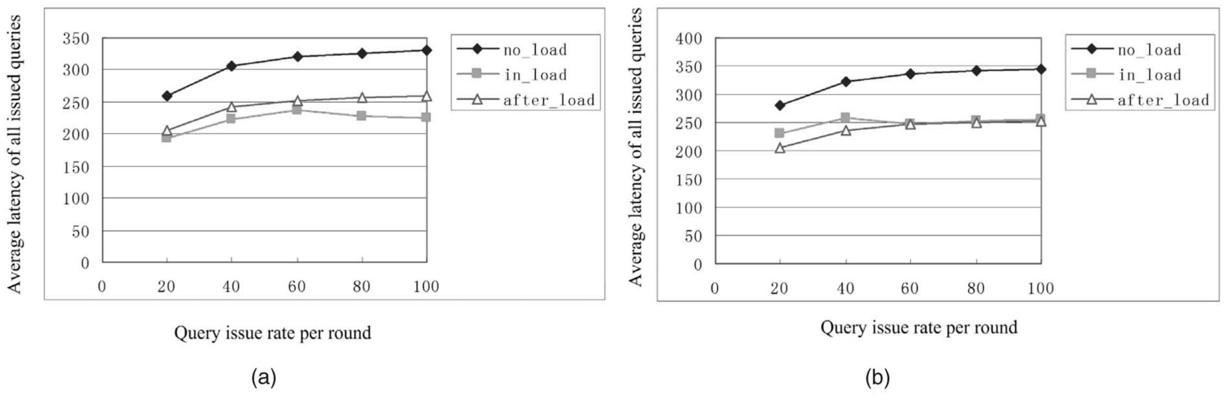


Fig. 10. Queries delays under the load balancing. (a) Queries are uniformly distributed and (b) queries are exponentially distributed.

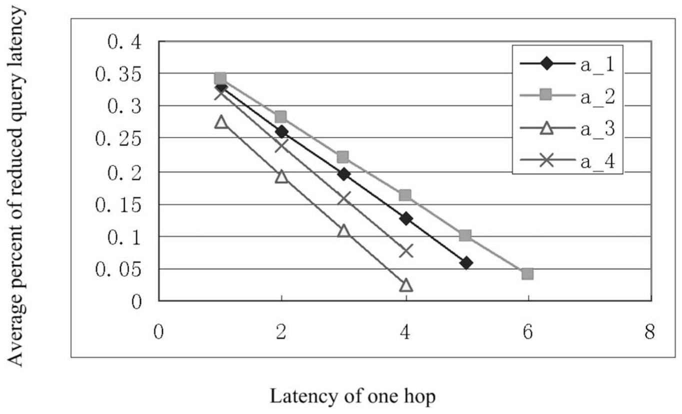


Fig. 11. Percent of reduced average query latency under different delay of one hop.

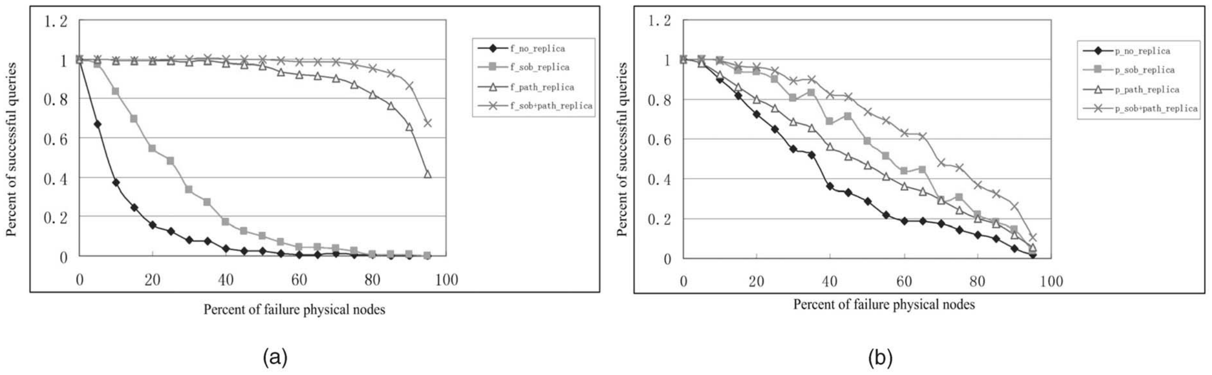


Fig. 12. The availability of the distributed trie indexes. (a) A Full trie with 2,000 keys (26 characters, max length is 20), 16,008 internal trie nodes. Network has 200 nodes. Average hops on the trie is 11.5920. (b) A pruned trie with 2,000 keys (26 characters, max length is 20), 595 internal trie nodes. Network has 200 nodes. Average hops on the trie is 2.66.

