# ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency

Felix Gessert[#1], Florian Bücklers[#1], Norbert Ritter[#1]

[#]*Computer Science Department, University of Hamburg*
*Vogt-Kölln Straße 33, 22527 Hamburg, Germany*
[1]{gessert,bueckler,ritter}@informatik.uni-hamburg.de

*Abstract*— **Today, the applicability of database systems in cloud environments is considerably restricted because of three major problems: I) high network latencies for remote/mobile clients, II) lack of elastic horizontal scalability mechanisms, and III) missing abstraction of storage and data models. In this paper, we propose an architecture, a REST/HTTP protocol and a set of algorithms to solve these problems through a Database-as-a-Service middleware called ORESTES (Objects RESTfully Encapsulated in Standard Formats). ORESTES exposes cloud-hosted NoSQL database systems through a scalable tier of REST servers. These provide database-independent, object-oriented schema design, a client-independent REST-API for database operations, globally distributed caching, cache consistency mechanisms and optimistic ACID transactions. By comparative evaluations we offer empirical evidence that the proposed Database-as-a-Service architecture indeed solves common latency, scalability and abstraction problems encountered in modern cloud-based applications.**

## I. INTRODUCTION

The emergence of cloud computing, Database-as-a-Service (DBaaS) and "NoSQL" databases has demonstrated a clear demand for scalable database systems with cloud-capable, web-based interfaces [1]. There has been a popular shift in application design towards relying on DBaaS systems to manage application data. A very recent development is that of "Backend-as-a-Service" (BaaS), where the cloud database service takes the place of a classic application server and allows applications (in particular mobile and web applications) to directly connect to it. Despite the surge of interest in DBaaS and BaaS, there are unsolved problems. The most prominent one is that of high network latencies incurred by database requests from remote clients. In this paper, we propose a solution to this problem which leverages the existing global web caching infrastructure to serve database objects with minimal latency.

Different studies have shown the dramatic effect of latency on user behavior. For instance, Amazon found that an additional latency of 100ms resulted in 1% less revenue and Google measured that increasing the load time of search results by 500ms decreased user traffic by 20% [2]. As an average web page load performs 90 HTTP requests [2] - many of which fetch data from the backend - the DBaaS/BaaS plays an eminent role for user-perceived latency. This is particularly true when data fetched from the DBaaS/BaaS is used to render the web site or web app and thus blocks other operations that incur latency.

ORESTES (**O**bjects **REST**fully **E**ncapsulated in **S**tandard Formats) is our proposed BaaS/DBaaS architecture to overcome these current limitations of the Backend-as-a-Service model. ORESTES targets the read-intensive, latency-sensitive workloads common for most web applications (e.g. blogs, social media or e-commerce platforms). A REST interface and server-side schema management allow the database to be accessed by globally distributed users (e.g. mobile devices), systems (e.g. a PaaS cloud) and applications (e.g. a web app).

Data is stored in a scalable underlying (NoSQL) database system that can be chosen according to functional and non-functional requirements. For example an application needing complex queries, linearizable consistency and scalability would employ the ORESTES middleware on top of a database system such as MongoDB whereas an application requiring high write-availability would choose an underlying database system such as Riak or Cassandra. ORESTES exposes the same CRUD (create, read, update, delete) operations and the same object-oriented schema interface for all systems, while allowing database-specific query languages and extensions. To achieve read scalability and solve the latency problem, caching is performed by various kinds of web caches at the HTTP level. To make this kind of caching feasible, we introduce a cache consistency algorithm based on Bloom filters that prevents stale cache reads and ensures consistency. As we found many applications in need of transactions for some infrequent operations (e.g. a reservation process), we introduce a generic mechanism for optimistic ACID transaction handling at the middleware level.

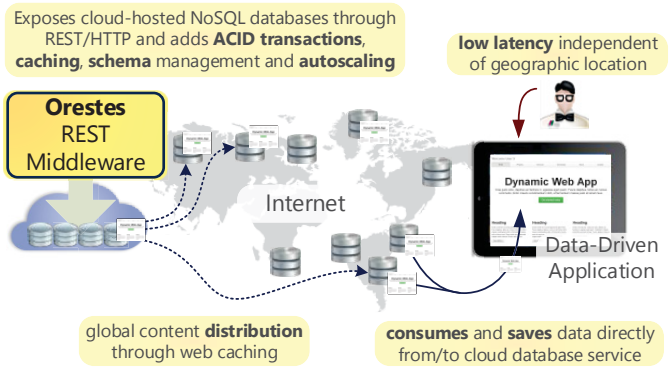


Fig. 1 Motivation for ORESTES: low latency Database-as-a-Service.

The contributions of this paper are threefold:

1. We introduce a Database-as-a-Service middleware that defines a universal REST API ("protocol") for object-oriented persistence, queries, schema management and ACID transactions.
2. We outline a solution to the latency problem of web and mobile applications by adapting the scalability model of expiration-based web caching and load balancing for

database systems. We introduce an algorithm for cache consistency without synchronous invalidation.

3. In experiments for a cloud computing scenario, we illustrate that the approach does indeed demonstrate the desired performance and scalability properties.

The paper is structured as follows. Section II motivates the problem and gives background. Section III and IV describe our approach. In Section V an evaluation is given and Section VI and VII present related work and conclusions.

## II. Background

Web-based applications have been gaining immense popularity over the last years. They have become the dominating application form across many industries and businesses. But with increasing complexity of these applications the number of network requests necessary to assemble the user interface (page) from HTML, Images, CSS and JavaScript files drastically increases. An average web application performs 90 such requests, transferring 1.3 MB of data [2]. However, page load time is governed almost solely by network latency rather than bandwidth [3]. At a fixed latency page load time saturates at a bandwidth of 4-5 Mbps, which is available in most networks. Decreasing latency at a fixed bandwidth on the other hand leads to a strictly proportional decrease in page load time [2].

