Partial Lookup Services

Qixiang Sun[1] Hector Garcia-Molina[2]

Computer Science Department
Stanford University, Stanford, CA 94305, USA
{qsun, hector}@cs.stanford.edu

Abstract

Lookup services are used in many Internet applications to translate a key (e.g., a file name) into an associated set of entries (e.g., the location of file copies). The key lookups can often be satisfied by returning just a few entries instead of the entire set. However, current implementations of lookup services do not take advantage of this usage pattern. In this paper, we formalize the notion of a partial lookup service *that explicitly supports returning a subset of the entries per lookup. We present four schemes for building a partial lookup service, and propose various metrics for evaluating the schemes. We show that a partial lookup service may have significant advantages over conventional ones in terms of space usage, fairness, fault tolerance, and other factors.*

1 Introduction

A *lookup service* translates a *key*, e.g., the name of a file, into an associated set of *entries*, e.g., the locations for the file. Lookup services are used by many Internet applications. The most well known example is the Domain Name Service [2] that translates machine names into IP addresses. More recently in file sharing applications (e.g. Gnutella [1], names of the files are translated into the IDs of computers storing those files. In the examples we have given, the entries point to the desired resources, but in other cases, the entries themselves may be the desired resources. In the latter case, the entries could be large.

In many cases, users do not need *all* the entries associated with a key, but only "a few." For example, a user looking for a popular song may only need two or three sites to contact for the song, not the hundreds of sites that may have a copy. In this paper we study how to exploit this observation to build lookup services that are much more "efficient" than traditional services where all entries are returned.

[1]Supported by NSF Graduate Fellowship
[2]Partially supported by NSF IIS-9817799

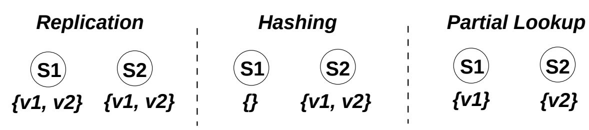

Figure 1. Three ways of managing a key with two entries v_1 and v_2.

To illustrate, consider Figure 1, where two servers, S_1 and S_2 implement a lookup service. Let us focus on a particular key k that maps to two entries $\{v_1, v_2\}$. With a traditional full-replication approach (left, Figure 1), the mapping of k to $\{v_1, v_2\}$ is stored on both servers. With a traditional hashing approach (center), key k is hashed to one server, say S_2, which stores the full mapping. With a partial lookup approach (right), servers need not keep the full mapping. In our example, S_1 stores the k to $\{v_1\}$ mapping, while S_2 stores the k to $\{v_2\}$ mapping. In storing partial mappings at the servers, we permit clients to contact either server, if they are only looking for one of the two entries, thus avoiding the overhead of full replication.

As we will study in this paper, the partial lookup strategy has some important advantages over the traditional approaches. Specifically, storage needs and update costs are reduced as compared to full-replication. Despite the large disk space available in modern servers, reducing storage cost is still critical because applications prefer to fit all the lookup data in physical memory, which is limited. Moreover, performance of modern servers is usually limited by the handling of network interrupts (i.e., remote messages and requests) in the operating system rather than processing power. Thus reducing update cost among the servers is strongly desirable. Partial lookups also provide better load balancing than traditional hashing. For instance, if k is very popular (i.e., a hot-spot), with hashing in Figure 1 (center), S_2 can be overloaded while with partial lookup in Figure 1 (right), the load is distributed. Furthermore, even if S_2 is down, partial lookups can still continue.

The basic idea of allowing a server to track fewer entries is simple, and has been used in a few systems (most notably the now-defunct Napster [4]). Yet, surprisingly, there are quite a few ways to implement the idea, and there are

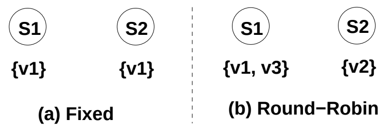

Figure 2. Two strategies of placing 3 entries on 2 servers, exploiting partial lookup.

important tradeoffs to consider. As far as we know, these tradeoffs have not been carefully evaluated, even by people implementing partial lookups. In this paper, we provide such a detailed evaluation of partial lookup schemes. As a preview, let us illustrate two of the options and the issues that arise. Let us return to our two server example, and consider the two schemes illustrated in Figure 2, for mapping a key k to entries $\{v_1, v_2, v_3\}$. With option (a), we place the same entry v_1 at both servers. In option (b), we use a round-robin scheme to assign individual entries to each server. Both options will return at least one entry to the user on a lookup. However, option (a) always gives out v_1 while option (b) can give out different answers depending on which server is contacted. Thus, option (b) provides more varied results, and is "fairer" because the resources represented by v_1, v_2 and v_3 are more evenly accessed. On the other hand, option (a) does not have any update cost if v_2 is deleted, while option (b) has to shuffle entries around to maintain the round-robin assignment. Option (a) also uses less storage space than (b). Other tradeoff include how resilient the service is to server failures, and the cost of processing each user lookup request.

In this paper, we make the following contributions:

- We formally define the partial lookup problem (Section 2).
- We present four fundamental schemes for partial lookups, first in a static scenario with no updates (Section 3) and then in a dynamic scenario (Section 5).
- We define metrics for evaluating partial lookup schemes (Section 4).
- We evaluate our four partial lookup schemes, in some cases via simulations, in both the static and dynamic scenarios (Sections 4, 6). We show that there can be significant performance, fairness, and scalability differences among the schemes.
- We provide guidelines or "rules of thumb" for selecting schemes.

