In: Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD2010), Miami, FL, USA, July 2010.

Flexible Data Access in a Cloud based on Freshness Requirements

Laura Cristiana Voicu, Data and Information Architecture, Credit Suisse, laura.c.voicu@credit-suisse.com*
Heiko Schuldt, Department of Computer Science, University of Basel, Switzerland, heiko.schuldt@unibas.ch
Yuri Breitbart, Department of Computer Science, Kent University, USA, yuri@cs.kent.edu
Hans-Jörg Schek, Professor Emeritus, ETH Zurich, Switzerland, schek@inf.ethz.ch

Abstract—**Data clouds are newly emerging environments in which commercial providers manage large volumes of data with individual quality of service (QoS) guarantees per customer. These guarantees mainly include keeping several replicas of each data item in different distributed data centers for availability purposes. However, as the cost of maintaining several updateable replicas per data object is very high, cloud providers rather offer only a limited number of synchronously updated replicas (i.e., replicas that are always up-to-date) together with several read-only replicas that are updated in a lazy way and thus might hold stale data. QoS agreements may also include the maintenance of dedicated archives (copies of data which are frozen at some point in time). Stale data allow cloud providers to offer a variety of read operations with different semantics, e.g., read the most recent data, read data not older than / not younger than some timestamp t, or read data produced between t_1 and t_2, or read data exactly as of t. These read operations can be supported by a read-only site using a stale replica. In this paper we present our approach to cloud data management, based on a recent protocol for data grids. We discuss in detail how the refresh of individual replicas is provided in a completely distributed way. Finally, we present the results of a performance evaluation in a data cloud setting.**

I. Introduction

Over the last years, cloud-based computing has become a very popular paradigm in industry [1]. It allows companies to rent hardware and/or software resources and helps them avoid major investments that would be needed for building and maintaining computing centers in-house as their ICT infrastructure can be partially or even completely outsourced to providers of cloud services. These cloud providers usually maintain different distributed data centers and offer (shares of) the resources therein according to dedicated quality-of-service (QoS) guarantees per customer. Typically, these guarantees include elastic behavior and a high degree of availability (replication across different data centers). Current approaches in cloud settings that deal with concurrent updates of replicas at different sites either use well-established protocols with blocking behavior (strict two-phase locking combined with two-phase commit), or relax ACID properties to increase the overall performance while reducing the costs of replica maintenance (e.g., PNUTS [8]). In the latter case, serializable executions are avoided and clients access stale data – but without any guarantees on the data staleness / freshness.

* The work has been done while the first author worked at the Department of Computer Science, University of Basel.

A more flexible approach is to offer only a limited number of synchronously updated replicas (always up-to-date) and several read-only replicas that are lazily updated (holding stale data). In particular, keeping several replicas of a data item with different levels of freshness can be highly beneficial in a data cloud since freshness can be exploited for replica selection. From the point of view of cloud providers, stale read-only replicas allow to trade latency and access time for freshness and make profitable use of available resources.

Consider the following application. Orinoco, a large global online bookseller, uses cloud resources to store their business-related data in the cloud. A contract with the cloud provider specifies the degree of availability of the data and the allowed access latency for clients independent of their location. Clients' data accesses include browsing book reviews, checking the availability of books, ordering books using credit cards, or tracking the progress of an order. Some of these reads demand the most up-to-date data while others can live with outdated data (e.g., the exact number of copies of a book in stock is not relevant; the number as of a few minutes ago is sufficient for most customers). Rather than keeping the complete history (versions) of data, and in order to fulfill the QoS agreements with Orinoco with limited resources, only a few updateable replicas are kept at different data centers and managed such that they can serve all operations that need to access up-to-date data. In addition, several read-only replicas are provided with as-fresh-as-possible (but not necessarily the most up-to-date) data for serving requests demanding moderately stale data (e.g., availability of books). Additional replicas which are less frequently updated are kept for read operations that need older data (e.g., time series). Assuming that these operations are less frequent, only few old replicas with coarsely distributed, low freshness levels have to be kept. This is a tradeoff between the potentially necessary and costly refresh operations, and the cost of keeping a larger number of replicas. In addition, as old replicas might also be eventually subject to refresh operations, some selected versions might be archived.

In our previous work we have developed the Re:GRIDiT system which proposes new protocols for the provably correct synchronization of concurrent updates to different updateable replicas in a data grid and their subsequent propagation to read-only replicas in a completely distributed way. For this, we employ a combination of eager and lazy replication protocols that take into account different levels of freshness. At the same time, Re:GRIDiT takes into account different semantics of data: mutable data can be subject to updates; immutable data, in turn, cannot be changed once created, but may be under version control. Furthermore, new replicas (updateable or read-only) may be created and/or removed on demand, according to current load and freshness metrics. The Re:GRIDiT protocols have described the details of distributed synchronization of updates [24], [25] and dynamic reconfigurations [23]. Data grids have the potential to provide valuable support for cloud data management problems and may act as starting point for novel data cloud infrastructures [22]. In this paper we take one step further and present Re:FRESHiT, a novel protocol capable of handling the propagation of updates to read-only nodes and the freshness-aware routing of requests in data clouds. Re:FRESHiT's unique features include the possibility to access up-to-date data as well as data with any freshness level – without keeping a complete history of all versions that have been created. Read-only sites are organized in an overlay tree structure, based on freshness. This paper discusses in detail how the trees are built and maintained, and how the refresh of replicas is provided in a completely distributed way. Finally, we present the results of a performance evaluation in a data cloud setting.

