DiSCO: A Distributed Semantic Cache Overlay for Location-Based Services

CONFERENCE PAPER · JULY 2011

DOI: 10.1109/MDM.2011.56 · Source: IEEE Xplore

5 AUTHORS, INCLUDING:

Carlos Lübbe
Universität Stuttgart
12 PUBLICATIONS 12 CITATIONS
SEE PROFILE

Andreas Brodt
IBM
20 PUBLICATIONS 109 CITATIONS
SEE PROFILE

DiSCO: A Distributed Semantic Cache Overlay for Location-based Services

Carlos Lübbe, Andreas Brodt, Nazario Cipriani, Matthias Großmann, Bernhard Mitschang
Universität Stuttgart, Institute of Parallel and Distributed Systems, Universitätsstraße 38, 70596 Stuttgart, Germany
Email: {luebbe, brodt, grossmann, cipriani, mitschang}@ipvs.uni-stuttgart.de

***Abstract*—Location-based services (LBS) have gained tremendous popularity with millions of simultaneous users daily. LBS handle very large data volumes and face enormous query loads. Both the data and the queries possess high locality: spatial data is distributed very unevenly around the globe, query load is different throughout the day, and users often search for similar things in the same places. This causes high load peaks at the data tier of LBS, which may seriously degrade performance. To cope with these load peaks, we present DiSCO, a distributed semantic cache overlay for LBS. DiSCO exploits the spatial, temporal and semantic locality in the queries of LBS and distributes frequently accessed data over many nodes. Based on the Content-Addressable Network (CAN) peer-to-peer approach, DiSCO achieves high scalability by partitioning data using spatial proximity. Our evaluation shows that DiSCO significantly reduces queries to the underlying data tier.**

I. Introduction

Currently, the mobile market faces an enormous boom boosting advertisement campaigns and encouraging immense infrastructure investments. On that account more and more people possess smart mobile devices which combined with widely spread wireless network infrastructure give users access to information anytime and anywhere. In consequence, location-based services (LBS) have gained immense popularity as they provide information considering the spatial context of users which is particularly handy for daily life. As relevant information is closely connected to behavioral patterns of users, the requested data possess high spatial, temporal and semantic locality. For instance, shortly before a soccer match people are concerned about finding parking spots near the arena, while after the match supporters of the winning team might be interested in pubs around the arena. As many people use the same applications on their devices this effect is even enforced. For example, an enhanced navigation application might use a remote LBS to automatically acquire local information, such as special offers of nearby supermarkets. Using such an application, people will even unknowingly create high locality just for the fact of being at the same place in the same time. This causes high load peaks at the data tier of a LBS which may seriously degrade performance.

As load peaks possess high locality, it is reasonable to extend the data tier with a dedicated cache which provides efficient access to the data currently needed. This disburdens the data tier and increases the overall performance of the system. As yet, different approaches to caching exist. Tuple caching and page caching techniques maintain a list of tuple identifiers and physical pages respectively, whereas semantic caching approaches maintain a semantic description of the cache content. In the application domain of LBS, a semantic cache management architecture is preferable to page caching or tuple caching approaches for two reasons: First, the locality of load peaks is not based on a tuple or page level but occurs on a semantic level, such as location or class membership. A semantic cache exploits such locality which makes it the more favorable candidate for handling load peaks. Second, a semantic model is able to use a descriptive query language to describe the requested data. This allows for placing a semantic cache on top of a data management system without any changes to the underlying back-end.

The high locality of load peaks at the data tier is caused by multiple clients simultaneously requesting the same information. Isolated semantic caches located on and exclusively owned by clients, as e.g. in [1], [2], cache subsequent queries of a single client and cannot benefit from former queries of other clients. To exploit the locality of queries across many mobile clients a distributed semantic cache is required.

Many distributed caching approaches still rely on a central component [3] to keep track of the cache content of participating nodes. This disqualifies them for handling load peaks, as any central component introduces a performance bottleneck. In contrast, the participating nodes should form a completely distributed semantic cache overlay, where each node manages a portion of the overall cache content independently and processes queries cooperatively using neighboring nodes in the overlay. Thus, to absorb load peaks in location-based systems,

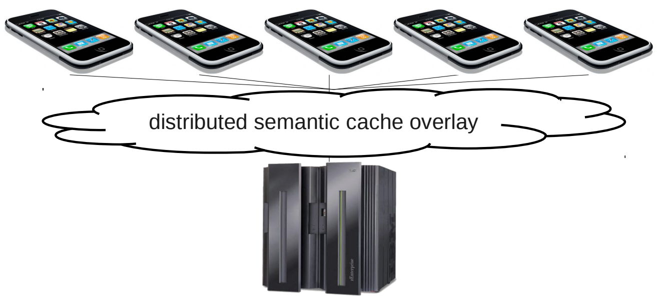


Fig. 1. Introducing a distributed semantic cache overlay between client and data back-end to alleviate load peaks

we introduce a distributed semantic cache overlay between clients and data back-end, as sketched in Fig. 1.

Scalability is a key requirement for a distributed semantic cache overlay. To provide it, the overall effort of processing a single request needs to be kept at a minimum, even when a huge amount of nodes participate in the overlay. Thus, only a small subset of nodes should be involved in processing of a request. This can be achieved by exploiting the locality of requested data. In the domain of LBS, spatial proximity is an important criterion for locality, as users often ask for information regarding a spatial region or position. Using spatial proximity to partition the overall data space between the nodes ensures that the data falling into a certain spatial region is kept by a small subset of nodes. This reduces the number of nodes needed for processing and increases scalability.

In this work we present DiSCO, a distributed semantic cache overlay for LBS. DiSCO uses a semantic model for caching location-based data, thus exploiting spatial, temporal and semantic locality of queries. It is a completely distributed approach avoiding performance bottlenecks. By sharing node caches in the overlay it exploits the high inter-client locality of LBS queries. Using spatial proximity to partition the overall data space it reduces the number of nodes needed to process a query and thereby greatly increases scalability.

Our evaluation shows that DiSCO reduces the load at the data back-end and therefore effectively alleviates load peaks. In addition, our experiments show that DiSCO scales with the number of nodes and clients.

The remainder of this paper is organized as follows. In Section II, we discuss related work. We lay out the theoretic foundations in Section III. In Section IV, we introduce a layered architecture for DiSCO. Section V and VI each detail on a certain layer of DiSCO. We evaluate DiSCO in Section VII and conclude the paper in Section VIII.

II. Related Work

S. Dar et al. [1] propose a semantic model for client-side caching and replacement in a client-server database system as opposed to page caching or tuple caching strategies. The cache consists of disjoint semantic regions each containing a set of tuples and a constraint formula which describes a common property of all tuples in that region. For processing a given query a semantic region is used whenever the constraint formula intersects with the query. In [2] a scheme for optimizing the maintenance of such constraint-based caches is proposed. The central idea of semantic caching was also transferred to other application domains such as XML databases [4], [5], Web Caching [6], [7], and location based services [8], [9]. However, all aforementioned approaches take client-server systems as a basis and do not consider cooperative aspects.