2 Service Definition and Assumptions

A lookup service manages keys and their associated entries and allows lookup operations. In a traditional lookup service, each lookup returns all entries for a given key. Formally, we define a traditional lookup service as a service that maintains the set S = $\{(k_i, V_i)\}_i$, where k_i denotes a key and V_i denotes the corresponding set of entries for key k_i, and supports the following interface and semantics:

- $place(k, \{v_1, v_2, \ldots, v_h\})$: if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow \{v_1, v_2, \ldots, v_h\}$. Else, $S \leftarrow S \cup \{(k, \{v_1, v_2, \ldots, v_h\})\}$.
- $lookup(k)$: if $k = k_i$ for some $(k_i, V_i) \in S$, then return V_i. Else, return $\emptyset$.
- $add(k, v)$: if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow V_i \cup \{v\}$. Else, $S \leftarrow S \cup \{(k, \{v\})\}$.
- $delete(k, v)$: if $k = k_i$ for some $(k_i, V_i) \in S$, then $V_i \leftarrow V_i - \{v\}$.

The $place$ interface specifies a set of entries for a key in batch whereas add and $delete$ provide incremental updates. And $lookup$ returns the current set of entries for a given key.

A *partial lookup service* is then a traditional lookup service that supports $partial\_lookup(k, t)$ instead of $lookup(k)$ where

- $partial\_lookup(k, t)$: if $k = k_i$ for some $(k_i, V_i) \in S$, then return $V \subset V_i$ such that $|V| \geq t$. Else return $\emptyset$.

In other words, $partial\_lookup(k, t)$ retrieves at least t entries for a client without getting all entries for a key k. The parameter t is the *target answer size* of the partial lookup.

The above definitions involve multiple keys. However, each key can be managed separately, e.g., replicate a single-key strategy to manage more than one key at a time. In fact, different strategies can be used to manage different types of keys. For instance, frequently updated keys require strategies with small update costs, while static keys want low lookup costs and fairness in which entries are returned. Since it is easy to generalize, we focus here on strategies that manage only one key. Hence we omit the key parameter in the interface and describe each strategy in terms of $place(\{v_1, v_2, \ldots, v_h\})$, $add(v)$, $delete(v)$, and $partial\_lookup(t)$.

This paper focuses on the basic tradeoff decisions in a centralized server-farm setting. In particular, we make the following two assumptions about the clients' usage of the service. When a client C performs a $partial\_lookup(t)$, we assume that C does not care which t entries are returned in the response and that C can access all n servers equally. We discuss issues when relaxing these two assumptions in the extended version of this paper [8].

3 Strategies for Static Placement

We first describe the strategies in a static placement setting where entries are placed on the servers via $place(v_1, \ldots, v_h)$, and followed by *partial_lookup* operations only. We give informal descriptions here. More details can be found in the extended version of this paper [8].

3.1 Full Replication Strategy

The naive strategy is to store all h entries for a given key on all n servers. This traditional full replication strategy does not take advantage of the partial lookup property at all. Since every server has the same entries, a client can contact any server during a lookup operation, which spreads out the workload among the servers.

3.2 Fixed-x Strategy

An obvious "improvement" is to only store a *fixed* subset of the h entries at all the servers, e.g., the first x of the h entries. One implementation of Fixed-x is to broadcast a client's $place(v_1, \ldots, v_h)$ request to all servers, and let each server keep the entries v_1 through v_x. Similar to full replication, a client can contact any server to do a lookup.

Fixed-x obviously uses less storage space and can ignore updates (e.g. delete requests) on entries that it does not have. However, the parameter x must be sufficiently large so that no client will ever want more than x entries for a lookup, i.e., the target answer size t of a lookup is less than the parameter x for all lookups.

3.3 RandomServer-x Strategy

In Fixed-x, each server has the same subset of x entries. Thus another "improvement" is to place different subsets of x entries at the servers, i.e., let each server choose a random subset of x entries instead of the subset v_1 through v_x after receiving the $place(v_1, \ldots, v_h)$ broadcast. For lookups, a client can contact any of the servers.

RandomServer-x has an advantage over the Fixed-x strategy in that x does not have to be larger than any conceivable target answer size t. If a client wants more than x entries during a lookup, it can contact multiple servers and merge the answers. Another advantage is that clients get more variety in which entries are returned on each lookup since servers store different sets of x entries.

3.4 Round-Robin-y Strategy

Fixed-x and RandomServer-x share a common feature that each server independently decides which entries to store after the client's $place$ request is broadcast, which implies that some entries may not be stored at any servers. An alternative is to use a round-robin placement of entries that ensures each entry is assigned to some servers. Specifically, when a client does a $place(v_1, \ldots, v_h)$, the request is sent to a coordinator who then stores v_1 on servers 1 through y, v_2 on servers 2 through $y + 1$, and so on.

Because the placement is deterministic, if a client must contact multiple servers to assemble t entries for a lookup, it can choose additional servers that have the least number of entries in common. For example, if a client initially contacts server 2, then contacting server $2 + y$ will yield the most number of new entries.

Strategy	Storage Cost
Full Replication	$h \cdot n$
Fixed-x, RandomServer-x	$x \cdot n$
Round-y	$h \cdot y$
Hash-y	$h \cdot n \cdot (1 - (1 - \frac{1}{n})^y)$

Table 1. Storage cost for h entries on n servers.

3.5 Hash-y Strategy

Instead of allocating entries to servers deterministically like Round-y, Hash-y uses y hash functions $f_1, f_2, \ldots, f_y$ to assign entries to servers, i.e. the coordinator, after receiving a $place(v_1, \ldots, v_h)$ request, assigns each entry v_i to servers $f_1(v_i), f_2(v_i), \ldots, f_y(v_i)$. If two hash functions assign an entry to the same server, it is stored only once. On a lookup, a client contacts servers in a random order until retrieving enough entries.

