A Survey on Caching and Data Consistency in P2P Networks

Dai Bing Tian
HT040826N

Zeng Yiming
HT040835M

Contents

Chapter 1

Introduction

Peer-to-Peer(P2P) technology is becoming extremely popular due to its self-organization, flexibility, scalability, fault-resilience and robustness. In order to improve the performance of P2P network, caching is a very frequently used technique to achieve fast response and reduce the overall load, so that the bandwidth can be utilized efficiently.

Generally, we can imagine caching technique as a layer between the applications and P2P routing schemes, such as CHORD, CAN, or PASTRY. When user issues a query for some object or some range, caching will first look for it among peers in the system, thus it needs those P2P routing schemes as their object location services. In case the searching fails, the request will still be routed to the peer or server who owns the object.

However, the caching technique can not just return whatever cached copy it has found, a data consistency check should be conducted before the cached copy can be returned to the requester.

Besides being a model for object and file retrieval, P2P system has the potential to serve as a platform for distributed databases. In this report, we discuss the techniques proposed to support caching of both objects and database tuples.

1.1 Organization of the Report

The rest of the report is organized as follows. In Chapter 2, we discuss a decentralized, peer-to-peer web cache called Squirrel. In Chapter 3, we cover an architecture for supporting online analytical processing (OLAP) queries, called PeerOLAP. Chapter 4 and Chapter 5 cover techniques for caching for range queries.

Chapter 2

Squirrel

Iyer, Rowstron and Druschel[1] presented a decentralized, peer-to-peer web cache called Squirrel. The key idea is to enable web browsers on desktop machines to share their local caches, to form an efficient and scalable web cache. Squirrel uses a self-organizing, peer-to-peer routing substrate called PASTRY as its object location service to identify and route to the node that caches a copy of the requested objects. PASTRY is resilient to concurrent node failures, and so is Squirrel.

Just like normal web browser cache proxy, when a request is issued to Squirrel proxy, it checks whether the local cache has one copy of the object. If a fresh copy of the requested object is not found, it will try to look for a copy on some other node.

As described in PASTRY, every object has its own 128-bit objectID, and it will be stored into the node with nodeID numerically closest to the objectID, where the node is called the *home node* for this object. In Squirrel, the objectID is obtained by applying SHA-1 function to the URL, and PASTRY will handle routing and locating the object.

There are two approaches for Squirrel, Home-store and Directory respectively. Home-store approach can be considered as a direct approach that the home node of an object will store it locally if the object is cacheable. In the directory approach, the function of the home node is to store the pointer to those peers who may have a copy of requested object, instead of to store it locally. In the following two sections, we will illustrate these two approaches. For the sake of simplicity, we assume the object we will be discussing are all cacheable. For uncacheable object, the requester can only fetch it from the origin server who owns the object.

2.1 Home-store

Before we discuss Home-store approach, we first introduce what are *GET* and *cGET* request. Web browsers generate HTTP *GET* requests for Internet objects like web pages, images, etc. For each request, the web browser proxy will try to look for this object in its own cache, there are two possibilities: the requested object is not found or the object can be found in its own cache. In the first case, the request is diverted to the origin server. For the second case, the cached object will be tested for freshness. If it is fresh, it will be returned directly, else a *cGET* request is send to the origin server, which will send back either a "Not-modified" message to confirm the freshness of the cached copy, or send a new copy of the object if it has been updated recently. This gives a basic description of web cache, it only makes use of local cache, there is no collaboration among the caches in the same network space. The contribution of Squirrel is that they proposed a way to share their local caches and make use of them.

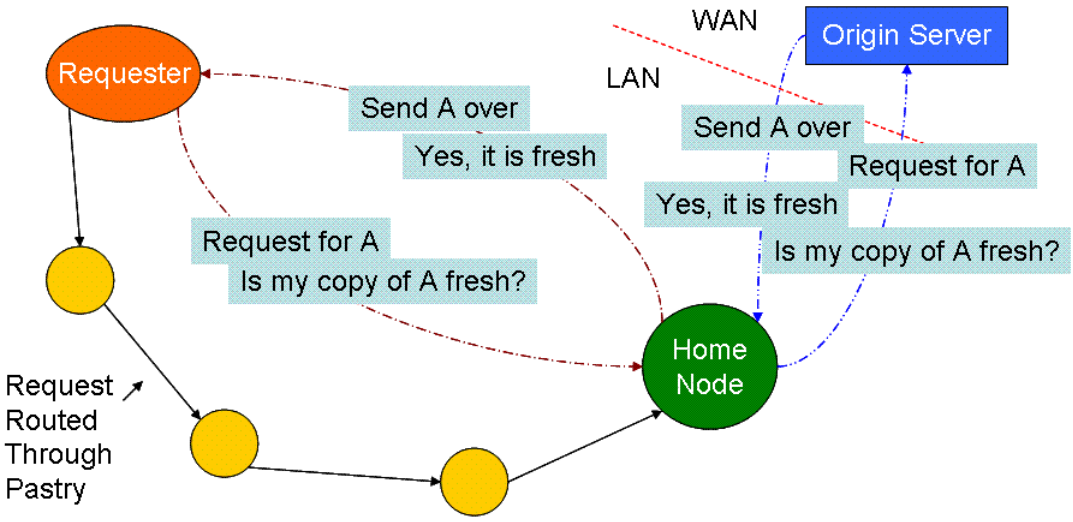

Figure 2.1: Home-Store Approach

In this approach, the object is stored at its home node. Suppose some client wants object A, we call the client as requester. As shown in Figure 2.1, the protocol works as follows:

- There is a fresh copy in the cache of the requester
  - return it directly

- If there is no copy in the cache of the requester
  Route through Pastry to find the home node of object A, and request A from its home node
  - If there is a fresh copy in the cache of home node
    - send A to the requester directly
  - If there is no copy in the cache of home node
    - request object A from the origin server, which will send A back, home node caches it and sends it requester.
  - If there is a copy, but the freshness is not granted
    Ask the origin server for A's freshness by just sending the A's version number
    - If origin server finds the A's version number is latest, it will just send a "Not-modified" message to the home node. The home node will update A's metadata and send A to the requester.
    - If origin server finds a later version of A, it sends A to the home node. Home node caches it and sends it to the requester as well.
- If there is a copy in the cache of requester, but it is not sure about A's freshness
  It will ask for A's freshness from its home node
  - If the home node can ensure A's freshness
    - Send "Not-modified" message to the requester if the copy at the requester side is fresh.
    - Send A over if the copy at the requester side is not fresh
  - If the home node do not know whether A is fresh, it will ask for A's freshness from the origin server
    - Origin server will send "Not-modified" message to the home node if the copy at the home node is fresh. Home node will just update A's metadata and forward the message to the requester.
    - Origin server sends A over if the copy is not fresh. Home node caches it and sends it to the requester.

Since all requests to the object are routed through its home node, the home node normally maintains the most up-to-date copy of the object in the Squirrel network. When cache is full, all objects stored are treated equally by the cache replacement policy.

2.2 Directory

In home-store approach, all peers are treated equally likely. However, it is quite obvious that some peer may have smaller cache compared to others, thus it results very frequently cache replacements and cache misses in the peers with smaller cache, while the caches of some other peers with larger cache are not fully utilized.