There is an abundance of studies showing the high impact of page load time on customer satisfaction, revenue and traffic. For this reason, the model of Single Page Applications (SPAs) has been proposed to decrease page load time and is gaining wide adoption. SPAs are web applications that only reload and redraw parts of the page using JavaScript and Ajax. This delivers a user experience similar to native mobile or desktop applications. By moving most application logic and working data into the client, the application is able to respond much faster by avoiding latency-sensitive roundtrips to the server for fetching an updated page for each user interaction. Authentication, data validation and data persistence is handled by the server side. The web application only fetches data that is necessary to update the user interface, for example further results from a product catalog. Most native mobile applications (e.g. Android apps) behave similarly, as they contain all parts of the user interface and contact a backend for fetching and storing data.

Even though SPAs reduce the number of requests to the backend, performance is still limited by the network latency of data requests. We propose Orestes to solve this problem. Orestes provides a generic framework to the serve the database and backend "as-a-Service", i.e. handles authentication, data validation and data persistence through a REST/HTTP-API. To achieve low latency for database requests, objects are cached through the existing HTTP-based web caching infrastructure, including client caches (browsers), proxies, web caches in carrier networks, CDNs, and server-side caching servers. These caches are normally used in caching-aware web applications to store immutable content like third-party JavaScript libraries for a fixed timespan. In Orestes objects are stored in these caches by serving them with appropriate caching information and handling consistency and versioning in the middleware (Figure 2).

For illustration, please consider the following simplified example of an SPA for blogging which requests an article the user likes to view. The JavaScript application loads the content of the article to update the respective parts of the user interface:

*entityManager.find(articleId)*

The Orestes client library therefore issues the corresponding HTTP call to the Orestes middleware:

*GET /db/Articles/{articleId}*

If any of the intermediate caches holds a cached representation of the object, it will be directly served to the client with strong latency benefits. The object is returned in the JSON (JavaScript Object Notation) format and is structured according to the corresponding class schema definition for articles. If none of the caches holds the object, the Orestes REST server loads it from the database system. This is similar for complex queries, updates, schema management and authentication which are always handled on the server side. Low latency is primarily important for read requests, as applications often perform writes asynchronously and reads synchronously, i.e. with a user waiting for a response.

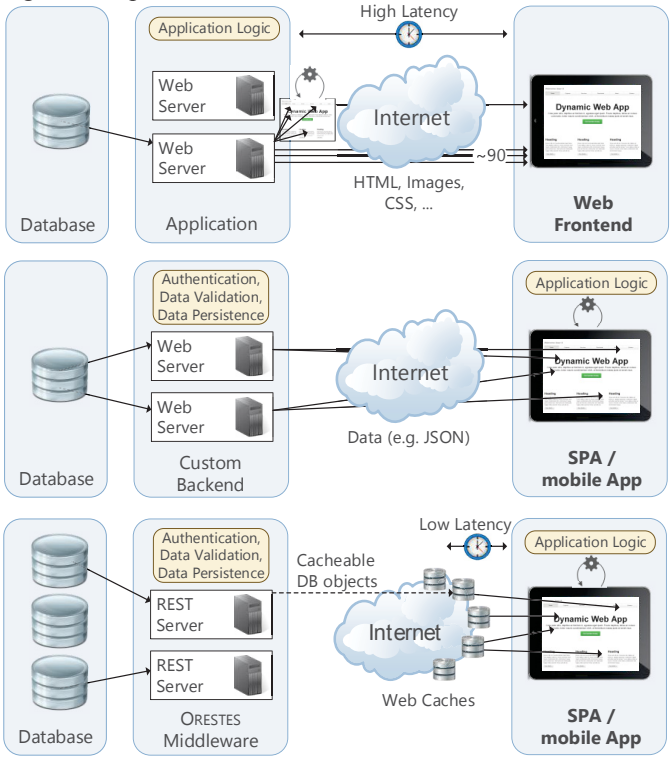


Fig. 2 Different models for dynamic web and mobile applications.

## III. Architecture

We define elastic read scalability as a system's ability to increase the possible request rate of (key/id-based) reads by using server nodes added to the running system. Latency is determined by two factors: the network round-trip time of a request and the processing overhead of the database (disk seeks, quorums, etc.). Read latency is low for any read request that takes less time than a full round-trip to the database server.

To our best knowledge, web caching for database access has not yet been researched and hence is a new strategy to make

