The Problem of Coherence of a Distributed Hash Table

Deepika Balachander
Department of Electrical and Computer Engineering, North Carolina State University
Raleigh, NC 27695-7911, USA
deepik@ncsu.edu

Rada Chirkova
Department of Computer Science, North Carolina State University
Raleigh, NC 27695-8206, USA
rychirko@ncsu.edu

Tiia. J. Salo
IBM Research Triangle Park
NC 27709, USA
tjsalo@us.ibm.com

May 8, 2013

(1)Introduction:

The fundamental problem being investigated is the maintenance of data consistency in a distributed system setup. Consider a set of some N >= 1 computers sharing a common hash table stored at a remote centralized repository. Each system is interested in a portion of the hash table, and these portions may overlap. So as to facilitate ease of access, we are allowed to cache a portion of the hash table at each system. Each system can read data cached at its location, or send requests straight to the remote repository.

There are three problems to be understood and analyzed here. One, when read requests come in from a computer, we need to figure out whether those requests can be answered by their local caches or whether such requests need to be propagated to the remote centralized repository. Two, based on the trail of read requests coming in from the computers, we should be able to predict the entries of interest and cache them in advance at the computer location. Three, the most important problem is to provide a consistent view of the shared resource to all the N computers at all times. We refer to these three problems collectively as "the problem of coherence of a distributed hash table."
Here onwards, the set of N computers is interchangeably referred to as clients and the remote centralized repository as server and vice versa.

Work on the problem at hand has the potential to impact a number of other important settings, and is hence an important problem to solve. The problem of coherence of a distributed hash table is closely related to a number of other problems, including maintenance of materialized views (MVs) in a distributed environment, cache coherence in multiprocessor systems, and concurrent processing of tasks in distributed computing. In Section 2 we discuss in length about how each of these problems are closely tied to our area of research, why they are important and how the work presented here is different from the work done in each of them so far.

A significant amount of research has been done in each of these related areas. However, this research can be best described as a generalization of the problem at hand. For example, with MVs, the view being materialized could be the answer to any query executed over the base data. However, with respect to the studied problem, the data being locally cached or “materialized” will always be a subset of rows of the centralized table. Numerous techniques have been developed for view maintenance of MV. However, each of these techniques maybe far more generalized and may take a lot more factors into consideration than what is required to provide an effective solution to the studied problem.

Hence, this project requires us to study techniques from a number of related areas of research and then tailor them suitably to act as effective solutions to the problem at hand. This makes it interesting and challenging at the same time.

In this technical report we study "the problem of coherence of a distributed hash table", and propose a number of solutions for the same. We then implement and test the different solutions and compare their relative performance under different input conditions. The goal of this report is not to provide one good solution to the aforementioned problem but to rather provide a series of solutions where some are more effective than the others, under certain input conditions.

(2)Related Work:

The problem of coherence of a distributed hash table is closely related to a number of other problems. These include maintenance of MV in a distributed environment, cache coherence in multiprocessor systems, and concurrent processing of tasks in distributed computing. We will now discuss in depth, how each of these problems are related to the studied problem, their individual importance and general methods proposed to provide solutions to them so far. In each case we will place special emphasis on how the scope or approach of these cited works differs from the work done in our current project.

A MV is a database object that contains the results of a query. It may be a local copy of data located remotely, or may be a subset of the rows and/or columns of a table or join result etc computed over base data located locally /remotely .
MVs are a natural embodiment of the ideas of pre-computing and caching in databases i.e. instead of computing a query from scratch over the base data, a database system can use results that have already been computed, stored and maintained. The ability of MVs to speed up queries, benefit most database applications ranging from traditional querying and reporting to data mining [1].

To draw an analogy between MVs and the problem at hand, we can think of the hash table at server to be the base data and the caches maintained at clients to be the derived/materialized data. Past research in MVs focuses on view selection (what are the right views to materialize to efficiently compute and maintain the results of a given query), view use (when should MVs be used to answer queries), and view maintenance (how to keep MVs consistent when changes are made to the base data).As you can see our problem is closely tied to research in the area of MVs and more so in the area of maintenance and selection.

Popular research in view maintenance analyze the problem along the timing dimension (immediate vs. deferred maintenance), information dimension (different levels of data accessible for the purpose of maintenance) and focuses on using auxiliary views (at times maintaining more can mean maintaining less) to ease the problem of maintenance. Time as a dimension has been exploited in a couple of our solutions. However our exploitation of time is not restricted to immediate and deferred maintenance (for more information refer to section on Scenario 4).

Other relevant research in the area of MVs focuses on developing systematic and automated solutions to the problem of view selection. In this work, view selection is done by exploiting the temporal locality of user queries. Wherein the views cached /materialized (if any) always correspond to entries read most recently in the past. A good direction of future work would be to develop another solution which would exploit the spatial locality of user queries.

In a shared memory multiprocessor with a separate cache memory for each processor, it is possible to have many copies of any one instruction operand: one copy in the main memory and one in each cache memory. When one copy of an operand is changed, the other copies of the operand must be changed also. Cache coherence is the discipline that ensures that changes in the values of shared operands are propagated throughout the system in a timely fashion. Caches were developed as a bridge between a high-speed processor and a relatively slow memory. By storing subsets of the memory (by the principle of locality of reference) in a fast access region (i.e. the cache), the performance of the system is improved because most memory accesses can be satisfied by the cache instead of main memory [2].

We can draw a parallel between the problem of cache coherence in multiprocessor systems and the problem at hand by equating the caches of different processors to the local caches at the client computers and the main memory to the storage at the remote centralized repository. The localities of reference which can be exploited include temporal and spatial locality. The only difference between in the 2 problems is that while the local copies cached at the individual processors can be directly modified in a multiprocessor system, our problem limits all direct modifications to the centralized repository or equivalently main memory. In other words, we support only reads by client and not both reads and writes.

The two basic methods for insuring cache consistency in multiprocessor systems are hardware implementations and software protocols.
In a hardware implementation, special hardware is added to the machine to detect cache accesses and implement a suitable consistency protocol which is transparent to the programmer and compiler. Popular hardware based approaches include directory based and snooping based protocols.
In a software approach, it is the responsibility of the compiler to generate consistent code. Parallel code may be made consistent by the compiler by inserting additional consistency instructions in the code. However, the static compiler analysis cannot detect run-time consistency conflicts. Hence, hardware based implementations are better as they can not only be implemented more efficiently but can also detect dynamic sharing sequences[2].
However, applying hardware solutions to the problem at hand would limit the clients to only those having a particular hardware configuration. This can be too restrictive, and hence we are interested in developing software solutions to the studied problem.

In addition, there are two basic enforcement strategies for insuring that cache lines are consistent. They are write-update (WU) and write-invalidate (WI). In the write-update protocol, after every write of a data item by a processor, the copy of the data item in other processor caches and main memory is updated to the new value. For write-invalidate, a write by a processor on a data item causes a message to be sent to other processor's caching the data item to invalidate the cache line containing this data item. If a processor requires a data item that was previously invalidated, it will reload it into the cache from main memory [2]. In this project, we develop a set of software protocols based on the aforementioned strategies of WU and WI.

As mentioned previously, another area of research closely related to the problem at hand, is distributed computing. In general, distributed computing is any computing that involves multiple computers remote from each other that each have a role in a computation problem or information processing. In business enterprises, distributed computing generally has meant putting various steps in business processes at the most efficient places in a network of computers. In the typical transaction using the 3-tier model, user interface processing is done in the PC at the user's location, business processing is done in a remote computer, and database access and processing is done in another computer that provides centralized access for many business processes. Typically, this kind of distributed computing uses the client/server communications model and a parallel can be drawn between distributed computing and the problem at hand by coalescing the operations of step 2 and step 3 of the 3 tier-model to a single step which is performed at a remote centralized repository.

A common way to address scalability requirements of distributed environment is to employ server replication and client caching of objects that encapsulate the service state. When there are a number of copies of the same object, there exists the problem of maintaining consistency among them. As we do not perform server replication we limit our interest (in distributed computing) to maintaining consistency of client caches.
Broadly speaking there are 2 models for consistency in distributed systems. Strong consistency, wherein after an update completes any subsequent access will return the updated value, the opposite of strong

consistency is weak consistency. We are interested in strong consistency and common approaches to achieve strong consistency include Lock-Based approaches and QUORAM-ASSEMBLY methods. Consistency with Concurrency is automatically achieved as a part of our current solution as we make an assumption that obviates the need to handle such cases explicitly (refer Section on problem specification). However in the absence of this assumption, our solution cannot guarantee strong consistency and ensuring the same under all conditions would be a good line of future work.

Another important point to note is that the aforementioned problems not independent of each other. Semantic Caching [3], is a technique which combines the ideas of caching and materialized views. A semantic cache remembers the semantic descriptions of its contents as view definitions, so that we can determine the completeness of query answers determined over caches and query the source only when needed. This idea has been applied to a wide range of settings, including caching for web [4] and mobile as well as other types of distributed systems.

(3) Specifications, Requirements and Measures:

The problem being studied is "the problem of coherence of a distributed hash table" .The system under study consists of N (where N>= 1) client computers and a single centralized server which houses the shared hash table. The system design makes a few general assumptions as follows-:

1) The system supports only reads (for (key, value) pairs in the hash table) by clients.