The remainder of the paper is organized as follows. In Section II we explain our system model. Section III gives details of the Re:FRESHiT protocol for freshness-aware read-only access to data, and Section IV presents the experiments that evaluate the performance of our protocol. Section V discusses related work and Section VI concludes.

II. System Model

The Re:GRIDiT suite of protocols targets problematic aspects of infrastructures that deal with huge data volumes. Our first focus has been on the complex and general case of distributed update transactions on replicated data. We have devised a protocol for the correct synchronization of concurrent updates to different updateable replicas that ensures their subsequent propagation to read-only replicas in a completely distributed way [24], [25]. This approach has been enhanced with efficient algorithms for selecting optimal replica placement locations so that the replica load is balanced [23].

The combination of these two protocols dictates how update sites behave and from a user's point of view the clients always access the most up-to-date data. We now refine this approach and introduce the Re:FRESHiT protocol, which allows to effectively trade freshness for performance and addresses freshness issues, without losing consistency.

A. Overview of the System Model

We assume Re:FRESHiT to be part of the middleware present on each site. The sites hold data objects, replicated in the network, and operations through which the data objects can be accessed. Depending on the access to data, the sites are divided between *update* and *read-only* sites [3], [17]. Read-only sites only allow read access to data. Updates occur on the update sites and are synchronized among the update sites and finally propagated to the read-only sites. We assume an eager replication among update sites and in addition we apply lazy replication mechanisms between update and read-only sites that take into account different levels of freshness.

A data object can be stored in a file on the local file system, or inside a database on the local database management system. Re:FRESHiT is capable of supporting arbitrary physical layouts ranging from full replication at the granularity of complete databases to partial replication. The partial replication scheme does not require all data to be replicated on each site. Smaller subsets of the entire set of data are replicated on the different sites — sites may contain different partitions which may even be overlapping. We refer to sites in the network as update and read-only sites. In theory, a site can be an update site for some data and read-only site for other data. In practice, for performance reasons, update sites should be considered updateable for all data at those sites (although our replication protocol makes no assumption in this respect).

Data are partially replicated in the network and updated accordingly as user update transactions are propagated through the network (eager synchronization between update sites and lazy propagation to read-only sites). We call these data *active replicas*. In addition, each site holds several archived copies of data used to support "as of" queries (called *archived replicas*). We make no assumptions about the archive creation (mainly motivated by legal reasons); we take for granted that sites hold one to several archived replicas in addition to active replicas.

Each site offers a set of semantically rich operations [21] which can be invoked within transactions. We consider the following update operations (shortly referred to as updates) available through the interface of update sites: insert, delete and replace. We consider specific read operations that are supported by read-only sites (see Section III).

With respect to transactions submitted by clients we distinguish between *read-only* and *update transactions* [3]. A read-only transaction only consists of read operations. Read operations within a single read-only transaction may run at several different read-only sites. An update transaction contains at least one update operation. *Propagation transactions* are performed during the idle time of a site in order to propagate changes to the read-only replicas. They are continuously scheduled as long as there is no running read or refresh transaction. Copies at read-only sites are kept as up-to-date as possible such that the work of refresh transactions (whenever needed) is reduced and the overall performance is increased. Decoupled *refresh transactions* propagate updates through the system on demand. A refresh transaction aggregates one or several propagation transactions into a single bulk transaction and is comprised of update operations. A refresh transaction is processed when a read-only transaction requests a version that is younger than the version stored at the read-only site. The refresh requires the updates to be propagated from the update site and may act on either active or archived replicas.

Update sites may have any number of read-only children to which updates are propagated. Read-only sites may further propagate their changes to other read-only sites, thus maintaining different levels of freshness in the system. We introduce the following naming convention, which defines a virtual tree structure (defined per data object). Since there are many update

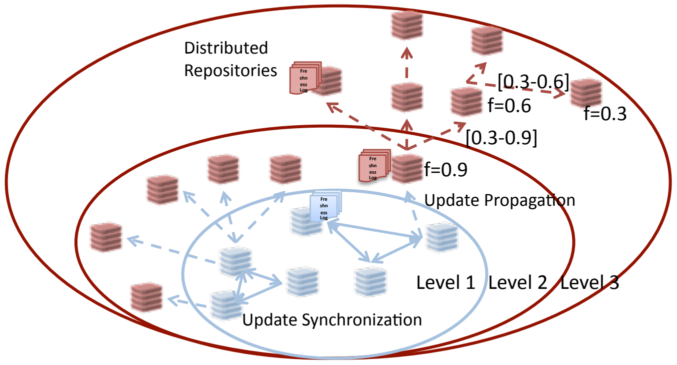


Fig. 1. System Model Architecture at the Middleware Level.

sites per data, the result is a forest of trees (see Figure 1).

- **Level 1** sites are update sites where data are synchronized (eager update everywhere replication). We assume a clock synchronization, needed for the calculation of freshness.
- **Level 2** sites are read-only sites where data are kept as up-to-date as possible. There is a *1-to-n* relationship between Level 1 and Level 2 sites.
- **Level 3** sites are read-only sites where data are not frequently updated. There is an *1-to-n* relationship between Level 2 and Level 3 sites. Level 3 sites are optional, but provide a definite advantage if the number of read-only sites in the system is considerably bigger than the number of update sites (trade-off between consistency and performance). Level 3 read-only sites may themselves be hierarchically structured, forming tree structures of different depths.