This pseudo-random assignment can result in some servers having more entries than others. Thus while Round-y can tell, in advance, how many new entries a client can get for contacting another server, Hash-y cannot. The tradeoff for this "determinism," as we will see later, is the much smaller update cost for Hash-y.

4 Static Placement Evaluation

We suggest five metrics for evaluating the different static placement strategies described in Section 3. The first two metrics capture the operating overhead of the strategies. The last three metrics evaluate the quality of the lookup answers. We compare strategies that incur similar overhead.

4.1 Storage Cost

The first overhead cost is the total storage required across all servers. We assume all the entries have the same size and compute the *storage cost* by counting the combined number of entries stored on all servers. Low storage cost is important if we want our entries (e.g., IP address) to fit entirely in physical memory for fast access or if our entries (e.g., music file) are so large that we might not have enough disk space.

Table 1 summarizes the costs. Note that storage cost for Fixed-x and RandomServer-x grows as a function of the number of servers, which is useful if the storage medium is the physical memory that cannot be adjusted dynamically. The cost for full replication, Round-y and Hash-y, on the other hand, grows as a function of the number of entries for a key which can be unbounded.

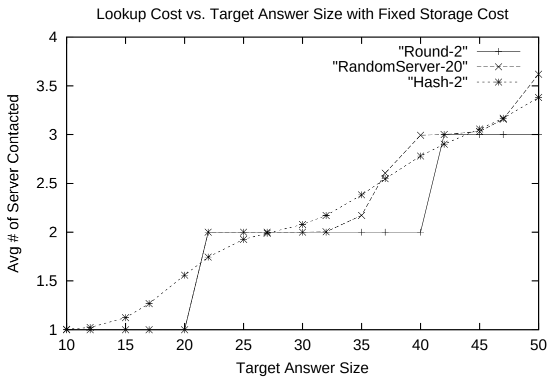

Figure 3. Lookup cost for managing 100 entries on 10 servers with a fixed storage cost of 200.

4.2 Client Lookup Cost

Another overhead cost is the client lookup cost. We define the *lookup cost* as the expected number of servers a client will contact during a lookup. We use this metric because the performance bottleneck for a modern server is usually in handling network interrupts in the operating system, i.e., the number of client requests received, rather than raw processing power. Ideally, we want the client lookup cost to be 1, i.e., a client contacts exactly one server during a lookup. However the cost could be more than 1 if a client contacts multiple servers to assemble t entries for a lookup.

To illustrate the different lookup cost for these strategies, we performed a small simulation of managing 100 entries using 10 servers while allowing each strategy to use up to 200 entries of storage space among the 10 servers. From the limit of 200 entries, we compute parameters x and y using the storage cost formula in Table 1. In this case, we get parameter x is 20 and parameter y is 2. Hence, we are comparing Fixed-20, RandomServer-20, Round-2, and Hash-2. The choice of 200 entries is simply for illustration. Other limits exhibit similar behavior.

Figure 3 shows the result of the simulation. The figure does not include Fixed-20 because it has a simple lookup cost 1 for $t \leq 20$ and cannot answer lookups with $t > 20$. For Round-2, we see a step curve behavior in the figure because each server stores 20 entries in a deterministic fashion which allows a $partial\_lookup$ to choose servers that share no common entries. Thus the lookup cost increases by 1 when the target answer size t increases by 20.

For RandomServer-20, the lookup cost in Figure 3 is consistently higher than Round-2's step curve because contacting additional servers during a lookup does not always yield 20 new entries (two servers may keep the same entry due to random choices). This effect is especially visible when the target answer size approaches a multiple of 20, e.g., $35 \leq t \leq 40$.

In contrast, Hash-2 is sometimes better than Round-2 and sometimes worse because it does not guarantee a minimum number of 20 entries per server. A small target answer size like 15, which always has lookup cost 1 under other strategies, has an average cost of 1.124 because some servers may have less than 15 entries. However, for a target answer size of 25, Hash-2 may still succeed in contacting only one server while all the other strategies need at least two servers as shown in the figure.

The data in Figure 3 suggests a couple of empirical rules of thumb. One, if the $partial\_lookup$ target answer sizes are smaller than the number of entries at each server, then avoid using Hash-y. Two, if the target answer sizes are *not* slightly more than the number of entries at each server, Round-y will give the lowest lookup cost.

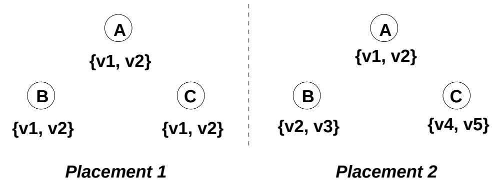

Figure 4. Two placements with different coverage

4.3 Maximum Coverage

The first quality metric is the maximum number of distinct entries retrievable by a client when it contacts *all* the servers. We call this metric the *maximum coverage*. Figure 4 shows two different placements of five entries onto three servers with different maximum coverage. Both placements can satisfy a target answer size 2. However, placement 1 has a coverage of two while placement 2 has a coverage of five. The coverage establishes an upper bound on the largest target answer size t supported by a strategy. For instance, placement 1 in Figure 4 can never support a target answer size of more than 2.

The maximum coverage is interesting for two reasons: (1) a larger coverage implies a strategy can support a more diverse group of clients with different target answer size requirements; (2) if entries associated with a key can be deleted, then the coverage also reflects how resilient a strategy is in continuing to support a specific target answer size. Suppose we delete the entry $v2$ in placement 1 of Figure 4. Then all three servers only have $v1$, thus can no longer support target answer size 2. In contrast, placement 2 is less affected by such a deletion, though some lookups may require contacting up to two servers instead of just one server.

