Quality-of-Service for Consistency of Data Geo-replication in Cloud Computing*

Sérgio Esteves, João Silva, and Luís Veiga

Instituto Superior Técnico - UTL / INESC-ID Lisboa, GSD, Lisbon, Portugal
sesteves@gsd.inesc-id.pt, {joao.n.silva,luis.veiga}@inesc-id.pt

Abstract. Today we are increasingly more dependent on critical data stored in cloud data centers across the world. To deliver high-availability and augmented performance, different replication schemes are used to maintain consistency among replicas. With classical consistency models, performance is necessarily degraded, and thus most highly-scalable cloud data centers sacrifice to some extent consistency in exchange of lower latencies to end-users. More so, those cloud systems blindly allow stale data to exist for some constant period of time and disregard the semantics and importance data might have, which undoubtedly can be used to gear consistency more wisely, combining stronger and weaker levels of consistency. To tackle this inherent and well-studied trade-off between availability and consistency, we propose the use of VFC^3, a novel consistency model for replicated data across data centers with framework and library support to enforce increasing degrees of consistency for different types of data (based on their semantics). It targets cloud tabular data stores, offering rationalization of resources (especially bandwidth) and improvement of QoS (performance, latency and availability), by providing strong consistency where it matters most and relaxing on less critical classes or items of data.

1 Introduction

The great success Internet achieved during the last decade has brought along the proliferation of web applications which, with economies of scale (e.g., Google, Facebook, and Microsoft), can be served by thousands of computers in data centers to millions of users worldwide. These very dynamic applications need to achieve higher-scalability in order to provide high availability and performance. Such scalability is typically realized through the replication of data across several geographic locations (preferably close to the clients), reducing application server and database bottlenecks while also offering increased reliability and durability.

Highly-scalable cloud-like systems running around the world often comprise several levels of replication, specially among servers, clusters, inter-data centers, or even among cloud systems. More so, the advantages of geo-distributed micro-data centers over singular mega-data centers have been gaining significant

* This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102250/2008, PTDC/EIA-EIA/108963/2008 and PEst-OE/EEI/LA0021/2011.

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 285–297, 20

attention (e.g., [1]), as, among other reasons, network latency is reduced to end-users and reliability is improved (e.g., in case of a fire or a natural catastrophe, or simply network outage). With this current trend that brings higher number of replicas, more and more data needs to be properly synchronized, carrying out the need of having smart schemes to manage consistency while not degrading performance.

In replication, the consistency among replicas of an object has been handled through both traditional pessimistic (lock-based) and optimistic approaches [2]. Pessimistic strategies provide better consistency but cause reduced performance, lack of availability, and do not scale well. Where optimistic approaches rely on eventual consistency, allowing some temporary divergence among the state of the replicas to favor availability and performance. Since it is not possible to fully have the best of both approaches in a distributed environment with arbitrary message loss, as stated in the CAP theorem [3], we envision that more can be done to approximate consistency from availability. One path not yet significantly explored, and that we intend to address in this work, consists of wisely and dynamically strengthen and weaken consistency in accordance to the importance of the data being replicated.

In the context of the web, replication has been extensively applied at the application and database tiers, significantly improving the performance and reducing workload. At the application level, popular in-memory caching solutions, such as memcached [4], defy developers in the following ways: they do not offer transactional consistency with the underlying database; and they offer only a key-value interface, and developers need to explicitly manage the cache consistency, namely invalidating cache data when the database changes. This management, which includes performing manual lookups and keeping the cache updated, has been a major source of programming errors.

At the database level, tuples are replicated and possibly partitioned across multiple network nodes, which can also execute queries on replicas. However, adding more database servers to a RDBMS is difficult, since the partition of database schemas, with many data dependencies and join operations, is non-trivial [5]. This is where non-relational NoSQL databases come to take place.