that, in a pruned trie index, using both path keys and semantic object replication has similar availability to only the path key replication or only the semantic object replication. The availability of the full trie with the replication is better than that of the pruned trie because the pruned trie has much shorter path length and there are fewer copies in path key replication. The pruned trie, however, has better availability without replication because it has much shorter search paths, i.e., it is less probably broken under the same failure distribution.

6.6 Summary of Experiments

The trie index is easier to be constructed and maintained than the B-tree when frequently inserting/deleting keys, while the B-tree index with large bucket size can achieve higher search efficiency. Search hops in the B-tree are determined by the bucket size and the number of keys. When deploying the B-tree, the bucket size and key movement should be carefully designed. In the trie index, search hops is only sensitive to the key string distribution, i.e., the distribution of string attributes and key string lengths. The pruned trie index is even insensitive to the key string length. The key string distribution is generally considered much steady in practice compared with the network size and the key number. Easy maintenance and steady search hops enable the trie index to scale in a large-scale and dynamic environment. Although publishing the trie index on the semantic overlay increases the whole query hops on the network, a broadcast query using the compressed pruned trie can achieve efficient query hops with a controlled number of messages.

The load-balancing method can relieve uneven load distribution in bounded rounds and incur a minor increased query hops. The convergent speed and the average movements of objects are almost independent of the network size. When the index paths become longer, the availability is improved by the replication method. In summary, the platform provides applications with various choices among the search hops, the storage utilization, and the availability of indexes to meet specific requirements.

7 Conclusion

To provide various types of query for intelligent applications on the Knowledge Grid, this paper presents a scalable platform IMAGINE-P2P that uses a semantic overlay to support distributed indexing services on a structured DHT P2P overlay. We demonstrate how a distributed trie index can be deployed to support path queries on key strings. It can scale well with the change of the size of network and the number of key. The load-balancing method can relieve uneven load distribution. It can efficiently improve the system utilization and reduce the query latency without using any global load information. The load-balancing method is independent of index structures and the network size. The replication method ensures longer indexing paths with higher availability. Experiments show that the platform is capable of making trade-offs among search hops, message cost, storage utilization, and index availability and, thus, scaling well in dynamic, large-scale environments.

Ongoing work includes studies of relieving query hotspots of trie index and deploying other tree-like index structures on the semantic overlay. Further, based on IMAGINE-P2P, we are developing indexes with richer semantics to support intelligent clustering of resources within the Knowledge Grid [40].

Acknowledgments

This work was supported by the National Basic Research Program (973 project no. 2003CB317001) and the National Science Foundation of China (Grants 60273020 and 70271007).

References

[1] K. Aberer, "P-Grid: A Self-Organizing Access Structure for P2P Information Systems," *Proc. Ninth Int'l Conf. Cooperative Information Systems,* vol. 2172, pp. 179-194, 2001.

[2] W. Litwin, M.A. Neimat, and D. Schneider, "LH*—A Scalable Distributed Data Structure," *ACM Trans. Database Systems,* vol. 21, no. 4, pp. 480-525, 1996.

[3] H. Balakrishnan, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking Up Data in P2P Systems," *Comm. ACM,* vol. 46, no. 2, pp. 43-48, 2003.

[4] A. Crespo and H. Garcia-Molina, "Semantic Overlay Networks for P2P Systems," technical reports, http://www-db.stanford.edu/~crespo/publications/, 2003.

[5] A. Crespo and H. Garcia-Molina, "Routing Indices for Peer-to-Peer Systems," *Proc. 28th Int'l Conf. Distributed Computing Systems,* pp. 23-32, July 2002.

[6] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram, "Querying Peer-to-Peer Networks Using P-Trees," *Proc. Seventh Int'l Workshop Web and Databases: Colocated with ACM SIGMOD/PODS2004,* pp. 25-30, 2004.