B. Distributed Repositories

Each site uses a set of tools to obtain partial, sometimes even out-dated information about the state of the system. We introduce to following components that facilitate the scheduling of read-only transactions and replica management decision. The components are not centrally materialized, but contain information that is globally distributed and replicated:

Replica catalog: used to determine the available replicas in the network. Update replicas contain distributed replicated information about the other update sites in the network and about their corresponding tree(s). In order to minimize the exchange of information and yet ensure that any update site can eventually reach any other update site (for example, for update transactions that include two or more data objects), update replicas need be aware of only one update site for each data object in the network. Read-only sites are aware of their parent and children in the tree. We replicate this information during replica synchronization to reduce communication overhead.

Freshness repository: used to collect the freshness of data periodically or on-demand. The freshness emphasizes the divergence of a replica from the up-to-date copy. Update sites will always have the highest freshness, while the freshness of the read-only sites will measure the staleness of their data. The sites at the leaves of the tree have the stalest data. Irrespective of their level, sites are aware of the freshness levels of the sites to which they propagate changes (see Figure 1).

Load repository: used to determine an approximate load information. This information can then be used to balance the load while routing read-only transactions to the sites or for the replica selection algorithm [23]. Update sites receive information regarding the load levels of other update sites and their subordinate children during replica synchronization. Read-only sites are only aware of their own load levels.

Propagation queues: used to enqueue changes to be applied to subordinate sites in the tree. The changes are bulked into propagation transactions and applied to the sites whenever possible. The propagation transactions execute changes from the local propagation queues in order. The queues are continuously updated as new updates arrive at the update sites.

C. Freshness Metrics

Freshness measures are closely related to the notion of coherence for which several ideas have been proposed in the literature [7], [14], [17]. We use the notions of *absolute* and *delay freshness* to characterize our freshness function.

Definition 2.1: ***(Absolute Freshness)*** *The absolute freshness of a data object d is defined by means of the timestamp $\tau(d)$ of the last committed update transaction that has updated d. These timestamps can be used by the clients in order to define their freshness requirements. A site s has a fresh copy of the data object d if the site timestamp associated to d, $\tau_s(d)$ is younger than the client timestamp requested for d, $\tau_c(d)$, i.e. if the following condition holds: $\tau_s(d) < \tau_c(d)$.* □

Using the absolute freshness implies that the younger the timestamp, the fresher the data.

Definition 2.2: ***(Delay Freshness)*** *A delay freshness defines how late in time a certain read-only site is compared to an update site that holds a copy of the same data object. We define $\tau(d)$ to be the commit time of the last refresh transaction that updates a copy of a data object d on a read-only site, and $\tau(d_0)$ the commit time of the most recent update transaction on the update site that updates d. Then the freshness function is defined as $f(d) = \frac{\tau(d)}{\tau(d_0)}$, with $f(d) \in [0, 1]$.* □

Each update site assigns to every committed transactions a unique timestamp greater than all timestamps assigned to previously committed transactions. Note that since we assume some clock synchronization between the update sites, clock skew can be ignored. Together with load information, each site propagates the freshness timestamps (of the last propagated update from the update site), such that all sites are aware of the load and freshness of their predecessors and/or successors.

A read-only site s is said to hold a copy of a data object d which fulfills a freshness level required by a client c, if $\tau_s(d) < \tau_c(d)$. Refresh transactions are used to bring all sites which hold data objects required by the client transaction to the same freshness level by executing a sequence of updates.

The freshness function is monotonically decreasing in the tree, from root to leaves. The further down the path in the tree, the less accurate the freshness function. This is an important observation upon which we base our routing strategy. Read-only sites are able to determine using their own local knowledge whether the current version of the data can be used to serve a user request or if this needs to be routed to a predecessor / successor in the tree. In other words, read-only sites are able to autonomously decide whether or not they satisfy a required freshness level and where to route the request, if necessary.

III. RE:FRESHIT PROTOCOL

Re:FRESHiT uses decoupled refresh transactions to propagate updates through the system on-demand, in order to bring read-only replicas to the freshness level specified by the read-only transactions [3], [17]. Re:FRESHiT exploits the sites' idle time by continuously scheduling propagation transactions as update transactions at the update sites commit, as long as there is no running read or refresh transaction.

A. *Freshness-aware Read Access*

The update sites are eagerly synchronized to produce a globally serializable schedule (although no central scheduler exists and thus no schedule is materialized in the system) [24]. Moreover, the update transactions' serialization order is their commit order. Each propagation transaction inherits the timestamp of the committed update transaction and this timestamp is propagates to all read-only sites. The extension required by Re:FRESHiT to support freshness without losing consistency are summarized in Algorithm 3.1.

