# Towards a Scalable and Unified REST API for Cloud Data Stores

Felix Gessert, Steffen Friedrich, Wolfram Wingerath,
Michael Schaarschmidt, Norbert Ritter

Database and Information Systems Group
University of Hamburg
{gessert, friedrich, wingerath, xschaars, ritter}@informatik.uni-hamburg.de

**Abstract:** In the last years, many database-as-a-service (DBaaS) systems have started to offer their functionalities through REST APIs. Examples are record stores like DynamoDB and Azure Tables, object stores such as Amazon S3 as well as many NoSQL database systems, for instance Riak, CouchDB and ElasticSearch. Yet today, there has been no systematic effort on deriving a unified REST interface which takes into account the different data models, schemas, consistency concepts, transactions, access-control mechanisms and query languages to expose cloud data stores through a common interface without restricting their functionality. This work motivates the design of such a REST API as well as the challenges of providing it in an extensible, scalable and highly-available fashion. To this end, we propose the REST middleware ORESTES that consists of an independently scalable tier of HTTP servers that map the unified REST API to aggregate-oriented (NoSQL) data stores. It extracts a wide range of DBaaS concerns (e.g. schema management and access control) and provides them in a modular, database-independent fashion at the middleware-level. To tackle the latency problem of cloud-based web applications we introduce the *Bloom filter-bounded staleness* cache consistency algorithm. It leverages the global web caching infrastructure for geo-replication to allow consistent low latency reads. We furthermore show the first steps towards a *Polyglot Persistence Mediator* that exploits the decoupling of the REST API from the data store to route data and operations based on SLAs.

## 1 Introduction

The success of the database-as-a-service model has brought a wide variety of commercial and research systems. Though many have been extensively studied, there is a lack of taxonomy. Throughout this work, we will distinguish between *managed database services*, *proprietary database services* and *Backend-as-a-Service* (BaaS). In a managed database service a DBaaS provider offers a cloud-deployed DBMS and automates operational tasks such as provisioning, multi-tenancy, backups, security, access control, elasticity, scaling, performance tuning, failover and replication [CJP+11]. Proprietary database services build on newly designed database systems (e.g. Google DataStore based on Megastore [SVS+13]) or integrate different databases to a polyglot persistence environment to offer them through provider-specific protocols and APIs. The Backend-as-a-Service model enhances the DBaaS model by adding abstractions for backend concerns of mobile appli-

cations and web sites (e.g. authentication, push notifications, data validation and assets).

DBaaS systems differ in the degree to which they provide automation of operational tasks, the underlying data store, pricing models, service level agreements (SLAs), multi-tenancy strategies and, most importantly, their interfaces. While Proprietary database services offer provider-specific interfaces, managed database services offer database-specific protocols that were not designed for cloud environments. In this paper, we will focus on *aggregate-oriented* NoSQL services. As described by Fowler et al. [SF12] this encompasses NoSQL databases that have a (potentially denormalized) aggregate as their primary unit of access: *rows* in wide-column and record stores, *documents* in document stores and *key-value pairs* in key-value stores. This aligns well with the resource-oriented model of REST. The key observation and motivation for this work is that aggregate-oriented DBaaS systems can be greatly enhanced by providing them through a single unified and scalable REST API. This is attractive as clients can be reused, data models be shared and applications be migrated. The contributions of this paper are threefold:

1. **Unified REST API**. We propose a Database- and Backend-as-a-Service REST API (see Figure 1) that captures functional concepts (transactions, queries, commutative updates, etc.) across a wide range of systems.
2. **Modular Middleware**. We introduce the modular ORESTES middleware approach, where backend dabatase systems can be exposed through the unified REST API by plugging them into reusable modules (e.g. for partial updates). This yields an architecture where core concepts like authentication and schema management only need to be implemented once and then can be reused for many database systems.
3. **Low latency through consistent web caching**. As achieving low latency is crucial for any user-facing application, we propose the *Bloomfilter-bounded staleness* algorithm to drastically reduce latency through web caching.