2) The client caches are initially empty. Their size is unbounded.

3) When a (key, value) pair is changed at server, the corresponding entry if previously cached at a client now becomes invalid. In order to inform clients of this now obsolete cache entry, the server broadcasts an invalidate message specific to this key to all active clients. On receiving the invalidate message, a client, checks to see if the corresponding (key, value) pair is cached in it, and if so, marks it invalid. Broadcasting ensures that all clients are synchronized in their awareness of an obsolete cache entry.

4) Clients become immediately aware of obsolete cache entries i.e. there is no race condition between the server announcing a hash-table entry to be invalid and a client requesting that entry. Thus, whenever the server announces a hash-table entry to be invalid, each client either has requested the old value previously, or will wait until reception of the broadcasted- invalidate message to request the new value. There are no read requests made in the interim of a change in (key, value) pair at server and the reception of a broadcasted invalidate message at client. This assumption guarantees consistency with concurrency.

5) Unless otherwise stated, the size of pull history (if maintained) at server is unbounded.

6) Broadcasting, i.e. pushing updates to all clients on change in a given (key, value) pair irrespective of their interest is NOT proposed as a solution.

The clients start by pulling hash entries from server and caching them. When a (key, value) pair to be read by client has a valid entry in its local cache , the read request can be satisfied at the client itself rather than being re-directed as a pull to server. Thus the goal of maintaining a cache is to satisfy read requests in an economical manner. However, hash entries are volatile and changes can be made to them at server. Client read requests cannot be satisfied by stale/obsolete cache entries and we need to develop one or more solutions to maintain the coherence of (key, value) pairs in client caches as changes are made to their counterparts at server.

Thus, the objective of the system design to satisfy all read requests made by clients in an economical manner while factoring in the cost of communication and the cost of maintaining coherence of client caches. To achieve the above objective we have a proposed a number of solutions.

Once we have developed different algorithms/solutions to solve the fundamental problem (of cache consistency in a distributed environment), we need to quantify each solution so that we can compare the performance of one against another under different input conditions. As the goal is to satisfy various read requests in an economical manner, we list out the various operations involved and the costs incurred in each case. Performance metrics at server are broadly categorized into Processing Overhead and Storage Overhead.

Processing Overhead-:

1) Pulls made by the client are charged to the server. Pulls are a consequence of obsolete or missed cache entries at client and each pull has an associated constant cost called the PULLCHARGE.

2) Though used pushes (a push made for a (key, value) pair which will be later read at client is called a used push, all other pushes are unused) made from server to client are free of charge. The server incurs an additional cost for every unused push it makes. However, the server has no way of knowing in advance whether a push is used or unused. To approximate the above calibration, the server charges all pushes with a blanket PUSHCHARGE and on termination divides the net PUSHCHARGE by the total number of read requests made. Lower the aforementioned ratio less is cost spent on pushes.

3) The cost to make changes to the centralized hash table is assumed to be zero. Also, the cost of all actions taken as a consequence of the aforementioned operation, (except for making a push) is assumed to be zero. E.g. broadcasting invalidates, scanning the pull histories of different clients to determine if they contain the changed key, etc.

4) The cost to retrieve values from the hash table or any other data structures is assumed to be zero. However, any additional processing done at the server other than the very basic fetch operation, corresponds to additional costs, and must be accounted for while quantifying an algorithm. The associated costs in each case equals COST OF ONE OPERATION * no. of times it is performed. E.g. creating pull history, maintaining pull history etc.

5) One exception to the above rule is the periodic scanning of pull histories (if done) at server. The periodic scanning of pull histories is accounted for like any other operation (COST OF ONE OPERATION * no. of times it is performed), however the net cost is amortized over the total number of read requests made at client. This is because the scanning of pull histories has a direct impact on cache maintenance similar to making pushes.

Storage Overhead-:

1) The cost associated with storing the centralized hash table is zero.

2) Though an assumption for unbounded cache size is made at client, the same does not hold good for the server. Therefore any data structure maintained at server with the exception of the original hash table is charged to server. The cost associated with storing additional data structures is given as COST OF STORING ONE INSTANCE* no. of such instances. E.g. storage of pull history, storage of THRESHOLDTIME etc

3) Each node of the pull history, irrespective of which Scenario it corresponds to, is associated with a constant storage cost given by PULLSTORAGECOST. Here, we do not take into consideration the different levels of information maintained by pull history nodes of different Scenarios for e.g. The pull history is Scenario 1 is just a list of keys, however the pull history in Scenario 4 is a list keys and their associated timestamps.

Thus we choose to conduct the performance evaluation of different scenarios with a greater emphasis on processing overhead than storage overhead. Analyzing the performances of different scenarios with equal or greater importance to storage overhead is a good line of future work.

The total (processing and storage) overhead at server is the sum of the net processing and storage costs incurred at server over all clients. For each Scenario we also determine the best case and worst case performance at server. In computer science, best or worst case performance of a given algorithm corresponds to conditions when the resource utilization by the algorithm, is at its very least (minimum), or at its very most (maximum) respectively. Usually the resource being considered is running time, but it could also be memory or other resources. We consider the processing and storage overhead for the purpose of our analysis.

To determine the best case and worst case of a given Scenario we use the performance measures and constants specified below-:

r: Number of Read requests made at client

R: Number of unique Read requests made at client

N: Number of clients connected to server

STORESIZE: Fixed size of centralized hash table

$f(C)^*$: frequency at which i th entry in hash table changes where i= 1, 2, 3…..STORESIZE =1/ (time to change 1 (key, value) pair in hash table * STORESIZE).

This can be achieved if entries in the hash table rather than being changed randomly are changed in a (systematic) round-robin fashion where we loop back to the beginning of the hash table on reaching the end.

PULLCHARGE = 100, is the cost incurred in pulling a (key,value) pair from server to client.

PUSHCHARGE = 30, is the cost incurred in pushing an update to a single (key,value) pair from server to client.

COSTOFADDEDPUSH= 10, is the cost incurred in adding an extra update to an existing push so as to gain the advantage of batching.

PULLCREATIONCOST = 1, is the cost incurred in traversing a single node in pull history (during the process of pull history creation) or adding a single node to the existing pull history.

STORAGECOST = 1, is the cost incurred in terms of memory usage in adding a single node to the existing pull history.

STORAGECOSTM = 1, is the cost incurred in terms of memory usage in storing a parameter m in Scenario 3.

STORAGECOSTN = 1, is the cost incurred in terms of memory usage in storing a parameter n in Scenario 3.

STORAGECOSTCOUNT = 1, is the cost incurred in terms of memory usage(per client) in storing the count of the # of pushes made so far, in Scenario 3.

STORAGECOSTTHRESHOLD = 1, is the cost incurred in terms of memory usage in storing parameter THRESHOLDTIME, in Scenario 4.

STORAGECOSTDEFAULTTIMEOUT = 1, is the cost incurred in terms of memory usage in storing parameter DEFAULTTIMEOUT, in Scenario 5.

PULLMAINTENANCECOST = 1,is the cost incurred in traversing a single node of pull history, during the process of making periodic scans in Scenario 5.

* For purposes of performance evaluation we assume that incoming read requests arrive at certain minimum rate such that f(C) cannot be increased to > 1/r

(4)Summary of the Proposed Scenarios:

The primary objective of this technical report is to study "the problem of coherence of a distributed hash table" and to provide one or more economical solutions for the same. To achieve the above objective we have proposed a number of solutions and tested them under varying input conditions. Each solution along with the broad class of input conditions for which it has been tested constitutes a scenario. We will now discuss briefly the different scenarios being proposed and study their comparative merits.

Scenario 0 is the most basic scenario where the server puts no effort into minimizing the cost of communication or the cost of cache maintenance. Here no assumptions are made about the client's requests/interests, they can either be stable or evolve with time. When a (key, value) pair to be read by a client has a valid entry in local cache, it is read from the cache, else the request is re-directed as a pull to server.

Under circumstances when all client reads are unique or when hash entries of client interest are slightly volatile or completely non-volatile, this system design can be shown to be the most economical among all solutions being proposed. This can be attributed to the fact that this scenario makes the least investment among all scenarios in order to reduce the overall system cost. However, in cases where the hash entries of client interest are moderate -to- highly volatile, the same key may be repeatedly pulled off server resulting in a high system cost.

In Scenario 1 the server maintains a data structure, the pull history, to reduce costs of communication and cache maintenance. The pull history is maintained on a per client basis and is an unordered list of the unique keys requested by a client in the past. This scenario makes an assumption that the client's requests/interests stabilize with time. Thus, when a change is made to a (key,value) pair at server, the serves pushes updates to all clients who contain the aforementioned key in their pull history. This ensures that client caches are always up to date. Since the requests of clients are stable, all pushes made are of client interest and the cost of cache maintenance is zero (as stated by assumptions, used pushes are free of charge).

In this scenario when a read (for a specific key) is made by a client, for the first time, it is pulled from server. Thereafter all requests for the key are satisfied by cache. Hence, this scenario is more economical than Scenario 0 when hash entries of client interest (which is assumed to be stable) are moderate-to- highly volatile.