Full replication has a complete coverage that includes every entry. Round-y and Hash-y have complete coverage only if the total storage of all servers is enough to store each entry $(v_1, \ldots, v_h)$ at least once. In the event of inadequate storage space, we assume Round-y and Hash-y keep a subset of $(v_1, \ldots, v_h)$; hence, the coverage is proportional to the storage limit until every entry is stored on some server.

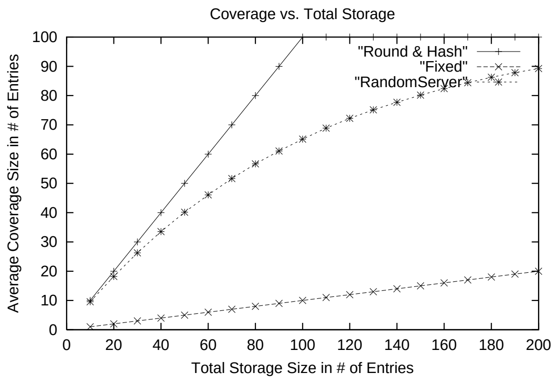

Figure 5. Coverage of strategies for managing 100 entries on 10 servers.

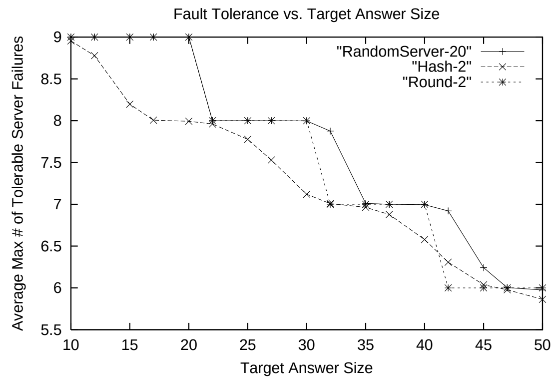

Figure 6. Fault tolerance of strategies for managing 100 entries on 10 servers using 200 entries of storage space.

Similarly, Fixed-x, has a coverage of x entries.

The coverage of RandomServer-x depends on how many entries are not stored at *any* servers. The probability that no server keeps a specific entry is $(1 - \frac{x}{h})^n$ where h is the total number of entries and n is the number of servers. Thus, the expected coverage is $h(1 - (1 - \frac{x}{h})^n)$. Figure 5 shows graphically the coverage of the strategies with different total storage limit. For this figure, we have 100 entries and 10 servers, and vary the total storage limit from 10 entries to 200 entries. In short, Round-y and Hash-y are ideal if clients require a large coverage.

4.4 Fault Tolerance

Another important quality metric is how many server failures can a strategy tolerate in the worst case before some client lookups fail. We consider a client lookup failed if it retrieves less than t entries specified in $partial\_lookup(t)$. We focus on the "worst case" scenario instead of random failures. Therefore, our *fault tolerance* metric reflects the minimum number of server failures from all possible failure patterns. Finding the minimum number of servers failures is NP-hard. For our evaluation, we use a heuristic, described in more detail in the extended version of this paper [8].

For strategies in Section 3, because all servers in full replication and Fixed-x are identical, they only require one operational server to service all client requests, hence can tolerate up to $n - 1$ failures. For the other three strategies, we ran a small simulation of managing 100 entries on 10 servers with a storage limitation of 200 entries. Figure 6 shows the result of the simulation where we compute the fault tolerance as a function of the target answer size (i.e., how many entries a client wants per lookup).

As Figure 6 shows, for the round-robin strategy, because entries are assigned deterministically, we get a step curve where increasing the target answer size by 10 reduces the fault tolerance by 1. For RandomServer-x, we observe a higher fault tolerance because of the potential common entries between multiple servers due to the random choices. However, the same reason contributed to the higher lookup cost of RandomServer-x than Round-y, as noted earlier.

Hash-y exhibits an S-shape curve as the overall fault tolerance declines. This shape is caused by two factors. The first factor is servers in Hash-y have different number of entries. If a server with more entries than the average fails, we lose more entries, which implies that the curve should drop like an exponential decay. However, the second factor of having multiple copies of each entry on a random set of servers makes it more difficult to render an entry irretrievable because the locations of the entries and their duplicates are not highly correlated as in the round-robin strategy. Therefore the failure of one server is unlikely to eliminate a large group of entries from the servers. As a summary, when coverage is not important, use Fixed-x for best fault tolerance; otherwise, use RandomServer-x and Round-y for large coverage and complete coverage respectively. Hash-y should be avoided unless the target answer size is very large (e.g., more than half of the total number of entries).

4.5 Unfairness in Lookup Answers

The maximum coverage and fault tolerance are concerned with how many entries can be returned for each lookup without examining whether some entries are returned more frequently than others. We capture this biasness in which entries are returned more frequently as the *unfairness* metric. The unfairness provides insights to the effects of a partial lookup service. For example, if each entry is an IP address of a service provider, then a biasness can overload a particular service provider.

A "fair" strategy should return each entry with equal likelihood during any individual lookups, i.e., if a key has h entries and a user wants t entries, the strategy should return each entry with probability $\frac{t}{h}$. The full replication strategy is obviously "fair" whereas Fixed-x is not because the unstored entries have probability 0 of being retrieved.