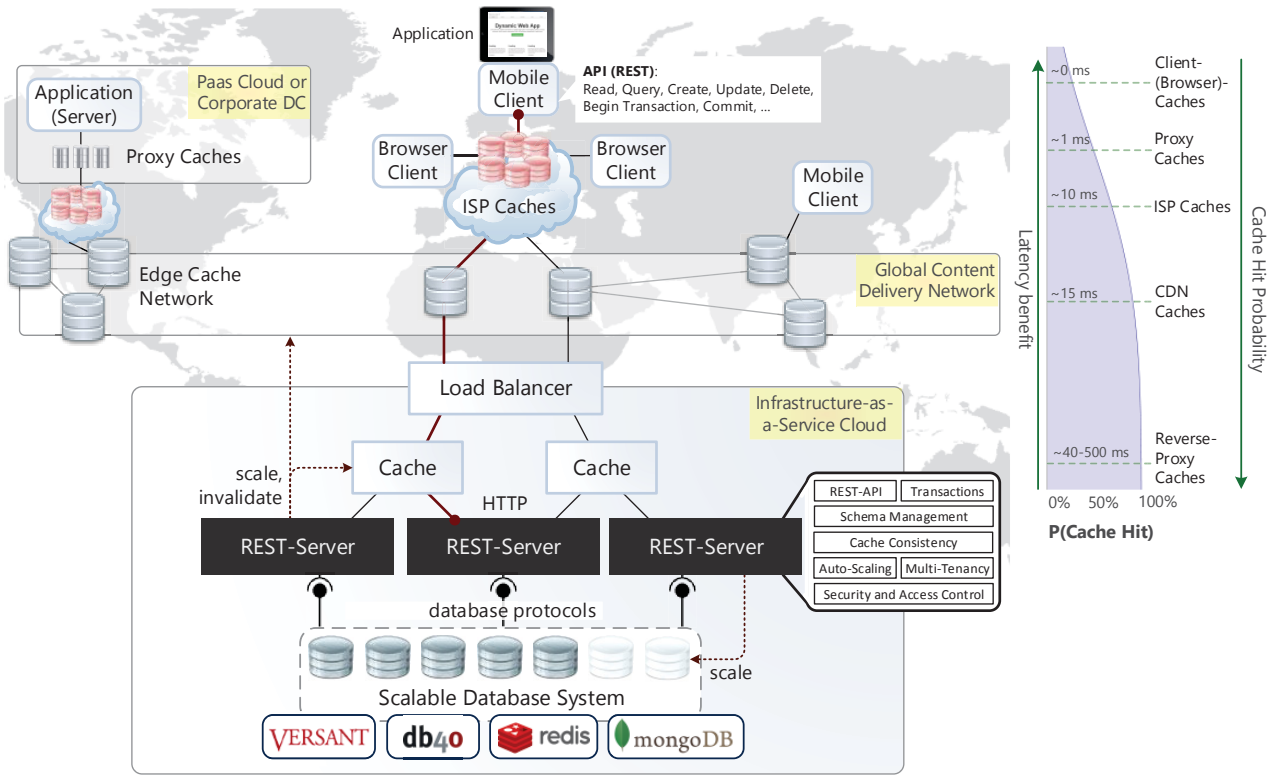


Fig. 3 ORESTES Architecture.

databases benefit from web technologies. The main problem which needs to be solved by our approach is the caching model of HTTP: the caching lifetimes of objects are predefined and ad-hoc invalidations usually impossible [3]. Web caches cannot natively provide cache coherence if ad-hoc changes may occur. Therefore we propose four consistency strategies:

1. **Read-Any** (*RA*): clients may receive any version; staleness is bounded by a defined cache expiration time.
2. **Read-Newest** (*RN*): clients receive the newest version using HTTP cache revalidation, i.e. a refresh.
3. **Transactional** (*TA*): the clients' read sets are validated and checked for stale reads at commit time.
4. **Bloom-Filter-Bounded** (*BFB*): by loading a Bloom filter of recent changes, clients are guaranteed to see only object versions that are at least as recent as the database state by the time the Bloom filter was generated.

*Read-Any* and *Read-Newest* follow from the HTTP caching model. *RA* has the strongest latency benefits while *RN* guarantees strong consistency. *Transactional* offers optimistic ACID transactions allowing arbitrary cache reads. *Bloom-Filter-Bounded* gives the best tradeoff between consistency and latency and can be combined with *RA* and *TA*. *TA* and *BFB* are explained in a later section. A consistency strategy can be chosen per operation, session, transaction or application and mixed according to the application's needs.

The proposed architecture is illustrated in Figure 3. The ORESTES middleware is comprised of stateless REST servers which are realized on top of a scalable database system. Building on the numerous advancements in the area of distributed databases, write scalability and query processing remain the duty of the underlying database system. Any database system supporting CRUD operations can be plugged into ORESTES. For transaction support a compare-and-swap and a consistent read operation are also required. The data model (schema), authentication, multi-tenancy, access control, cache consistency and object versioning are completely performed in the ORESTES middleware. Database-specific capabilities (queries, counters, etc.) form additional parts of the REST API. Server-side caches and CDN caches are managed by the middleware, i.e. object updates are propagated as cache invalidations. If ORESTES is deployed in an IaaS Cloud environment, it can leverage elastic resource provisioning to start additional caching servers, database nodes and REST servers.

Clients, which can either be rich clients (SPA, mobile applications) or classic applications (e.g. application servers) access the ORESTES middleware through the caching hierarchy of existing HTTP Caches. Concrete latency numbers and cache hit ratios depend on the workload, geographic position and carrier networks but rough estimations are given in the right part of Figure 3. Incoming client requests are load-balanced over the server-side caches and REST servers, which is enabled by the stateless REST API. The properties the proposed architecture tries to satisfy are summarized in Table 1.

TABLE I
REQUIREMENTS AND THEIR IMPLEMENTATION

| Property | Mechanism |
|---|---|
| Low latency | Existing HTTP caches, e.g. browser caches and Content Delivery Networks |
| Schema | Server-side schema management (schema definition, evolution and validation) |
| Standard formats | HTTP content negotiation, default JSON representations |

| | |
|---|---|
| Cache consistency | Probabilistic algorithm based on a Bloom filter of potentially stale database objects |
| ACID transactions | Scalable optimistic concurrency control |
| Read scalability and elasticity | Web caching, load balancing, workload-aware spawning of new web caches |

### 1. REST/HTTP API

In the ORESTES REST API abstractions are represented by resources identified by URLs, e.g. *queries*, *transactions*, *objects*, *schema*, etc. Operations are expressed through the HTTP verbs GET, PUT, POST, and DELETE. Resources are integrated through Hypermedia, i.e. mutual referencing, similar to web links. For example, a resource for query results has a list of references to the objects matching the query predicate (see example in Figure 4). This is necessary as the HTTP caching model is URL-based and thus only accelerates point lookups by object id. Objects can be received (GET), created/replaced (PUT), updated (POST) and destroyed (DELETE).