High-performance NoSQL data stores emerged as an appealing alternative to traditional relational databases, since they achieve higher performance, scalability, and elasticity. For example, Google has built its own NoSQL database, BigTable [6], which is used to store google entire web search system. Other solutions include Cassandra [7], Dynamo [8], PNUTS [9] and HBase [10] (which we focus on in this work).

To sum up, these several approaches to replication, existent in well-known cloud systems and components, usually treat all data at the same consistency degree and are blind w.r.t. the application and data semantics, which could and should be used to optimize performance, prioritize data, and drive consistency enforcement.

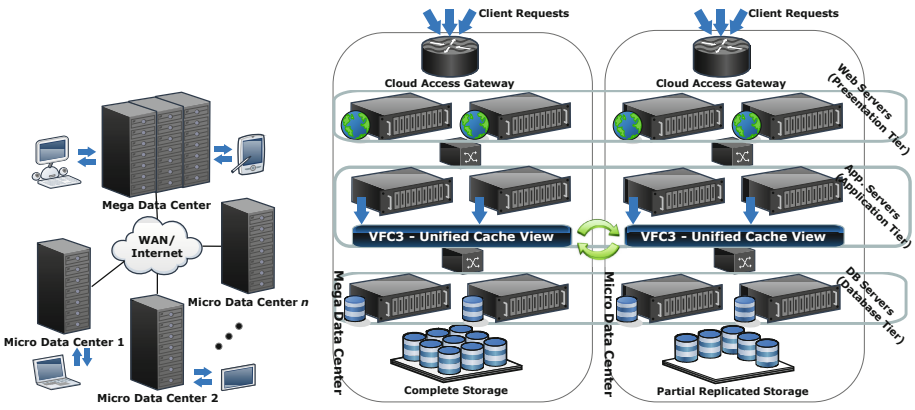
Given the current context, we propose the use of a novel consistency model with framework and programming library support that enables the definition

and dynamic enforcement of multiple consistency degrees over different groups of data, within the same application, across very large scale networks of cloud data centers. This model is driven by the different semantics data might have; and the consistency levels can be automatically adjusted based on statistical information. Moreover, this framework, named VFC^3 (*Versatile Framework for Consistency in Cloud Computing*), comprises a distributed transactional in-memory cache and is intended to run on top of NoSQL high-performance databases. This way, we intend to improve QoS, rationalize resource usage (especially bandwidth), and deliver higher performance.

The remainder of this paper is organized as follows. Section 2 presents the architecture of the VFC^3 framework, and Section 3 its underlying consistency model. In Section 4, we offer details of relevant implementation aspects. Then, Section 5 presents a performance evaluation, and Section 6 reviews related work.

2 Geo-distributed Cloud Scenario and Architecture

The VFC^3 consistency model was specially designed to address very large-scale and dynamic environments (e.g., cloud computing) that need to synchronize large amounts of data (either application logic or control data) between several geographically dispersed points, while maintaining strong requirements about the quality of service and data provided. Figure 1a depicts a scenario of geo-replicated data centers where a mega data center needs to synchronize data with micro data centers scattered throughout different regions. Such micro centers replicate part of the central database, with only the more relevant data to a given corresponding region. In more detail, Figure 1b shows the constituent parts of the mega and micro data centers, where the VFC^3 middleware operates and how the interaction is carried out among data centers.



(a) Target environment (geo-distributed data centers)

(b) data center detailed view

Fig. 1. VFC^3 overall scenario and network architecture

We present an archetypal architecture (Figure 2) of the VFC^3 framework that is capable of enforcing different degrees of consistency (or conversely, bounding divergence) and runs atop very large-scale (peta-scale) databases residing in multiple data centers. The consistency constraints over the replicas are specified in accordance to the data semantics and they can be automatically adjusted at run time. Moreover, this framework comprises a distributed transactional in-memory cache system enhanced with a number of more components, described as follows.

Monitor and Control. This component analyses all requests directed to the database. It decides from where to retrieve results to queries, cache or database, and controls the workflow when an update occurs. It also collects statistics regarding access patterns to stored data in order to automatically adjust the divergence levels.