Cooperative semantic caching schemes for relational data and XML have been extensively discussed by the data base community, e.g., [10], [11], [3]. However, these approaches do not consider the intrinsic characteristics of location-based services and thus cannot exploit spatial proximity of data access patterns.

Cooperative caching of spatial data in mobile ad hoc networks has been intensively researched, e.g., in [12], [13]. However, these works focus on use cases with marginal or no network infrastructure facing discontinuous connectivity and network segmentation, which is often the case for disaster or military scenarios. This leads to strategies which are optimized for network segmentation. Thus, nodes cache data of their current vicinity assuming these to be most relevant to close-by nodes in the ad hoc network when disconnected from the data back-end. In this work, we regard disconnection or network segmentation more as failure than as normal condition which allows distributing data between the nodes more flexibly.

Basing upon existing physical network infrastructure, we use common distributed hash table technology to build our cache overlay. The most popular are Chord [14], Pastry [15], Tapestry [16] and CAN [17]. The first three base upon one dimensional key spaces. Considering location-based queries, we would have to transform multidimensional keys to the one dimensional key space of the DHT. As proposed in [18], space-filling curves [19] (e.g., the Lebesgue Curve, Hilbert Curve or Peano Curve) can be used for transforming the key spaces. However, efficient processing of spatial range queries in a data space partitioned by a space-filling curve is intricate, because some parts of the space-filling curve intersecting a given two-dimensional spatial region might not be adjacent in the one dimensional key space defined by the curve. For that reason, we built upon CAN [17], which internally organizes the overlay network using a n-dimensional coordinate space. This guarantees that neighbors in the overlay network are also neighbors in the data space.

Aspects of workload-aware partitioning in overlay networks have been discussed in [20], [21]. However, these approaches base on overlay networks which permanently store data items. Applications using these systems have to explicitly insert and remove data. In contrast, DiSCO transparently caches data offering the same interface as the data back-end of a LBS.

III. Foundations

Our caching-scheme bases on the object-oriented data model of Nexus [22]. It defines an *object* as a set of attribute instances. An *attribute instance* is a tuple consisting of an attribute name and a value. An obligatory attribute instance, named *type*, carries type information. Types form a hierarchical system with subtypes extending their supertypes.

On top of this simple data model queries are formulated as Boolean expressions over predicates, using the Boolean operators $\wedge, \vee$, and $\neg$. A predicate is expressed as $A\phi C$, where A is an attribute name, C a constant and ϕ a comparison. Comparisons can be formulated over multiple data domains, such as numerical ($=, \neq, \leq, \geq$, etc.), string (eq, neq, etc.), type ($instanceof$) or spatial ($within$, $intersects$, etc.). For instance, a query to retrieve all schools in a certain region – e.g., Berlin whose spatial extent is given by the geometry G – having less than than 500 pupils could be formulated as follows:

$$\texttt{type}\ \mathit{instanceof}\ \texttt{School} \wedge \texttt{pupils} < 500$$

$\wedge$ `extent` *intersects* G

Based on this query algebra, a *semantic cache* can be described as follows: A semantic cache maintains a set of cached objects S and a semantic description D of the cache content. D is a Boolean expression describing a common property of the cached objects. The cached object set S must be complete in respect to the cache description, i.e., no object exists in the data source, which satisfies D and which is not cached by the semantic cache. By the means of the cache description, a given query Q can be split into two disjoint parts: one part that can be completely processed using cached objects and one part that requires objects from the data back-end. Similar to [1], we denote the former part as *probe query*, whereas we denote the latter part as *remainder query*. Formally, the *probe query* is defined as $Q \wedge D$, the *remainder query* as $Q \wedge \neg D$.

Based on these concepts, we constitute the notion of a *distributed semantic cache*, which is composed out of a network of N nodes. Each node holds a set of cached objects, which number is restricted to a certain finite capacity. In addition, each node maintains a cache description describing the local cache content. On a global level, the cache content and the cache description can be defined as the union of all local cache contents and cache descriptions. I.e., the global cache description can be defined as $D_1 \vee D_2 ... \vee D_N$ and the global cache content as $S_1 \cup S_2 ... \cup S_N$, where D_i and S_i are the local descriptions and local cache contents respectively.

The local caches may overlap since local cache descriptions are not necessarily disjunct. In order to reduce overlap between the nodes, we partition the data space into cache zones $Z_1, Z_2, ..., Z_N$, where each node is responsible for one zone. The Z_i are Boolean expressions describing the possible content of a node cache. Consequently, the probe query for a given query Q and a node i can be formulated as $Q \wedge Z_i \wedge D_i$ and the remainder query as $Q \wedge Z_i \wedge \neg D_i$.

IV. The DiSCO Approach

DiSCO composes a layered architecture and processing model. As visualized in Fig. 2, applications query a node from the cache overlay, which routes the request to the responsible nodes. A node looks up the requested data in its node cache and may forward the query to the data back-end in case of a cache miss. The results are routed in the reverse direction, merged and sent back to the application. To achieve this, each node runs the protocol stack of Fig. 3. Each layer provides a well defined interface to its upper layer and groups closely related functions. Implementations of layers are easily exchangeable. Typically, each client device runs the complete stack of protocols processing a query top-down from higher to lower layers. In certain cases, it may be reasonable to discard some layers and use a reduced protocol stack only. In the remainder of this section we will first describe key functionality and interfaces of each layer and successively present some reasonable reduced stack configurations.

The topmost layer (*cache overlay layer*) organizes the cache overlay, i.e., partitions the data into cache zones, assigns each node one zone, routes requests to the designated nodes and updates the overlay, when nodes join or leave. For applications this layer provides a simple interface which essentially consists of the method `query`. It takes a Boolean expression as parameter and returns all objects which satisfy the expression.