[7] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-Area Cooperative Storage with CFS," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP' 01),* pp. 202-215, Oct. 2001.

[8] S. El-Ansary, L.O. Alima, P. Brand, and S. Haridi, "Efficient Broadcast in Structured P2P Networks," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 304-314, 2003.

[9] M.J. Freedman and R. Vingralek, "Efficient Peer-to-Peer Lookup Based on a Distributed Trie," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 66-75, Mar. 2002.

[10] P. Ganesan, M. Bawa, and H. Garcia-Molina, "Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems," *Proc. Very Large Data Bases Conf.,* pp. 444-455, 2004.

[11] L. Garcés, P.A. Felber, E.W. Biersack, G. Urvoy-Keller, and K.W. Ross, "Data Indexing in Peer-to-Peer DHT Networks," *Proc. 24th Int'l Conf. Distributed Computing Systems,* pp. 200-208, Mar. 2004.

[12] B. Gedik and L. Liu, "PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System," *Proc. 23rd Int'l Conf. Distributed Computing Systems,* pp. 490-499, May 2003.

[13] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Dynamic Structured P2P Systems," *Proc. IEEE INFOCOM Conf.,* vol. 4, pp. 2253-2262, Mar. 2004.

[14] M. Harren and J.M. Hellerstein, "Complex Queries in DHT-Based Peer-to-Peer Networks," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* pp. 242-259, Mar. 2002.

[15] N.J.A. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," *Proc. Fourth USENIX Symp. Internet Technologies and Systems (USITS '03),* pp. 113-126, Mar. 2003.

[16] M.F. Kaashoek and D.R. Karger, "Koorde: A Simple Degree-Optimal Distributed Hash Table," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* F. Kaashoek and I. Stoica, eds., pp. 98-107, 2003.

[17] A. Kementsietsidis, M. Arenas, and R.J. Miller, "Mapping Data in Peer to Peer Systems: Semantics and Algorithmic Issues," *Proc. ACM SIGMOD Conf.,* pp. 325-336, June 2003.

[18] D.E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching,* second ed. Addison-Wesley, 1973.

[19] A. Kothari, D. Agrawal, A. Gupta, and S. Suri, "Range Addressable Network: A P2P Cache Architecture for Data Ranges," *Proc. Third Int'l Conf. Peer-to-Peer Computing,* pp. 14-22, Sept. 2003.

[20] B. Kröll and P. Widmayer, "Distributing a Search Tree Among a Growing Number of Processors," *ACM SIGMOD Record,* vol. 23, no. 2, pp. 265-276, June 1994.

[21] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 190-201, Nov. 2000.

[22] G. Li, "Project JXTA: A Technology Overview," Sun Microsystems, Inc., http://www.jxta.org, 2002.

[23] W. Litwin, M. Neimat, and D.A Schneider, "RP*: A Family of Order Preserving Scalable Distributed Data Structures," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB94),* pp. 342-353, Sept. 1994.

[24] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and Replication in Unstructured Peer-to-Peer Networks," *Proc. 16th Int'l Conf. Supercomputing,* pp. 84-95, June 2002.

[25] X. Qian and Q. Yang, "Load Balancing on Generalized Hypercube and Mesh Multiprocessors with LAL," *Proc. 11th Int'l Conf. Distributed Computing System,* pp. 402-409, May 1991.

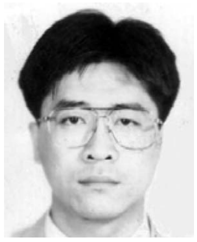
[26] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load Balancing in Structured P2P System," *Proc. Int'l Workshop Peer-to-Peer Systems (IPTPS),* F. Kaashoek and I. Stoica, eds., pp. 119-128, 2003.

[27] A. Rowstron and P. Druschel, "Storage Management and Caching in PAST, A Large-Scale, Persistent Peer-to-Peer Storage Utility," *ACM SIGOPS Operating Systems Rev.,* vol. 35, no. 5, pp. 188-201, 2001.

[28] H.T. Shen, Y. Shu, and B. Yu, "Efficient Semantic-Based Content Search in P2P Network," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 7, pp. 813-826, Aug. 2004.