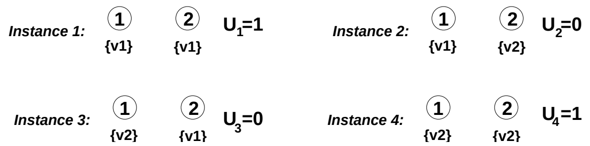

Figure 7. The four possible instances for RandomServer-1 managing 2 entries on 2 servers

To quantify the unfairness, we first define the *unfairness of an instance*, where an instance is a specific placement of entries onto the servers as a result of executing a strategy. Let $p_I(j)$ be the probability of returning the j-th entry on a lookup in instance I. Then the unfairness U_I is

$$U_I = \frac{h}{t}\sqrt{\frac{\sum_j \left(p_I(j) - \frac{t}{h}\right)^2}{h}} \qquad (1)$$

The definition above is commonly known as the coefficient of variation. The square root portion computes the standard deviation of individual entry's retrieval probability from the ideal value $\frac{t}{h}$. The $\frac{h}{t}$ factor then normalizes the standard deviation. A large coefficient (e.g., on the order of 0.1 to 1) indicates some entries are returned much more frequently than others. A small coefficient implies that all the entries have about a $\frac{t}{h}$ probability of being returned on a random client lookup. For example, if we manage 2 entries on 2 servers using Fixed-1 strategy, we retrieve the first entry with probability 1 and the second entry with probability 0. This instance has an unfairness $2 \cdot \sqrt{\frac{(1-\frac{1}{2})^2+(0-\frac{1}{2})^2}{2}} = 1$. Since the coefficient is 1, the strategy is very unfair.

Not all strategies have only one instance like Fixed-x. For example, random choices in RandomServer-x and Hash-y create different placements of entries onto servers and result in different instances. Each instance can have a different unfairness. Consider RandomServer-x with 2 servers, 2 entries, and the parameter x is 1. Depending on which entry is stored at each server, there are four instances, as shown in Figure 7. For target answer size 1, instances 1 and 4 have an unfairness 1 while instances 2 and 3 are completely fair. To accommodate the different instances, we take the average unfairness over all instances. Formally, let S be the set of all instances possible under a strategy. Let I be an instance in S. Let U_I denote the unfairness of instance I computed by Eq. (1). Then the *unfairness of a strategy* $\bar{U} = \frac{1}{|S|}\sum_{I \in S} U_I$. For the example in Figure 7, the unfairness of RandomServer-1 is $\frac{1}{2}$.

Of the strategies described in Section 3, full replication and Round-y has 0 unfairness. Other strategies' unfairness depends on the amount of storage. Figure 8 illustrates the unfairness of RandomServer-x and Hash-y when managing 100 entries on 10 servers with a client target answer size 35. We vary the total storage limit from 100 entries to 1000 entries to see how unfairness changes. Fixed-x is

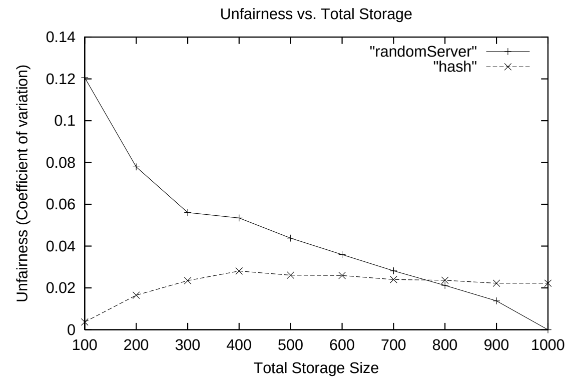

Figure 8. Unfairness when managing 100 keys on 10 servers with a target answer size 35.

not included in this figure as it exhibits similar behavior to RandomServer-x except an order of magnitude worse.

From Figure 8, we see the unfairness of RandomServer-x improves in two different phases as we increase the storage. In the first phase, the unfairness decreases rapidly like an exponential decay and corresponds to low storage limit where we need to contact multiple servers for each lookup. In the second phase, the unfairness decreases linearly and corresponds to high storage limit where one server is sufficient to answer the lookup.

The exponential decay in the first phase is strongly correlated to the maximum coverage. For instance, using 200 storage space in RandomServer-x has a coverage of about 89 entries. This coverage means 11 of the 100 entries have retrieval probability 0 and imposes a lower bound on unfairness. Since the maximum coverage for RandomServer-x grows like an inverted exponential, the unfairness mirrors the effect in the first phase. The second phase has a linear characteristic because a lookup contacts exactly one server and depends heavily on which entries that server has. As we increase the storage, each server keeps more and more entries until eventually it has all the entries.

Interestingly, for Hash-y, the opposite trend is true. In the first phase, the unfairness increases rather than decreases. In the second phase, the unfairness decreases only slightly. The intuition behind the increase is that hash functions create an inherent biasness in which entries are stored at each server. This bias is masked in the first phase by contacting multiple servers during a lookup. When we increased the storage, a lookup contacts fewer servers, and the unfairness reaches the inherent bias of the initial placement. When we increase the storage further to in the second phase, the number of entries stored at each server does not increase very much because multiple hash functions could map the same entry to the same server. Hence we do not get as much benefit as RandomServer-x.

In short, if we want to be fair, then we must to use either full replication or round-robin. If we can relax the unfairness constraint, then Fixed-x, RandomServer-x, and Hash-y offers several alternatives with different degrees of unfairness. One would choose the alternatives based on other metrics such as client lookup cost or dynamic update costs.

5 Dynamic Updates for the Strategies

This section describes how each strategy handles dynamic updates. Again we only give informal description. Details are in the extended version of this paper [8].

5.1 Full Replication

For each add or $delete$ request, a client selects a coordinator server S at random, and sends S the request. The coordinator then broadcasts instructions to all servers that perform the actual operation.

5.2 Fixed-x Strategy

Fixed-x uses the same approach as full replication for updates. However, since Fixed-x only keeps track of the first x entries and each server has the same entries, the coordinator server S, after receiving a client $add(v)$ request, can ignore v if it already has x entries. Thus server S only does a broadcast when it has fewer than x entries. Similarly on $delete(v)$, a broadcast is needed only if server S currently stores v locally. These semantics allow Fixed-x to selectively broadcast and generate less update traffic than full replication.