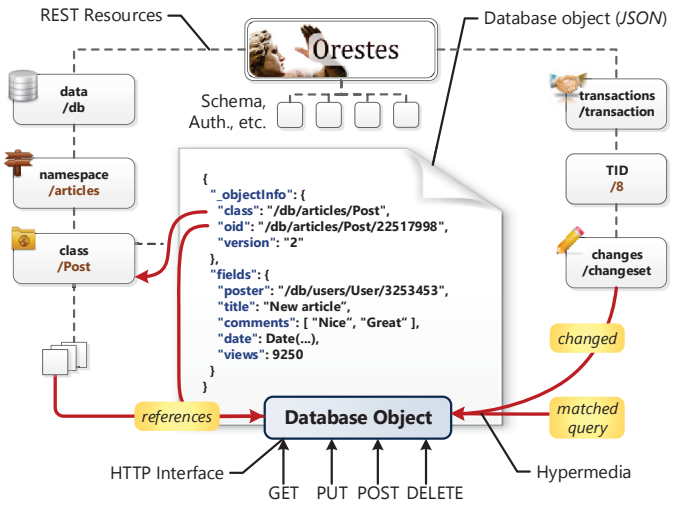


Fig. 4 Example part of the ORESTES REST interface.

ORESTES requires objects to carry version numbers (*Etags*) in order to allow optimistic concurrency. The nature of version numbers is opaque, so any versioning scheme of the underlying database can easily be exposed including version counters, timestamps, vector clocks and content hashes. By default, all resources in ORESTES are represented as JSON objects. ORESTES follows the REST architectural style as described by Fielding [4]. Statelessness of communication and thus load balancing is enabled by not relying on implicit state from request to request (e.g. cookies) [5]. Other constraints (caching, client-server, uniform interface, layered system) are similarly met, yielding a property that other protocols like TCP wire protocols, RPC and SOAP services cannot deliver: inherent read scalability and low latency provided by the infrastructure [6].

### 2. Schema Management

ORESTES introduces an object-oriented data model based on concepts of object databases (e.g. Versant, db4o), object-relational mapping (e.g. Hibernate) and persistence APIs (e.g. JDO, JPA, Entity Framework). For schema-free database systems schema management is completely handled in the ORESTES middleware. The schema consists of classes which define typed fields. Types can be primitives (Integer, String, etc.), typed references and collections (Sets, Lists and Hashes). Nesting of classes is allowed for denormalization to achieve best performance on aggregate-oriented NoSQL databases. Inheritance is supported through horizontal partitioning ("table-per-class"), i.e. inheritance of class fields. This does not require joins for polymorphic reads/queries but can be handled through a union operation over the class hierarchy in the middleware. Fields can have constraints (e.g. not null, domain checks) which can be checked in ORESTES. Consistency Constraints that limit availability (e.g. uniqueness constraints [7]) are disallowed.

Access Control Lists may be associated with a schema to constrain reads and writes to certain users, groups and roles at field or class level. That way a schema for user profiles could limit updates to the creator and limit general read access to public fields. Objects of classes that constrain read access are not cached, so permissions can be checked in ORESTES.

For schema updates, communication between the REST servers is necessary: each server has to know the schema. ORESTES supports two kinds of schema updates:

1. **Safe Updates** (adding fields, changing field types to a parent type): updates are commutative, associative and idempotent and can be performed asynchronously
2. **Unsafe Updates** (deleting and renaming fields, changing field types to a non-parent type): updates can lead to update anomalies and have to be coordinated

Schemas are constructed as state-based CRDTs (Commutative, Convergent, Replicated Data Types) [8] for safe updates. Any REST server receiving a safe schema update asynchronously broadcasts the update to all other servers. Every server applies a *merge* function to the current and received schema. Due to the properties of this function, schema updates can be batched (*associativity*), concurrently performed (*commutativity*) and resent arbitrarily (*idempotence*). Safe Updates thus are non-blocking, efficient and fault-tolerant. Unsafe updates on the other hand need coordination to prevent race conditions and update anomalies. In ORESTES they are therefore coordinated through a two-phase commit protocol, which blocks the database between the prepare and commit phase and is potentially unavailable in case of network partitions.

## IV. SOLVING THE LATENCY PROBLEM

Recent cloud computing services and NoSQL database interfaces are often exposed as REST/HTTP services [1]. Unlike these, ORESTES uses infrastructure-level HTTP caching through mechanisms explained in this section.

### 1. Leveraging Web Caching

In ORESTES, all database objects are explicitly marked as cacheable for a fixed timespan *TTL* (e.g. 30 minutes). This decreases database utilization and reduces network latency, as web caches are optimized for serving many clients concurrently and with minimal delay - without contacting the server. We distinguish between six types of caches that are leveraged in ORESTES, based on their network location (see Figure 5) [3], [5]. A *Client Cache* can be directly embedded in the application (1), e.g. a browser cache. *Server Caches* (5), e.g. in-memory data

grids and *Reverse Proxy Caches* (4) as well as *CDN caches* (6) are controlled and invalidated by the REST servers. *Forward Proxy Caches* (2) are shared caches in the client's network while *Web Proxy Caches* (3) are deployed in carrier networks.

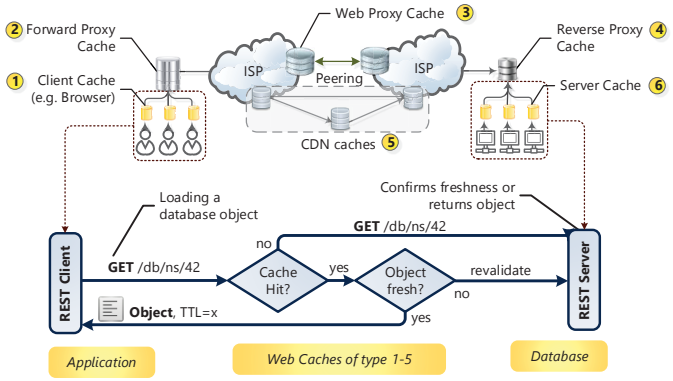


Fig. 5 Types of web caches and their behavior in ORESTES.