Scenario 2 uses the solution proposed in Scenario 1 and tests its effectiveness against evolving client interests. In accordance to the proposed solution, whenever a change is made to a (key, value) pair at server, the server pushes updates to all clients who contain the aforementioned key in their pull history. This results in an increased system cost because, clients' interests evolve with time, and a key pulled in the past may no longer be of client interest, and as stated by assumptions, the server is charged for every unused push it makes. Thus Scenario 2 may be as bad Scenario 0 if not worse when client interests (which evolve with time) are volatile.

To minimize the number unnecessary/unused pushes, the solution proposed in Scenario 2 is extended in Scenario 3. In Scenario 3 the pull history per client is an ordered (based on time of latest pull) list

of unique keys, pulled by clients in the past. Also, the length of the pull history is limited to some maxLengthOfPullHistory. If a new entry has to be added to the pull history, which is of length maxLengthOfPullHistory, the oldest entry is deleted. Further, in Scenario 3 the number of contiguous pushes made per client is also limited to some value, once the threshold is reached,pushes to the client resume only on receiving a pull.

If the maxLengthOfPullHistory is fixed to equal (or be slightly greater/lesser than) the number of keys in client's current interest and the threshold number of pushes is kept as high as infinity, the number of unused pushes can be reduced to zero(or made close to zero).Under such conditions Scenario 3 is far more economical than Scenario 2 and Scenario 0. However, if the maxLengthOfPullHistory is much lesser or greater than the number of keys in client's current interest , the threshold number of pushes can be kept as low as 1 so that the performance is no worse than Scenario 0. It should be noted for stable requests the number of keys of client's current interest equal the number of unique requests made in the past, and for this specific case can be approximated to infinity.

However, there is no way to know how many keys would be in client interest. Hence it is difficult to fix the value of maxLengthOfPullHistory to get a performance improvement over Scenario 0and Scenario 2. To overcome the above drawback, Scenario 4 was proposed. The solution in Scenario 4 is similar to that proposed in Scenario 2, except that each key in pull history is also associated with the timestamp of its latest pull. The system also has a threshold parameter for staleness, THRESHOLDTIME, such that, each time the server scans the pull history of a client, it deletes all pull entries time stamped at time Current Time- THRESHOLDTIME or earlier. THRESHOLDTIME is chosen to be the minimum possible time such that the probability of a client being interested in a key pulled at Current Time –THRESHOLDTIME or earlier is low. As long as THRESHOLDTIME is chosen well, the number of unused pushes can be minimized and even reduced to zero. Though the number of pulls /number of unused pushes maybe slightly more than the number of pulls /number of unused pushes made in ideal case in Scenario 3, Scenario 4 is more practical than Scenario 3 and with a high probability provides a performance improvement over Scenario 0 and Scenario 2. It should be noted for stable requests THRESHOLDTIME should equal infinity, as all keys pulled in the past are of client's current interest.

All scenarios discussed so far make immediate client updates, however Scenario 5 explores the idea of deferred updates. In Scenario 5, the server maintains pull histories per client similar to Scenario 1. In addition, each key in the pull history has an associated field to specify whether it has been changed since last pull/push. This helps the server scan through the pull histories of all clients once every DEFAULTTIMEOUT, and push updates to only those clients who have 1(or more) keys which have been changed since last pull/push. Hence updates to clients are not immediate but potentially periodic. This scenario has been tested for both stabilized and varying client interests. For stabilized interests, (provided DEFAULTTIMEOUT is chosen to be suitably small), this scenario is as good as Scenario 1. In case of varying client interests, this scenario can perform as well as Scenario 2,(by choosing small to intermediate values of DEFAULTTIMEOUT), in cases where Scenario 2 is better than Scenario 0 , and no worse than Scenario 0 otherwise (by choosing DEFAULTTIMEOUT = ∞).

Thus we have proposed a number of solutions to fulfill the objective of satisfying read requests at clients in an economical manner. The effectiveness of a given solution is dependent on input parameters such as the stability of incoming read requests and the frequency of changes made at the centralized hash table etc. Our goal with this project is to test the different solutions under various input conditions and to determine a niche case for each , where its performance is the best amongst all solutions being proposed. This would help us develop a feel/ intuition for the performance of each solution (under various input conditions) and help us pick the best given a set of arbitrary input conditions.

(5)Scenario 0:

High Level Intuition :

In Scenario 0 the client does all the work and the server does zero work. When a previously read (key ,value) pair needs to be read again at the client, it is simple read off local cache if no changes have been made to the (key, value)pair since it was last read. However, if the (key, value) pair to be read is missing in local cache or has an obsolete cache entry, a pull needs to be issued from client to server.

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete a pull is issued by client to server. All other requests are satisfied by valid cache entries.

Details of Implementation:

Server:

At the server a hash table (i.e. a set of (key, value) pairs) is read off a notepad file and is stored in an array of hash nodes of size STORE SIZE. Different values hashing to the same location are saved using separate chaining. The size of the array can be changed such that the separate chains are at maximum 3 nodes deep to ensure a reasonably good performance.

Once the hash table is setup, the server waits on a socket listening for client connections. Once the server connects to a client, the number of active clients is incremented by one and the client is allocated its own individual socket. The client can use this socket to send read requests to server and this mechanism helps the server communicate with many clients simultaneously. When a client requests a value corresponding to a given key (Pull), the server fetches the same off the hash table and returns it to the client. Each time a pull takes place, the server outputs a) the ID of the client that has done the pull b) the total number of pulls so far from that client and c) the total number of pulls overall so far.
Clients can also request to be disconnected from the server. Each time a client disconnects, the server decrements the number of active clients and outputs a) the ID of the client which has disconnected, b) the amount by which it charged the server c) the list of keys it read during its lifetime and the number of reads it made for each. The server also maintains an “importance” co-efficient for each client. By default the importance co-efficient for all clients is unity. This co-efficient helps the server weigh in the costs of different clients by factoring in their relative importance such that the net cost at server is skewed in the direction of the more important clients.

A module associated with server can be used to make changes to (key, value) pairs stored in hash table. When a change is made to a given (key, value)pair, the server broadcasts an invalidate message to notify all clients in the system.

The server can also quit if the number of active clients is zero. Just before the server quits/exits, it outputs a) the total number of pulls made per client and b) the total number of pulls made over all clients, over the course of its service duration.

Client:

Each client maintains a local cache, which is initially empty. The cache is designed to be a list of unique keys requested by client in the past. Each key in the cache has an associated value, a state and a count of the number of it has been read. The state helps differentiate valid and in-valid entries and the count plays a role in performance evaluation. The client first connects to the server so that any read requests which lead to cache misses or obsolete/in-valid cache entries can be redirected to it (as pulls).

When a pull for a given key is made from client to server, the pulled key is added to local cache (if not already present) , its value and count are updated to reflect current pull and its state is set to "valid". If instead the read request is satisfied by a local cache entry, its count is simply incremented by one.

In addition to receiving replies for pulls , clients also receives key specific "in-validate" messages. When such a message is received, the client checks to see if the aforementioned key has an entry in its local cache. If so, the state of such an entry is now made "in-valid". This ensures that read requests made for the same key in the future are re-directed to server (as pulls) rather than being satisfied by the now "invalid" cache entry.

Clients have the ability to disconnect from server. Just before disconnection it sends a message to server containing the list of keys it read during its lifetime and the number of reads it made for each. This is done with the help of the count values for entries in local cache, and is used by server to compute amortized (per client) maintenance costs.

Performance Evaluation:

In scenario 0, by definition, there is zero work done at the server. Also no additional data structure is maintained at server.

Whenever clients request to read (key, value) pairs from server, the server responds by retrieving the same from the centralized hash table and delivering the result to the client. Hence the only cost incurred by the system is the net pull charge. Where, the "pull charge" accounts for the cost incurred in re-directing read requests from client to server and in communicating the returned result/(key, value) pair from server to client.

The NET PROCESSING OVERHEAD per client is the sum of all pull charges incurred at the server due to that client.

$$\therefore NET\ PROCESSING\ OVERHEAD = NET\ PULL\ CHARGE = \#\ of\ Pulls\ made\ to\ Server * PULLCHARGE$$

Since no additional data structure is maintained at server, NET STORAGE OVERHEAD per client is 0.

$$\therefore NET\ STORAGE\ OVERHEAD = 0$$

The total processing and storage overhead over all clients is calculated as,

$$\therefore TOTAL\ PROCESSING\ OVERHEAD = \sum_{i=1}^{i=\max \#\ of\ Clients} NET\ PROCESSING\ OVERHEAD\ \ for\ Client\ i$$

$$\therefore TOTAL\ STORAGE\ OVERHEAD\ = 0$$

The best case performance for Scenario 0 occurs under conditions when no changes are made to the centralized hash table, and the number of clients associated with the server = 1. Under such conditions, each read request made for the first time gets pulled off the server and any read request made for a previously read (key, value) pair is satisfied using local cache.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
0	r >= R f(C)= 0 N=1	$= R * PULLCHARGE$ = 100R =Θ(R)	0

The worst case performance for Scenario 0 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some N >= 1. Specifically we require f(C) =1/r ,i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client .Under such conditions, each read request by a client results in a pull being made to server, causing the server to incur an extremely high pull charge.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
0	r>=R f(C)= 1/r N>= 1	$= N * r * PULLCHARGE$ =100nr =Θ(Nr)	0

(6) Scenario 1:

High Level Intuition :

In Scenario 1 the server maintains per client pull histories. An implicit assumption is made that a client is interested in all (key,value) pairs that feature in its pull history. When any change is made to a (key, value) pair at server, it pushes updates to all clients interested in it immediately. Thus only when a key is being read for the first time at client, a pull is issued to server , thereafter all read requests are satisfied by local cache entries. As the interests of the clients stabilize with time, all push updates are valid and since valid pushes are free of charge (as opposed to Pulls) Scenario 1 provides a significant performance improvement over Scenario 0.