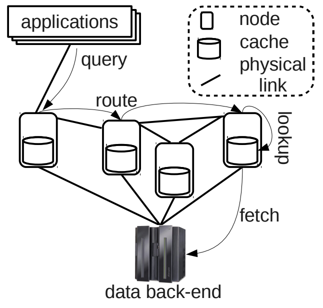


Fig. 2. Processing model

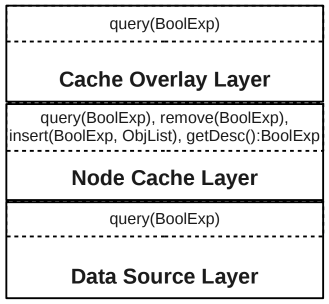


Fig. 3. Protocol stack

The *node cache layer* caches previous queries and their results in volatile memory. It also implements a specific replacement strategy which determines the cache parts to replace when the cache capacity has been reached. It receives queries from upper layers, typically from the cache overlay layer after the request has been routed to its destination. For a given query the layer first processes the query on cached data. If it can be answered completely, it returns the results to the upper layer. Otherwise it sends the remainder of the query to the data source layer. The remainder and the local results are merged and returned to the upper layer afterwards. For upper layers the node cache layer provides a `query` method with identical semantics as the query method of the cache overlay layer. When a new node joins the network, the cache overlay layer needs to split the cache of an existing node and transfer one portion of the split cache to the new node. To provide the means, the node cache layer offers a set of methods: The `getDesc` method can be used to retrieve description of the current cache content of a node. The `insert` method inserts objects into the local cache. The first parameter, a Boolean expression, describes the set of objects which ought to be inserted. The second parameter is a set of objects satisfying the first parameter. The object set must be complete, meaning no object exists which satisfies the Boolean expression and which is not included in the object set. To remove objects from the cache the `remove` method can be used. It removes objects from the cache and updates the internal cache description. For practical reasons both, insert and remove, are not required to implement a strict semantic, i.e., they are not required to insert and likewise to remove all objects specified. For instance, if the removal of certain objects is too expensive, the cache may abstain from removing until they eventually become replacement victims.

The *data source layer* offers an interface to an arbitrary remote data source. It integrates the data back-end of a LBS. The layers main task is to map Boolean expressions to the query language used by the data back-end and to transform the data received from the data back-end to our object-based data model. It may even integrate a federated database system consisting of multiple physical data sources. For the sake of simplicity, we assume a single logical unit in this paper. Upper layers can query this layer using the `query` method with aforementioned semantics.

Each layer mentioned above offers a `query` method having identical semantics. As applications will find the needed functionality on each layer, they can easily skip layers which allows for alternative stack configurations. For instance, an application which does not want to use the distributed cache because it expects little inter-client locality in its queries can skip the topmost layer. Furthermore an application skipping the top two layers and communicating with the data source directly, typically expects no locality at all. Obviously, these different configurations offer optimization opportunities as it might be possible to determine the optimal configuration for a given query before processing. However, this is not the focus of this paper.

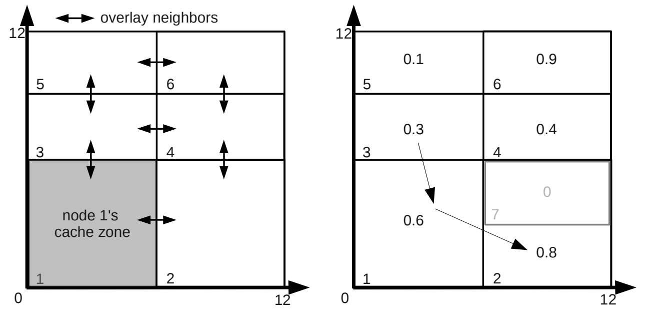
The Boolean expressions used by the interfaces are not domain specific making it open for arbitrary application domains. However, for this work we focus on the domain of LBS. In the following sections, we present protocols of the *cache overlay layer* and *node cache layer* suited for spatial query processing.

V. Cache Overlay Layer

The *cache overlay layer* manages the cache overlay. On that account it has to dynamically partition the data space among the nodes. To keep routing and processing overhead at a minimum it has to ensure that for a given query as few nodes as possible have to participate in processing. For this reason an adequate partitioning and routing scheme is required. In the domain of LBS we expect high spatial locality. For this reason a partitioning and routing scheme based on spatial proximity is reasonable. Our approach builds on the content addressable network (CAN) which can be adapted to spatial partitioning and routing. In the following, we describe base functionality of CAN, explain how CAN can be extended for the spatial domain, show how spatial queries are processed using our spatial extension and describe the maintenance in case of node departure.

A. Content Addressable Network (CAN)

CAN [17] is a distributed infrastructure providing hash table-like functionality. CAN's design bases on a d-dimensional Cartesian coordinate space on a d-torus, which is dynamically partitioned among the nodes. Thus, each node in the network is in charge of its individual, distinct zone within the overall space. Nodes form an overlay representing the coordinate space by learning the IP addresses of nodes with adjoining zones. The set of immediate neighbors in coordinate space and their zones constitute the routing tables. Within the coordinate space (key, value) pairs are stored. Keys are d-dimensional coordinates in that coordinate space. Each node keeps the (key, value) pairs, whose keys lie in its zone. To retrieve a value with key K, a node looks up the corresponding entry and returns the value. If the node's zone does not contain K, the request must be routed through the CAN network until it reaches the node owning K. A node uses its local routing table to forward the request to the neighbor whose zone has the smallest Euclidean distance to the K.



(a) Spatial Topology (b) Node Split

Fig. 4. Spatial overlay

B. Spatial Extension of CAN

We extent CAN to built our cache overlay. Targeting spatial queries and data we straightforwardly use CAN's coordinate space as a container for spatial data. Consequently, this reduces the n-dimensional coordinate space to two dimensions containing geographic coordinates.

Based on the geographic address space, a partitioning and join mechanism is required, that considers the specific domain properties of LBS. To exploit spatial locality, we partition the data space using spatial proximity. For that purpose, each node obtains a cache zone which can be expressed as predicate desccribing the spatial extent of the cache zone. For instance, in Fig. 4a the cache zone of node 1 can be expressed as: $Z_1 := \texttt{extent}\ \textit{intersects}\ G_1$, where G_1 is a geometry delineating the spatial extent of the node's cache zone, i.e., a rectangle spanned by the lower left corner $(0, 0)$ and the upper right corner $(6, 6)$. Note that the *intersects* comparison does not create disjunct cache zones. Consequently, objects which overlap a zone border may be cached by all nodes sharing that border. This is an important property as it enables a node to efficiently decide whether a range query can be answered locally. For instance, consider the query $Q := \texttt{extent}\ \textit{within}\ G_Q$, whereas G_Q is a rectangle spanned by the lower left corner $(0, 0)$ and the upper right corner $(5, 6)$. It can be completely answered by node 1 because its cache zone completely covers the query region.

The nodes in the overlay resemble the geographic topology of the cache zones, meaning that neighbors in the overlay are also neighbors in the geographic coordinate space. Thus, a node can easily determine if one of his immediate neighbors cache zones intersect with a given query and forward it if necessary. Thus, nodes which own portions of the query range can cooperatively process the query without affecting other parts of the overlay. This is a crucial property for efficiently handling range queries which overlap multiple zone borders.

The partitions are created dynamically, whenever a new node joins the overlay. The first node simply obtains the complete address space. Any subsequent node sends a join request to a node which is already part of the overlay.

Bootstrapping, i.e., identifying a node for sending the join request to is not focus of this paper. A practical approach may be to provide a list of common entry points via a DNS server, or ship client software with a contact list. Furthermore, addresses may be probed randomly as discussed in [23], or statistical knowledge may be used to probe the most promising addresses first, as described in [24]. Whichever bootstrapping scheme is used, it must prevent that some nodes are congested with join requests. For this work, we assume that join requests are uniformly distributed among the nodes.