Algorithm 3.1: Scheduling Thread

```
(1)  begin
(2)    while $true$ do
(3)        // user request $read(d, t_1, t_2)$
(4)        // for data object $d$ with $t_1$ and $t_2$ timestamps
(5)        if $(d \ \nexists \ locally)$
(6)          then
(7)             // find a site $s_{out}$ from the replica catalog
(8)             // which contains $d$
(9)             route request to $s_{out}$
(10)         else
(11)            case: $t_1$ is null and $t_2$ is null
(12)            // read the latest data
(13)            $readLatestData()$
(14)            case: $t_1$ is null
(15)            // read data not older than $t_2$
(16)            // transform timestamp into freshness;
(17)            // cf. Def. 2.2
(18)            $transform(t_2) = f_{client} = \frac{t_2(d)}{\tau(d_0)}$
(19)            $readNotOlderThan(f_{client})$
(20)            case: $t_2$ is null
(21)            // read data not younger than $t_1$
(22)            // transform timestamp into freshness;
(23)            // cf. Def. 2.2
(24)            $transform(t_1) = f_{client} = \frac{t_1(d)}{\tau(d_0)}$
(25)            $readNotOlderThan(f_{client})$
(26)            case: none is null
(27)            // read data between $t_1$ and $t_2$
(28)            $readBetween(t_1, t_2)$
(29)       fi
(30)   od
(31) end
```

Algorithm 3.2: Read Latest Data

```
(1)  funct $readLatestData(f_{client})$ $\equiv$
(2)    $readNotOlderThan(1)$
(3)  .
```

Algorithm 3.3: Read Not Older Than

```
(1)  funct $readNotOlderThan(f_{client})$ $\equiv$
(2)    if $f(d) >= f_{client}(d) \ \wedge \ s \notin overload$
(3)      then
(4)         execute read
(5)     else $f(d) >= f_{client}(d) \ \wedge \ s \in overload$
(6)         // find a child which is best replica
(7)         if $child \notin overload \ \wedge \ f_{child}(d) >= f_{client}(d)$
(8)           then
(9)              route request to child
(10)          else
(11)             //no site to route
(12)             $s_{new}$ = createNewReplica(d, f(d))
(13)             route request to $s_{new}$
(14)        fi
(15)    else $f(d) < f_{client}(d)$
(16)        route request to parent
(17)        if $(\nexists \ \ parent \ \vee \ parent \in overload)$
(18)          //no site to route
(19)          then
(20)             refresh(d)
(21)             rotate(tree)
(22)        fi
(23)   fi
(24) .
```

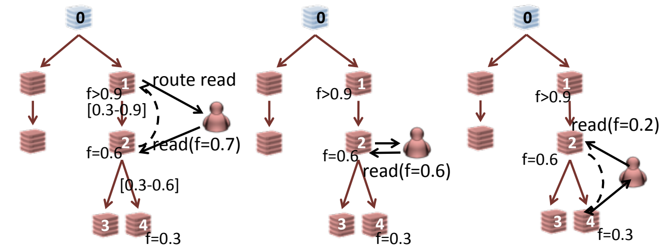
We use *freshness locks* [3], [17] in order to prevent propagation and/or refresh transactions to update a data object above the freshness level required by a running read-only transaction. A freshness lock placed on a data object d at a site s with an associated absolute freshness, required by a running read-only transaction, will not allow an update operation on the data object d at site s to bring it to a younger absolute freshness. As proven in [3], the freshness locks ensure one-copy serializable executions. We build on the same assumptions, and thus guarantee consistent access to data. Defresh transactions act on archived replicas and do not require freshness locks.

We allow users to directly query any read-only site in the system and to specify freshness requirements as QoS constraints. However, these requirements might be implicitly changed by the middleware if none of the objects involved in the transaction satisfies the required freshness level.

We foresee the following situations which may occur during the lifetime of a read-only site, s:

1. Read requests for data above a certain freshness level: A read-only transaction T_j submits a read operation $r_j(d, null, t_2)$ to a read-only site s that stores an active replica of d. The following rules apply:

- If $f_s(d) \geq f_{T_j}(d)$ and s is not in an overload situation, then the read operation is executed (for example, Figure 2 (b)). If $f_s(d) \geq f_{T_j}(d)$ and s is in an overload situation, then the read operation is routed downward in the tree until the best site s_l is found which fulfills the condition $f_{s_l}(d) \geq f_{T_j}(d)$ and s_l is not in an overload situation (for example, Figure 2 (c)). We use the definition of best replica previously introduced in [23]. Accordingly, the best site fulfills the required freshness level and is not in an overload situation. Very importantly, the query routing is possible as each site knows the freshness of its children. If there is no site among the transitive children of s which is not in an overload situation where the request can be routed, a new replica of d is created. The details on replica



(a) User request read(f=0.7) (b) User request read(f=0.6) (c) User request read(f=0.2)

Fig. 2. Query Routing for Read-Only Sites.

creation are presented in Section III-B.

- If $f_s(d) < f_{T_j}(d)$, then the read operation is routed upward in the tree until the best site s_m is found among the (transitive) parents of s which fulfills the condition $f_{s_m}(d) \geq f_{T_j}(d)$ and s_m is not in an overload situation (for example, Figure 2 (a)). If no such site is found, s requests a refresh transaction until $f_s(d) \geq f_{T_j}(d)$. The read operation can be processed at s after the refresh transaction has been executed. The dynamic tree rotation, associated with a refresh, is detailed in Subsection III-C.