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete a pull is issued by client to server. All other requests are satisfied by valid cache entries.

3) The Server maintains the pull history for each client and when a (key,value) pair is changed at server, it pushes updates to all interested* clients immediately.

* In Scenario 1a client is interested in a (key, value) pair if he pulled it off server previously. The above assumption is true as client interests in Scenario 1 are required to be stabilized.

Details of Implementation:

Server:

The Server as implemented in Scenario 1 is the same as Scenario 0 with the exception of maintaining per client pull histories.

When a client requests the server for a value corresponding to a given key (issues a pull), the server fetches the same off the hash table and returns it to the client. The requested key is then made a part of the client's pull history. The pull history for each client is a linked list of previously requested keys and is identified by the client's unique ID. In the future when the server modifies a (key, value) pair it will immediately push updates to all clients which contain the aforementioned key in their Pull history. This helps keep the client cache entries current and valid at all times.

It is important to keep in mind that in Scenario 1 the Server is charged for every Pull issued to it, but all pushes are free of charge as the client interests stabilize with time.

Client:

The Client as implemented for Scenario 1 is exactly the same as that in Scenario 0.

However unlike Scenario 0 , clients in Scenario 1 receive regular updates from the server. When an update for a given key is received at client, the clients checks to see if an entry corresponding to the given key is present in local cache; If so the entry is modified to reflect the current update.

Here as the interests of Clients stabilize with time all cache entries are significant and all push updates are necessary.

Performance Evaluation:

In Scenario 1 the server does a certain amount of additional work and bookkeeping in order to reduce the net pull charge . For each active client the server maintains a "pull history". This data structure is a reflection of all the unique pull requests made by a given client in the past. The additional work at server can be categorized into creating and maintaining the pull history while the additional bookkeeping/storage corresponds to the space needed to store the pull history itself. The cost of maintaining the pull history , is the processing burden incurred by the server in pushing updates to interested clients when changes have been made to the centralized hash table. This helps ensure that the client cache entries always up to date.

Thus the net processing overhead per client is calculated as,

$$\therefore \textit{NET PROCESSING OVERHEAD} = \textit{NET PULL CHARGE} + \textit{NET COST FOR CREATING AND MAINTAINING PULL HISTORY}$$

where,

$$\textit{NET PULL CHARGE} = \textit{PULLCHARGE} * \#\ of\ \textit{Pulls made by client to server}$$

$$\textit{COST OF CREATING PULL HISTORY} = \text{PULLCREATIONCOST} \times \#\ of\ \textit{unique Pull requests made by client}$$

$$\text{COST OF MAINTAINING PULL HISTORY} = \text{NET PUSH CHARGE} = \frac{1}{Total\ \#\ of\ read\ requests\ by\ client} \sum_{i=1}^{i=Total\ \#\ of\ changes} (PUSHCHARGE \times boolean\ value\ indicating\ whether\ client\ has\ previously\ requested\ key\ changed\ at\ i)$$

The net storage overhead per client is calculated as,

$$\therefore \textit{NET STORAGE OVERHEAD} = \text{STORAGECOST} \times \#\ of\ \textit{unique Pull requests made by client}$$

The total processing and storage overhead over all clients is calculated as,

$$\therefore TOTAL\ PROCESSING\ OVERHEAD = \sum_{i=1}^{i=\max \#\ of\ Clients} NET\ PROCESSING\ OVERHEAD\ for\ Client\ i$$

$$\therefore TOTAL\ STORAGE\ OVERHEAD = \sum_{i=1}^{i=\max \#\ of\ Clients} NET\ STORAGE\ OVERHEAD\ for\ Client\ i$$

The best case performance for Scenario 1 occurs under conditions when no changes are made to the centralized hash table, and the number of clients associated with the server = 1. Under such conditions, each read request made for the first time gets pulled off the server and any read request made for a

previously read (key, value) pair is satisfied using local cache. Any extra overhead incurred by this scenario as opposed to Scenario 0 is due to the creation of pull history.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
1	r >= R f(C)= 0 N=1	$=R * (PULLCHARGE + PULLCREATIONCOST)$ = 101R =Θ(R)	$=R * STORAGECOST$ =Θ(R)

The worst case performance for Scenario 1 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some n >= 1. Specifically we require f(C) =1/r ,i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client .Under such conditions, the first time a key is read at a client, it is pulled off the server, any read requests made thereafter (for the same key) are satisfied using local cache. This is possible because each time a (key, value) pair is changed at server, an update is sent to all interested* clients immediately.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
1	r>=R f(C)= 1/r N>=1	$= N\ (R * (PULLCHARGE + PULLCREATIONCOST)\ + \frac{1}{r} * PUSHCHARGE\left(\frac{R(R-1)}{2} + R(r-R)\right))$ $^{\$}$ =O(NR)	$=N * R * STORAGECOST$ =Θ(NR)

*In Scenario 1a client is interested in a (key, value) pair if he pulled it off server previously. The above assumption is true as client interests in Scenario 1 are required to be stabilized.

$^{\$}$ Under the conditions for Worst Cast performance, the number of pushes received per client = 1+2+ 3+……+R-1+ ……R+R(r-R times) = $\sum_{i=1}^{i=R-1} i +$ r − R(R) $= R(R-1)/2$ + r-R(R)

(7) Scenario 2:

High Level Intuition :

The operation of Scenario 2 is the same as Scenario 1 except that the interests of clients evolve with time. Hence it may so happen that a client may receive a large number of pushes for keys outside its current interest. Unused pushes cost the Server, and hence under certain circumstances the net cost of sending unused pushes may exceed the net cost of satisfying requests by equivalent pulls and the performance Scenario 2 may deteriorate to be worse than that of Scenario 0.

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete a pull is issued by client to server. All other requests are satisfied by valid cache entries.

3) The Server maintains the pull history for each client and when a (key,value) pair is changed at server, it pushes updates to all interested* clients immediately.

*In Scenario 2, a client is assumed to be interested in a (key, value) pair if he pulled it off server previously. The above assumption is false as client interests in Scenario 2 evolve with time.

Details of Implementation:

Server:

The Server as implemented for Scenario 2 is exactly the same as Scenario 1.

Client:

The Client as implemented for Scenario 2 is exactly the same as that in Scenario 1.

However in case of Scenario 2 where Client interests evolve with time, only cache entries of current interest are significant. Other cache entries are insignificant and push updates for them are unnecessary. This sending of unnecessary information by server leads to an increased system cost in Scenario 2 as compared to Scenario 1.

Performance Evaluation:

The formulae to compute the processing and storage overhead of Scenario 2 is the same as Scenario 1. However the two Scenarios differ in their input conditions and hence differ in the actual amount of cost incurred. In Scenario 1, the interests of the clients have stabilized with time, hence all pushes made from server to client are of client's interest and are ideally charged zero. In Scenario 2, the interests of the clients evolve with time; hence pushes made from server to client may not always serve a purpose and contribute to unnecessary overhead

The best case performance for Scenario 2 occurs under conditions when no changes are made to the centralized hash table, and the number of clients associated with the server = 1. Under such conditions,

each read request made for the first time gets pulled off the server and any read request made for a previously read (key, value) pair is satisfied using local cache. Any extra overhead incurred by this scenario as opposed to Scenario 0 is due to the creation of pull history. The evolving interests of the clients are modeled by the fact that r=c'R, where 1<=c'<=1.5.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
2	r = c' R 1<= c' <= 1.5 f(C)= 0 N=1	$=R * (PULLCHARGE + PULLCREATIONCOST)$ = 101R =Θ(R)	$=R * STORAGECOST$ =Θ(R)

The worst case performance for Scenario 2 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some N >= 1. Specifically we require f(C) =1/r ,i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client .Under such conditions, the first time a key is read at a client, it is pulled off the server, any read requests made thereafter (for the same key) are satisfied using local cache. This is possible because each time a (key, value) pair is changed at server, an update is sent to all interested* clients immediately.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
2	r = c' R 1<= c' <= 1.5 f(C)= 1/r N>=1	$=N\ (R * (PULLCHARGE + PULLCREATIONCOST\ ) + \frac{1}{r} * PUSHCHARGE\left(\frac{R(R-1)}{2} + R(r-R)\right))^{\$}$ =O(NR)	$=N * R * STORAGECOST$ =Θ(NR)

*In Scenario 2, a client is assumed to be interested in a (key, value) pair if he pulled it off server previously. The above assumption is false as client interests in Scenario 2 evolve with time.

$ Under the conditions for Worst Cast performance, the number of pushes received per client = 1+2+ 3+……+R-1+ ……R+R(r-R times) = $\sum_{i=1}^{i=R-1} i +$ r − R(R) $= R(R-1)/2$ + r-R(R)

(8) Scenario 3:

High Level Intuition :

To overcome the drawback of receiving unused pushes, Scenario 3 makes use of 2 additional parameters. One, being the maxLengthOfPullHistory and two, being threshold number of pushes(called DEFAULTPUSHTHRESHOLD). Of the two, maxLengthOfPullHistory is the more important parameter.