Whenever a node receives a join request, a split candidate has to be found. Making the right split decision is essential for efficiently handling load peaks, as it affects four key performance parameters of the cache overlay:

1) **Hit-rate:** The split decision determines the size of the cache zones. Typically, the size of a cache zone influences the amount of data within the zone. Thus, a split reduces the amount of data which potentially qualifies for each zone. As the capacity of a node remains the same, this increases the probability of an object being cached and improves the hit-rate.
2) **Cache utilization:** A split also divides the cached objects between the new node and the split node. Thus, it affects the cache utilization, i.e., the ratio of the number of currently cached objects and its capacity. To increase efficiency the cache utilization needs to be maximized.
3) **Workload:** As queries possess high spatial locality and splitting reduces the zone sizes the probability for a query to fall into a cache node's zone is affected. Thus, splitting influences the current workload of a node. For instance, a node owning the complete coordinate space has to process all queries, while a node only owning a small portion of it will probably have to process much fever queries. This can substantially increase the overall performance of the system, as load is shared between the nodes. The former holds as long as the zones are substantially larger than the average query regions. When the average query regions become larger than the cache zones, a split enlarges the number of nodes needed to process one query and therefore increases the overhead introduced by distributed query processing.
4) **Routing overhead:** A split enlarges the number of nodes in the overlay. This inevitably increases the number of hops a message has to be routed in order to reach its destiny. When splits occur equally distributed, the average number of hops is independent of the message destination. Splitting the coordinate space non-uniformly results in additional overhead for routing messages through dense split regions.

The split candidate can be chosen randomly which on average results in equally distributed splits. Facing load peaks with high spatial locality certain hot spot regions will be accessed more frequently. Using random splitting, nodes owning a hot spot zone will receive significantly more requests than average and in consequence risk being congested. In addition, caches in hot spot regions have to handle large amounts of data causing many data objects to be replaced too early. For these reasons, we introduce a split mechanism which considers run-time statistics of the cache nodes.

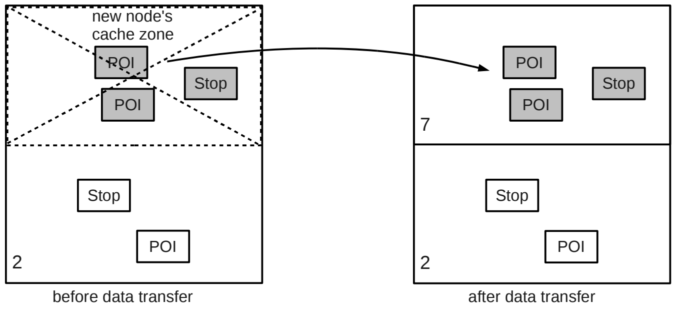


Fig. 5. Transfer of node 2's cache content

A node receiving a join request must be able to locally decide whether to split its own cache zone or whether to forward the join request to some more suitable neighbor. For that purpose, we extend the routing tables to hold an additional entry, called *split indicator*. The *split indicator* is a quantity derived from the run-time statistics of a cache node, such as current workload or cache utilization. It enables a node to compare different split candidates. The node receiving the join request compares its own and the neighbors split indicators and recursively forwards the request to the node with the highest split indicator, until no neighbor has a higher split indicator.

Our split mechanism does not guarantee to find the best split candidate on a global level. For instance, Fig. 4b shows a join request which was forwarded to node 2, even though node 6 has the highest split indicator in the overlay. Nevertheless, if join requests occur uniformly distributed, node 6 will eventually be split. If the used bootstrapping mechanism distributes joins non-uniformly, single nodes might never receive a join request even if they have a high split indicator. To counteract such situations, join requests could be routed to a random node first, before it forwards the join according to the split indicator.

Once a join request has reached its final destination, the node splits its cache zone along one dimension in two equidistant portions. Each time the node is split, it alters its split dimension causing the node to split its zones in turn along the x-axis and y-axis. The node computes a geometry G_N and a cache zone expression $Z_N := \texttt{extent}\ \textit{intersects}\ G_N$ which defines the cache zone of the new node. In addition, the node computes the new nodes routing table N_N containing the new node's neighbors, their cache zones and their split indicators. Furthermore, the node needs to transfers the parts of its cached content which are covered by the new node's cache zone, as depicted in Fig. 5. To do so, it creates a predicate describing the cache content which ought to be transferred. This predicate can be defined as $E := \texttt{extent}\ \textit{within}\ G_N\ \wedge\ D$, where D is the current cache description retrieved by the `getDesc` method. The node first retrieves all objects S_N from its cache satisfying E by calling query(E) and afterwards removes those objects by calling remove(E). The routing table N_N, the new node's cache zone expression Z_N, the removed objects S_N and their description E are sent to the new node. The new node initializes its routing table according to N_N. Then it sets its own cache zone to Z_N, inserts the objects into its cache by calling insert(E, S_N) and finally acknowledges the sender. After receiving the new node's acknowledgment the node updates its own routing table and notifies his old neighbors to do the same. At this moment the join is complete and the new node is ready to receive and process queries.

C. Query processing

The cache overlay layer processes queries cooperatively exploiting their spatial locality. For the matter of simplicity, we will presume that queries are conjunctions of simple comparisons including at least one spatial comparison and extend our model to arbitrary Boolean expressions afterwards.

Whenever a client sends a query to an arbitrary node in the cache overlay, the nodes which can contribute results have to be found. For that purpose, we exploit the routing mechanism of CAN which is capable of routing messages to a certain coordinate in its coordinate space. A coordinate for routing the query can be easily determined by investigating the reference geometry in the spatial comparison. Any coordinate intersecting the reference geometry belongs to a node which must participate in processing the query. We simply choose the first point defining the reference geometry and route the query to the node owning this point.

Fig. 6 depicts a range query which has been routed to node 3 by choosing the lower left corner $(3, 8)$ of the reference geometry as address-coordinate. Node 3 initializes the processing of the query. In our example the query region overlaps with zones of other nodes, one of which not being an immediate neighbor of node 3. Node 3 generates a unique process ID, attaches it to the query and forwards it to all neighbors intersecting the query region. All neighbors receiving the request process the query locally using the local cache layer and recursively forward the query to all intersecting neighbors except the one from which they received the request. It may happen that a node receives a duplicate request. To prevent processing the same query twice, each node keeps the unique process ID in a ignore list. In our example, node 6 receives a duplicate request, which is silently dropped according to the nodes ignore list information. Each participating node sends results to the initial node, which merges them. In addition, the node may also remove duplicates due to objects intersecting multiple cache zone borders. Finally, the node sends the results back to the client issuing the query.