2. Read requests for data below a certain freshness level: A read-only transaction T_j submits a read operation $r_j(d, t_1, null)$ to a read-only site s that stores an active replica of d. The following rules apply:

- If $f_s(d) \leq f_{T_j}(d)$ and s is not in an overload situation, then the read operation is executed. If $f_s(d) \geq f_{T_j}(d)$ or s is in an overload situation, then the read operation is routed downward in the tree until the best site s_l is found which fulfills the condition $f_{s_l}(d) \leq f_{T_j}(d)$ and s_l is not in an overload situation.

3. Read requests for data that are not replicated locally: A read-only transaction T_j submits a read operation $r_j(d)$ to a read-only site s that does not store a copy of d. Then the read operation is re-routed to a site in the tree that contains a copy of that data object, from the replica repository.

4. No read requests: In this case only propagation transactions can occur at s. Propagation transactions execute changes from the local propagation queues (where updates propagated downwards from the update sits are stored). S delays a propagation transaction P until all propagation transactions with smaller freshness levels than P have committed at s. A propagation transaction is dropped if the site has already seen and processed a propagation transaction with the same or higher freshness level or a refresh transaction.

Algorithm 3.4: Create New Replica

```
funct $createNewReplica(x, f(x))$ $\equiv$
  // this happens if local site and all children are in overload
  // determine sites in the geographic neighborhood, $S^*$
  for $s \in S^*$ do
     if $s \notin overload$
       then
          copy x to s
          exit
     fi
  od
.
```

Algorithm 3.5: Read Between

```
funct $readBetween(t_1, t_2)$ $\equiv$
  // find x, the closest local archive or replica with $t_x \leq t_2$
  if $(t_1 <= t_x <= t_2)$
    then
          execute read
    else $t_x < t_1$
          refresh(d) until $t_1 <= t_x <= t_2$
          rotate(tree)
    else (no local archive or replica)
          // request archive or replica y in the tree
          // with $(t_1 \leq t_y \leq t_2)$
          if $(s_{remote} \quad has \quad y)$
            then
                 // route upwards or downwards
                 route request to $s_{remote}$
          fi
    else (no remote archive or replica)
          // notify user that archive or replica does not exist
          return $d$ with $t_x$(d) closest to $t_{client}$(d)
  fi
.
```

B. Tree Dynamics

As mentioned in Section II the virtual trees are created per data object basis. In order to create a tree, we require a minimum number of initial update replicas of a data object (for example, at least three) that are used to bootstrap the tree and a set of fixed IP addresses of the replicas. These initial IP addresses point to (some) sites in the system equipped with our middleware. In the beginning, any new replica will become a Level 2 read-only site. At later stages, new sites can enter the replication scheme by becoming read-only sites at all levels. If a new site s wants to join as read-only site for a data object d, it selects a site in a tree by using the replica catalog. Let s_k be the selected site. If s_k is not in an overload situation s will join the replication scheme as its child (see Figure 3 (c)). Otherwise, its join request is forwarded downwards in the tree, starting from s_k, until a subordinate is found which is not in an overload situation and which will become the parent of s (see Figure 3 (b)). Hence, depending on local load conditions, the tree may either grow in breadth or in depth. If a leaf is reached and no suitable site is found, a new replica will be created in the geographic vicinity at a site which is not in an overload situation (Algorithm 3.4). Complex selection algorithms can be employed by monitoring user access patterns or by selecting the sites in the geographic vicinity of s which do not hold copies of d but data previously accessed together with d.

Continuous propagation transactions ensure that new replicas will eventually hold a copy of the data object(s) of interest with a certain freshness level. The dynamic load balancing between the replicas presented in [23] and the dynamic tree changes induced by refresh transactions will ensure that a site can dynamically move up or down in the tree hierarchy and be promoted to an update site or again demoted to a read-only site based on user access patterns.

We rely on local information only to balance the load among replicas. In widely distributed environments choosing the globally least loaded replica to forward requests is an impractical task that requires continuous and complete knowledge about the entire system. It has been mathematically proven that

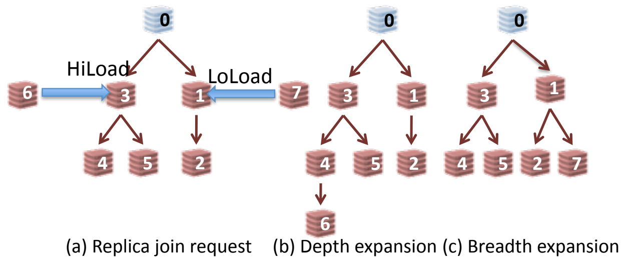


Fig. 3. Breadth vs Depth Expansion of a Tree.

having just two random choices yields a large reduction in the maximum load over having one choice, while each additional choice decreases the maximum load by just a constant factor [11]. Consequently, load balancing is achieved by allowing join requests to be addressed to any site, which chooses between itself and its immediate subordinate(s).

C. Replica Refresh

The tree structure changes dynamically. The following examples illustrate the cases that require changes in its topology.

Consider the example in Figure 4. Assume a user query (of the type $readNotOlderThan$, Algorithm 3.3) is sent to the Level 3 read-only site 2, which contains older data (Figure 4) and is not able to service the request. It will check among its predecessors in the tree to see if its parent is capable of serving this request. If no site is found (see Figure 4 (a)), site 2 would request a refresh transaction and it will be updated with the most up-to-date data from the update site (site 0), becoming the freshest read-only site in the path. Leaving it in the current position in the tree would violate the monotony of the freshness. Consequently, site 2 becomes a direct child of site 0 and a Level 2 site, by rotating the tree structure in order to preserve the monotony of the freshness (see Figure 4 (b)).