The maxLengthOfPullHistory in the ideal case is exactly equal to the number of keys of current interest. In this case, the threshold number of pushes can be as high as infinity. An implicit assumption made here, is that the keys of current interest are always the keys requested most recently in time. This assumption holds good in case of both stabilized and evolving client interests. Since pushes are made based on the keys in pull history, by restricting it to only keys of current interest, we can reduce the number of unused pushes to zero. However, if we end up overestimating/underestimating the length of Pull History by a large margin, we may receive a large number of unused pushes. Under such circumstances it is better to keep the threshold of number of pushes as low as possible (i.e. 1).As it may be more economical to satisfy requests by making required pulls than receiving a large number of unnecessary pushes to keep a few entries (correspond to keys of current interest)of local cache valid.

NOTE:In case of stable requests the maxLengthOfPullHistory and threshold number of pushes can be made equal to some infinity to have the same effect as Scenario 1

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete, a pull is issued by client to server. All other requests are satisfied by valid cache entries.

3) The Server maintains the pull history for each client and orders the keys in the Pull History in a manner such that the head always reflects the oldest Pull and tail the latest. Each key appears at most once in the pull history and the pull history is updated as a consequence of pulls alone.

4) The size of the pull history is restricted to some m; which is a command-line input parameter. Once m has been reached and a new entry has to be made, the oldest entry or the head of the pull history is deleted.

5) When a (key,value) pair is changed at server, it pushes updates to all interested[*] clients immediately. The max number of pushes which can be made by server per client is restricted by a pre-defined parameter n. Once the server has counted to n-1 for a particular client, it does not do the next push, and waits until a pull comes from the client.

6)Once the pull has come, the server: a) resets the n-counter; b) responds to the pull; c) adds to the client's pull history the key at the tail position, dropping if necessary the oldest entry/head from the pull history.

[*] It is assumed in Scenario 3 that a client is interested in a (key, value) pair only if it is a part of its pull history at the time when the check is made. The validity of this assumption depends on the value of maxLengthOfPullHistory

Details of Implementation:

Server:

The Server in Scenario 3 is identical to Scenario 1/2 . However, it does some additional bookkeeping in order to reduce the number of unused pushes when client requests evolve with time. The first extra information maintained at Server in this scenario is the ordered pull history. Here the pull history is ordered such that, the head corresponds to the key pulled earliest in time, the tail corresponds to the one pulled latest in time. The purpose of this is to help the server exercise control over the contents of pull history and to limit it to only keys of current interest or the ones requested most recently in time. This is achieved with the help of a pre-defined parameter maxLengthPullHistory. maxLengthPullHistory, is the longest the pull history of each client can get. When a new entry has to be made to the client's pull history and it is already at its maximum length, then the oldest entry or the head is deleted and the new entry is added at the tail. If the pull history is kept short automatically the number of pushes made by the server per client will also be kept small.

Another additional parameter maintained at the server is the number of pushes made per client. We have a default threshold parameter, DEFAULTPUSHTHRESHOLD, which ensures that at max number of pushes that can be made before receiving a pull is DEAULTPUSHTHRESHOLD-1. Once the maximum number of pushes has been reached, no more pushes are made until a Pull is issued. This ensures that updates are not unnecessarily pushed to clients who have stopped making read requests for (key,value) pairs currently being updated.

Client:

The Client in Scenario 3 is identical to the clients in Scenario 1/2 .

However, clients in Scenario 3 may not always receive an update when a (key,value) pair cached within it is updated. The server has a pre-defined threshold called DEFAULTPUSHTHRESHOLD such that it can never send more than DEFAULTPUSHTHRESHOLD-1 pushes to any one client before receiving a pull from it. Also the length of the pull history (per client) maintained at server is limited, therefore it may so happen that a (key,value) pair is cached at client but it does NOT feature in the server's pull history for it. As the server pushes updates only for (key,value) pairs which feature in a client's pull history, changes made to other (key,value) pairs do not result in updates.

Performance Evaluation:

Scenario 3 (when compared with Scenario 2) does some additional bookkeeping and processing to reduce the number of unused pushes made when client requests evolve with time. The additional bookkeeping includes storing the max length of pull history (m) threshold # of pushes (n), and a count of the number of pushes made per client.While, the additional processing involves, one, creating and maintaining an ordered pull history of length at most 'm'and two, ensuring that no pushes are made (until such time a pull is received) once the threshold of 'n-1' has been reached. The latter operation requires us to check a condition in constant time and is hence not accounted for explicitly, while computing Processing Overhead.

The formula for computing NET STORAGE OVERHEAD and NET PROCESSING OVERHEAD per client is as follows-:

$$\therefore \textit{ NET PROCESSING OVERHEAD = NET PULL CHARGE + NET COST FOR CREATING AND MAINTAINING PULL HISTORY}$$

where,

$$\textit{NET PULL CHARGE} = \textit{PULLCHARGE} * \#\ of\ \textit{Pulls made by client to server}$$

$$\textit{COST OF CREATING PULL HISTORY}^{*} = \sum_{i=1}^{i=\max \#\ of\ \textit{Pull Requests by client}} \text{PULLCREATIONCOST} \times 2 * \textit{Length of Pull History on the completion of Pull Request } i$$

$$\textit{COST OF MAINTAINING PULL HISTORY}^{\#} = \text{NET PUSH CHARGE} = \frac{1}{\textit{Total \# of read requests by client}} \sum_{i=1}^{i=\textit{Total \# of changes}} (\textit{PUSHCHARGE} \times \textit{boolean value indicating whether client is interested in key changed at } i\ )$$

*The cost of Creating Pull history is grossly overestimated here, where we assume it is equal to 2 times the cost to traverse the entire length of pull history post insertion. This is the cost only in the rare case when a duplicate node is present, and is located at the very end of the pull history. In general duplicates are rare and the cost is typically equal to the cost of traversing the length of pull history once.

It is assumed in Scenario 3 that a client is interested in a (key, value) pair only if it is a part of its pull history at the time when the check is made. The validity of this assumption depends on the value of maxLengthOfPullHistory.

$$\therefore \textit{NET STORAGE OVERHEAD} = \text{STORAGECOST} \times \max \textit{length of pull history over all requests made by client} + \textit{STORAGECOSTCOUNT}$$

The total processing and storage overheads over all clients are as follows-:

$$\therefore \textit{ TOTAL PROCESSING OVERHEAD} = \sum_{i=1}^{i=\max \#\ of\ \textit{Clients}} \textit{NET PROCESSING OVERHEAD for Client } i$$

$$\therefore \textit{TOTAL STORAGE OVERHEAD} = \sum_{i=1}^{i=\#\ of\ \textit{Clients}} \textit{NET STORAGE OVERHEAD of Client } i + \textit{STORAGECOSTM} + \textit{STORAGECOSTN}$$

The best case performance of Scenario 3 is similar to Scenario 0, and occurs under conditions where no changes are made to the centralized hash table and the number of clients associated with the server equals 1. Since we know that f(C) = 0, we can adjust the maxLengthPullHistory / m to be 0 as there are no changes made to the centralized hash table for which updates need to be pushed to clients. This helps bring down the pull history creation and maintenance cost to zero giving us a performance improvement over Scenario 1 & 2.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
3	r >= R f(C)= 0 N= 1 maxLengthPullHistory = 0 DEFAULTPUSHTHRESHOLD = 1	$= R * PULLCHARGE$ = 100R =Θ(R)	$=O(1)$

The worst case performance for Scenario 3 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some N >= 1. Specifically we require f(C) =1/r ,i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client . Under extreme these conditions (high value of f(C)), it is cheaper to make a required pull than to make pushes for all changed keys immediately. By choosing maxLengthPullHistory/ m and DEFAULTPUSHTHRESHOLD/n to be ∞ as we can ensure that whenever possible pushes are made in place of pulls. This guarantees worst case performance and hence, the first time any key is read at a client, it is pulled off the server, and any read request made thereafter (for the same key) is satisfied using local cache.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
3	r >= R f(C)= 1/r maxLengthPullHistory = ∞ DEFAULTPUSHTHRESHOLD = ∞ N>=1	$=N(R * PULLCHARGE + R(R+1) * (PULLCREATIONCOST) + \frac{1}{r} * PUSHCHARGE\left(\frac{R(R-1)}{2} + R(r-R)\right))^{\$}$ $=O(NR^2)$	$=N * R * STORAGECOST + (N+2) * O(1)^{\%}$ =Θ(NR)

% The O (1) storage costs correspond to the constant amount of memory spent in storing parameters m , n and a count of the # of pushes made per client.

[$] Under the conditions for Worst Cast performance, the lengths of pull history traversed ,post pull history creation = 1+ 2+ 3…+R = $\sum_{i=1}^{i=R} i = \frac{R(R+1)}{2}$ and thus the cost of Pull history Creation = 2* Length of pull history ,post insertion = (R+1) *PULLCREATIONCOST

(9) Scenario 4:

High Level Intuition :

Scenario 4 proposes another solution to minimize the number of unused pushes. The solution is based on a system wide accepted value called THRESHOLD TIME such that, any entry in pull history time stamped, at current time- THRESHOLD TIME or earlier is considered stale and eliminated. If THRESHOLD TIME is too small then the solution proposed would be identical to Scenario 0. If THRESHOLD TIME is too large then the solution would have the same effect as Scenario 2. Hence in the ideal case THRESHOLD TIME should be chosen to be the minimum possible value such that any key pulled at time current time -THRESHOLD TIME or earlier would no longer be of client's current interest and can be eliminated. Thus by limiting the Pull History to only keys of client interest we can minimize the number of unused pushes, while simultaneously keeping the number of pulls as low as possible.