There are two caveats in this scheme. First, there is no concurrency control. If two add requests arrive at two different servers simultaneously when there are $x - 1$ entries per server, both requests will be processed. Second, servers may have fewer than x entries after $deletes$ because it is impossible to find a replacement for the deleted entries. Hence to support a client target answer size t, pick parameter x as $t + b$ where b is a cushion for having $deletes$ without new $adds$. The cushion b does *not* guarantee the service will always return at least t entries per lookup; it merely reduces the probability of getting fewer than t entries. Note that the cushion size b could be much smaller than the number of consecutive $deletes$ since not all of them will be one of the x stored entries. Section 6.2 looks at what are reasonable cushion size b.

5.3 RandomServer-x Strategy

An update in RandomServer-x could affect any subset of the servers. Therefore after receiving an update request at server S, S must broadcast the request to all servers and let each server perform its randomized decisions as follows. On $add(v)$, if a server has fewer than x entries, v is stored locally on the server automatically. If a server has x entries, then it decides whether to keep its current subset of x entries or replace one of the old x entries with the new entry v. It has been shown in [9] that to maintain a uniformly random subset of x entries, a server should, with probability $\frac{x}{h+1}$, keep v and remove one of the x entries at random, where h is the current number of entries in the system. However, this incremental scheme only guarantees a uniformly random subset of x entries if there are no $deletes$.

On $delete(v)$, we use the same cushion scheme as Fixed-x. An alternative is to actively find a replacement for a deleted entries by contacting other servers, since two servers are not likely to have the same entries. This replacement alternative uses less storage because we do not need to keep cushion entries. However, neither option preserves RandomServer-x's advantage of much lower unfairness (Section 4.5) than Fixed-x. In fact, the replacement alternative results in higher unfairness than the cushion scheme when there are $deletes$. Because finding a replacement is a costly operation, we chose the simpler cushion scheme. Section 6.3 looks at how quickly the fairness deteriorate when there are $deletes$.

5.4 Round-Robin-y Strategy

Ensuring round-robin placements of entries onto servers with updates is problematic. For example, a series of $deletes$ that affect only one server will create holes in the round-robin sequence. If these holes are not fixed, we lose the benefits of low lookup cost, low unfairness, and high fault tolerance. Also, $adds$ require knowledge of the current round-robin sequence. The details are given in the extended version of the paper [8].

5.5 Hash-y Strategy

Since Hash-y uses the hash functions to determine which servers are affected by the update request, it avoids dealing with dedicated counters or broadcasts. After receiving an $add(v)$ or $delete(v)$ request at server S, S informs the affected servers $f_1(v), \ldots, f_y(v)$ directly to add or remove the entry. Note Hash-y is essentially the dual of Fixed-x. Fixed-x does a lot of work (the broadcasts) for a small number of updates while Hash-y does little work on each update. We compare of the two strategies under dynamic updates in Section 6.4.

6 Dynamic Update Evaluation

We first describe how we simulate dynamic updates. We then focus on individual strategies.

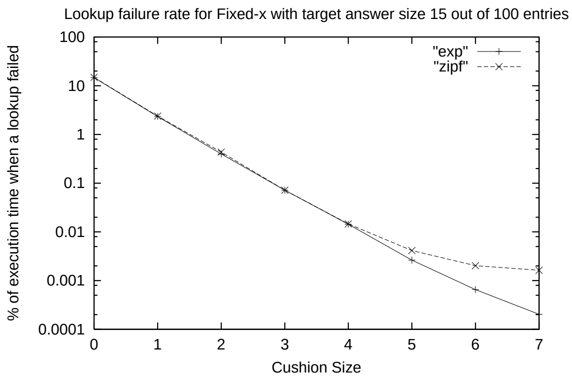

Figure 9. Percentage of time when a client failed to retrieve 15 of the 100 entries in Fixed-x with different cushion factors.

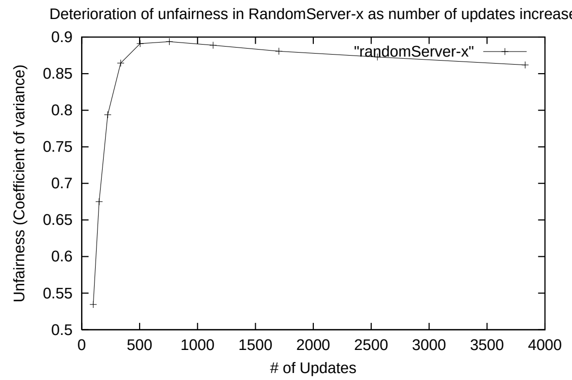

Figure 10. With more updates, the unfairness of the RandomServer-x increases rapidly

6.1 Generating Synthetic Updates

We use a discrete-time event-driven simulation to study the dynamic behavior of the strategies. We create update events with timestamps in advance and replay these events in the simulation. In order to focus on the steady-state behavior, we generate the add events separately from the $delete$ events such that the expected number of entries maintained by the servers is constant over time.

Specifically, we generate the add events using the Poisson arrival model with an expectation $\lambda = 10$, i.e., one add event per 10 time units. For each $add(v)$ event, we then determine the lifetime of the entry v using a second distribution and record the corresponding $delete(v)$ event at the end of its lifetime. For this study, we experiment with both an exponential distribution and a Zipf-like distribution for an entry's lifetime. We chose these two distributions because one is tail-heavy while the other is not.

6.2 Cushion Factor for the Fixed-x Strategy

The update behavior of the Fixed-x strategy, described in Section 5.2, often results in fewer than x entries on each server when $delete$ occurs. Consequently, even if we know, a priori, that no clients will request more than t entries per lookup, we cannot simply run Fixed-x with $x = t$. Instead, we need to set $x = t + b$ where b is the cushion factor. One natural question then is how big is this cushion factor b.