In contrast to home-store approach, directory approach make the home node of some object to store pointers to those peers whose cache contains this object. In this approach, the home node for an object stores up to K pointers to the nodes that have most recently accessed the object, which are called delegates. The delegates store same version of the object. Any node that recently accessed an object can be a delegate, the key idea is to redirect the request to a randomly chosen delegate from the home node. Figure 2.2 describe the process when object A is requested.

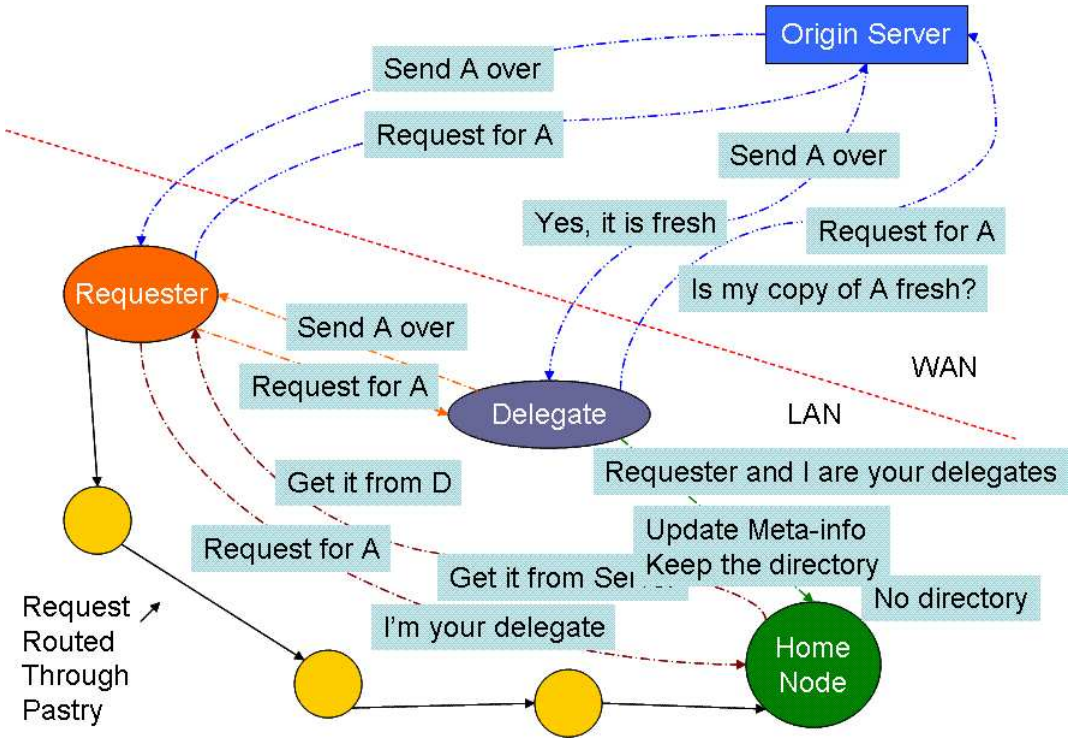

Figure 2.2: Directory Approach

Since the home node of an object also stores its metadata, when a *cGET* request is received by the home node, it can answer directly if it knows its freshness, otherwise, it will ask for its freshness from the server through a

similar manner as requesting.

The request for A route through Pastry to find its home node

- If there is no directory stored at the home node
  Home node asks the requester to get a copy directly from the origin server. Once the requester receives the copy, it will send the metadata of object A to its home node; and the home node will create a new directory for A, taking the requester as the first delegate.

- If there is a directory at the home node
  Home node will randomly select one delegate D, and tell the requester to get A from D. In the meanwhile, the home node will also acknowledge D about A's metadata, in case D has a fresh copy, but it is not sure about the freshness. We shall see that the home node always maintains the most updated metadata about the object.
  The requester then asks A from delegate D.

  - If there is a fresh copy at D
    D sends A to the requester directly.
  - If there is no copy at D
    This happens because D's cache is full, so object A is replaced, but the home node is not aware of this and route the request to D.
    In this case, D has to ask A from the origin server.
  - If there is a copy, but D is not sure its freshness
    D will ask for A's freshness from the origin server by sending the metadata.

    * Origin server has an updated version, it then sends the latest version to D.
    * Origin server finds there is no change between its copy and the copy on D, so it sends "Not-modified" to D.

  Once D receives a newer version of A, D knows that the directory stored at home node is useless, then it sends a message to the home node, telling it that the directory needs to update, and D and requester are the two new delegates.
  If D receives "Not-modified", it also sends a message to the home node of A to update A's metadata at home node.

In fact, even when there are already K delegates in the directory of the home node, it can still choose the requester as its new delegates by replacing

one in the directory. The advantage is to keep the directory updated, because the peer with newly cached object has higher probability to keep this object, therefore less chance to divert some request to a peer who has already replaced the object. However, if some delegate has large cache capacity, and it is known to the home node, the home node can also implement a mechanism to always keep the delegate with large cache capacity.

2.3 Comparison

Home-store approach is easier to implement with less message flying over the network, while directory approach is more complicated and requires more message transmission. However, directory approach is more collaborative than home-store. It is more dynamic to store a object in more peers rather than single peer, thus it results more flexibility in directory approach. By setting more pointers to the peers with large cache capacity, directory approach can actually use the cache space more efficiently, thus it is more collaborative.

2.4 Improvements

In the directory approach of Squirrel, the delegates selection is quite simple, as in the home node always keeps the peers which has most recently retrieved the object as its delegates. Hence if there is a very active peer, which always requires some objects, then many home nodes will make this peer as a delegate. Due to frequent query and limited cache size, there will be a lot of replacement in its cache, therefore, when some home node divert other query to this delegate, it will result cache misses. So there is a need to improve the performance by improving the delegate replacement algorithm.

There are two points to consider when the home node wants to make some peer as its delegate. Firstly, how large is the available cache size, and secondly, how recent it caches this object. Therefore, we can give an delegate replacement algorithm that is similar to LRU, by setting the weight for each peer who may store the object.

- When a delegate has large available cache size, it is assigned with higher weight than those delegates with smaller available cache size.
- The weight of a peer decreases if there is another new peer which just recently obtained this object.
- If a delegate can provide the object to some other peer who needs it, its weight will be increased.

Chapter 3

PeerOLAP

This paper, by Kalnis, Ng, Ooi etc.[2], proposed an architecture for supporting On-Line Analytical Processing(OLAP) queries, which is PeerOLAP. Each peer contains a cache with the most useful results, and they are connected through any P2P network. When a query is initiated at some peer, if it could not answer it locally, it will ask for the peers in the network. However, it is possible that a peer can only answer it partially, thus the answer can be constructed by partial result from many peers. In the case where no combination of peers can give a complete answer, the query will be forwarded to the data warehouse. This is always the last option because it is assumed that to retrieve data from data warehouse is always slow. Hence, in order to efficiently use the bandwidth, the query should be answered by the caches as much as possible.