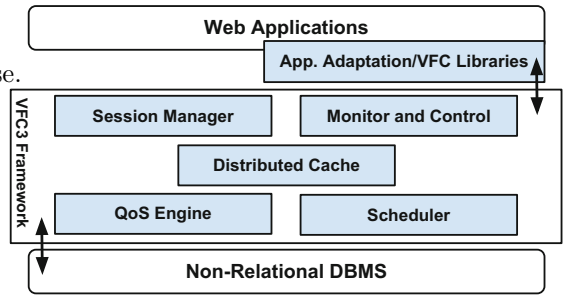


Fig. 2. VFC^3 middleware architecture

QoS Engine. It maintains data structures and control meta-data to decide when to replicate and synchronize data, obeying to consistency specifications.

Scheduler. This component verifies the time constraints over the data. When the time for data being replicated expires, the Scheduler notifies the QoS engine.

Distributed Cache. It represents an in-memory, transactional and distributed database cache for: i) temporary storing frequently accessed database items; and ii) keep tracking of which items need to be replicated.

Session Manager. It manages the configurations of the consistency constraints over the data through extended database schemas (automatically generated) defined for each application.

Application Adaptation: Applications may interact with the VFC^3 framework by explicitly invoking our libraries, but VFC^3 can also automatically adapt and intercept the invocation of other libraries, such as the HBase API, where developers need only change the declarations referencing the HBase libraries (also being automated), with remaining code unmodified. Legacy code is adapted by using annotations or pre-processor directives, during loading-time, where database tier code is transformed into calls to VFC^3 API.

Caching: The VFC^3 framework comprises a distributed and transactional in-memory cache system to be used at the application-level. It has two main purposes: i) keep tracking of the items waiting for being fully replicated and ii) temporarily store both frequently used database items and items within the same locality group (i.e., pre-fetch columns of an hot row), in order to improve read performance. Specifically, this cache stores partial database tables, with associated QoS constraints, that are very similar from the ones in the

underlying database, but with tables containing many less rows. Moreover, the VFC^3 cache is completely transparent to applications; it guarantees transactional consistency with the underlying database and the data is automatically maintained and invalidated, relieving developers from a very error-prone task.

This cache can be spanned across multiple application servers within the same data center, so that it can grow in size and take advantage of the spare memory available in the commodity servers. Although the work is distributed, it still gives a logical view of a single cache. The partition of data follows an horizontal approach, meaning that rows are divided across servers and the hash of their identifiers works as key to locate the servers in which they should be stored. Hence, this cache is optimized for our target database systems, since rows constitute indexes in the multi-dimensional map and every query must contain a row identifier (apart from scans). Furthermore, each running instance of the VFC^3 cache knows all others running on neighbor nodes.

Replication: The VFC^3 framework handles the whole replication process asynchronously, supporting the two general used formats, statement-based and row/column-based. However, if the statement-based strategy is used, the element (row, column, or table), referenced in a query, with the more restrictive QoS constraints will command the replication of the statement, leading other data with less restrictive constraints to be treated at stronger consistency degrees (which can be useful for some use cases). When a maximum divergence bound, associated to an element (row, column, or table), is crossed, the changed items, and only these, within that element are broadcasted to the other interested servers. These modified items are identified through *dirty bits*.

The system constantly checks bandwidth usage of messages exchanged with other nodes. The framework can trigger data replication and synchronization on low bandwidth utilization periods, even if consistency constrains do not impose it. Replication messages, using the column-based strategy, contain the necessary indexes to identify items placement within the database map and are compressed with gzip.

3 Consistency Model

The VFC^3 consistency model is inspired on our previous work [11]. It defines three-dimensional consistency vectors (κ) that can be associated with data objects. κ bounds the maximum objects divergence; each dimension a numerical scalar defining the maximum divergence of the orthogonal constraints: time (θ), sequence (σ), value (ν).