In Figure 9, we show the result of a simulation where our steady-state is 100 entries in the system and a client only wants 15 entries per lookup. We vary the cushion factor b from 0 to 7 entries and plot the percentage of time in which the client failed to retrieve 15 entries. The percentage is plotted in log scale. For 0 cushion, we get over 10 percent failures. As we increase the cushion, the failure time drops exponentially. Note that the heavy-tail Zipf-like distribution tapers off at the end.

The trend in Figure 9 is similar for different target answer sizes. But the exact cushion size needed to achieve a certain failure rate depends on two factors: the target answer size t and the expected length of an entry's life time in the system. These two factors are opposites of each other. As a client wants more entries per lookup, the cushion size grows proportionally. But as the expected life time of an entry increases, the cushion size decreases proportionally. For example in Figure 9, a cushion size 3 yields a failure rate 0.1% when the average life time is 1000. If the average life time doubles to 2000 time units, a cushion size 2 is sufficient. This behavior is because longer entry life time means the probability of multiple $deletes$ occurring between two $adds$ is smaller. In practice, a non-zero failure chance is not detrimental to most applications. Since failures are quickly repaired by new add events, we can mask the failures by asking the client to retry after a pause.

6.3 RandomServer-x and Round-y

We now argue qualitatively that RandomServer-x and Round-y are not appropriate when update rate is high. Consider RandomServer-x versus Fixed-x. The advantage of RandomServer-x is lower unfairness regarding which entries are returned to clients. However when there are many $delete$ operations, the unfairness for RandomServer-x approaches the level of Fixed-x.

Figure 10 shows the effect on the unfairness as more $deletes$ occur. In this experiment, there are 10 servers and each server holds at most 20 entries out of the expected 100 entries in the system. From the graph, we see the unfairness deteriorates rapidly and then stabilizes as the number of $deletes$ increase. This rise is because deleted entries are replaced by newer insertions, thus creating a bias towards the newer entries. Even if we try to actively find replacement immediately after a deletion as mentioned in Section 5.3, the unfairness will not improve because we then bias towards the older entries. In fact, finding replacements immediately is worse in terms of unfairness.

In comparison, Fixed-x, for this same experiment, has an unfairness of 2. Thus RandomServer-x is only a factor of 2 better than Fixed-x in unfairness as opposed to an order of magnitude better for the static case. While not getting much better unfairness than Fixed-x, RandomServer-x is also incurring five times more broadcasts than Fixed-x because RandomServer-x does a broadcast on each update compared to one-fifth of the updates for Fixed-x (keeping 20 entries out of 100). Hence Fixed-x is much more attractive under high update rates. The trade-off for using Fixed-x is the smaller coverage. For the same experiment, Fixed-x usually has half as many distinct entries as RandomServer-x. Similarly, high update rates cause problems for Round-y. The trade-off in using Hash-y over Round-y is that some client lookups may contact one extra server because some servers may have fewer entries than others.

In short, RandomServer-x and Round-y are much better in a static environment than Fixed-x and Hash-y. If a smaller coverage or a higher client lookup cost can be tolerated, Fixed-x and Hash-y are more efficient because of lower overhead in processing each update.

6.4 Comparing Fixed-x and Hash-y

The previous section has argued for choosing Fixed-x over RandomServer-x and Hash-y over Round-y. This section focuses on quantitatively comparing Fixed-x and Hash-y in terms of the overhead cost for handling updates. For the overhead cost, we count the total number of messages received and processed by all the servers in the system during simulation. Since we are counting processed messages, a broadcast has overhead cost n where n is the number of servers. A point-to-point message has cost 1.

Under this simple cost model of counting processed messages, we study how the strategies behave when clients want a small fraction of all the entries per lookup and when they want a large fraction. This notion of small or large fraction is captured in the ratio between the client target answer size t and the number of entries in the system h. A small $\frac{t}{h}$ ratio implies clients want a small fraction while a large ratio means a large fraction. Thus in the experiment, we vary this $\frac{t}{h}$ ratio and simulate the overhead cost for different ratios.

For the simulation, we fix the target answer size at 40 and vary the number of entries in the system from 100 to 400, thus giving us a range of ratios between 0.1 and 0.4. We also use 10 servers for this experiment. With this setup, we choose $x = 50$ for the Fixed-x strategy. This choice gives a cushion of size 10, ensuring that requests for 40 entries can be satisfied with very high probability. For Hash-y, we cannot just fix the value of y because if we choose a large y such that the lookup cost is close to 1 when the ratio is 0.4, the strategy is unnecessarily penalized when the ratio

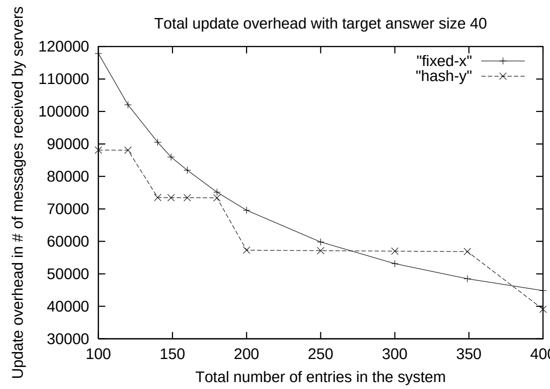

Figure 11. Update overhead for Fixed-x and Hash-y with varying number of entries in the steady-state.

is 0.1 where a smaller y is sufficient for the task. Similarly, if we choose a small y for the ratio 0.1, the lookup cost is much bigger than 1 (the lookup cost for Fixed-50) when the ratio is 0.4. To resolve this complication, we decided to choose an optimal y for each ratio we study, where the optimal is when the expected number of entries per server is at least the target answer size 40. This choice implies that the lookup cost is always close to 1. In the context of this experiment, we use $y = 1$ when the total number of entries h is 400, $y = 2$ when h is between 200 and 400, $y = 3$ when h is between 133 and 200, and $y = 4$ when h is between 100 and 133.