Besides the assumption we have made just now that the bandwidth to data warehouse is always low, we also assume that only chunks at the same aggregation level as the query are considered. A chunk is defined by an aggregation of the cells in data cube over one or more dimensions. Because this paper focuses on chunk locating, caching, storing and replacing, we need to assume that the selection predicates is a subset of grouping-by predicates. This is due to the nature of aggregating cells to chunks, when cells are aggregated to chunks, the information on the aggregated dimensions is lost, thus if selection predicated is applied on these dimensions, the chunks become useless.

PeerOLAP is build on BestPeer and also extends it for OLAP application. Therefore, PeerOLAP is fully distributed except for a few servers to maintain the global name lookup, and it can reconfigure itself on-the-fly with the hope that the new configuration can reduce the workload of PeerOLAP network.

3.1 PeerOLAP Network

The goal of PeerOLAP is to act as a combined virtual cache for the purpose of answering the query at low cost. Each peer of the PeerOLAP network has a local cache and publishes its cache contents and its computational power. Besides the connection between peers, there are also data warehouses and LIGLO, which is location independent global name lookup servers. Its function is to uniquely recognize nodes whose IP address may change because of entering or leaving from the PeerOLAP network.

The searching is carried in a propagation manner with depth restricted in order to avoid flooding the network with messages. Each message keeps the path that it has visited for breaking message loops. If any peer has any chunks that the requester needs, it will return the chunks with a benefit value for these chunks. A time-out mechanism is needed for the requester to decide when to request chunk from data warehouse.

After the requester has received all chunks with their benefit value, it will decide which chunks to keep in its local cache. The chunks sent from data warehouse will be assigned with a high benefit value, and will be tried to keep locally or among nearer neighbor.

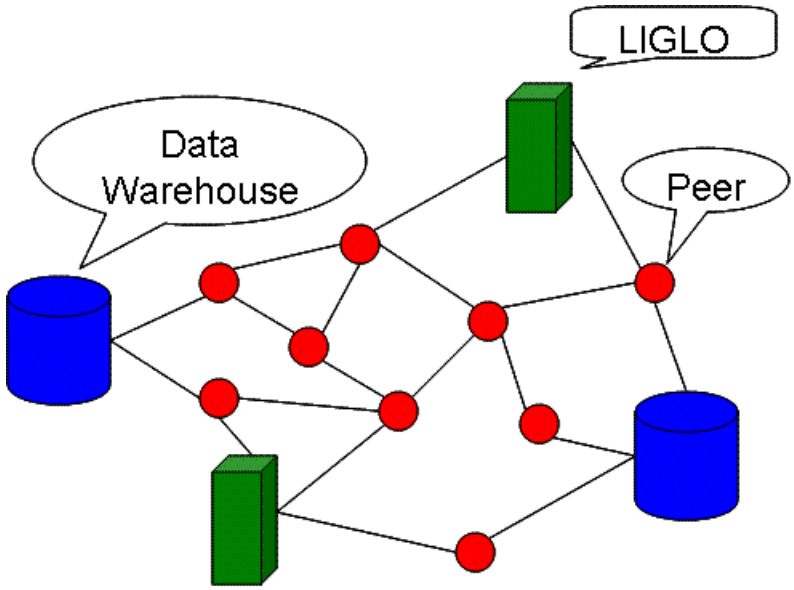

Figure 3.1: PeerOLAP Network Example

LIGLO servers are in charge of maintaining the list of active peers while peers can join or disconnect dynamically. The information about their connections, bandwidth and neighborhood is also stored in LIGLO.

An advantage of PeerOLAP network is that the neighborhood of a peer is dynamic. The number of neighbors a peer can have is limited to network parameter, every peer evaluates his neighbor's benefit, and neighbor with low benefit value will be dropped if the peer wants to add a new neighbor and the neighborhood is at limit.

3.2 PeerOLAP Architecture

Like most caching architecture, the cache control layer is designed between application layer and P2P layer. Once the application get the query agent from data warehouse server, all the data requests must go through cache layer. In local cache, every piece of data is a chunk, identified by their unique ID. Each peer can support simultaneous access to multiple warehouses and store some chunks that do not belong to any warehouses it connects to. By storing such chunks, it can benefit some of his neighbors.

3.3 Query Processing

3.3.1 Cost Model

Let $Cn(P \rightarrow Q)$ be the cost of establishing a connection between the two peers, k be the number of chunks that will be transferred together in a batch operation, $size(c)$ be the size of chunk c in tuples and $Tr(Q \rightarrow P)$ be the transfer rate from Q to P, then the network cost N for transferring chunk c from Q to P is

$$N(c, Q \rightarrow P) = \frac{Cn(P \rightarrow Q)}{k} + \frac{size(c)}{Tr(Q \rightarrow P)}$$

If $S(c, Q)$ denotes the cost of computing c at peer Q, then the total cost T of answering c at P by using data from Q is

$$T(c, Q \rightarrow P) = S(c, Q) + N(c, Q \rightarrow P)$$

3.3.2 Eager Query Processing(EQP)

As described in earlier section, when a query is initiated at some peer P, the query is decomposed into chunks C_{all}. P first checks its local cache, denote C_{local} be the set of chunks can be found in local cache and C_{miss} be the remaining chunks. Then it starts propagating the request for the entire C_{miss} to his neighbors $Q_1, Q_2, \ldots, Q_k$ with depth restricted. If Q_i has

a subset of C_{miss}, which means it has a partial answer, it then estimates the cost $T(c_j, Q_i \rightarrow P)$ for each chunk c_j, and sends to P.

After P has received all response from Q_i for some time t, denote the set of chunks which can be found in the PeerOLAP network be C_{peer}. P constructs the query plan in a greedy manner. A chunk c_i is randomly selected, and find its corresponding peer Q_i with lowest cost. Next, a chunk c_j is selected. The peer which gives lowest cost for evaluating c_j may not be Q_i. However, if Q_i can also provide c_j, the cost can be saved by sharing connection cost among c_i and c_j, in this case, we shall take the minimum between $T(\{c_i, c_j\}, Q_i)$ and $T(c_i, Q_i) + T(c_j, Q_j)$.

Once query execution plan is ready, P can request chunks from other peers. However, some of the chunks might not be available any longer since the peers could evict them out in order to store new chunks. So the actual set of chunks that P can get from PeerOLAP network is $C_{peer} - C_{evicted}$. Hence the set of chunks that P needs to request from data warehouse is $C_{miss} - (C_{peer} - C_{evicted})$.

3.3.3 Lazy Query Processing(LQP)

LQP is similar to EQP except for the propagation step. For P's neighbors $Q_1, Q_2, \ldots, Q_k$, instead of propagating the request to every neighbor, they propagate to its most beneficial neighbor. In addition, Q_i will remove those chunks it can answer from the request before propagating it. Let the maximum number of hops be h_{max}, which is the restricted depth in propagation, the number of messages will reduce from EQP's $O(k^{h_{max}})$ to $O(k \cdot h_{max})$.