For queries which contain arbitrary Boolean expressions the aforementioned technique can also be reused. We transform an arbitrary Boolean expression into disjunctive normal form (DNF) containing disjunctions of conjunctions of comparisons. For each conjunctive term containing a spatial comparison we use the aforementioned scheme. I.e., the node initially receiving the request extracts an address coordinate from

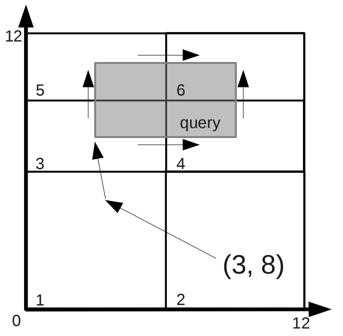


Fig. 6. Query processing

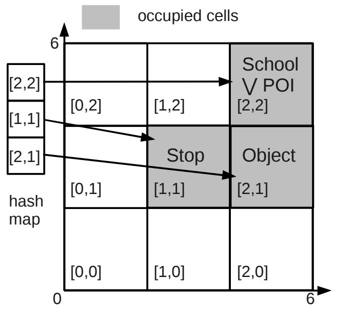


Fig. 7. Virtual grid structure

each reference geometry of the spatial comparison, routes the respective conjunctive term query to these coordinates using CAN and finally merges the partial results of each conjunctive term query. Likewise the result may be stripped of duplicates caused by overlapping conjunctive terms.

A conjunctive term without any spatial comparison potentially qualifies for every node in the network. Processing such a query in the overlay would essentially require flooding the entire network. For this reason, we forward such terms to the data source directly.

D. Maintenance of DiSCO

Continuous maintenance of DiSCO is required for two reasons: First, the split indicator entry in the routing tables has to be updated to ensure good split decisions. Second, in presence of node departures or node failures, we have to find a node which takes over the leaving node's zone. For both, we can build upon the maintenance mechanisms of CAN [17].

To detect node failures in CAN, nodes periodically send update messages to their neighbors. The absence of an update message indicates node failure. To update the split indicator, we exploit this mechanism and send an up-to-date split indicator value piggybacking each update message.

Once a failure is detected or a node explicitly leaves the overlay, two mechanisms are triggered: At first, an immediate neighbor takes over and temporarily manages the zone besides his own zone. While CAN uses zone size to determine the take-over node, we delegate the zone to the immediate neighbor with the smallest split indicator. Subsequently, this node starts a background reassignment process to find a pair of nodes which zones can be merged and to transfer the adopted zone to the left-over node. The basic idea is to perform a depth first search in the binary tree the partitions form conceptually to find a pair sibling leaf nodes which partitions can be merged. Details can be found in [17].

During both the take-over process and the background reassignment process nodes solely rearrange cache zone management responsibilities. This is opposed to the CAN maintenance mechanisms, where nodes explicitly parting the overlay also transfer their data content to a remaining node. In contrast to the CAN model, the DiSCO nodes possess a limited capacity. This would cause the node which takes over the leaving nodes data to replace huge amounts of its own cache. For this reason no cache content is transferred when a node leaves the overlay.

VI. Node Cache Layer

The node cache layer manages a node's local cache. It organizes cached objects in memory, maintains a cache description of the cached objects and implements a strategy to replace outdated objects when the cache capacity is reached. It implements the node cache layer interface given in Fig. 3.

To handle load peaks, efficient cache management is required. This is particularly important for constructing the remainder query, as complex remainders can lead to higher processing costs at the data back-end. The complexity of the exact remainder $Q \wedge \neg D$ of query Q, as proposed in e.g. [1] or [25], strongly depends on the complexity of the current cache description D. The complex geometries of spatial queries rapidly increase the complexity of an exact cache description and consequently increase the complexity of remainder queries. Therefore, we use a simplified remainder representation R which satisfies $Q \wedge \neg D \Rightarrow R$. This enables efficient query processing, but may lead to fetching more objects from the data back-end than needed to answer Q.

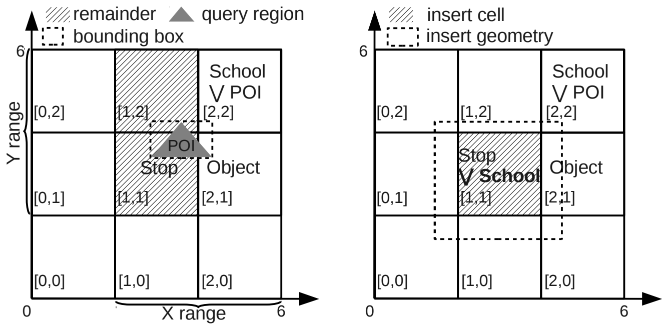
Our approach bases on a uniform grid, which significantly reduces the complexity of geometries involved in remainder construction. The grid structure is completely virtual and empty cells do not allocate any memory. Only the occupied cells are kept in a hash map indexed by their cell keys, as depicted in Fig. 7.

The hash map has a fixed capacity, which requires a replacement strategy, once the capacity is reached. Targeting load peaks of LBS, we expect queries to have temporally high locality. For instance, social events, such as soccer matches, temporally bring many people with shared interests together. This causes temporally accumulated requests for similar information. Motivated by this observation, we implemented a LRU strategy, replacing least recently used cells first.

Each grid cell corresponds to a spatial region and manages objects intersecting that region. Objects intersecting a cell border may be cached by multiple cells. A cell has two keys x and y which define its position in the grid. In addition, a cell keeps a Boolean expression describing its current content. We denote the set of objects managed by cell $[x, y]$ as $S_{[x,y]}$ and the Boolean expression as $T_{[x,y]}$. For instance, cell $[2, 2]$ in Fig. 7 covers a spatial region, which is delineated by the rectangle $G_{[2,2]}$ with the lower left corner $(4, 4)$ and the upper right corner $(6, 6)$. The cell caches schools and POIs, i.e., $T_{[2,2]} := \texttt{type}\ \textit{instanceof}\ \texttt{School} \vee \texttt{type}\ \textit{instanceof}\ \texttt{POI}$. For simplicity, we just give the type name, when we mean a *instanceof* comparison, i.e. $T_{[2,2]} := \texttt{School} \vee \texttt{POI}$.