Web caches are transparent to ORESTES, as all caching is strictly performed at HTTP level (*Cache-Control*, *Edge-Control*, *Expires* header). Autoscaling and invalidation of CDN and Reverse Proxy Caches through ORESTES relies on a plug-in mechanism for the respective protocols. While scalability of CDN caches depends on individual SLAs and contracts, Reverse Proxy Caches can be scaled towards higher request rates through load balancing and peering (HTCP, ICP, Cache Digests [3]) and higher data volumes through partitioning (CARP [3]).

Figure 5 illustrates the steps of a web cache for a *Read-Any* object requests in ORESTES: if the object was not previously cached, the web cache will forward the request. Otherwise, i.e. if a cache-hit occurs, the web cache will determine whether the local copy is still fresh by checking the object's static lifetime. If the object is still fresh, it is returned to the client without any communication to the database. If the object turns out to be stale, the web cache will revalidate the object by asking ORESTES to send the latest object should the *ETag* (version) have changed or just to indicate that the current version is still fresh.

### 2. Consistent Caching

The *Bloom-Filter-Bounded* consistency strategy bounds staleness of objects to the age of a Bloom filter that clients fetch at connection and transaction begin and update periodically. The Bloom filter is a compact, probabilistic set indicating object ids changed in a sliding time window of size *TTL*. The ORESTES middleware maintains it as a central, in-memory Counting Bloom Filter. It contains the flat Bloom filter that is actually transferred but also allows removal of objects. Every REST server maintains a priority queue to remove an object it has served an update request for and has not been updated for *TTL* minutes, i.e. cannot lead to any stale reads. We skip many details for brevity, but the basic read algorithm of a client using *BFB* to load an object *o* is:

1. If *o.id* is not contained in the Bloom filter *o* can be normally loaded (from caches) and is guaranteed to be at least as recent as the Bloom filter.
2. If *o.id* is contained, a HTTP revalidation request is issued, as cached copies are likely stale. The Bloom filter has a false positive rate *f*, which is the small share of all revalidations (typically 1%) that are unnecessary.

Consider the example that during the last $TTL = 600$ seconds a total of 10 000 distinct objects were updated and the Bloom filter's false positive rate is $f = 0.01$. The Bloom filter then has a size of only 12 KB which is similar to a very small image. There are only $2.081 * \ln(1/f) \approx 9.6$ bits required per object. If the number of distinct stale objects increased by 50% due to an unexpected surge of updates $f$ would increase from 1% to only 4.6%. Thus, even large update spikes only have a small effect on false positives. The optimal choice of parameters mainly depends on the workload distribution and latency. The (Counting) Bloom filter solution is very well-suited for the scenario as it combines space-efficiency, salability and very high lookup and update performance [9].

### 3. Scalable Caching-aware Optimistic Transactions

Recent NoSQL databases designed for scalability (like BigTable or Dynamo) mainly sacrifice the transaction concept [10]. Considerably fewer systems support transactions but constrain them to predefined data partitions, stored procedures or weaker consistency guarantees (like HAT, Megastore, ElasTras, Calvin or H-Store [11]). As serializable transactions are provably irreconcilable with high availability [7] we argue that trading availability for full ACID properties is a good solution for many applications, in particular if transactions are only performed occasionally.

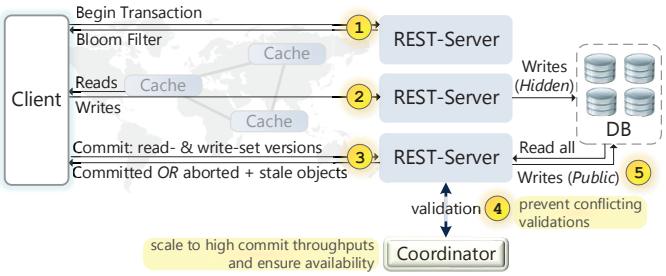


Fig. 6 High-level architecture of our implementation.

We propose a modification of BOCC+ (Backward Oriented Optimistic Concurrency Control) [12] called **SCOT** (**S**calable **C**aching-aware **O**ptimistic **T**ransactions) that allows the occurrence of stale cache reads without degrading ACID semantics. A SCOT transaction is illustrated in Figure 6:

1. Upon transaction begin the client receives a Bloom filter marking potentially stale objects.
2. All reads of objects not contained in the Bloom filter can be served from caches. Writes always reach the server, where they are isolated and invisible to others.
3. On transaction commit, version numbers of all objects that were read (*read-set*) are transferred to the server.
4. The REST server handling the commit uses a coordination service (Zookeeper) to exclude validations of transactions with overlapping *read-* and *write-sets*.
5. If all versions of the read-set are still valid (no conflicting transactions) the private writes are made visible. Otherwise the transaction is rolled back.

We have to omit many details here, but the above procedure ensures atomicity and isolation. Durability is a property of the

underlying database and consistency is checked for each individual write. By the use of a coordinator *SCOT* transactions may suffer unavailability in case of network partitions, but non-transactional operations are not affected. The major benefit of *SCOT* is its ability to be added to a non-transactional database by using the ORESTES middleware and at the same time being compatible with caching. Giving a formal correctness proof and empirical evaluation of *SCOT* is a core part of our future work.

### 4. Implementation

The ORESTES implementation consists of several Java projects which realize the REST server(s), the networking, scaling and caching logic, *SCOT*, schema management, *BFB* data structures and algorithms as well as a generic web dashboard for data browsing, administration and development. Currently four database implementations (Versant, db4o, Redis, MongoDB) plug into the different ORESTES interfaces which abstract different database concepts (CRUD, data listing, queries, versioning, transactions, schema). For Java clients, a low-level API and JDO are implemented. For SPAs and other web applications we designed a JavaScript port of the Java Persistence API (*JSPA*) extended by Backend-as-a-Service functionalities.

We will publish the complete ORESTES implementation as an open-source project, soon. We already contributed our distributed Bloom filter library (*http://divinetraube.github.io/*ORESTES-*Bloom-filter)*, as we found existing implementations to be lacking in various aspects. By $\chi^2$ goodness of fit tests, we found that the trade-off for $f$ and speed was best for the Murmur hash (detailed evaluations and statistics online).