Time: Specifies the maximum time a replica can be without being updated with is latest value. Considering $\theta(o)$ provides the time (e.g., seconds) passed since the last replica update of object o, constraint κ_θ, enforces that $\theta(o) < \kappa_\theta$ at any given time.

Sequence: Specifies the maximum number of updates that can be applied to an object without refreshing its replicas. Considering $\sigma(o)$ indicates the number

of applied updates, this sequence constraint κ_σ enforces that $\sigma(o) < \kappa_\sigma$ at any given time.

Value: Specifies the maximum relative difference between replica contents or against a constant (e.g., top value). Considering $\nu(o)$ provides that difference (e.g., in percentage), this value constraint κ_ν enforces that $\nu(o) < \kappa_\nu$ at any given time. It captures the impact or importance of updates on the object's internal state.

Evaluation and Enforcement of Divergence Bounds: The evaluation of the divergence vectors σ and ν takes place every time an update request is received by the middleware. Upon such event, it is necessary to identify the affected tables/rows/columns, increment all of the associated vectors σ and verify if any σ or ν is reached when compared with the reference values (i.e., the maximum object allowed divergences). If any limit is exceeded, all updates since last replication are placed in a FIFO-like queue to be propagated and executed on other replicas. When there are multiple versions for the same mapping (table/row/column), the most recent ones are propagated first.

To evaluate the divergence vector, θ, VFC^3 uses timers (each node holding one timer per application) to check, e.g., every 1 second, if there is any object that should be synchronized (timestamp about to expire). Specifically, references to modified objects (identified by the dirty bits) are held in a list ordered ascending by time of expiration, which is the time of the last object synchronization plus θ. The Scheduler component, starting from the first element of the list, checks which objects need to be replicated. As the list is ordered, the Scheduler has only to fail one check to ignore the rest of the list; e.g., if the check on the first element fails (its timestamp has not expired yet), the Scheduler does not need to check the remaining elements of the list.

We consider 3 main events, perceived by the Monitor and Scheduler, that influence the enforcement and tuning of QoS, with their handling workflow described next.

Upon **Read Event:** 1) Try to fetch results from cache with sufficient QoS; 2) On success, return results immediately to the client application; 3) Otherwise, perform query on the database, return the respective results to the application, and store the same results in cache.

Upon **Write Event:** 1) Perform update on the database; 2) Update the cache; 3) Update θ and increment σ on table, column, or row; 4) Verify if the divergence bound, κ, is reached; 5) If so, the data is replicated to the other nodes, θ receives a new timestamp, and σ is reset.

Upon **Time Expiration Event:** 1) The data is replicated to the other interested nodes, θ is timestamped, and σ goes to 0.

Dynamic Adjustment of Consistency Guarantees: Users can specify intervals of values on the QoS vectors to let the framework automatically adjust the consistency intensity. This adjustment, performed by the QoS Engine component, is based on the observation of the frequency of read and write operations to data items during a given time frame. The general idea behind this is that many write operations, performed on different nodes over the same object (or

replica), will cause conflicting accesses, and thus it is necessary to guarantee stronger consistency. Conversely, few updates, or updates concentrated only on one node, allow weakening consistency guarantees within the specified vector intervals. The frequency of read operations also contributes to tuning. Many read operations on data that is mainly written in other nodes will strengthen consistency; if the data is written in the same nodes, consistency is relaxed. Conversely, few reads, or reads concentrated on one node, will weaken consistency.

Concurrent Updates: When two or more updates occur simultaneously over the same data in different data centers, both are preserved as the all data items are versioned. We resort mostly to *last-writer-wins* rule and handlers to make data centers converge on the same values. If stronger agreement is needed in more critical (and restricted) data, rotating leases allow data centers to perform writes without contention.