Figure 11 shows the result of the simulation as we vary the total number of entries in the system from 100 to 400. The x-axis shows the total number of entries, from which we can get the corresponding ratio. The y-axis shows the total number of messages processed by the servers. The curve for Fixed-x is inversely proportional to the total number of entries in the system because the fractions of add and $delete$ that affect the chosen x entries is decreasing at the same rate. For Hash-y, the overhead cost is purely determined by the choice of y as described in the previous paragraph, hence follows a step curve shape with break points at 133, 200, and 400.

Notice that the two curves cross each other at a couple points. The crossover points can be captured analytically by computing when the two strategies have the same overhead cost. For Hash-y, we perform approximately $1 + y$ operations for each update: one to process the client request and y for storing or deleting y copies in the system (barring collisions among the y hash functions). Therefore, the total cost for Hash-y is $(1 + y)U$ where U is the number of updates. For Fixed-x, each update incurs cost 1 to check whether the broadcast is necessary plus the cost n if there is a broadcast, which occurs with probability $\frac{x}{h}$. Hence the total cost is $(1 + \frac{x}{h}n)U$. Comparing the two costs yield the equality $\frac{x}{h}n = y$. When $\frac{x}{h}n$ is smaller than y, Fixed-x has lower overhead than Hash-y, and vice versa. This equality

Strategy	Storage		Coverage	Fault Tolerance	Fairness		Lookup Cost	Update Overhead	
	Few entries	Many entries			Few updates	Many updates		Small target size	Large target size
Fixed-x	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	⋆⋆⋆⋆	⋆	⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆
RandomServer-x	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆	⋆⋆⋆	⋆⋆	⋆⋆
Round-y	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆	⋆
Hash-y	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆⋆⋆

Table 2. Informal summary of the strategies with respect to our proposed metrics. (More stars is better.)

test only gives us one crossover point. However recall that we are using different values for y as the number of entries increase. Specifically, $y = \lceil \frac{t \cdot n}{h} \rceil$ where t is the target answer size. Therefore, the equality test is really $\frac{x}{h} n = \lceil \frac{t \cdot n}{h} \rceil$. The ceiling function creates discontinuity in the overhead cost, thus creates multiple crossover points. Note that if we continue to increase the number of entries in the system to beyond 500, we get a third crossover point where Fixed-50 has lower overhead than Hash-1. Beyond the 500 point, Hash-1 always has higher overhead because it has to store each entry at least once, thus cannot take advantage of the small ratio $\frac{t}{h}$ where $\frac{t}{h}$ is less than $\frac{1}{n}$.

As a rule of thumb, if the client target answer size is a small fraction of the total number of entries (typically less than $\frac{1}{n}$), then Fixed-x will have less update overhead. Another consideration in deciding between Fixed-x and Hash-y is the lookup cost described in Section 4.2. Since Hash-y has higher lookup cost, the ratio between lookups and updates will also be a factor in choosing Fixed-x or Hash-y.

7 Related Work

The most well known lookup service is the Domain Name Service[2]. More recently, peer-to-peer (P2P) systems [4, 3, 7, 6] have dominated the discussion of lookup services in locating shared files. Some P2P systems like Napster[4] use a centralized lookup service with the $partial\ lookup$ characteristic. When clients search for a file, only the first 50 or 100 hits are returned. However, the servers usually keep track of a large amount of information rather than discarding extra information. Other P2P systems like Chord[7] and CAN[6] partition the key space rather than partition the entries of a key as we suggest for partial lookups. And Gnutella-style[1] systems do not even reorganize keys.

Aside from P2P, the partial lookup idea of storing just enough entries of a key at each server also resembles vertical partitioning of relations in distributed databases[5]. In vertical partitioning, a relation table is broken up by attributes and only a subset of the attributes of the table are stored at individual servers. However, the main focuses of various partitioning schemes are completeness and reconstruction, which are not concerns for partial lookup.

8 Conclusion

In this paper, we demonstrated a *partial lookup service* that maintains fewer entries on a server provides significant performance improvements over the traditional full replication or hashing-based lookup services. Specifically, storage cost for partial lookup services are lower, which in turn leads to lower overhead in processing updates. Furthermore, partial lookup services are insensitive to the popular key or hot-spot problems which plague traditional hashing-based lookup services. These two advantages make partial lookup services very attractive to applications with high workload. Our paper also showed that keeping fewer entries at servers does not adversely affect most metrics that are important to users such as client lookup cost, coverage, and fairness. Table 2 informally summarizes the tradeoffs between different strategies, with respect to our proposed metrics. In the table, four stars imply the strategy is very suitable, while one star implies the strategy performs poorly. Note that no strategy is the best in all situations.

References

[1] Gnutella. Website http://gnutella.wego.com.

[2] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Procoeedings of ACM SIGCOMM*, pages 123–133, Stanford, CA, 1988.

[3] Morpheus. Website http://www.musiccity.com.

[4] Napster. Website http://www.napster.com.

[5] M. T. Oszu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, August 2001.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 160–177, San Diego, August 2001.

[8] Q. Sun and H. Garcia-Molina. Partial lookup services (extended version). Technical report, Stanford University, 2002. Available at http://dbpubs.stanford.edu/pub/2002-15.

[9] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. on Math. Software*, 11(1):37–57, March 1985.