## V. EVALUATION

This section gives an evaluation of the latency benefits ORESTES achieves by leveraging the global caching infrastructure. A workload is generated through clients using the object-oriented persistence API JDO. As the underlying database of ORESTES we chose the Versant Object Database (VOD), as other benchmarks like PolePos [13] indicate that it outperforms object-relational mapping and hence gives a significant comparison. We compare access through our REST/HTTP middleware to the baseline of the native TCP access protocol of VOD.

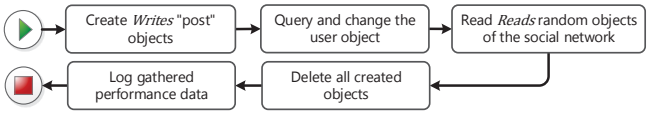


Fig. 7 Workload executed by each client.

The experiments use a complex object model for a social networking scenario relying on object-oriented concepts like aggregation, association, generalization, etc. The clients perform a navigating access pattern by serially and randomly loading objects stored in the database (either drawn from a uniform or Zipf distribution) and writing others using transactions, reads, queries, updates and deletes. This navigational access is the most common pattern in object-oriented persistence [14]. The workload generating clients (Figure 7) can be configured by defining their number, the ratio of reads to writes, the amount of objects in the database, the probability distribution objects are drawn from and the number of consecutive runs.

The random read operations are central for the experiments. We give a simple stochastic model for the expected number of cache hits for objects drawn from a uniform distribution over several runs by applying a model known for universal hashing and the birthday paradox: $m$ buckets represent the objects in the cache and loading an object is equivalent to assigning a read marker to a bucket with probability $1/m$. When loading the $k$th object, the probability of a cache miss is:

$$P(k,m) = \left(\frac{m-1}{m}\right)^{k-1} =: P_{miss}^{k-1}$$

Let the random variable be the indicator variable $I_k$ which is 1 for a cache miss when loading the $k$th object and 0 otherwise. Summing up the number of cache misses over all $n$ load operations gives the expected number of cache misses:

$$E(\#misses) = I_1 * 1 + I_2 P_{miss}^1 + \cdots + I_n P_{miss}^{n-1} = \sum_{k=1}^{n} P_{miss}^{k-1}$$

$$= \sum_{k=0}^{n-1} P_{miss}^{k} = \frac{1-P_{miss}^{n}}{1-P_{miss}} = m * \left(1 - \left(\frac{m-1}{m}\right)^{n}\right)$$

This follows from the properties of geometrical series and linearity of expected values. The expected amount of cache hits is $E(\#hits) = n - E(\#misses)$. So 300 objects and 500 operations with a read ratio of 99% gives 252.46 expected cache hits for the first, 448.55 for the second and 486.10 for the third run. Numbers extracted from the cache logs of an actual execution confirm this (258, 446, 486). This indicates that for read-intensive workloads a considerable speedup is possible.

### 5. Parallel Cloud-based Scenario

For the cloud-based scenario, we use Amazon's Elastic Compute Cloud (EC2) as an IaaS platform. The experimental setup is depicted in Figure 8. Clients and ORESTES/database are distributed with a network latency of $165ms \pm 2ms$. 50 client instances (VMs) are triggered simultaneously. VOD uses client-embedded caches, while ORESTES uses a shared web cache (Squid). Compute capacity is measured in EC2 Compute Units (ECUs) which roughly equal the capacity of a 1.0-1.2 GHz 2007 Xeon. Clients have 1 ECU and 1.7 GB RAM, while the cache and VOD have 4 ECUs and 7.5 GB RAM.

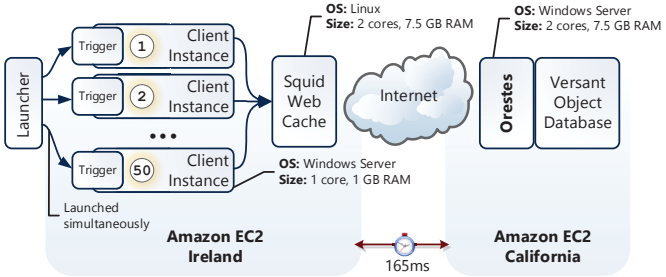


Fig. 8 Cloud computing scenario.

The average overall execution times are shown in Figure 9. We defined a read/write ratio of 90%/10%, considered different sizes of the database (300, 3000, 30000 objects) and executed the workload in three consecutive runs. The results show how our approach outperforms the native VOD protocol and increasingly so as the cache hit ratio rises over the runs (cache warming). For higher numbers of total objects, the execution

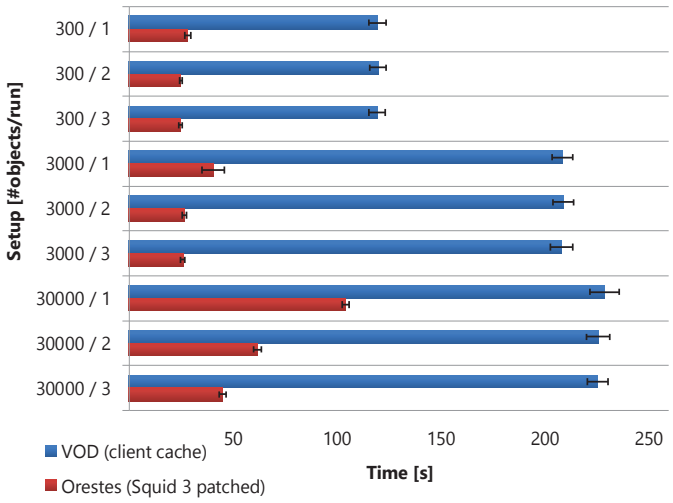


Fig. 9 Average execution time of 50 concurrent clients. Bars indicate the empirical standard deviation. The y-axis shows stored objects and the run.