3.4 Caching Policy

The caching policy used in PeerOLAP is called Least Benefit First(LBF). LBF is an LRU-like algorithm, which replaces pages with least weight. The benefit of a chunk c in a peer P is defined as

$$B(c, P) = \frac{T(c, Q \rightarrow P) + a \cdot H(P \rightarrow Q)}{size(c)}$$

where $H(P \rightarrow Q)$ is the number of hops from peer P to Q, and a is a constant representing the overhead of sending one message. Intuitively, the higher the value H is, the more difficult to locate the chunk, therefore, it is beneficial to keep it. The benefit value is normalized by dividing the total cost to obtain a chunk by the its size.

The algorithm can be summarized as follows:

- Once a chunk is reused, its weight is set to the original benefit value.
- The weight decreases every time when there is a new chunk coming in, and it drops by the benefit value of the new comer.
- Sort the chunks when there is a new chunk entering the cache, and this takes $O(\log n)$ at most, where n is the number of chunks in the cache.
- Those chunks with smaller weight are replaced in ascending order of weight until there is enough space for the new chunk.

LBF describes the local cache behavior of each peer, there are three variations of global behaviors which enforce progressively higher degree of collaboration.

1. **Isolated Caching Policy(ICP)**

   With ICP, every peer is completely autonomous. The peer P publishes its cache contents, however it does not count the hits on its cache by other peers. Therefore, if a chunk c is used by another peer, its weight will not be restored to the original benefit value.

2. **Hit Aware Caching Policy(HACP)**

   HACP is opposite to ICP, whenever a chunk c is used by another peer, its weight will be restored to the original benefit value. Its aim is to minimize the totaly query cost.

3. **Voluntary Caching**

   The voluntary caching searches for the peers whose cache is under utilized in the neighborhood of a peer with heavy workload, and transfer some of the valuable chunks from this peer to such peers, thus any query whose partial answer can be made from these chunks, the peer then divert the query to the peer that it transferred the chunks to, to give a better overall performance to answer the query.

On the issue of collaboration, ICP is the worst in the sense that a peer does not care about whether other peers need some of the chunks, and will remove them as the peer does not need them any longer. In contrast to ICP, HACP is more collaborative, this policy takes care of the local usage and global usage. For a chunk in a peer's local cache, as long as some other peer uses this chunk, the peer has to keep it. In the extreme case, assume there is a chunk in a peer which is always used by other peers but never used by

itself, then eventually, the chunk will be removed from the cache under ICP; but it will never be removed if HACP is used.

The voluntary caching is the most collaborative among them. Its goal is to make full use of the caches from all peers, and trying to keep all chunks in the distributed cache as many as possible to improve the performance. In addition, the voluntary caching policy may work either in conjunction with ICP (v-ICP) or with HACP (v-HACP).

3.5 Network Reorganization

Intuitively, peers with similar query pattern should be grouped together. This is because if they are within one another's neighborhood, there is a high probability to obtain missing chunks directly from its neighbors, and hopefully with a lower cost. Therefore, to optimize PeerOLAP performance, a virtual neighborhood should be created in order to group peers with similar query patterns. This is done by assigning a benefit value to each connection, a peer then selects its most beneficial connections as its virtual neighbors.

Chapter 4

A Peer-to-peer Framework for Caching Range Queries

Most P2P attempts have been restricted to exact match lookups and therefore are only suitable for file-based or object-based applications. In [4], O. D. Sahin et al. tried to build an P2P database that supports caching of range query results, based on the structured P2P system called CAN.

One question that may arise is why we need caching. Caching is beneficial due to the following reasons. Firstly, the data source may be too far away from the querying node which leads to inefficient query processing. Secondly, the data source will be heavily loaded with all the queries from the system. Thirdly, the system will not be fault-tolerant because the data source is a single point of failure. Thus, by caching the results of previous queries at some nodes in the system, we hope that new queries can be answered without contacting the data source.

4.1 Problem Formulation

The system primarily aims to answer range queries. A typical range query would be *SELECT Student.Name WHERE 20<Student.Age<30* in SQL-like syntax. One or more peers in the system are used to store the result tuples of previously asked queries. Later, if a new query q_{new} whose range is subsumed by a previous query q_{old}, the querying peer can find out the answers by contacting the peers who store the result tuples of q_{old} in their local cache. A more formal formulation of the problem can be stated as follows:

Problem Given a relation R, and a range attribute A, we assume that the results of prior range-selection queries of the form *R.A(LOW, HIGH)* are stored at the peers. When a query is issued at a peer which requires the

retrieval of tuples from R in the range *R.A(low, high)* with low$\geqslant$LOW and high$\leqslant$HIGH, we want to locate the peer in the system which already stores tuples that can be accessed to compute the answer.

Distributed Hash Table is used to support the range lookup. One nice property of DHT is that the only knowledge that peers need is the function used for hashing. Once this function is known, given a lookup request, the peers only need to hash the range locally and use the hashed value to route the request to a peer that is likely to contain the answer. This is the reason why the system is built on CAN. However, the hashing function is tailored to support non-exact match range queries, i.e., we are also interested in the results falling in the ranges that are a superset of the query range.

4.2 System Model

The system uses a 2d virtual space in a manner similar to CAN. Assume the domain of a 1-dimensional attribute is [a,b], the corresponding virtual hash space is a 2-dimensional square with the coordinates of the 4 corners being (a,a), (b,a), (b,b) and (a,b). The virtual hash space is further partitioned into smaller rectangles, each of which is called a *zone*. The union of these zones cover the whole virtual hash space and none of the two overlap. Each zone is uniquely identified by the bottom left and upper right corner coordinates. Figure 4.1 shows an example of a possible partitioning of a virtual hash space for a range attribute whose domain is [10,70].

Unlike CAN, not all peers participate in the partitioning. Each participating peer owns a zone. These peers are called *active* peers. Those that do not participate are called *passive* peers. Each passive peer has to register with one active peer. All active nodes have a list of passive nodes registered with them.

Each active node also keeps a routing table with the IP addresses and zone coordinates of its neighbors, which are owners of adjacent zones.

Given a range query with range $[q_{low},q_{high}]$, it is hashed into point (q_{low},q_{high}) in the virtual hash space. This is referred to as the *target point* of the query. The zone where the target point belongs to is called the *target zone*, and the node which owns the target zone is called the *target node*. Once the answer to this range query is obtained, the querying peer will cache the answer locally if it has enough space and is willing to share the answer. The target node will keep a pointer to the querying node. The target node will also cache the answer if it has enough space. In either case, we say that the target node stores the result.