On top of this simple structure, we elaborate the interface for the node cache layer defined in Fig. 3. In the following, we briefly discuss the mode of operation for each method.

query: The cache overlay layer calls this method, passing the query Q as a Boolean expression. Without loss of generality, we presume Q to be a conjunction of simple comparisons, as arbitrary DNFs can be processed by calling



(a) Query (b) Insert

Fig. 8. Grid-based cache index

`query` for each conjunction and merging the partial results of each call. We also assume at least one spatial comparison, as non-spatial queries are forwarded to the data source directly. The method computes the geometry G_Q covering the query region of all spatial comparisons. For instance, multiple *within*-comparisons are covered by the geometric intersection of their reference geometries. Afterwards, it determines the key range for each dimension intersecting G_Q's bounding box, as shown in Fig. 8a. For each dimension the method iterates over all cell keys in the key range. For each key set $[x, y]$, it checks, whether the cell region $G_{[x,y]}$ intersects with G_Q. If so, it probes the hash map of occupied cells. If the hash map contains an entry for that key set and $T_{[x,y]}$ subsumes the conjunction of the non-spatial comparisons T_Q (i.e., the set of objects matching T_Q is a subset of the objects matching $T_{[x,y]}$), all objects satisfying Q are added to a preliminary result set S. Otherwise, a cell remainder expression $R_{[x,y]} := T_Q \wedge \texttt{extent}\ \textit{intersects}\ G_{[x,y]}$ is built. Finally, the method builds the remainder query by combining all the cell's remainders: $R := R_{[x_1,y_1]} \vee R_{[x_2,y_2]} ... \vee R_{[x_k,y_k]}$. The method passes the remainder query to the data source layer, merges the results returned by the data source layer with the results retrieved from the cache and returns the merged result. Subsequently, we sketch the method in pseudo code:

```
function QUERY($Q$)
    $T_Q$: non-spatial parts of $Q$
    $G_Q$: geometry covering the spatial region of $Q$
    $LoKeyX, HiKeyX, LoKeyY, HiKeyY$: key ranges intersecting $G_Q$'s bounding box for each dimension
    $S \leftarrow \emptyset$, $R \leftarrow$ null
    for $x \leftarrow LoKeyX, HiKeyX$ do
        for $y \leftarrow LoKeyY, HiKeyY$ do
            if $G_{[x,y]}$ intersects $G_Q$ then
                if probe($[x, y]$) $\wedge$ $T_{[x,y]}$ subsumes $T_Q$ then
                    $S \leftarrow S \cup \{s | s \in S_{[x,y]} \wedge Q(s)\}$
                else $R \leftarrow R \vee R_{[x,y]}$
                end if
            end if
        end for
    end for
    if $R \neq$ null then $S \leftarrow S \cup$ queryDataSource($R$)
    end if
    return S
end function
```

Note, that testing subsumption for arbitrary Boolean expressions can be computionally intensive, as it requires normalizing the expressions. Thus, we restrict the cache granularity to type predicates and thereby reduce the complexity of the subsumption test. Assuming that queries contain simple type comparisons, each $T_{[x,y]}$ reduces to a simple disjunction of type comparisons. The types used in the comparisons form a hierarchical system, as depicted in Fig. 9. Thus, subsumption can be tested by checking if the queried type has an equal-to or an is-a relationship with one of the cell types. E.g., in Fig. 8a POI is subsumed by $T_{[2,2]}$ and $T_{[2,1]}$, but not by $T_{[1,1]}$.

getDesc: The current cache description is constructed by iterating over all occupied cells. For each cell, the method builds the expression $D_i := T_{[x_i,y_i]} \wedge \texttt{extent}\ \textit{intersects}\ G_{[x_i,y_i]}$. For instance, the expression of cell $[2,2]$ in Fig. 7 can be defined as: $D_1 := (\texttt{School} \vee \texttt{POI}) \wedge \texttt{extent}\ \textit{intersects}\ G_{[2,2]}$. The method returns the cache description $D := D_1 \vee D_2 ... \vee D_K$, where K is the number of occupied cells.

insert: The method inserts a set of objects S satisfying the Boolean expression Q into the cache. It computes geometry covering the spatial comparisons of Q and determines all cells that are fully covered by this geometry. For each cell where $T_{[x_i,y_i]}$ does not subsume the non-spatial parts T_Q, it adds all objects satisfying the cell's region and sets $T_{[x_i,y_i]} := T_{[x_i,y_i]} \vee T_Q$. Fig. 8b depicts inserting `School` objects in cell $[1,1]$.

remove: The remove method works conversely to the insert method and computes the geometry covering the spatial comparisons of the remove expression E and extracts the non-spatial parts T_E. For each cell $[x_i, y_i]$ fully covering the geometry, it removes all objects satisfying the T_E from $S_{[x_i,y_i]}$ and sets $T_{[x_i,y_i]} := T_{[x_i,y_i]} \wedge \neg T_E$. For instance, removing the previously inserted `School` objects from cell $[1,1]$ in Fig. 8b, leads to $T_{[1,1]} := \texttt{Stop} \wedge \neg\texttt{School}$. Considering the type hierarchy of Fig. 9, this is equivalent to $T_{[1,1]} := \texttt{Stop}$.

VII. Evaluation

We evaluated our approach by simulation using PeerSim [26], which bases on a discrete event-driven simulation model. PeerSim is able to simulate peer-to-peer overlays of nodes, where each node runs a set of arbitrary protocols. We implemented all layers of the DiSCO architecture (Fig. 3) as protocol stack in the PeerSim environment.

Our simulation bases on real world data of Berlin extracted from OpenStreetMap [27]. We converted their data format into a format based on the data model introduced in Section III. The data set comprises a total number of 23625 objects.

On top of this data, we simulated range queries of mobile clients, asking for information surrounding their current position. The positions determine the centers of uniform square query regions with a side length of 500 m. In addition, each query contains a predicate determining the type of objects which ought to be returned. The object types form a hierarchical system of different selectivities per type. Fig. 9 shows some exemplary types and their selectivities in parentheses.

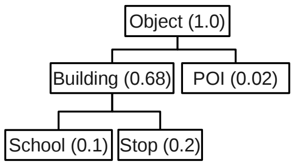


Fig. 9. Type hierarchy

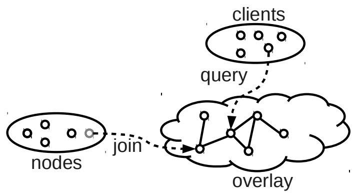


Fig. 10. Simulation model

For simulating client positions, we consider two scenarios:

1) **Hotspot Scenario:** To consider the spatial locality of load peaks, we simulate query hot spots. We generated client positions according to a Gaussian distribution with a deviation of 1000m around a hot spot.
2) **Mobility Scenario:** We used the mobility simulation environment CANUMobiSim [28] to create client positions. As simulation model a vehicle-like movement was chosen, where clients move on streets randomly changing their speed in between 30 km/h and 60 km/h.

Based on these general scenarios, we derive five different behavioral patterns of clients:

- **n = 1|10|100:** We simulate n clients moving on streets requesting all objects (i.e., instances of `Object`) in the query region.
- **n = 100 MIX:** In reality, we expect that many clients request similar information, but at distinct places. This resembles different shared applications running on multiple client devices which automatically request similar information per default. Thus, we simulate node groups, where all nodes of a group query for the same object type. We created five node groups with equal cardinality of 20 nodes per group. Each of the five node groups query one of the five types depicted in Fig. 9.
- **n = 100 HOT SPOT:** We selected five distinct hot spots in the center of Berlin. At each hot spot, nodes ask for one of the types depicted in Fig. 9

We simulated the aforementioned patterns using a discrete event-driven model, i.e., the simulation is represented by a chronological sequence of events, where each event occurs at an instant in time. The events represent queries or join requests which are both sent in equidistant time intervals to a random node in the overlay. During the simulation, the overlay is dynamically constructed.