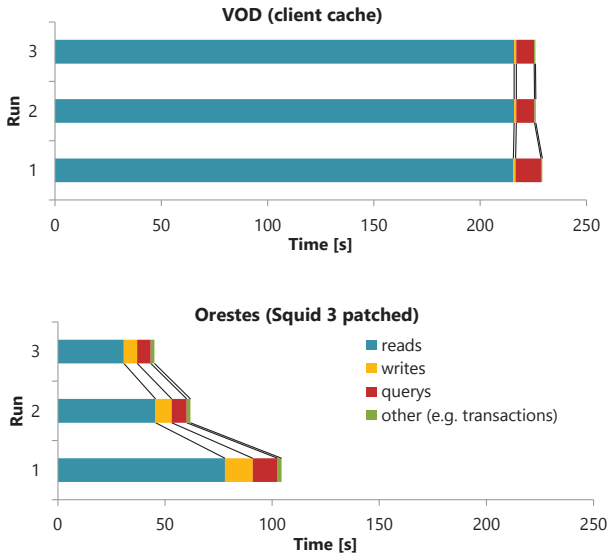


Fig. 10 Average execution time of the cloud computing scenario broken down into types of operations for a read ratio of 90% and 30000 database object.

time increases as some purposefully costly operations (like an unindexed query) are performed (Figure 10). The execution times in the third runs using ORESTES demonstrate that the database can perform these operations better, when the effort of serving objects is shifted to the cache. Writes are slightly slower in ORESTES as they are issued as separate requests ("HTTP cache invalidation by side-effect"), while VOD clients buffer writes and transfers them in bulk at commit time.

### 1. *Effects of geographical distribution*

We study the performance of ORESTES for the case of geographical application/database distribution. Web caching is performed in the client's network, which is located in Hamburg, Europe. The database is deployed in the California, USA, creating the typical Backend-as-a-Service setup. Client and web cache are VMs with 2GB of RAM and one core of a 3.4 GHz Xeon Sandy Bridge processor. The round-trip time between client and database is $180ms \pm 5ms$ over a virtual private network (VPN). We compare different web caches: Squid 2, Squid 3, Microsoft TMG and a patched version of Squid 3 for which we contributed a fix for a mistake in the TCP specification of the Nagle algorithm that is out of the scope of this paper.

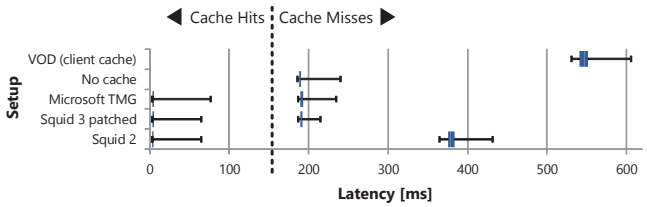


Fig. 11 Latency of reads, blue box: 25 to 75 percentile, bars: total range.

The experiments are performed using a working set size of 300 objects, 3 consecutive runs and different read/write ratios (50%/90%/99% reads). Figure 11 shows the latency of fetching an object for the different setups comparing ORESTES without web caching, ORESTES with different web caches and the native VOD protocol. VOD's in-memory cache hit latencies are too small for the millisecond scale. An average HTTP cache hit has a network latency of roughly 1-2 $ms$ which is faster than a local disk access. The figure also shows that Squid 2 and VOD need two resp. three TCP round-trips.

As shown in Figure 12, ORESTES outperforms native VOD for all read ratios. As foreshadowed by the stochastic analysis, the increasing number of cache hits in the second and third run further reduces the overall execution time. Microsoft TMG and the patched Squid 3 web caches yield the best performance: the performance advantage of ORESTES (web caching) over native VOD (client caching) is factor 2.5 in the first, 6.46 in the second and 10.87 in the third run. ORESTES profits from read-intensive workloads. This becomes obvious when considering the share of reads in the total execution times as illustrated in Figure 13 for a read ratio of 90%. Read operations dominate the execution of all configurations, but the impact on VOD is strongest.

In summary, the experiments show that the proposed ORESTES middleware is indeed capable of achieving a massive latency reduction speeding up read-intensive applications while still allowing complex queries and transactions. We are currently working on extending the evaluations to *SCOT* transactions, different caching topologies and database backends, *BFB* strategies and parameters as well as exactly quantifying horizontal scalability and availability of ORESTES.

## VI. RELATED WORK

Work on REST interfaces in systems such as CouchDB, Riak, Azure Table Storage, PNuts, Neo4J, HBase, SimpleDB, database.com, Datomic [15], [16] focuses on interoperability and accessibility. ORESTES builds upon this work and extends it to actively use infrastructure support (caching and load balancing) as well as more complex database concepts (schema management, transactions). A scalability pattern often found in large-scale web applications is that of Memcache or other in-memory caches serving requests in place of the primary data store. ORESTES is similar to this approach but also reduces wide-area latency (required for BaaS), automates the process, offers *BFB* consistency and supports transactions. Approaches

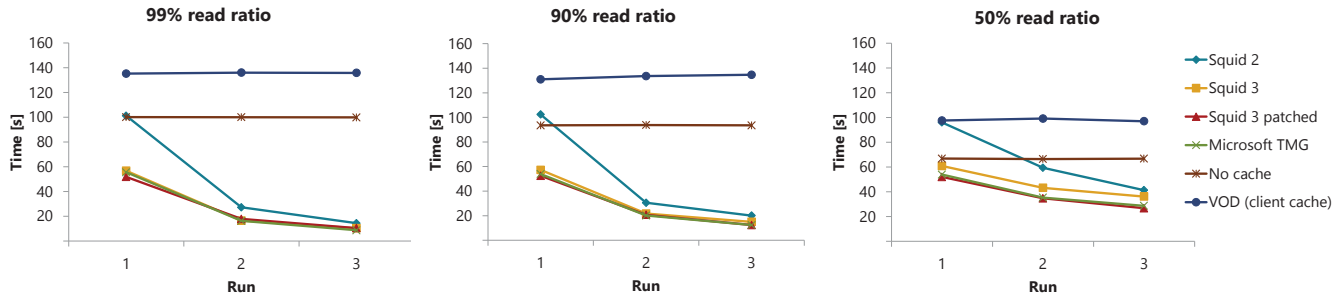


Fig. 12 Runtimes of the single-client benchmark for different read ratios. The benchmark compares the ORESTES protocol using several web caches to the native performance of the Versant Object Database and to ORESTES without any caching at all. Three consecutive runs of the benchmark are performed.