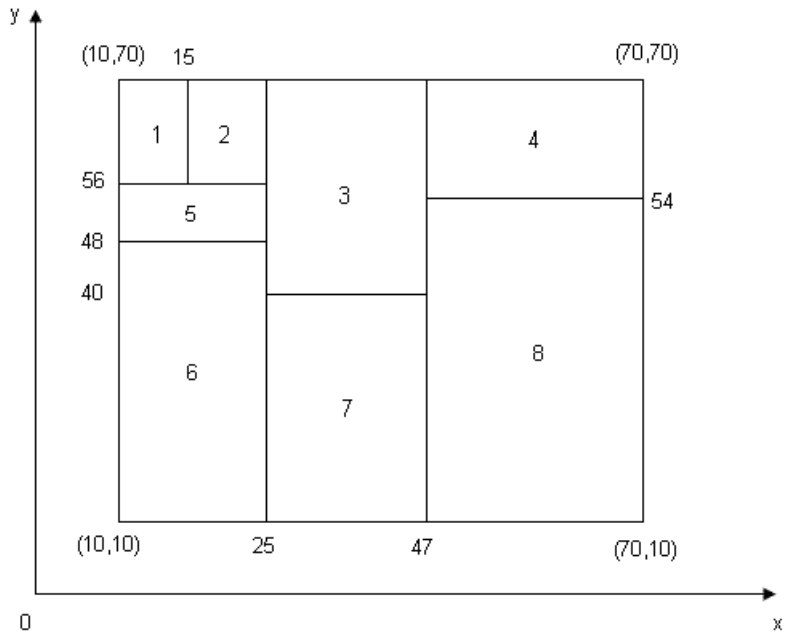

Figure 4.1: Partitioning of a virtual hash space

4.3 Zone Maintenance

How the zones are split and assigned are the core part of the system. Initially, the whole space is a zone and the node responsible for the zone is the data source. When a zone split occurs, owner of the zone contacts its own list of passive peers or its neighbors' lists of passive peers to assign one of the new zone to one of the passive peers.

Zone splits occur is due to either of the following two reasons.

Heavy Answering Load Too many queries are directed to the peer for answers. In this case, the peer finds the *x-median* and the *y-median* of the stored results to determine if a split at the *x-median* or *y-median*will lead to an even distribution of the stored results as well as zone space.

Heavy Routing Load Too many queries pass by the peer when they are routed in the system. In this case, the zone split is along the mid-point of the longer side of the zone.

After the split, a passive peer is assigned the new zone (the bottom/right half). The neighborhood relationships are updated accordingly. Figure 4.2 shows the partitioned zones after zone-6 in Figure 4.1 splits along the y-axis.

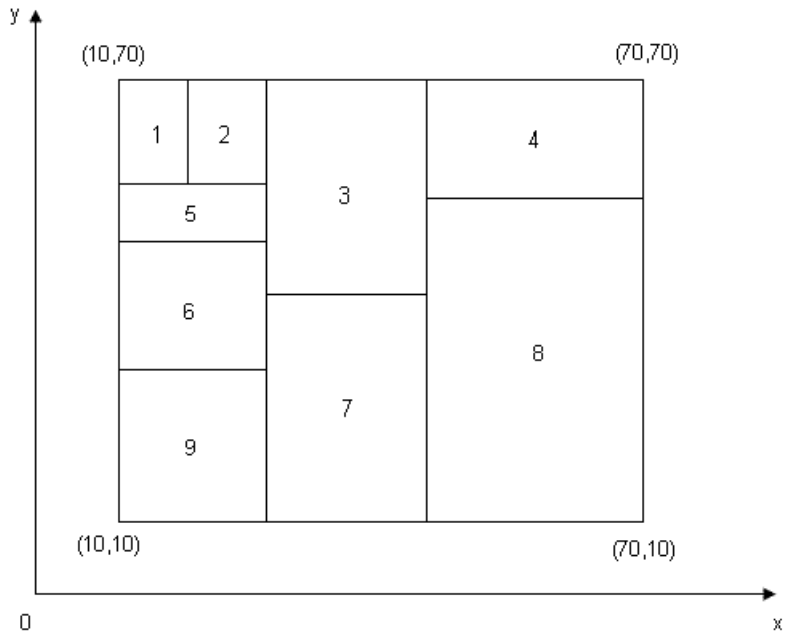

Figure 4.2: Partitioning of the virtual hash space after zone-6 of Figure 4.1 splits

4.4 Query Routing

Every peer in the system can initiate a range query. In the virtual hash space, if the peer is active, the query starts routing from the zone that it is responsible for; if the peer is passive, it sends the query to any of the active peers and the routing starts from there.

Upon receiving a query, the peer first checks if the query range maps to its zone. If so, the zone will be returned to the querying peer. Otherwise, the peer will compare the range with all of its neighbors and route the query to the neighbor whose coordinates are the closest to the target point. Figure 4.3 shows an example of query routing, as the solid arrows indicate. The analysis shows that in an equally partitioned hash space, the average routing length of a range query is $O(\sqrt{n})$, where n is the number of zones in the system.

4.5 Forwarding

When the query is routed to the target zone, the target node checks if it knows the results whose range contains the query range. If such results exist, the querying peer can get the results either directly from the target

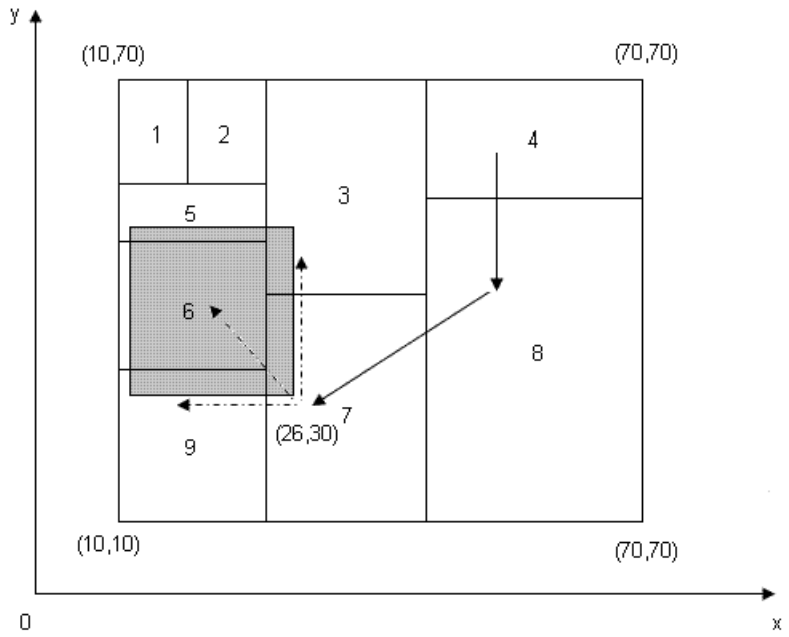

Figure 4.3: Routing and forwarding in virtual space. The shaded region shows the *Acceptable Region* for the query

node (if it cache the results locally) or contact the node who has the results (if the target node has a pointer to the node who has the results). If the target node does not know the results whose range subsume the query range, there is still a chance that the results can be found in some other zones. Hence, the query should still be forwarded to those zones.

The zones other than the target zone that may have the answers are those located on the upper left area of the target point. Consider a range query whose range is [26,30]. It is mapped to point (26,30) in the virtual hash space in Figure 4.3. The shaded area is the possible place where the answer may be found. The reason is that for any point in the shaded area, its range subsumes [26,30].

Hence, when the answer to a range query is not found in the target zone, the query is forwarded to the left and top neighbors that may contain a potential result. Those nodes also check their local results and can forward the query to their top and left neighbors recursively if need be. In Figure 4.3, if no results can be found in zone-7, the query is forwarded to zone-6, zone-9 and zone-3 as the arrows indicate.