4 Implementation Details

As a proof of concept, we developed, in Java, a prototype of VFC^3 to demonstrate the advantages of our consistency model when deployed as a replication middleware for high-performance databases (i.a., not supporting a full relational model). Although the framework may be adapted to other storages, our target, in the scope of this particular work, is BigTable [6] open-source Java clone, HBase [10]. This database system is a sparse, multi-dimensional sorted map, indexed by row, column (includes family and qualifier), and timestamp; the mapped values are simply an uninterpreted array of bytes. It is column-oriented and designed to scale into the petabyte, while ensuring that write and read performance remain constant. In the following we provide the more relevant details of the VFC^3 implementation.

Schema and Database Management: VFC^3 requires the registration of each application, which includes providing the schema of the required databases. For each table, row, column (and optionally sets of rows and columns), therein, it is necessary to specify the maximum object divergence, κ. Otherwise, the default κ will be used meaning no divergence at all. This schema can be built manually, specifying tables and columns, or simply introduced as the standard auto-generated XML-based schema (given by the HBase Schema Manager tool), which can be processed by VFC^3. After this, the user should create and associate divergence bounds with database objects (i.e., tables/rows/columns) through code annotations, XML specification or a UI.

W.r.t. the creation of divergence bounds, users may specify intervals of values as elements for vectors, rather than scalar constants, so that consistency constraints can be automatically adjusted within an interval. The association of κ with: i) tables is useful when the semantics of the data therein contained indicates that the data should be treated at the same consistency level (e.g., a guest list); ii) rows is beneficial to handle single records independently; iii) columns may be practical if they are supposed to hold very large values, such as

media content. Furthermore, the vector ν is only applied on numeric values, and thus text or byte fields are initially precluded and supported only as byte-wise differences (e.g., number of different characters in a string). After the association of divergence bounds with schemas, a similar database is created in the domain of the VFC^3 framework for caching and tracking purposes. This database also contains the QoS attributes and a dirty bit for each cell telling if the cell was modified since the last synchronization occurred.

QoS Management: Different applications may specify overlapping QoS constraints; in this case, more restricted constraints override any others. Thus, such a scenario may happen: application1 requires $\kappa_{1,x}$ and $\kappa_{1,y}$, and application2 requires $\kappa_{2,y}$ ($\kappa_{1,y} > \kappa_{2,y}$). It could make no sense for application1 to have different consistency levels for the items x and y (as $\kappa_{1,y}$ is overridden by $\kappa_{2,y}$). To tackle this, we also allow users to define groups over items that should be handled at the same consistency level, i.e., ensuring atomicity upon serial consistency constraints, over a set or rows and/or columns, to comply with the application semantics. In the previous example, and considering application1 grouped x and y, $\kappa_{1,x}$ is thus assigned with the value of $\kappa_{1,y}$.

The QoS constraints, referring to the data consistency levels, are specified along with standard HBase XML schemas and given to the middleware with an associated application. Specifically, we introduced in the relative XSD the new element *vfc3*, which can be used inside the elements *table*, *column_family*, or *row*, to specify data requirements in relation to a table, column, or row. The vector ν is optional. Enhanced XML schemas are known by all data centers.

Library Support and API: In order to adapt HBase client applications, we provide a similar API to HBase,[1] where we only changed the implementation of some classes in order to redirect HBase calls to the VFC^3 framework, namely *Configuration.java*, *HBaseConfiguration.java*, and *HTable.java* were modified to delegate the HBase configurations to VFC^3. VFC^3 performs the management of the multiple distributed stand-alone HBase instances (without the Hadoop replication) in a transparent manner.

Cache and Metadata: The cache uses similar data structures to HBase itself, such as the *ConcurrentHashMap*, but with extensions to include metadata (the divergence bound vectors) and living in memory (albeit its state can be persisted in the underlying HBase for reliability purposes). The size of the cache and number of items to replace is configurable and new implementations of the cache eviction policy can be provided (default is LRU). Also, the types of the vector elements can be configurable.