NOTE:In case of stable requests the THRESHOLD TIME can made equal to some infinity to have the same effect as Scenario 1

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete a pull is issued by client to server. All other requests are satisfied by valid cache entries.

3) The Server maintains the pull history for each client and each key in the pull history is associated with the timestamp of its last pull(by the given client). Each key appears at most once in the pull history and the pull history is updated on pulls and pushes.

4) When a (key,value) pair is changed at server, it pushes updates to all interested* clients immediately.

5)There is a single system-wide threshold for the staleness (relative) of a timestamp in pull history and each time the server scans the pull histories of clients(on a pull or before a push), it will piggyback on this scanning to drop all stale entries in pull history.

* In Scenario 4 a client is interested in a (key, value) pair, if the key is a part of its recorded pull history at the time the check is made. The validity of this assumption depends on the accuracy of THRESHOLDTIME.

Details of Implementation:

Server:

The Server in Scenario 4 is identical to Scenario 1/2. However, the Server in Scenario 4 does additional bookkeeping in order to reduce the number of unused pushes made when clients requests evolve with time. The first extra information maintained at server in this scenario, is the time stamped pull history. Here each key in pull history is associated with the time stamp of when the last pull for the same was made. Also ,the server holds a pre-defined a universal constant ,THRESHOLDTIME to determine staleness of pull entries. A stale/obsolete entry is one whose associated time stamp is current time –

THRESHOLDTIME or earlier. Since the server regularly scans the Pull history (one, when a (key,value) pair is changed at hash store and two, when a pull from the corresponding client arrives), each time it does so it can automatically discard all pull entries which are now obsolete. Thus by this approach the pull history is always limited to those keys which have been requested most recently in time, and are hence of client's current interest.

Since pushes are made to clients based on their pull histories, limiting the pull histories of clients to the most recently requested keys also limits the number of unused push updates made and in the ideal case reduces them to zero.

Client:

The Client in Scenario 4 is identical to the clients in Scenario 1/2.

However, clients in Scenario 4 may not always receive updates when a (key,value) pairs cached within them are changed. This can be attributed to the fact that the Server in Scenario 4 always limits the pull histories of clients to keys which have been pulled most recently in time(Current time – THRESHOLDTIME or earlier) and only pushes updates for the same.

Performance Evaluation:

Scenario 4 (when compared with Scenario 2) does some additional bookkeeping and processing to reduce the number of unused pushes made when client requests evolve with time.

The additional bookkeeping involves storing a threshold value, THRESHOLDTIME to indicate staleness, and the additional processing involves creating and maintaining a pull history containing only those entries which are fresh (i.e. entries whose associated timestamp is strictly earlier than Current Time-THRESHOLDTIME). The formula for computing NET PROCESSING OVERHEAD and NET STORAGE OVERHEAD per client is as given below-:

$$\therefore \textit{ NET PROCESSING OVERHEAD = NET PULL CHARGE + NET COST FOR CREATING AND MAINTAINING PULL HISTORY}$$

where,

$$\textit{NET PULL CHARGE} = PULLCHARGE * \#\ of\ Pulls\ made\ by\ client\ to\ server$$

$$\textit{COST OF CREATING PULL HISTORY} = \sum_{i=1}^{i=\max \#\ of\ Pull\ Requests\ by\ client} \text{PULLCREATIONCOST} \times Length\ of\ Pull\ History\ on\ completion\ of\ Pull\ Request\ i$$

$$\textit{COST OF MAINTAINING PULL HISTORY}^{\#} = \text{NET PUSH CHARGE} =$$

$$\frac{1}{Total\ \#\ of\ read\ requests\ by\ client} \sum_{i=1}^{i=Total\ \#\ of\ changes} (PUSHCHARGE \times boolean\ value\ indicating\ whether\ client\ is\ interested\ in\ key\ changed\ at\ i\ )$$

$$\therefore NET\ STORAGE\ OVERHEAD = STORAGECOST \times \max length\ of\ client\ pull\ history\ \ over\ entire\ service\ duration$$

[#]A client is interested in a (key, value) pair in Scenario 4 only if it is a part of its pull history at the time when the check is made. The validity of this assumption depends on the accuracy of THRESHOLDTIME.

The total processing and storage overheads over all clients are as follows-:

$$\therefore TOTAL\ PROCESSING\ OVERHEAD = \sum_{i=1}^{i=\max \#\ of\ Clients} NET\ PROCESSING\ OVERHEAD\ \ of\ client\ i$$

$$\therefore TOTAL\ STORAGE\ OVERHEAD = \sum_{i=1}^{i=\#\ of\ Clients}(NET\ STORAGE\ OVERHEAD\ of\ client\ i\ ) + \text{STORAGECOSTTHRESHOLD}$$

The best case performance of Scenario 4 occurs under conditions when no changes are made to the centralized hash table and the number of clients associated with server = 1. As there are no changes being made to the hash table, we have no updates to send and hence no information about past requests needs to be maintained. Thus, the THRESHOLDTIME is chosen to be infinitesimally small such that, every pull request deletes all previously requested keys from the client's pull history (if any) such that the NET STORAGE OVERHEAD at server per client at is at most one. This gives us a performance (with respect to storage) similar to Scenario 0.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
4	THRESHOLDTIME << 1/r r >= R f(C)= 0 N=1	$=R * PULLCHARGE + (R-1) * (2 * PULLCREATIONCOST) + PULLCREATIONCOST$ = R(102) -1 = Θ(R)	= STORAGECOSTTHRESHOLD = O(1)

The worst case performance for Scenario 4 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some N >= 1. Specifically we require f(C) =1/r, i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client.

As THRESHOLDTIME is chosen to be ∞, no key is ever deleted from a client's pull history, and clients receive updates when any (key, value) pair they requested in the past is changed. Thus under the extreme condition, where f(C) = 1/r, between any 2 read requests issued by a client, it receives as many updates as the current length of its pull history.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
4	THRESHOLDTIME = ∞	$=N(R * PULLCHARGE + \frac{R(R+1)}{2} *$	=NR + STORAGECOSTTHRESHOLD

	r >= R f(C)= 1/ r n>=1	$(PULLCREATIONCOST) + \frac{1}{r} * PUSHCHARGE\left(\frac{R(R-1)}{2} + R(r-R)\right)$ $=O(NR^2)$	=Θ(NR)

(10) Scenario 5:

High Level Intuition :

In Scenario 5 deferred updates are made in place of immediate updates. When client's interests stabilize with time, all updates are valid and all pushes are free of charge. Under such circumstances, deferred updates cannot provide a performance improvement over immediate updates. However by ensuring periodic updates occur often enough (small values of DEFAULTTIMEOUT) Scenario 5 can be made to perform equivalent to Scenario 1.

When client interests evolve with time, unused pushes pose a problem. By applying deferred updates in place of immediate updates, the cost of pushes (unused) can be minimized(due to the benefits of batching). In the extreme case where it may be more economical to satisfy requests by making equivalent pulls than receiving a large number of (unused)pushes, Scenario 5 can be tweaked (DEFAULTTIMEOUT= ∞) to perform equivalent to Scenario 0, where the number of pushes made equal zero.

Pseudo Code:

1) Client starts by pulling requests from server into an empty cache.

2) When a (key,value) pair to be read is missing in the client cache or is declared obsolete a pull is issued by client to server. All other requests are satisfied by valid cache entries.

3) The Server maintains the pull history for each client and each key in the pull history is associated with a field which indicates whether this (key,value) pair has been changed since the last pull/push("changed", "unchanged").

4) When a client pulls an entry from server. The key is made a part of client pull history (if not already present) and its associated field is set to "unchanged" to indicate that no changes have been made to the key since its last pull.

5) When a (key,value) pair is changed at server, the pull histories of all clients containing this key are updated to reflect the now "changed" state.

6) The Server periodically scans the pull history of all clients .If there are 1 or more keys in a client's pull history which have been changed since the last push/pull, the server pushes updates for all such keys at once via batching .The associated fields of these keys are then set to reflect the current "unchanged" state.

Details of Implementation:

Server: The Server in Scenario 5 is identical to Scenario 1/ 2. However ,it performs deferred updates in place of immediate updates. To facilitate the above operation, each key in a client's pull history is associated with a field indicating whether a given key has been changed since its last push/pull. When a (key,value) pair is changed at server, the pull histories of all clients containing this key are updated to reflect the now "changed" state. Conversely, when a pull for a given key arrives from a client, the server adds the key to corresponding pull history(if not already present) and sets its state to "unchanged".

Periodically the Server checks the pull histories of all clients, if there are 1 or more keys in a given client's pull history which have been changed since the last push/pull, the server pushes updates for all such keys simultaneously (batching). It then sets their associated fields to reflect the current "unchanged" state. This periodic scanning by Server is done at the rate of once per DEFAULTTIMEOUT, where DEFAULTTIMEOUT, is a pre-determined constant.