[29] M. Singhal, "Deadlock Detection in Distributed Systems," *Computer,* vol. 22, no. 11, pp. 37-48, Nov. 1989.
[30] 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 SIGCOMM Conf.,* pp. 149-160, Aug. 2001.
[31] S. Voulgaris, A. Kermarrec, L. Massoulié, and M.V. Steen, "Exploiting Semantic Proximity in Peer-to-Peer Content Searching," *Proc. 10th IEEE Int'l Workshop Future Trends of Distributed Computing Systems,* pp. 238-243, May 2004.
[32] C. Xu, B. Monien, R. Lüling, and F.C.M. Lau, "Nearest Neighbor Algorithms for Load Balancing in Parallel Computers," *Concurrency: Practice and Experience,* vol. 7, no. 7, pp. 707-736, 1995.
[33] C. Xu and F.C.M. Lau, "Iterative Dynamic Load Balancing in Multicomputers," *J. Operational Research Soc.,* vol. 45, no. 7, pp. 786-796, 1994.
[34] J. Xu, A. Kumar, and X. Yu, "On the Fundamental Tradeoffs between Routing Table Size and Network Diameter in Peer-to-Peer Networks," *IEEE J. Selected Areas in Comm.,* vol. 22, no. 1, pp. 151-163, 2004.
[35] B. Yang and H. Garcia-Molina, "Improving Search in Peer-to-Peer Networks," *Proc. 28th Int'l Conf. Distributed Computing Systems,* pp. 5-14, July 2002.
[36] B. Yang and H. Garcia-Molina, "Designing a Super-Peer Network," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 49-63, Mar. 2003.
[37] B.Y. Zhao, H. Ling, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, "Tapestry: A Resilient Global-Scale Overlay for Service Deployment," *IEEE J. Selected Areas in Comm.,* vol. 22, no. 1, pp. 41-53, 2004.
[38] Y. Zhu, H. Wang, and Y. Hu, "Integrating Semantics-Based Access Mechanisms with P2P File Systems," *Proc. Third Int'l Conf. Peer-to-Peer Computing,* pp. 118-125, Sept. 2003.
[39] H. Zhuge, "China's E-Science Knowledge Grid Environment," *IEEE Intelligent System,* vol. 19, no. 1, pp. 13-17, 2004.
[40] H. Zhuge, *The Knowledge Grid.* World Scientific, 2004.
[41] H. Zhuge, "The Future Interconnection Environment," *Computer,* vol. 38, no. 4, pp. 27-33, Apr. 2005.

Hai Zhuge is the chief scientist of the China Semantic Grid project funded by the National Basic Research Program of China. He is a professor and the director of the Key Lab of Intelligent Information Processing at the Institute of Computing Technology in Chinese Academy of Sciences, and the founder of the China Knowledge Grid Research Group (http://kg.ict.ac.cn), which employs more than 30 young researchers. He presented more than 10 keynotes at international conferences. He was the cochair of the Second International Workshop on Knowledge Grid and Grid Intelligence, the program cochair of the Fourth International Conference on Grid and Cooperative Computing, and the cochair of the First International Conference on Semantics, Knowledge, and Grid. He organized several journal special issues on knowledge grid and semantic grid. He is serving as the area editor of the *Journal of Systems and Software*, the associate editor of *Future Generation Computer Systems*, the area editor of the *Journal of Computer Science and Technology*, and the editorial member of the *Information and Management and the Electronic Commerce Research and Applications*. His major research interest is the model, theory, and methodology on the future interconnection environment. His monograph *The Knowledge Grid* is the first book in the area, and received the 2005 Top Award of SONY Excellent Research. He is the author of more than 90 papers that appeared mainly in leading international journals such as *Communications of the ACM*, *Computer*, *IEEE Transactions on Knowledge and Data Engineering*, *IEEE Intelligent Systems*, *IEEE Computing in Science and Engineering*, and *IEEE Transactions on Systems, Man, and Cybernetics*. One of them was among the top 1 percent of highly cited papers in the area according to ISI Essential Science Indicator. He is a senior member of the IEEE and a member of the ACM. He was among the top scholars in systems and software engineering area (1999-2003) according to the assessment report published in the *Journal of Systems and Software*.



Xiaoping Sun is a PhD student in the China Knowledge Grid Research Group at the Institute of Computing Technology at the Chinese Academy of Sciences. His research interests include peer-to-peer computing, grid computing, and future networking techniques. He has published three papers in international journals.