To restrict the length of the super range, a parameter called *acceptableFit* is used. It is a controlling parameter that specify how big an answer range is acceptable for a given range, and therefore also determine how far the

forwarding can go. It defines an allowed offset for the query range s.t. *offset* = *acceptableFit*×|*domain*|, where |*domain*| is the length of the domain of the range attribute. Hence, for a range [low, high], the acceptable cached results must fall in the range of [low-offset, high+offset]. The square defined by these offsets and the target point is referred to as the *Acceptable Region*. When a node receives a forwarded query, it checks if it has local results whose ranges is within the allowed offset. When *acceptableFit* is set to 0, no results from super range is acceptable to the query. If *acceptableFit* is set to 1, all results whose range mapped to the top left area of the target point is acceptable. There are two strategies for query forwarding.

4.5.1 Flood Forwarding

This is a naive approach of query forwarding. Every peer receives the forwarded query will check if it has the qualifying results. If not, it will forward the query to the top and left neighbors if they fall in the permitted acceptable region.

4.5.2 Directed Forwarding

Out of all the neighbors in the upper left region of the zone, the peer picks the neighbor whose zone overlaps largest with the *Acceptable Region*. A query can set a limit d on the directed forwarding. Whenever a query is forwarded, the limit d is decremented. When d reaches 0 and there is still no results found at the peers, the querying node is notified to query the data source directly. In this way, the querying peer can get a control of the maximum number of hops during forwarding.

4.6 Improvements

There are several improvements over the system, as suggested in the paper.

4.6.1 Lookup During Routing

During query routing, the query is passed from the querying zone to the target zone. Along the way, the query visits many nodes in the system. These nodes may have asked queries whose ranges subsume the query range previously and have cached the results in their local cache. Hence, during routing, it is worth checking if the visited nodes have the results along the way from the querying zone to the target zone.

4.6.2 Issuing Warmup Queries

When a passive node is assigned a zone, it can compute and cache the result of the query whose range is mapped to the upper left corner of its zone in order to warm up its cache. In this way, further queries mapped to this zone will always be answered locally without forwarding because the range of the warmup query contains all ranges mapped to this zone.

4.6.3 Supporting Exact Match

Exact match queries is supported by the system too because all we need to do is to set both the lower bound and upper bound of the range to be the same number. For example, to answer *SELECT XXX WHERE Student.Age=25*, we can issue a range query [25, 25] on attribute *Age*.

4.6.4 Updates

To ensure cache consistency, when a tuple t with attribute A=k is updated, all nodes whose zones lie on the upper left area of point (k,k) needs to update their cache. This is done by sending an update message to the target zone of (k,k). The message will also be forwarded to the upper left neighbors too. Upon receiving the message, the nodes responsible for the zones do the update accordingly.

4.6.5 Handling Multi-attribute Range Queries

The system can be easily extended to support range queries of multiple attributes. The virtual hash space will be a 2d dimensional hypercube, where d is the number of attributes of the relation. A range query over the d attributes can be written as $[l_1, h_1], [l_2, h_2], ..., [l_d, h_d]$. It is mapped to the point $(l_1, h_1, l_2, h_2, ..., l_d, h_d)$ in the hypercube. The first two dimensions corresponds to the first attribute, and the second two dimensions corresponds to the second attribute and so on. If the result is not found in the target zone containing the point, the query can be forwarded by moving towards the upper left of the 2d-dimensional space, which corresponds to increasing the coordinates of the even dimensions and decreasing the coordinates of the odd dimensions.

Chapter 5

Range Addressable Network: a P2P Cache Architecture for Data Ranges

In [3], A. Kothari et al. approached the problem of caching range query results using a different strategy. They developed a network topology called Range Addressable DAG and mapped every active node in the P2P system to a group of nodes in the DAG. Each node is responsible for caching results and answering queries falling into a specific range.

5.1 Range Addressable Network Topology

The basic caching and retrieval policies are similar to [4] in that the nodes responsible for a particular range need to store the results in their cache or a pointer to other nodes that know the results; and the range query is directed through neighboring nodes from the querying node to the destination node. In the case that no peer knows the answers, the query is directed to the source(s).

The tuples stored in the peers are labeled 1,2,...,N. A range [a,b] is a continuous subset of 1,2,...,N with $0 \leqslant a \leqslant b \leqslant N$. Given a query of range [a,b], peers cooperate to find the shortest superset of [a,b], if there is one. This is achieved by firstly map the universe 1,2,...,N to a Direct Acyclic Graph, and secondly map different part of the DAG to nodes in the system.

5.1.1 Range Addressable DAG

Given the entire universal set 1,2,...,N, it is mapped into the root node of the DAG, and the node will be recursively divided into 3 overlapping sub-intervals. Figure 5.1 illustrates the idea of constructing a DAG for the range of [1,8]. Each interval corresponding to a node in the DAG is divided into 3 equal-length overlapping sub-intervals, each of which corresponds to a child node. This recursive partitioning continues until each interval has length 2, in which case we create two leaf nodes. It is a DAG rather than a tree because a node may have 2 parents due to the overlapping partitioning.

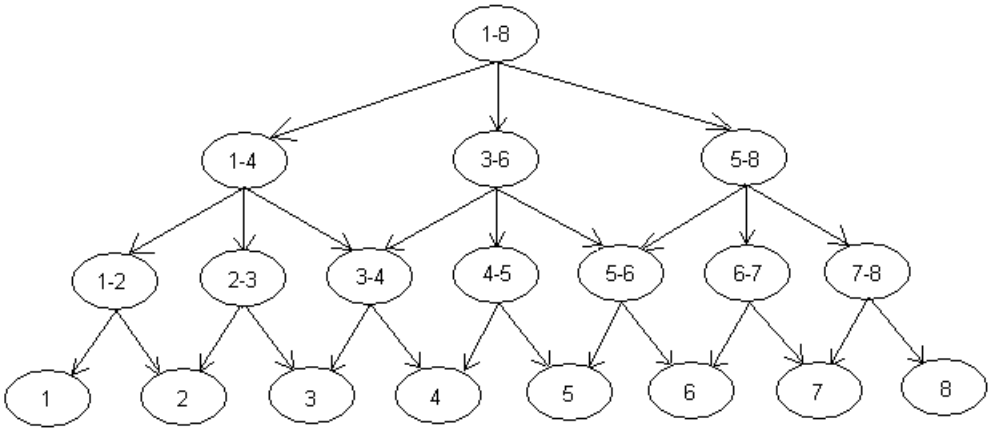

Figure 5.1: Range Addressable DAG

A range r=[a,b] is associated to a unique DAG node v_r whose interval $i(v_r)$ contains r and none of the child-intervals of v_r contains r. Node v_r is called the *topology node* for r. The search for a query range r=[a,b] has 3 cases to consider. Initially, the boolean value *down* is set to true.

1. If $r \not\subseteq i(v)$, then the search moves up to one of the ancestors of v whose interval overlaps q;

2. If $r \subseteq i(w)$, for some child w of v, and *down* is true, then the search moves to w.

3. If some range stored at v is a superset of r, then we report the shortest range containing r that is stored at either v or a parent of v, and stop. Otherwise, we set *down* to false, and the search moves to one or both parents of v whose intervals overlap with r.