Client:

The Client in Scenario 5 is identical to the clients in Scenario 1/2.

However, clients in Scenario 5 do not receive immediate updates when a (key, value) pairs cached within them are changed. This can be attributed to the fact that the Server in Scenario 5 pushes updates periodically rather than as an immediate consequence of changes being made to the centralized hash table.

Performance Evaluation:

In Scenario 5, we perform deferred updates in place of immediate updates. Hence, we do not push updates as an immediate consequence of changes made to the centralized hash table. Rather, we periodically (every DEFAULTTIMEOUT) scan the pull histories of all clients, and push updates to only those clients who contain one or more changed keys in their pull histories. Thus the additional work done at server in Scenario 5 (when compared with Scenario 0) is the periodic scanning of pull histories followed by possible pushing of updates. The additional bookkeeping at Scenario 5(when compared with Scenario 0) corresponds to storing a per client "pull history". The "pull history" is a list of all unique keys requested by a client in the past and each key in the pull history is associated with a Boolean value indicating whether it has been changed since last pull/push.

Thus the net processing overhead per client is calculated as,

$$\therefore \textit{NET PROCESSING OVERHEAD} = \textit{NET PULL CHARGE} + \textit{NET COST FOR CREATING AND MAINTAINING PULL HISTORY}$$

where,

$$\textit{NET PULL CHARGE} = \textit{PULLCHARGE} * \#\ of\ \textit{Pulls made by client to server}$$

$$\textit{COST OF CREATING PULL HISTORY} = \textit{PULLCREATIONCOST} \times \sum_{i=1}^{i=\max \#\ of\ \textit{Pull Requests by client}} \text{PULLCREATIONCOST} \times \textit{Length of Pull History on completion of Pull Request } i$$

$$\textit{COST OF MAINTAINING PULL HISTORY} = \frac{1}{\textit{Total \# of read requests by client}} * (\textit{NET PUSH CHARGE} + \textit{NET COST FOR SCANNING PULL HISTORY ENTRIES})$$

$$NET\ COST\ OF\ SCANNING\ PULL\ HISTORY\ ENTRIES^{*}$$
$$= \sum_{i=1}^{i=\max \#\ of\ scans\ in\ client\ lifetime} PULLMAINTENANCECOST \times 2 * length\ of\ Pull\ History\ at\ the\ time\ of\ scan\ i$$

$$NET\ PUSH\ CHARGE$$
$$= \sum_{i=1}^{i=\max \#\ of\ scans\ in\ client\ lifetime} Cost\ of\ a\ single\ push \times boolean\ value\ indicating\ if\ push\ was\ made\ during\ scan\ i$$

It should be noted that Cost of a single push in Scenario 5 is >=PUSHCHARGE depending on the number of keys for which an update is being pushed.

$$Cost\ of\ a\ single\ push = PUSHCHARGE + (COSTOFADDEDPUSH * \#\ of\ keys\ for\ which\ an\ update\ is\ being\ pushed - 1)$$

The net storage overhead per client is calculated as,

$$\therefore NET\ STORAGE\ OVERHEAD = STORAGECOST \times \#\ of\ unique\ Pull\ requests\ made\ by\ client$$

*The cost of Scanning Pull history is grossly overestimated here, where we assume it is equal to 2 times the cost of traversing the entire length of pull history each time it is scanned. This is the cost only in the rare case when there the one node to be updated is located at the very end of pull history. Typically the cost is closer to traversing the length of pull history 1.5 times.

The total processing and storage overhead over all clients is calculated as,

$$\therefore TOTAL\ PROCESSING\ OVERHEAD = \sum_{i=1}^{i=\max \#\ of\ Clients} NET\ PROCESSING\ OVERHEAD\ of\ client\ i$$

$$\therefore TOTAL\ STORAGE\ OVERHEAD = \sum_{i=1}^{i=\#\ of\ Clients} NET\ STORAGE\ OVERHEAD\ of\ client\ i + STORAGECOSTDEFAULTTIMEOUT$$

The best case performance of Scenario 5 occurs under conditions when no changes are made to the centralized hash table and the number of clients associated with server = 1. As there are no changes being made to the hash table, we have no updates to send and hence the time period to scan for deferred updates can be kept as high as ∞, giving us a performance as close as possible to Scenario 1.

Scenario #	Conditions for Best Case	Total Processing Overhead for Best Case	Total Storage Overhead for Best Case
5	DEFAULTTIMEOUT	$= R * PULLCHARGE +$	$=R * STORAGECOST\ +$

	= ∞ r>=R f(C) = 0 N=1	$(\frac{R(R+1)}{2} * PULLCREATIONCOST)$ $=O(R^2)$	$STORAGECOSTDEFAULTTIMEOUT$ $=\Theta(R)$

The worst case performance for Scenario 5 occurs when the (key, value) pairs stored at the centralized hash table are changed with a high frequency, and the number of clients associated with server is some N >= 1. Specifically we require f(C) =1/r, i.e. all (key, value) pairs are changed exactly once between any two read requests issued by a client.

Since DEFAULTTIMEOUT is chosen as the time to change 1 (key, value) pair in the hash table, and as all hash table entries are changed between any 2 read requests made by a client, the number of scans made between any 2 read requests issued by client is equal to size of the hash table i.e. STORESIZE. At each timeout, the entire pull history is at max scanned twice and possibly updates for one or more keys are issued. However, we scan the pull history after each change, and hence there is at most one key updated in pull history since last scan, and thus rather than gaining the advantage of batching, each time only an update for a single key is pushed. The pushing of individual updates and high rate of scanning under conditions where f(C) itself is high gives the worst case performance of Scenario 5.

Scenario #	Conditions for Worst Case	Total Processing Overhead for Worst Case	Total Storage Overhead for Worst Case
5	DEFAULTTIMEOUT = time to change 1 (key,value) pair in the entire hash table r>=R f(C) = 1/r N>=1	$=N(R * PULLCHARGE + \frac{R(R+1)}{2} * (PULLCREATIONCOST) + \frac{1}{r} * PUSHCHARGE\left(\frac{R(R-1)}{2} + R(r-R)\right) + \frac{1}{r} * PULLMAINTENANCECOST * STORESIZE(R(R-1) + 2 * R(r-R)))^{\#}$ $=O(NR^2)$	=NR + $STORAGECOSTDEFAULTTIMEOUT$ $=\Theta(NR)$

#Under conditions of the worst case performance the PULLMAINTEANCECOST is multiplied by parameter STORESIZE, as STORESIZE is the maximum number of scans which can be performed between any 2 read requests made by the client.

(11) Experimental Results:

In this section, via a series of test cases, we developed niche cases for each scenario. For each test case we ran, we picked the winning scenario, i.e .the one which performed best under the given set of test conditions and designated it to be the niche case for that scenario. The aim of these experiments/test cases was to develop a feel/intuition for the performance of each scenario. Now that we are done with our experiments, given any arbitrary set of input conditions, we can predict with reasonable accuracy the scenario (among all the scenarios being proposed) which will perform best under the aforementioned conditions.

TEST 1: Stable requests with no modifications at server

Time stamped sequence of operations performed in TEST 1:

Time	Pulls/Reads at Client in Scenario 0	Action at Server
1	Read A	
2	Read B	
3	Read C	
4	Read A	
5	Read B	
6	Read C	
7	Read A	

Numerical Results *:

Scenario #	Net Processing Overhead	Net Storage Overhead
Scenario 0	300	0
Scenario 1	303	3
Scenario 2	Not applicable for stabilized requests	
Scenario 3(Parameters set to model Scenario 0, i.e maxLengthPullHistory = 0, DEFAULTPUSHTHRESHOLD= ∞)	303	1
Scenario 4(Parameters set to model Scenario 0, i.e THRESHOLDTIME = 0.1)	305	1
Scenario 5(Parameters set to model Scenario 0, i.e DEFAULTTIMEOUT = ∞)	306	3

Inference: Scenario 0 wins, as it is the one which incurs the least cost among all scenarios being proposed. This can be attributed to the fact that Scenario 0 makes the least investment among all scenarios to reduce the cost of cache maintenance, which is zero by default for the conditions of this test case.

Behavior of Winning Scenario:

Time	Pulls/Reads	Action at Server
1	Pull A	
2	Pull B	
3	Pull C	
4	Read A	
5	Read B	
6	Read C	
7	Read A	

TEST 2: Stable requests with few updates to keys between read requests for the same (key, value) pairs at client.

Time stamped sequence of operations performed in TEST 2:

Time	Pulls/Reads	Action at Server
1	Read A	
2	Read B	
3	Read C	
4		Update A
5		Update B
6		Update C
7	Read A	
8	Read B	
9	Read C	
10	Read A	

Numerical Results [*]:

Scenario #	Net Processing Overhead	Net Storage Overhead
Scenario 0	600	0
Scenario 1	315.8571	3
Scenario 2	Not applicable for stabilized requests	

Scenario 3(Parameters set to model Scenario 1, i.e maxLengthPullHistory = ∞ ,DEFAULTPUSHTHRESHOLD= ∞)	318.8571	4
Scenario 4(Parameters set to model Scenario 1, i.e THRESHOLDTIME = ∞)	318.8571	3
Scenario 5(Parameters set to model Scenario 1, i.e DEFAULTTIMEOUT = 1.5(largest value of timeout such that a key pulled in the past need not be pulled again at client))	323.7142	3

Inference: Scenario 1 wins, as it is the one which incurs the least cost among all scenarios being proposed. This can be attributed to the fact that Scenario 1, under the conditions of this test case, makes no unused pushes and reads as much as possible from local cache. All requests for a given key (except the first) are satisfied by local cache.