5 Evaluation

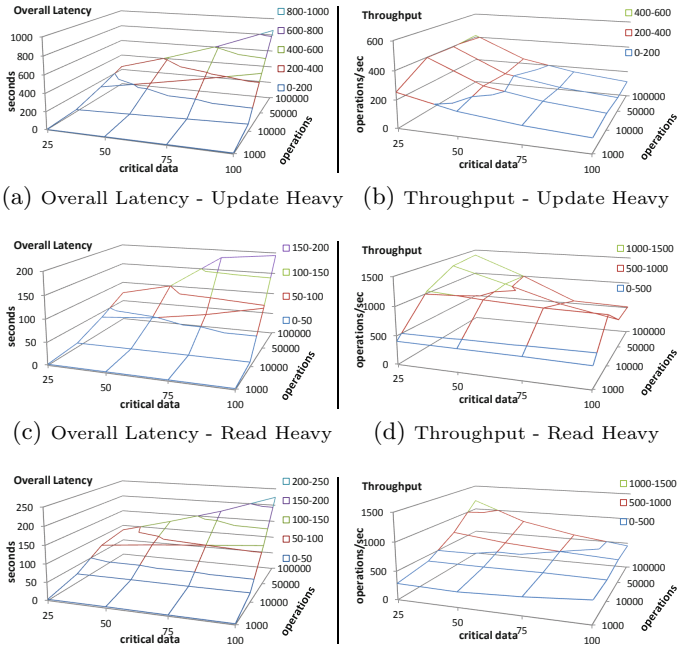
This section presents the evaluation of the VFC^3 framework and its benefits when compared with the regular HBase/Hadoop replication scheme. All tests were

[1] `http://hbase.apache.org/apidocs/overview-summary.html`

conducted using machines with an Intel Core 2 Quad CPU Q6600 at 2.40GHz, 7825MB of RAM memory, and HDD SATA II 7200RPM 16MB. As for the network, we scattered nodes around two different and relatively distant locations and the available bandwidth was around 60Mbps (downstream and upstream). Moreover, each node/machine had a standalone HBase instance running under VFC^3.

To evaluate the performance of our replication middleware we modified and adapted the YCSB benchmark [12] to work with VFC^3, thereby only redirecting the imports of some classes. Our scenario consisted of running this benchmark, with three different workloads (95/5, 5/95, and 50/50 %updates/%reads), to measure the overall latency and throughput (operations/second) for series of 1000, 10000, 50000, and 100000 operations (reads and writes), assuming in each case that 25, 50, 75, and 100% of the data is critical and required maximum consistency. The non-critical data was associated with σ and ν constraints, meaning its replication could be postponed and not all versions of the data are required. Additionally, the case of 100% means full replication, i.e., the same as using the regular HBase replication.

The (straight) lines of figures 3a, 3c, and 3e show that the overall latency is reduced with VFC^3 (25, 50, and 75%) when compared with HBase full replication (100%): i) latency gains were almost linear (latency of single operations is nearly constant, especially for writes), e.g., under heavy updates, for 100000 ops.,



(a) Overall Latency - Update Heavy (b) Throughput - Update Heavy

(c) Overall Latency - Read Heavy (d) Throughput - Read Heavy

(e) Overall Latency - 50/50 Update/Read (f) Throughput - 50/50 Update/Read

Fig. 3. VFC^3 throughput and latency performance

the gain was about 200 sec. every time critical data decreased; ii) from 10000 to 100000 ops., in all workloads, the latency increased linearly for each critical data segment; iii) for 1000 ops., the latency is not stable as the critical data increases, except for the first workload that incurs a slight improvement through the critical data axis (naturally, workloads with more updates will favor VFC^3, even for a small number of ops.); iv) the level of critical data is almost irrelevant in workloads under heavy reads, and the gains therein are mostly supported by our cache; v) the effects of the non-critical data replication were almost not noticed, since most of these data allowed version loss through vector σ (and the cache also absorbed some of these latencies); and vi) in a workload with a balanced mix of reads and writes, the average latency gain with VFC^3 is very satisfactory.