A range of length L is stored at a node whose interval length is close to L. In particular, if a range of length L is stored at node v, then $\frac{i(v)}{4} < L \leqslant i(v)$.

This is true by observing that each interval is divided into 3 overlapping sub-intervals of equal length. If L$<\frac{i(v)}{4}$, the range would be stored in the child node of v rather than v. This also guarantees that by search both v (the lowest node whose range contains the query range) and the parent node of v, we can find the shortest superset matching the query, if it exists. This is because any node w that is at least two levels above v will have i(w)$\geqslant 2^2$i(v), and for any range [a,b] stored at w, $|b-a|$ is strictly greater than $\frac{i(w)}{4}\geqslant$i(v).

Another important property of the range addressable DAG is that, given a range selection query, one can find the shortest superset by searching O(logn) nodes in the worst case.

5.2 The Peer Protocol

Peer protocol concerns how the DAG is mapped physically into the peers in the system. Peer protocol has two components: *peer management* and *range management*. The peer management component handles the joining, leaving and failure of a peer in the system. The range management component handles how the underlying database ranges are mapped to the current set of peers in the system. It also defines the routing protocol used by peers to perform the query lookup.

5.2.1 Peer Management

The peer management component handles the joining, leaving and failure of peers. It ensures that at any point of time, every node in the DAG are assigned to some peer. The *zone* of a peer is the nodes that are assigned to it. A peer's zone is always a connected graph in the DAG and the union of all peers' zones is the entire DAG.

Initially, the entire DAG is assigned to one peer. Afterwards, as new peers join the system, they request for part of the DAG from the peer(s) already in the system. We say that two peers are neighbors if there is a parent-child relationship among any of the nodes in their respective zones. Conventionally, we define the node to be in the zone same as its left parent.

Previously, we say that the for a DAG having N leaves, the range lookup operation takes O(log$N$) time, which is undesirable. Now, we assign nodes to peers, if two nodes of the DAG belongs to the same peer, then no query forwarding is needed. Hence, the time for range lookup operation should be a function of the number of peers, n, rather than the number of leaves. In the zone splitting and assignment process, the system will try its best to maintain a *balanced* division. Consider a *collapsed* DAG, where we collapse

each peer's zone into a single node. We call the system to be balanced if the range addressable DAG is divided among the n peers in such a way that the corresponding collapsed DAG has a height of O($\log n$). If this holds, then the range lookup time is O($\log n$) too where n the number of peers in the system.

Join Request

When a new peer joins the system and request one of the old peer for a zone, the old peer will assign one of the 3 zones rooted at its child nodes to the new peer. The old peer now becomes the parent neighbor of the new peer. Figure 5.2 gives an example of this zone partition, where the peer responsible for the root is the parent neighbor of the peers responsible for the 3 child zones.

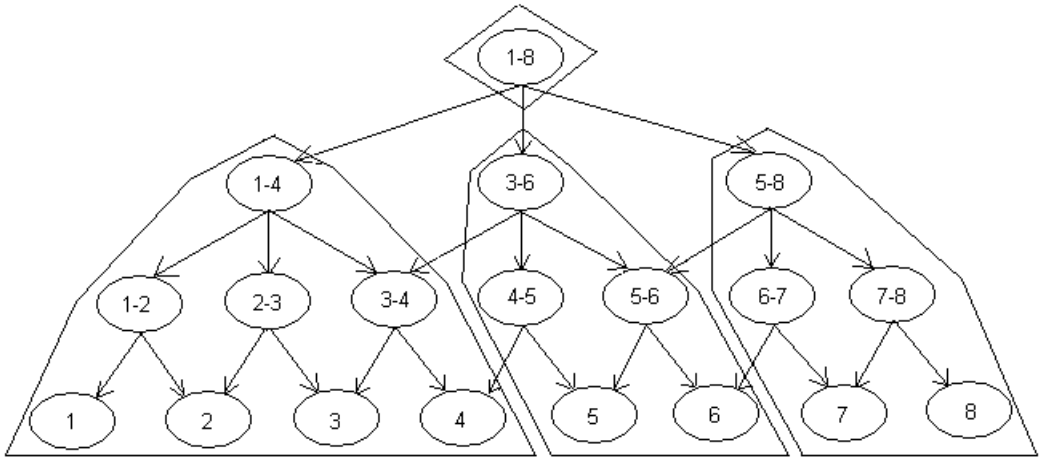

Figure 5.2: Zone Partition of DAG

Leave Request

When a peer leaves, its zone is merged with its parent neighbor or child neighbor's zone. In order to balance the zone sizes, the leaving peer's zone is merged with the neighboring zone whose size is the smallest. Figure 5.3 illustrates how the zone merge is performed for the zones in Figure 5.2, when the peer responsible for the middle child zone of the root leaves.

Peer Failure

To maintain the connectivity of the peers in the case of peer failure, it is not enough for a peer to maintain the information about only its parent

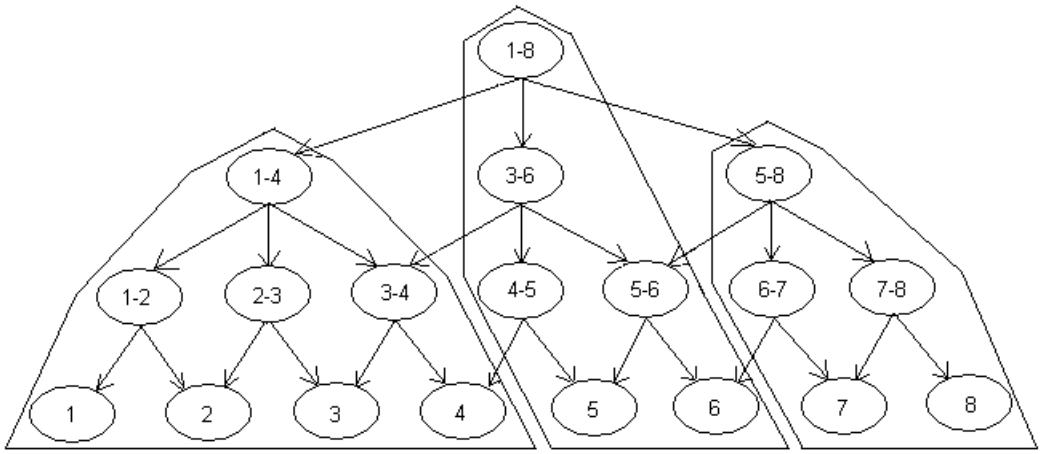

Figure 5.3: Zone Partition After A Peer Leave Request

and child nodes. The scheme is modified in the way that a peer also has the information of all its ancestor nodes. During range lookup, when a peer finds out that its parent has failed, it sends a *zone takeover* request to its first alive ancestor. The ancestor checks if some other peer has already taken over the zone or part of it. If not, the requesting peer is allowed to take over the zone, the requesting peer's ancestor list is updated and the process is repeated, where the peer talks to its new ancestor to take over the remaining part of the zone. For example, in Figure 5.4, during query routing, peer 3 noticed that the peer responsible for its left parent had failed, it contacted its nearest alive ancestor peer 2 for zone takeover. Peer 2 checked that no other peer has requested for node 2-3, so it granted the request. Peer 3 was then responsible for node 2-3 and node 3. Later, peer 4 also noticed that the peer responsible for node 4-5 and node 3-6 had failed, it contacted the nearest alive ancestor peer 1 for zone takeover. Peer 1 granted the request because no one had asked for the two nodes before. Now, during another query routing, peer 3 again notices that its right parent, and right ancestor has failed. Peer 3 contacts peer 1. Peer 1 knows that in fact peer 4 is the new nearest ancestor of peer 3, so peer 1 asks peer 3 to contact peer 4 instead. Peer 3 finally will take control of node 3-4 after contacting peer 4.