Behavior of Winning Scenario:

Time	Pulls/Reads	Action at Server	Pull History at Server
1	Pull A		A
2	Pull B		B->A
3	Pull C		C->B->A
4		Update A(Push A)	‘’
5		Update B(Push B)	‘’
6		Update C(Push C)	‘’
7	Read A		‘’
8	Read B		
9	Read C		‘’
10	Read A		‘’

TEST 3: Stable Requests with many updates to key between read requests for the same (key, value) pair at client.

Time stamped sequence of operations performed in TEST 3:

Time	Pulls/Reads	Action at

		Server
1	Read A	
2	Read B	
3	Read C	
4		Update A
5		Update B
6		Update C
7		Update A
8		Update B
9		Update C
10	Read A	
11	Read B	
12	Read C	

Numerical Results [*]:

Scenario #	Net Processing Overhead	Net Storage Overhead
Scenario 0	600	0
Scenario 1	333	3
Scenario 2	Not applicable for stabilized requests	
Scenario 3(Parameters set to model Scenario 5, i.e maxLengthPullHistory = ∞ ,DEFAULTPUSHTHRESHOLD= ∞)	336	4
Scenario 4(Parameters set to model Scenario 5, i.e THRESHOLDTIME = ∞)	336	3
Scenario 5(Parameters set to provide an improvement over Scenario 1, i.e DEFAULTTIMEOUT = 2.5(largest value of timeout such that a key pulled in the past need not be pulled again at client))	331	3

Inference: Scenario 5 wins as it is the one which incurs the least cost among all scenarios being proposed. This can be attributed to the fact that Scenario 5, makes periodic updates in place of immediate updates, and is thus able to push updates to more than a key at once, gaining the advantage of batching. Batching helps reduce the cost of cache maintenance.

Behavior of Winning Scenario:

Time	Pulls/Reads	Action at Server	Pull History at Server
1	Pull A		A(U)
2	Pull B		A(U)->B(U)
2.5	Scanning all Pull Histories to make deferred updates		
3	Pull C		A(U)-> B(U)->C(U)
4		Update A	A(C)-> B(U)->C(U)
5		Update B	A(C)-> B(C)->C(U)
5	Scanning all Pull Histories to make deferred updates ; Push A,B		
6		Update C	A(U)-> B(U)->C(C)
7		Update A	A(C)-> B(U)->C(C)
7.5	Scanning all Pull Histories to make deferred updates ; Push C,A		
8		Update B	A(U)-> B(C)->C(U)
9		Update C	A(U)-> B(C)->C(C)
10	Read A		
10	Scanning all Pull Histories to make deferred updates ; Push B,C		
11	Read B		A(U)-> B(U)->C(U)
12	Read C		‘’

TEST 4: Evolving Requests with Modifications at Server

Time stamped sequence of operations performed in TEST 4:

Time	Pulls/Reads	Action at Server
1	Read A	
2	Read B	
3	Read C	

4		Update A
5		Update B
6		Update C
7		Update D
8		Update E
9		Update F
10	Read C	
11	Read D	
12	Read E	
13		Update A
14		Update B
15		Update C
16		Update D
17		Update E
18		Update F
19	Read E	
20	Read F	

Numerical Results [*]:

Scenario #	Net Processing Overhead	Net Storage Overhead
Scenario 0	800	0
Scenario 1	Not applicable for evolving requests	
Scenario 2	636	6
Scenario 3(Parameters set to provide an improvement over Scenario 2, i.e maxLengthPullHistory = 1(number of keys of interest from the past) ,DEFAULTPUSHTHRESHOLD= ∞)	628.5	2
Scenario 4(Parameters set to provide an improvement over Scenario 2, i.e THRESHOLDTIME = 5.5(the minimum possible value such that no key requested at time current time – THRESHOLDTIME or earlier is of current interest))	628.75	3

Scenario 5(Parameters set to provide an improvement over Scenario 0, i.e DEFAULTTIMEOUT = 1.5(largest value of timeout such that a key pulled in the past need not be pulled again at client))	657.25	6

Inference: Scenario 3 and 4 win as they are the ones which incur the least cost among all scenarios being proposed. This is because both Scenario 3 and Scenario 4 make an effort to reduce the number of unused pushes made when client requests evolve with time, thereby reducing the cost of cache maintenance.

Behavior of Winning Scenarios:

Scenario 3:

Time	Pulls/Reads	Action at Server	Pull History at Server
1	Pull A		A
2	Pull B		B
3	Pull C		C
4		Update A	‘’
5		Update B	‘’
6		Update C (Push C)	‘’
7		Update D	‘’
8		Update E	‘’
9		Update F	‘’
10	Read C		‘’
11	Pull D		D
12	Pull E		E
13		Update A	‘’
14		Update B	‘’
15		Update C	‘’
16		Update D	‘’
17		Update E (Push E)	‘’
18		Update F	‘’
19	Read E		‘’
20	Pull F		F

Scenario 4:

Time	Pulls/Reads	Action at Server	Pull History at Server
1	Pull A		A(1)
2	Pull B		A(1)->B(2)
3	Pull C		A(1)-> B(2) -> C(3)
4		Update A (Push A)	‘’
5		Update B (Push B)	‘’
6		Update C (Push C)	‘’
7		Update D	B(2)-> C(3)
8		Update E	C(3)
9		Update F	
10	Read C		
11	Pull D		D(11)
12	Pull E		D(11)-> E(12)
13		Update A	‘’
14		Update B	‘’
15		Update C	‘’
16		Update D (Push D)	‘’
17		Update E (Push E)	E(12)
18		Update F	
19	Read E		
20	Pull F		F(13)

*The numerical results stated above are an accurate estimate of the costs incurred per client per scenario. In fact, the formula to compute the pull creation cost in certain scenarios is an overestimation of the actual cost incurred in most cases. However, all numerical data presented here are accurate measurements and no approximation has been performed. However, it should be noted that the Net Processing Overhead and Net Storage Overhead mentioned above are the costs incurred per client and not the costs computed over all clients in the system, hence they do not include system wide O(1) costs such as STORAGECOSTTHRESHOLD, STORAGECOSTM etc.

(12)Conclusion:

The fundamental problem being investigated is the problem of data consistency in a distributed system setup. Given a system of N>=1 client computers, interested in a common centralized hash table, we wish to design one more techniques to facilitate easy and inexpensive access to the aforementioned table by taking advantage of local client caches. Through the course of this technical report, we have studied “the problem of coherence of a distributed hash table” (under the realm stated assumptions) in detail and proposed as many as five solutions for the same.

The objective of this report is to not provide one solution to the problem at hand, but to rather develop a suite of solutions where one is more appropriate than others based on applied input conditions. To achieve the above objective, the performance of each solution has been quantized, and the solutions have been implemented and tested. Based on the costs incurred by the different solutions under different input conditions, we have developed a good feel/intuition of the performance of each algorithm, and given a set of test input conditions, we can now predict (with a reasonable degree of accuracy) the algorithm which would perform the best, under the aforementioned conditions.

(13)Future Work:

This technical report studies "the problem of coherence in a distributed hash table", in detail, but does so under a realm of stated assumptions. The assumptions limit the scope of the problem, and changes in assumptions stated, can broaden or enhance the scope of the problem.

However, even within the realm of stated assumptions there are a few problems which are yet to be investigated. Firstly, though we have proposed few solutions, which achieve cache consistency, by taking advantage of the temporal locality of user queries, we are yet to propose solutions which take advantage of spatial locality of user queries. Secondly, we have implicitly assumed that strong consistency is expected by all user queries, however not all applications require strong consistency of user queries and under certain conditions slightly stale data can be tolerated. Thirdly, the performances of all solutions proposed thus far have been thoroughly analyzed from the point of processing overhead. However, the storage overhead (or memory requirements) of the algorithms have not been considered as important. Hence modifying existing solutions to overcome the aforementioned limitations would be a good line of future work.

A second line of future work can be achieved by relaxing the assumptions which limit the scope of the problem. Changes to stated assumptions could include but not be limited to, one, supporting both reads and writes at clients and two, removing the assumption which obviates the need to worry about concurrency while achieving consistency.

References :

[1] R. Chirkova and J. Yang, "Materialized Views," *Foundations and Trends in Databases*, vol. 4, no. 4, pp. 295–405, 2011.

[2] R. Lawrence, "A Survey of Cache Coherence Mechanisms in Shared Memory Multiprocessors", May 1998.

[3] S. Dar, M. J. Franklin, B. T. J´onsson, D. Srivastava, and M. Tan, "Semantic data caching and replacement," in *Proceedings of the 1996 International Conference on Very Large Data Bases*, pp. 330–341, Mumbai (Bombay), India, September 1996.

[4] A. Labrinidis, Q. Luo, J. Xu, and W. Xue, "Caching and materialization for web databases," *Foundations and Trends in Databases*, vol. 2, no. 3, pp. 169–266, 2009.