Similarly, user queries of the type $readBetween$ may request a refresh transaction, hence the tree structure may change as shown in Figure 4. Therefore, propagation transactions always follow the tree paths. Refresh transactions on the other hand change the tree topology.

IV. Experimental Evaluation

We have implemented a prototype on which we ran a series of experiments that evaluate the performance of Re:FRESHiT.

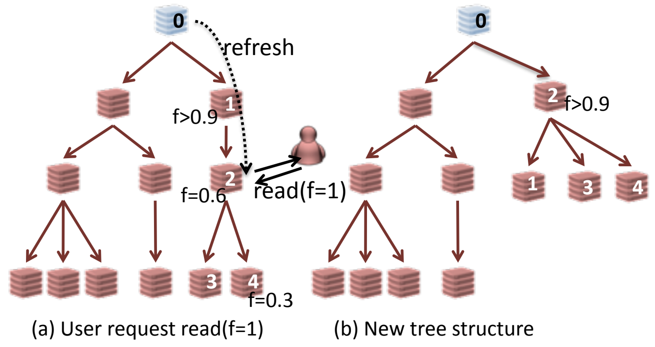


Fig. 4. Dynamic Tree Rotation after Replica Refresh.

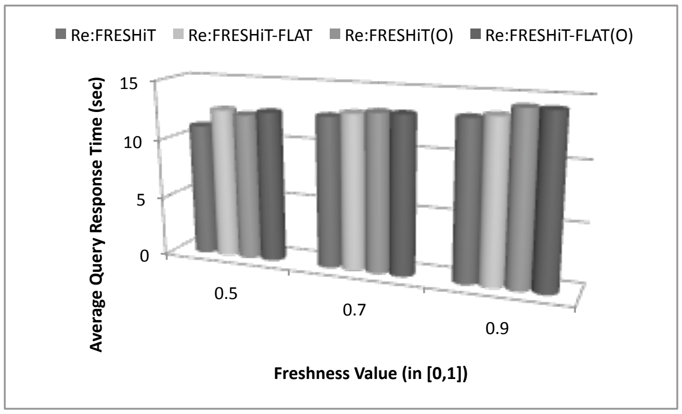


Fig. 5. Average Query Response Time for Different Network Topologies.

A. Experimental Setup

The evaluation has been conducted on 48 sites, and shows the performance of Re:FRESHiT in different network topologies and different routing strategies.

Updates at the update sites are continuously propagated to read-only sites. Hence, in addition to refresh transactions, which occur on demand (as a consequence of user queries), changes are bulked into propagation transactions and applied to the sites whenever possible. We have used a ratio of concurrent update transactions to queries of 1:10. We considered the update rate at update sites (100 updates per second) as the unit size of bulked updates. The tests were repeated for client requests for data with freshness levels in the following freshness intervals: [0.5;1], [0.7;1] and [0.9;1]. The freshness belongs to a given interval and is mapped to transaction timestamps. Since defresh transactions are very infrequent, their impact has not been evaluated in this paper.

We compared two network topologies in order to evaluate the advantages of our proposed architectural structure:

- In the first setting we propose a tree structure, where sites are classified into three levels: Level 1 update sites, Level 2 read-only sites (fresher copies) and Level 3 read-only sites (staler copies). We call this setting Re:FRESHiT.
- In the second setting, we eliminate Level 3 sites completely. We call this setting Re:FRESHiT-FLAT[1].

In the rest of the experiments, the Re:FRESHiT network topology has been used. The next set of experiments compared two query routing strategies:

- In the first setting we route queries according to freshness and load. We refer to this setting as Re:FRESHiT.
- In the second setting we force requests to be processed locally, by triggering a refresh instead of routing queries. We refer to this setting as FORCE-Refresh.

Our third series of experiments evaluated the advantages of tree rotation whenever refresh transactions occur. We compared two refresh strategies:

- In the first setting we dynamically rotate the tree whenever a query that arrives at a site requires a refresh transaction. We refer to this general setting as Re:FRESHiT.

[1]This setting is conceptually equivalent to the PDBRep approach [3].

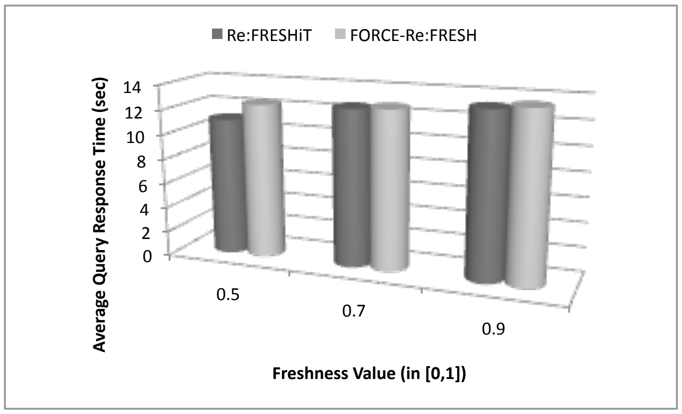


Fig. 6. Average Query Response Time for Different Route Strategies.