5.3 Range Management

The range management component is responsible for mapping ranges to peers and updating tuples in the peers. Recall that the range query results are stored at the topology node of the DAG. Hence, the peer whose zone contains

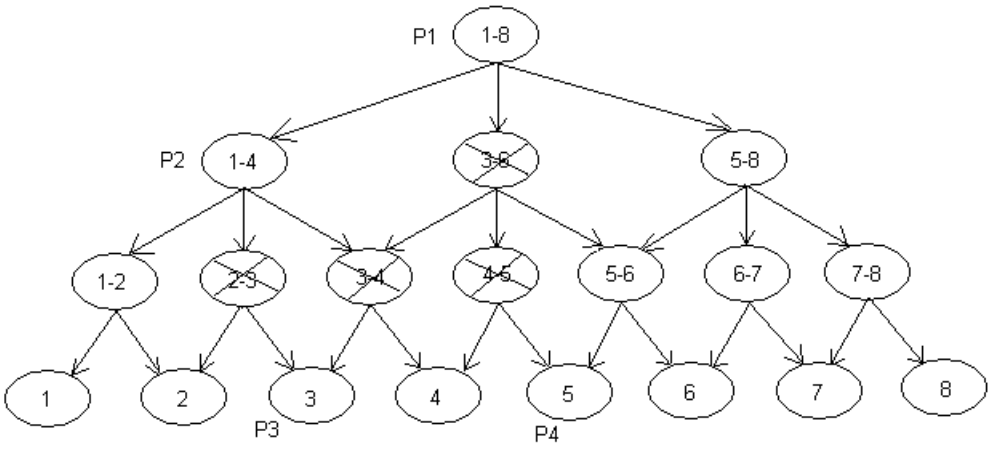

Figure 5.4: Peer Fail Event

the topology node is responsible for caching the results.

Range lookup algorithm is similar to the one introduced previously. Note that the query now needs to be forwarded to a neighboring peer only when the traversal of the DAG crosses zone boundaries.

When a tuple in the database is updated, we first locate the topology node of the tuple. The peer whose zone contains the topology node needs to update the tuple in its cache. It then contact all its ancestors for the update too.

5.4 Improvements

Two improvements are discussed in the paper. *Cross Pointers* is used to provide shortcuts during query routing; and *Peer Sampling* tries to maintain a balanced collapsed DAG by finding peers with large zones to split.

5.4.1 Cross Pointers

In the worst case, a query need to traverse from one leaf node of the DAG to the root and then goes down again to another leaf node. This can be avoided by adding *cross pointers* among the same level of nodes. When cross pointers are present, queries can be routed faster without the need to go through the hierarchical route.

If a node is the left child of its parent, then it keeps cross pointers to all the left child nodes of the nodes that are in its parent's level. Similarly,

middle children keeps cross pointers to all the middle children of nodes that are in their parents' level. Note that a cross pointer needs to be stored at a peer only if it points to a topology node in other peer's zone.

Figure 5.5 shows an example DAG with cross pointers.

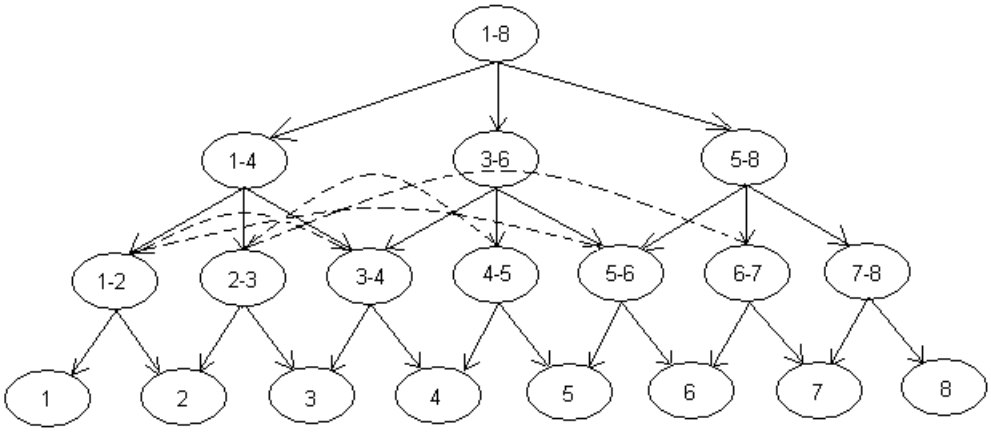

Figure 5.5: DAG with Cross Pointers

5.4.2 Peer Sampling

To maintain a balanced collapsed DAG, we need to ensure that when a new peer joins, it knows to which old peer it should send the zone request. However, this is hard to achieve since we do not have a centralized server to keep the information of all existing peers. Hence, the new peer will try to randomly poll k old peers in the system and choose the one whose zone is rooted closest to the root to send the zone request.

Chapter 6

Conclusion

In this report, we have surveyed four papers on different perspectives of caching techniques.

Squirrel is a decentralized, P2P web cache system. It makes use of caches in web browsers on desktop machines to form an efficient and scalable distributed web cache, without the need for dedicated hardware and the associated administrative cost. PeerOLAP is an architecture for supporting On-Line Analytical Processing queries. An OLAP query can be answered by aggregation of partial answers from many different peers; PeerOLAP also gives algorithms for cache management, like chunk replacement and network reorganization. Range query results can be cached in peers too. To direct a range query to the peers that may have cached the previous range query results, two techniques are proposed. One is based on CAN. An attribute interval is mapped into a 2D Euclidean space. A query range [low, high] is mapped to a point with coordinates (low, high) in the 2D space. Resolving a range query involves query routing and forwarding. The other technique based on range addressable DAG maps a range into a node in the DAG and recursively divide the node into 3 nodes, each of which corresponds to a subinterval. Searching for a range involves traversing up and down the DAG. To maintain data consistency, updates are handled pretty much as searching for the target peers and updating the tuples in the peers.

Bibliography

[1] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache, 2002.

[2] Panos Kalnis, Wee Siong Ng, Beng Chin Ooi, Dimitris Papadias, and Kian-Lee Tan. An adaptive peer-to-peer network for distributed caching of OLAP results. In *SIGMOD Conference*, 2002.

[3] Anshul Kothari, Divyakant Agrawal, Abhishek Gupta, and Subhash Suri. Range addressable network: A p2p cache architecture for data ranges, 2003.

[4] Ozgur D. Sahin, Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. A peer-to-peer framework for caching range queries, 2004.