Figures 3b, 3d, and 3f show that the gains of throughput are more accentuated when critical data represents a smaller slice (25% in this case). Plus, the number of ops. only affected significantly the throughput for smaller amounts of critical data (e.g., from blue to green lines in the Read Heavy workload). The throughput for full replication was practically the lowest, comparing with other critical data levels, and almost the same in the non read heavy workloads (irrespective of the number of operations). Also, single read operations have higher latency than write operations (practically zero sec. since they are written in memory first on HBase) and that explains the instability on the Read Heavy figure.

Regarding network usage, we reduced the number of messages and also the volume of data with VFC^3. Note that only the last versions of the data were sent (like what typically happens) when σ expired; and the middleware synchronization performed compression and agglomeration of replication messages when they were inside a same small time window. For 25, 50, and 75% of critical data we saved on average about 75, 48, and 20% respectively, since we mostly relied on the σ vector (message skip).

For evaluating the cache component, we relied on different workloads (taken from the YCSB benchmark) performing 100000 operations each: a) 50%/50% reads and writes (e.g., session store recording recent actions); b) 95%/5% reads/ write mix (e.g., photo tagging); c) 100% reads; d) read latest workload (e.g., user status updates on social networks); e) read-modify-write (e.g., user database activity). The cache size was 30% of the size of each workload.

Figure 4 shows that our cache is effective and can reduce latency and save communication hops. Not surprisingly, the workload *d* obtained best results; i.e., since we read the most recently inserted records and have LRU as the cache eviction policy. For the other workloads, the gains were good, between 23-35% (*a* and *b* at the extremes). Note that the cache size and eviction policy can impact

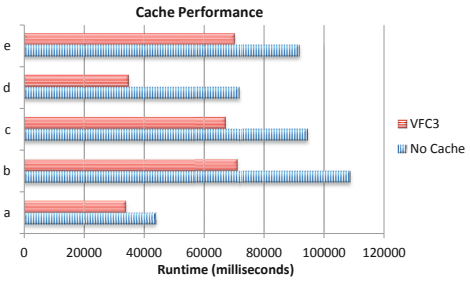


Fig. 4. Cache Performance

these results; so these parameters should be adjusted to better suit the target applications.

The average hit rate of the cache for all experimented workloads was about 51% (except for *d*, which had 77%), revealing that VFC^3 cache can significantly improve performance, by avoiding expensive trips to the database, for a set of typical scenarios.

6 Related Work

Many work has been done in the context of replication, transactional and consistency models. Our model is based on those already established (standard book [13]) and can be seen as an extension of them by allowing multiple levels of consistency.

In [14], a system is presented for detection-based adaptation of consistency guarantees. It takes a reactive approach to adjust consistency: upon detection of inconsistencies, this system tries to increase the current consistency levels so that they satisfy certain requirements. Contrastingly, we follow in VFC^3 a more proactive approach by trying to avoid inconsistencies from the beginning.

In [15], authors propose three metrics to cover the consistency spectrum (numerical order, order error, and staleness) which are expressed by a logical unit (conit). However, it could be difficult to associate conits with the application-specific semantics, specially in terms of granularity; whereas our work goes with a more fine-grained approach, capturing the semantics from the data itself.

In [16], authors propose creating different categories of data that are treated at different consistency levels. They demonstrate through the TPC-W benchmark that object-based data replication over e-commerce applications can definitely improve availability and performance. We go further with VFC^3, whereas we do not have such a restrictive model regarding target applications and degrees of consistency.

In [17], authors propose a system that allows developers to define consistency guarantees on the data (instead of at the transaction level) that can be automatically adjusted at runtime. Different strategies are explored to dynamically adjust consistency by gathering temporal statistics of the data. Moreover, those strategies are driven by a cost model, which associates penalty costs with different degrees of consistency. This project shares our goals of having a data-driven consistency model, however it could be difficult to associate (real) costs with transactions; and they provide only 3 consistency levels. In VFC^3, consistency degrees are not pre-categorized and hence can be used to fine-tuning and better optimize the overall system performance.