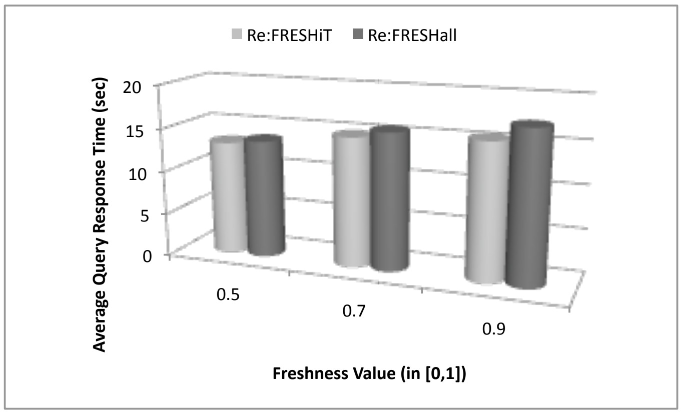


Fig. 7. Average Query Response Time for Different Refresh Strategies.

- In the second setting we modify the Re:FRESHiT protocol to refresh the entire tree structure whenever a query arrives at a site and cannot be serviced locally. We refer to this setting as Re:FRESHall.

B. *Re:FRESHiT Network Topology*

Our first experiments show how the network topology and the freshness requirements influence the query response time (see Figure 5). The results show the advantages of the tree topology versus the flat topology. By reducing the number of Level 2 sites in comparison to Re:FRESHiT-FLAT, we reduce the propagation time, which explains the increase in performance. Furthermore, in a high workload, even though propagation transactions are slower than refresh transactions, they are still able to keep sites fresh enough for queries with lower freshness requirements. By taking advantage of the tree structure, queries have better chance of finding a site that satisfies a lower degree of freshness, which can be seen by the increase in query response time for Re:FRESHiT.

C. *Re:FRESHiT Query Routing*

Our second set of experiments shows how the query re-routing and the freshness requirements influence the query response time (see Figure 6) and the advantages of Re:FRESHiT versus FORCE-Refresh, when processing queries with a lower degree of freshness. By routing the queries within the tree structure we reduce the time required by the refresh transactions. Since the freshness is monotonically decreasing within a tree, a site is able to properly route a query upwards in the tree if the client request has a higher timestamp than the local one, or downward in the tree for a lower timestamp. The difference in performance is reduced for queries with a higher freshness levels, as in this case the refresh transactions are still needed. Re:FRESHiT shows considerable increase in the query response time for queries with lower freshness levels.

D. *Re:FRESHiT Refresh Strategies*

Our third set of experiments show how the tree dynamic structure and the freshness requirements influence the query response time (see Figure 7) and the advantages of Re:FRESHiT versus Re:FRESHall, when processing queries with all degrees of freshness. By dynamically rotating the tree structure we reduce the time required by the refresh transactions, as they are applied at one site only rather than at a subset of sites along a portion of the path. Using our routing strategy, queries are still routed by benefiting from the monotony of the freshness. However, by splitting a potentially long path in the tree in several shorter paths we ensure that data with higher freshness levels are also available and that the re-routing of queries takes less time.

V. Related Work

According to Brewer's CAP theorem [6], Strong *Consistency*, High *Availability* and *Partition Tolerance* are three requirements that exist in a special relationship when it comes to designing and deploying applications in a distributed environment. In this work we relax the notion on consistency and allow users to read data with different freshness levels. Previous work on lazy replication has concentrated on performance and correctness only. As observed from the example scenario, today's applications have different requirements. Older work on lazy replication like [5], [13], [15], [20] assumed that transactions execute entirely at the initiation site, which may not be the case in practical settings. A recent protocol for finer grained data replication that supports freshness and lazy update propagation for many read-only nodes has been presented in [3]. In [2], we explain how we can adapt this protocol to the Grid. However, this protocol suffers from a strict assumption on the existence of a central component used to collect and serialize the updates. In [4], several updateable replicas are supported but a single, global replication graph is required. This graph corresponds to a global agreement on the order of propagating updates in the system.

Another important aspect of the work presented in this paper relates to temporal aspects in database research. Schema versioning [9], [10], [16] is a technique used for managing database evolution in order to preserve versions of schema and data during the evolution of a database. To provide the most generality, bi-temporal databases [18], [19] can be used to realize schema versioning, since they allow both retroactive and proactive updates to the schema and database. However, there is a cost tradeoff between the flexibility of retroactive and proactive schema changes and the cost of implementing these mechanisms. The complexity is high because changes not only affect the current versions of data but also the past and even the future versions which makes the database conversion much more complicated than for conventional snapshot databases. By using archives we eliminate this complexity from our approach (since archives are essentially frozen versions of the past that are not affected by current changes). Another way to implement this functionality is log-only temporal databases [12]. In contrast to log-only approaches which are write-optimized and where object retrieval can be a bottleneck, our archiving mechanism is aimed at improving read performance.