Jie Liu is a PhD student in the China Knowledge Grid Research Group at the Institute of Computing Technology at the Chinese Academy of Sciences. Her research interests are P2P computing and semantic heterogeneous data integration. She has published 14 papers in international journals and conferences.

Erlin Yao is a PhD student in the China Knowledge Grid Research Group at the Institute of Computing Technology at the Chinese Academy of Sciences. His current research interests include the theory and formalization on the knowledge grid. He has published three papers in international journals.

Xue Chen is a PhD student in the China Knowledge Grid Research Group at the Institute of Computing Technology at the Chinese Academy of Sciences. Her research interests include peer-to-peer systems and self-organized networks. She has published three papers in international journals.

Appendix: Analysis of Constructing Comparison-Based Structured P2P Overlays

Definition 1. Assume that for a given finite key set $S = \{k_1, k_2, \ldots, k_n\}$, all the keys in the set S are unique, π is a linear ordering relation defined on S. For a permutation $V = \{k_{p1}, k_{p2}, \ldots, k_{pn}\}$, $k_{pi} \in S$, if for every pair of k_{pi} and k_{pi+1}, $k_{pi}\, \pi\, k_{pi+1}$, then V is a sorted permutation of the set S under the linear ordering relation π.

Reference [18] gives the lower bound of the sorting in terms of the comparison times, where one comparison can determine the partial order of two keys.

Theorem 1 [18]. In the worst case, to achieve a sorted permutation V of a finite set S by comparison of keys, the lower bound of comparisons among keys is $O(n \log_2 n)$, $n = |S|$.

I is a predefined ID set. In a P2P network $N = \{id_1, id_2, \ldots, id_n\}$, each node is uniquely identified by an $id_i \in I$ and maintains a non-null set of neighbor nodes $H_i = \{id_{p1}, id_{p2}, \ldots, id_{pk}\}$, $id_{pi} \in N$. The relationship between the IDs of a node and its neighbors is represented as R_h. Note that the inverse relation $\tilde{}R_h$ also exists in the overlay. Then, an abstracted definition of a P2P overlay network is given as follows.

Definition 2. A P2P overlay network P is a tuple $<N, R_h>$, where $N = \{id_1, id_2, \ldots, id_n\}$ is the set of IDs of nodes in the network and R_h is a binary relation that defines the relation between the IDs of a node and its neighbors, that is, if id_j is a neighbor of id_i, then $id_i\, R_h\, id_j$. However $id_i\, R_h\, id_j$ does not necessarily mean that id_i must have id_j as a neighbor.

Definition 2 makes several assumptions: First, all the nodes have the same R_h to form a neighbor node set. Second, all the nodes are connected through neighbors, though if the graph is disconnected, we can view the separate parts as different networks. And third, a node can communicate directly only with its neighbors. The definition implies that R_h determines the route on the overlay. In an overlay network without a central directory, the messages between nodes have to be sent along a path called a route.

Definition 3. A route path on an overlay network $P = <N, R_h>$ is a finite path, either $id_{p1}, id_{p2}, \ldots, id_{pn}$ from id_{p1} to id_{pn} such that $id_{pi}\, R_h\, id_{pi+1}$ and $\forall i, j$, $id_{pi} \neq id_{pj}$, or $id_{p1}, id_{p2}, \ldots, id_{pn}$ from id_{p1} to id_{pn} such that $id_{pi}\, {}^{\sim}R_h\, id_{pi+1}$, $\forall i, j$, $id_{pi} \neq id_{pj}$.

Definition 4. A well-structured P2P overlay network $P=<N, R_h>$ is an overlay such that for any two id_i and id_j ($i \neq j$), there must be a deterministic and unique route from id_i to id_j and vice versa through R_h. Otherwise, it is an ill-structured overlay since unless the route is unique, routing will be uncertain.

Theorem 2. In a well-structured overlay $P=<N, R_h>$, R_h should be reflexive and asymmetric if there is to be a deterministic and unique route between any two nodes.

Proof. Clearly, for all id_i, we have $id_i\, R_h\, id_i$. If $id_i\, R_h\, id_j$ and $id_j\, R_h\, id_i$, $i \neq j$, there is always a looped routing path that ends up with the starting node, which breaks the uniqueness and incurs uncertainty to a routing path, that is, P is an ill-structured overlay. Thus in a well-structured P2P overlay, if $id_i\, R_h\, id_j$ and $id_j\, R_h\, id$, $id_i = id_j$. So R_h is asymmetric.