Caching at the application-level can significantly improve the performance of both web servers and underlying databases, since it can save expensive trips to the data base (e.g., [4,18]). The granularity may vary, storing partial database tables, SQL queries, entire webpages, arbitrary content, etc. The major problems: i) they usually do not provide transactional consistency with the primary data; and ii) application developers have to manually handle consistency and

explicitly invalidate cache items when the underlying data changes (a very error-prone task). In VFC^3, we offer a complete solution that transparently handles consistency and data is always updated w.r.t. the local database.

Regarding cloud data stores, Amazon S3[2] and BigTable provide eventual consistency. PNUTS [9] argue that eventual guarantees do not suit well its target applications; they provide a per-record timeline consistency: all updates to a record are applied in the same order over different replicas. Conflicting records cannot exist at the same time as it is allowed by Dynamo [8].

7 Conclusion

This paper presented a novel consistency model and framework capable of enforcing different degrees of consistency, accordingly to the data semantics, for data geo-replication in cloud tabular data stores.

We implemented and evaluated a prototype architecture of the VFC^3 framework revealing promising results. It is effective on improving QoS, thereby reducing latency, bandwidth and augmenting throughput for a set of key (and typical) workloads; this, while maintaining the requirements about the quality of data provided to end-users.

To the best of our knowledge, none of the existing solutions in the areas of web-caching, database replication offer a similar flexible and data-awareness control of consistency to provide high-availability without compromising performance. Most of them only allow one level of stale data to exist, that is usually configured per application, and follow a blind approach w.r.t. the data that could require or not strong consistency guarantees; while other solutions, that share some of our goals, impose fixed consistency levels that and limited number of application classes or categories.

References

1. Church, K., Greenberg, A., Hamilton, J.: On delivering embarrassingly distributed cloud services. In: HotNets (2008), CR-ENS-GRID
2. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. 37, 42–81 (2005)
3. Brewer, E.A.: Towards robust distributed systems (abstract). In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 2000, p. 7. ACM, New York (2000)
4. Fitzpatrick, B.: Distributed caching with memcached. Linux Journal 2004, 5 (2004)
5. Coulouris, G.F., Dollimore, J.: Distributed systems: concepts and design. Addison-Wesley Longman Publishing Co., Inc., Boston (1988)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2006, vol. 7, p. 15. USENIX Association, Berkeley (2006)

[2] `http://aws.amazon.com/s3/`

7. Lakshman, A., Malik, P.: Cassandra: structured storage system on a p2p network. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, p. 5. ACM, New York (2009)
8. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 205–220. ACM, New York (2007)
9. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1, 1277–1288 (2008)
10. George, L.: HBase: The Definitive Guide, 1st edn. O'Reilly Media (2011)
11. Veiga, L., Negrão, A., Santos, N., Ferreira, P.: Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. J. Internet Services and Applications 1, 95–115 (2010)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, pp. 143–154. ACM, New York (2010)
13. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
14. Lu, Y., Lu, Y., Jiang, H.: Adaptive consistency guarantees for large-scale replicated services. In: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage, pp. 89–96. IEEE Computer Society, Washington, DC (2008)
15. Yu, H., Vahdat, A.: Design and evaluation of a continuous consistency model for replicated services. In: Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, OSDI 2000, p. 21. USENIX Association, Berkeley (2000)
16. Gao, L., Dahlin, M., Nayate, A., Zheng, J., Iyengar, A.: Application specific data replication for edge services. In: Proceedings of the 12th International Conference on World Wide Web, WWW 2003, pp. 449–460. ACM, New York (2003)
17. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency rationing in the cloud: Pay only when it matters. PVLDB 2, 253–264 (2009)
18. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: GlobeCBC: Content-blind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands (2006)