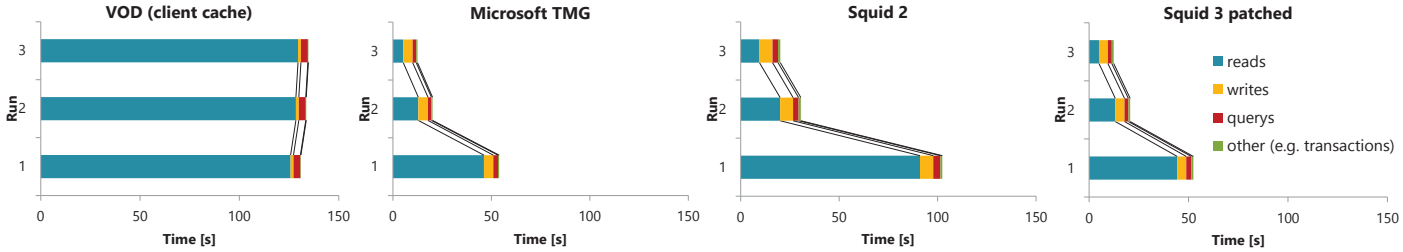


Fig. 13 Share of different operations in the total execution time with a read ratio of 90% on 500 operations and 300 database objects.

based on a caching RBMS like DBProxy and DBCache [17] cannot solve the problem of wide-area latency for scattered (web/mobile) clients and suffer from scalability limitations.

The DBaaS model promises to shift the problem of configuration, scaling, provisioning, monitoring, backup, privacy, and access control to a service provider [18]. However, low latency access, elastic scalability, efficient multitenancy and database privacy remain to be solved [18], [19]. In this paper we addressed the first two: low latency and elastic scalability. Related work on scalability often trades transactions for scalability (most NoSQL systems) or limits them to partitions (Megastore, ElasTras), weaker guarantees (HAT) or stored procedures (H-Store) [7], [10], [19]. Google's F1 [20] uses optimistic transactions similar to ORESTES. It builds on other Google internals (Spanner, Colossus File System), special hardware and does not support web caching or DBaaS/BaaS access.

## VII. CONCLUSIONS

The Database- and Backend-as-a-Service model offer many opportunities for cloud-based application development. However, wide-area latency between accessing clients and cloud services is a largely unsolved research problem that leads to poor performance of web applications. In this paper we described ORESTES, a REST middleware for database systems that employs the existing web caching infrastructure to solve this problem and serve database requests with minimal latency - independent of clients' geographic location. ORESTES uses a set of strategies to mitigate inconsistencies caused by stale cache reads. The *Bloom-Filter-Bounded* algorithm produces efficient staleness bounds, while *Scalable Caching-Aware Optimistic Transactions* (SCOT) realize ACID transactions that detect and prevent stale reads at commit time. ORESTES allows for easy plugging of new database system backends and provides the generic schema, caching and networking capabilities for exposing them as a low-latency service.

## REFERENCES


[1] R. Cattell, "Scalable sql and nosql data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.

[2] I. Grigorik, *High performance browser networking*. [S.l.]: O'Reilly Media, 2013.

[3] M. Rabinovich and O. Spatscheck, "Web caching and replication," *SIGMOD Record*, vol. 32, no. 4, p. 107, 2003.

[4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," University of California, 2000.

[5] R. Fielding et al., "RFC 2616: Hypertext Transfer Protocol–HTTP/1.1, 1999," *URL http://www. rfc. net/rfc2616. html*, 2009.

[6] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly Media, 2007.

[7] P. Bailis et al., "Highly Available Transactions: Virtues and Limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, 2013.

[8] M. Shapiro et al., "A comprehensive study of convergent and commutative replicated data types," 2011.

[9] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.

[10] G. DeCandia et al., "Dynamo: amazon's highly available key-value store," in *ACM SOSP*, 2007, vol. 14, pp. 205–220.

[11] J. Baker et al., "Megastore: Providing scalable, highly available storage for interactive services," in *Proc. of CIDR*, 2011, pp. 223–234.

[12] G. Weikum and G. Vossen, *Transactional information systems*. Morgan Kaufmann Pub, 2002.

[13] "PolePos Benchmark." [Online]. Available: http://polepos.org/. [Accessed: 28-Feb-2013].

[14] M. Grossniklaus, "The case for object databases in cloud data management," *Objects and Databases*, pp. 25–39, 2010.

[15] S. Sakr et al., "A survey of large scale data management approaches in cloud environments," *Communications Surveys & Tutorials, IEEE*, vol. 13, no. 3, pp. 311–336, 2011.

[16] D. Sitaram and G. Manjunath, *Moving To The Cloud: Developing Apps in the New World of Cloud Computing*. Syngress, 2011.

[17] C. Bornhövd et al., "Adaptive database caching with DBCache," *Data Engineering*, vol. 27, no. 2, pp. 11–18, 2004.

[18] C. A. Curino et al., "Relational cloud: A database-as-a-service for the cloud," 2011.

[19] S. Das et al., "Elastras: An elastic transactional data store in the cloud," *USENIX HotCloud*, vol. 2, 2009.

[20] J. Shute et al., "F1: A distributed SQL database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1068–1079, 2013.