As shown in Fig. 10, our simulation model bases on two distinct node sets: the *clients*, which send queries to the overlay and the *nodes*, which successively join the overlay. Nodes may participate in query processing as soon as they are joined. We simulated an overlay construction phase, during which both queries and join events occur, and a query-only phase. During the overlay construction phase nodes are split according to two split schemes: random split and load split. The *random split* scheme simply splits the node which receives the join request. As join requests are randomly sent to nodes, this results in

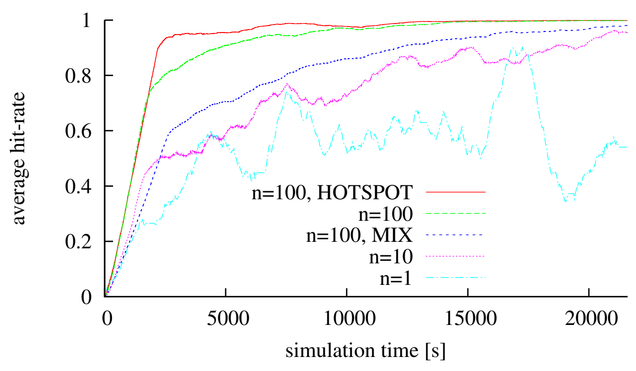


Fig. 11. Average hit-rate

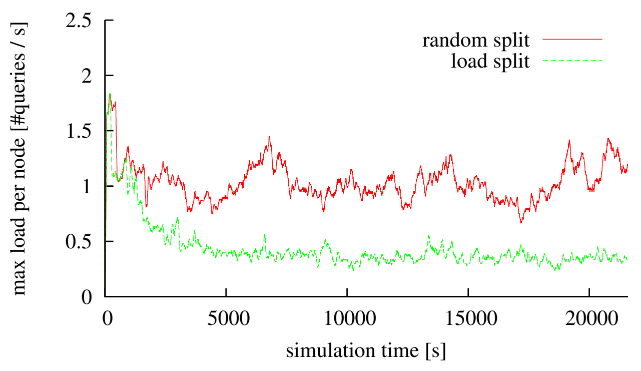


Fig. 13. Maximum of load per node using the “n=100, MIX” pattern

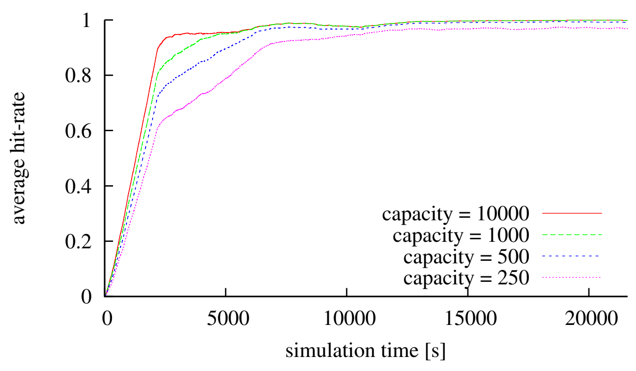


Fig. 12. Impact of node capacity on the hit-rate

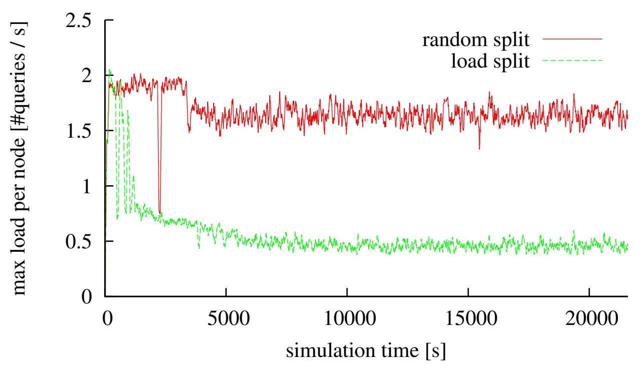


Fig. 14. Maximum of load per node using the hot spot pattern

uniformly distributed splits. The *load split* scheme recursively forwards the join request to the neighbor with the highest workload. The simulation runs for six hours of simulated time (21600 s) and the overlay construction is complete after three hours (10800 s). Each client sends queries in equidistant time intervals of 30 s during the whole simulation time. This results in 720 queries per client, i.e., 72000 queries and approximately three queries per second when 100 clients are simulated. The join requests occur during the overlay construction phase, i.e., every 108 seconds, until the size of 100 nodes is reached.

Based on the general simulation model described above, we defined concrete simulation settings. As the simulation was executed in a deterministic fashion, we ran the simulation exactly once per setting. In the following, we describe the main results of these simulation runs.

A. Hit-Rate

We start with examining the cache hit-rate, i.e., the ratio of the number of objects retrieved from the cache and the total number of objects satisfying the query.

Fig. 11 shows the moving average of 1500 s. In this setting, the capacity per node is 10000 objects, leading to very few cache misses due to replacement. The network was split using the current workload per node as split indicator. The graph depicts the hit-rate for the five different simulated behavioral patterns of clients. The evaluation shows the impact of inter-client locality on the hit-rate. The pattern “n=1” cannot profit at all, as only one client is involved. Thus, it has the lowest overall hit-rate, even though it has occasional high hit-rate periods. These are caused by the client moving into regions he has already visited or moving at low speed so that regions of subsequent queries overlap. The other patterns show higher hit-rates caused by higher inter-client locality. Thus, 100 nodes lead to a higher hit-rate than ten nodes and one node. The hot spot pattern shows the highest hit-rates, because the geographic regions of queries overlap with high probability. Naturally, 100 nodes asking for the same object types produce more locality than 100 nodes asking for different types.

The results in Fig. 12 show the impact of the per node capacity on the hit-rate. We simulated the hot spot pattern in load split mode and reduced the capacity of each node. This reduces the hit-rate at most during the overlay construction phase, which is due to the reduced overall capacity of the overlay during that time.

In a nutshell, the results show rapidly increasing cache hits outreaching 90 % for patterns with high inter-client locality. Facing load peaks with high locality, this significantly reduces workload at the data back-end, as most of the data can be retrieved from the cache.

B. Workload per Node

The workload per node is an important indicator for the overall performance of the system. We define the workload per node as the number of processed queries per node and second. This includes the part requests of queries overlapping zone borders, but excludes routing. The workload per node

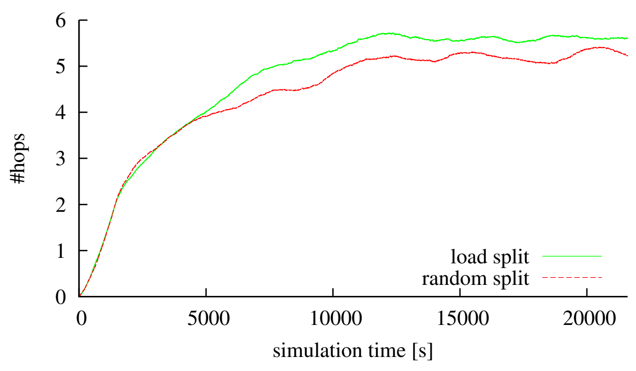


Fig. 15. Average routing overhead

indicates whether the system will scale with increasing number of requests and nodes. Ideally, the overall workload should be equally distributed to all participating nodes.