Since no node will be able to reach a node with a smaller ID according to the partial relation R_h, either use the inverse relation ${}^{\sim}R_h$ or add a short cut from the largest ID to the smallest ID. This does not affect the linear ordering relation requirement. This model shows that to build a structured P2P overlay, the neighbor relationship should be reflexive and asymmetric. Now we give a definition of mapping a data object to a node by the key and the node identification.

Definition 5. For each $k_i \in K$ there must be a unique $id_j \in N$ that corresponds to each k_i. The tuple (k_i, id_j) is called the location relation R_d. If $D = K \cup N$, R_d is a relation on D.

Definition 6. Deterministically mapping a $k_i \in K$ to an $id_j \in N$ is to find such an id_{j} that $k_{\mathrm{i}}\, R_d\, id_j$ and there is no id_m that follows $k_j\, R_d\, id_m$ and $id_m\, R_h\, id_j$.

Definition 6 ensures that a k_i is uniquely mapped to an id_j. For simplicity, we assume that each data object d_i has one unique key $k_i \in K$. This definition shows that k_i and id_i are selected from the same domain so that they can be compared with each other and that k_i can equal to id_j.

Lemma 1. If a comparison R is to determine that which one is larger than or equal to the other for two elements, then according to the definition of the linear ordering relation, if R can distinguish every two elements of a set S, R must be a linear ordering relation on the finite set S.

Theorem 3. R_d must be a linear ordering relation on the set D.

Proof. From Definition 5 the only way to determine the unique location is the comparison. From Lemma 1, if R_d is not a linear ordering relation on D, then there must be a k_i and an id_i that cannot be determined by the relation R_d, which means that the location of k_i cannot be determined. Thus R_d must be a linear ordering relation.

Corollary 1. To build a well structured P2P overlay $P = <N, R_h>$, where each key can be deterministically mapped onto a unique node, R_h must be a linear ordering relation on the set N.

Proof. Since R_d is a linear ordering relation on D and $N \subseteq D$, it is also a linear ordering relation on N, i.e. R_d is a reflexive and asymmetric relation on N. From Theorem 2, R_{d} can be used to form a well-structured overlay network, i.e. R_d can be an R_h. If R_h is the only relation used to construct $<N, R_h>$ and R_h is also used to define the deterministic map between keys and nodes, according to Theorem 3, R_h must be a linear ordering relation on N, that is $R_h = R_d$

Theorem 4. To build a well structured P2P overlay $P = <N, R_h>$, where each key can be deterministically mapped onto a node, in the worst case, at least $O(n \log_2 n)$ comparisons are needed, $n = |N|$.

Proof. From Corollary 1, to build such a P, R_h must be a linear ordering relation on N. Then, the whole *id*s will form a sequence L satisfying $id_{p1}\, R_h\, id_{p2}\, R_h\, id_{p3} \ldots R_h\, id_{pn}$. Assume that there are only two adjacent nodes v_i and v_j that break the rule $id_i\, R_h\, id_{\mathrm{j}}$. Since R_h is asymmetric and transitive, in the sequence L, if $id_i\, R_h\, id_{\mathrm{j}}$ does not hold, the only possible permutation of id_i and id_j is that id_i is in the position right to the id_{j}, However, according to Definition 2, since id_i has the neighbor id_j, then $id_i\, R_h\, id_{\mathrm{j}}$ must hold. Thus, if two adjacent nodes cannot follow $id_i\, R_h\, id_{\mathrm{j}}$, then R_h cannot be a linear ordering relation and adjustment is required. Now considering a well-structured $P = <N, R_h>$ and R_h is a linear order relation on the set N, that is, it already has a right sequence L, if a new node id_k is to join the overlay, then $<N, R_h>$ become $<N', R_h'>$, and R_h' may not be a linear ordering relation on N'. In this case, id_k has to be compared to the IDs in N' to find its right location in R_h to make R_h' a linear ordering relation on N'. This process is exactly the same as the sorting process on N'. Thus, whatever method used, it takes the lower bound of $O(\log_2 n)$ comparisons in the worst case as in Theorem 1 for each new node, and for all n new nodes, it will be $O(n \log_2 n)$.