VI. Conclusions and Outlook

A core challenge in the context of cloud computing is the management of very large volumes of data with dedicated service guarantees. One of the most common QoS guarantees is the availability of data which requires data to be replicated across data centers. In this paper we propose Re:FRESHiT, a protocol that specifically addresses the problems of replication management in data clouds. Re:FRESHiT organizes read-only sites in virtual trees based on the sites' freshness levels, and introduces a clever routing mechanism for reading data, while at the same time allowing users to specify their own freshness requirements. Trees are automatically reorganized after individual nodes are refreshed or when new replicas join. For the latter, the sites' load is taken into account. We provide an evaluation of our replication protocol and compare it in different settings with different network topologies and routing strategies.

In future work, we plan to explore the benefits of trading access time for data freshness using an economic model that takes into account the provider's real costs for replica management. In addition, we will dynamically adapt the refreshment policy of replicas to the profiles of clients in the cloud (schedule propagation transactions less frequently for users that often access very stale data). Another extension will address the support for read-only transactions: read operations will either be supported by a read-only site using a stale replica, or by exploiting one of the locally managed archives. In each case, depending on the timestamps specified by a client, read operations might be served directly, might require a refresh of a copy which is too old, or a defresh of a copy which is too recent. For this, we plan to complement the concept of refresh operations that raise the level of freshness of a replica with defresh operations that conversely lower the degree of freshness of a replica (an operation counterpart to refresh which is introduced to allow users to even further refine their read requests). As a consequence, also the concept of archive replica will be extended. As old replicas might also be eventually subject to refresh operations, selected versions might be archived in order to limit the possibly high costs of potential defresh operations if requested by read operations.

References

[1] R. Agrawal et al. The Claremont Report on Database Research. *Communication of the ACM*, 52(6):56–65, 2009.

[2] F. Akal, H. Schuldt, and H.-J. Schek. Grid-Enabled Data Replication for Digital Libraries with Freshness and Correctness Guarantees. In *3rd VLDB Workshop on Data Management in Grids, Vienna, Austria*, 2007.

[3] F. Akal, C. Türker, H.-J. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Proceedings of the 32st International Conference on Very Large Databases (VLDB 2005)*, pages 565–576, 2005.

[4] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, Consistency, and Practicality: are these mutually exclusive? In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 484–495. ACM, 1998.

[5] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update Propagation Protocols for Replicated Databates. In *Proceedings ACM SIGMOD*, pages 97–108, 1999.

[6] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 7. ACM, 2000.

[7] J. Cho and H. Garcia-Molina. Synchronizing a database to Improve Freshness. In *ACM SIGMOD International Conference on Management of Data*, pages 117–128, 2000.

[8] B. Cooper, R. Ramakrishnan, et al. PNUTS: Yahoo!'s Hosted Data Serving Platform. *PVLDB*, 1(2):1277–1288, 2008.

[9] C. DeCastro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, 22(5):249–290, 1997.

[10] W. Kim and H.-T. Chou. Versions of Schema for Object- Oriented Databases. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB 1988)*, pages 148–159. Morgan Kaufmann Publishers Inc., 1988.

[11] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, 2001.

[12] K. Norvåg. The Vagabond Approach to Logging and Recovery in Transaction-Time Temporal Object Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(4):504–518, 2004.

[13] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 126–137. Morgan Kaufmann Publishers Inc., 1999.

[14] E. Pacitti and E. Simon. Update Propagation Strategies to improve Freshness in lazy master replicated Databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[15] E. Pacitti, E. Simon, and R. Melo. Improving Data Freshness in Lazy Master Schemes. In *Proceedings of the The 18th International Conference on Distributed Computing Systems (ICDCS 1998)*, page 164, Washington, DC, USA, 1998. IEEE Computer Society.

[16] J. F. Roddick. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology*, 37(7):383–393, 1995.

[17] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. FAS: a Freshness-sensitive Coordination Middleware for a Cluster of OLAP Components. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 754–765, 2002.

[18] R. Snodgrass. The Temporal Query Language TQuel. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of Database Systems (PODS 1984)*, pages 204–213. ACM, 1984.

[19] R. T. Snodgrass. *The TSQL2 Temporal Query Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.

[20] I. Stanoi, D. Agrawal, and A. E. Abbadi. Using Broadcast Primitives in Replicated Databases. In *Proceedings of the 16th ACM symposium on Principles of Distributed Computing*, page 283. ACM, 1997.

[21] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H.-J. Schek. Unifying concurrency control and recovery of transactions with semantically rich operations. *Theoretical Computer Science*, 190(2), 1998.

[22] L. C. Voicu and H. Schuldt. How Replicated Data Management in the Cloud can benefit from a Data Grid Protocol – the Re:GRIDiT Approach. In *Proceedings of the 1st International Workshop on Cloud Data Management (CloudDB'09)*, November 2009.

[23] L. C. Voicu and H. Schuldt. Load-aware Dynamic Replication Management in a Data Grid. In *Proceedings of the 17th International Conference on Cooperative Information Systems (CoopIS'09)*, November 2009.

[24] L. C. Voicu, H. Schuldt, F. Akal, Y. Breitbart, and H.-J. Schek. Re:GRIDiT – Coordinating Distributed Update Transactions on Replicated Data in the Grid. In *Proceedings of the 10th IEEE/ACM on Grid Computing (Grid'09)*, October 2009.

[25] L. C. Voicu, H. Schuldt, Y. Breitbart, and H.-J. Schek. Replicated Data Management in the Grid: the Re:GRIDiT Approach. In *ACM Workshop on Data Grids for eScience (DaGreS'09)*, May 2009.