Fig. 13 depicts the maximum workload per node at a given time simulating the "n = 100, MIX" pattern. The graphs show that the splitting reduces the workload per node, as more nodes share the same overall query load. Furthermore, using the workload as split indicator significantly reduces the maximum load per node. This effect is even enforced when simulating the hot spot pattern, as shown in Fig. 14.

C. Routing Overhead

The routing overhead is a key performance indicator. It should not increase over-proportionally with increasing number of nodes.

Fig. 15 shows the average hop count for a query to reach the node which starts the processing. We selected the "n=100, MIX" pattern for this setting. The graphs show that random splitting performs slightly better than load-based splitting. The reason for this is that load-based splitting leads to non-uniformly distributed splits which increase the node density in certain spatial regions. Routing messages through these dense nodes regions causes additional overhead. However, this overhead is comparatively low.

VIII. Conclusion

We addressed the problem of load peaks at the data back-end of location based systems (LBS). To alleviate load peaks, we introduced DiSCO, a distributed semantic cache overlay for LBS. Placing DiSCO between client and data back-end alleviates load peaks and increases the overall performance of the LBS.

Our evaluation showed that DiSCO reduces the amount of data requested from the data back-end and therefore effectively alleviates load peaks. Furthermore, we showed that partitioning according to the workload decreases the load per node. As our experiments included a multitude of nodes and a huge amount of queries, we showed that DiSCO scales with the number of nodes and clients.

Encouraged by our positive results with workload-aware partitioning, we plan to consider alternative run-time statistics, such as cache hit-rate or cache utilization, inorder to partition the data space. Furthermore, we aim at investigating the impact of replication mechanisms on the per node workload. In addition, as our caching mechanism is currently optimized for queries containing spatial and type predicates, we envisage introducing additional dimensions, such as time, into cache management. This will increase the cache hit-rate, but may also render cache management more difficult, as cache descriptions will become more complex. Finally, we did not consider updates at the data back-end, which may lead to inconsistency of cached data and back-end data. We plan to extend DiSCO to handle such inconsistencies in future work.

References

[1] S. Dar et al., "Semantic data caching and replacement," in *VLDB '96*. San Francisco: M. K. Publishers Inc., 1996, pp. 330–341.

[2] J. Klein et al., "Optimizing maintenance of constraint-based database caches," in *ADBIS '09*. Berlin: Springer, 2009, pp. 219–234.

[3] J. Colquhoun et al., "A p2p database server based on bittorrent," School of Computing Science, Newcastle University, Tech. Rep., 2010.

[4] B. Mandhani et al., "Query caching and view selection for xml databases," in *VLDB '05*. VLDB Endowment, 2005, pp. 469–480.

[5] L. Chen et al., "Xcache: a semantic caching system for xml queries," in *SIGMOD '02*. New York: ACM, 2002, pp. 618–618.

[6] B. Chidlovskii et al., "Semantic caching of web queries," *The VLDB Journal*, vol. 9, no. 1, pp. 2–17, 2000.

[7] D. Lee et al., "Towards intelligent semantic caching for web sources," *Journal of Intell. Inform. Systems*, vol. 17, pp. 23–45, 2001.

[8] K. C. K. Lee et al., "Semantic query caching in a mobile environment," *SIGMOBILE Mob. Comput. Comm. Rev.*, vol. 3, no. 2, pp. 28–36, 1999.

[9] Q. Ren et al., "Using semantic caching to manage location dependent data in mobi. com." in *MobiCom*. New York: ACM, 2000, pp. 210–221.

[10] K. Lillis et al., "Cooperative xpath caching," in *SIGMOD '08*. New York: ACM, 2008, pp. 327–338.

[11] A. Vancea et al., "A Cooperative Semantic Caching Architecture for Answering Selection Queries," IFI., Uni. of Zurich, Tech. Rep., 2009.

[12] Y. Wang, E. Chan, W. Li, and S. Lu, "Location dependent cooperative caching in manet," in *ICPP '08*, sep. 2008, pp. 470 –477.

[13] F. J. Maymí et al., "A cooperative spatial-aware cache for mobile environments," in *MobiDE'10*. New York: ACM, 2010, pp. 65–72.

[14] I. Stoica et al., "Chord: A scalable peer-to-peer lookup service for internet appl." in *SIGCOMM*. New York: ACM, 2001, pp. 149–160.

[15] A. I. T. Rowstron et al., "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. Int. Conf. on Dist. Systems Platforms*. London: Springer, 2001, pp. 329–350.

[16] B. Y. Zhao et al., "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Berkeley University, Tech. Rep., 2001.

[17] S. Ratnasamy et al., "A scalable content-addressable network," in *SIGCOMM '01*. New York: ACM, 2001, pp. 161–172.

[18] M. Knoll et al., "Optimizing locality for self-organizing context-based systems," in *Self-Organizing Systems*. Springer Berlin, 2006, vol. 4124, pp. 62–73.

[19] H. Sagan, *Space-Filling Curves*. New York: Springer-Verlag, 1996.

[20] K. Aberer et al., "Indexing data-oriented overlay networks," in *VLDB '05*. VLDB Endowment, 2005, pp. 685–696.

[21] T. Scholl et al., "Workload-aware data partitioning in community-driven data grids," in *EDBT '09*. New York, NY, USA: ACM, 2009, pp. 36–47.

[22] D. Nicklas et al., "On building location aware applications using an open platform based on the NEXUS augmented world model," *Software and System Modeling*, vol. 3, no. 4, 2004.

[23] J. Dinger et al., "Decentralized bootstrapping of p2p systems: A practical view," in *Proc. of the 8th Int. IFIP-TC 6 Networking Conf.* Berlin: Springer, 2009, pp. 703–715.

[24] C. GauthierDickey et al., "Bootstrapping of peer-to-peer networks," in *Proc. of the Int. Symposium on Applications and the Internet*. Washington: IEEE, 2008, pp. 205–208.

[25] Y. Ishikawa et al., "A semantic caching method based on linear constraints," in *DANTE '99*. Washington DC: IEEE, 1999.

[26] M. Jelasity et al., "Peersim," Oct. 2010, `http://peersim.sf.net`.

[27] OpenStreetMap, Nov. 2010, `http://www.openstreetmap.org`.

[28] I. Stepanov et al., "A meta-model and framework for user mobility in mobile networks," in *ICON*, 2003, pp. 231 – 238.