The paper is structured as follows. Section 2 outlines the requirements and overall concept before presenting the proposed architecture. In Section 3 an evaluation is given, followed by a discussion of networking aspects relevant for the applicability of REST in the cloud database setting. Section 4 overviews related work and section 5 presents conclusions.

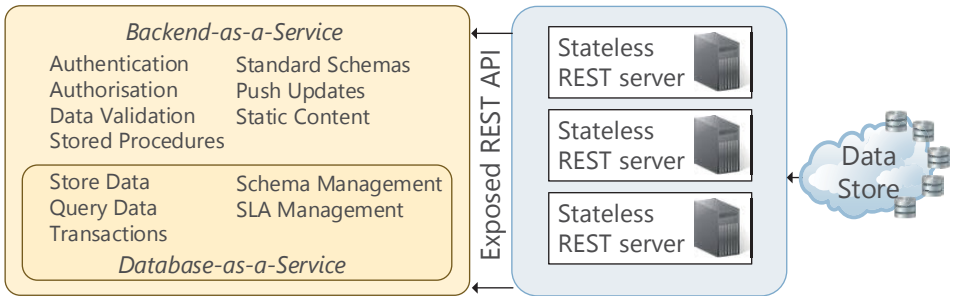



Figure 1: Vision: a scalable REST tier for unmodified, aggregate-oriented data stores.

## 2 Concept and REST middleware architecture

To expose existing data stores as a Backend/Database-as-a-service without prior modification, the ORESTES middleware and its *unified REST API* have to be powerful enough to expose all possible capabilities of the underlying database (e.g. conditional updates) without compromising its nonfunctional properties (e.g. scalability of data volume or linearizable consistency). We distinguish between two types of modules in the middleware. *Data Modules* express the mapping of data operations in the REST API to the underlying database (CRUD operations, queries, indexing, system configuration). Data Modules have to be implemented for each database that is to be exposed through the unified REST API. *Default Modules* on the other hand implement abstractions that can be provided by default on top of databases through data modules and middleware services (authentication, authorization, data validation, stored procedures, push notifications, transactions, schema management, SLA management, elastic scaling). Default modules can be overwritten to leverage existing native capabilities (e.g. table-level ACLs for authorization).

Today, web performance is governed almost completely by latency [Gri13]. Numerous studies by Amazon, AOL, Google, Bing, Akamai and others demonstrated the severe implications of latency[1]. For this reason, web sites are increasingly built as Single-Page Applications (SPAs) to eliminate the overhead of server-side page generation, client redraws and asset refetching. However, data requests remain extremely latency-critical. We solve the latency problem through web caching of database objects. This is a novel approach, as the web caching model is based on a-priori cache expiration times which necessitates a cache coherence mechanism. In this work we propose a dual strategy: *cache invalidations* perform asynchronous purging of server-controlled web caches (reverse-proxies and CDN nodes) and *Bloomfilter-bounded Staleness* ships probabilistic summaries of recently changed objects to clients that can then revalidate (bypass) stale cached copies (in client caches, forward proxies and ISP caches).

**Architecture.** Figure 2 shows the ORESTES middleware architecture. Clients that can either be pure DBaaS clients (application servers) or BaaS clients (browsers and mobile devices) use the unified REST API supported by persistence APIs (JDO/Java and JSPA/-JavaScript). At the server-side, HTTP servers wrap a (potentially distributed) data store which currently can be VOD, db4o (OODBMSs), Redis (in-memory key-value store) or MongoDB (document store). Through the stateless design of the servers, latency and throughput is only bounded by the saturated database system as the middleware can scale horizontally. Objects delivered over REST/HTTP are transparently cached for a fixed duration. To allow server-side stored procedures and data validation through `before` and `after` triggers, every server runs a Node.js process which executes registered client code written in the same JSPA browsers run. This server-side code execution is essential for BaaS when business logic should not be exposed and validation of user input is needed.

**REST API.** The unified REST API is composed of different modules, as shown in Figure 3a. It is specified through a new REST specification format similar to routing lan-

[1] Amazon found that with every 100ms of additional page load time, revenue decreases by 1%. AOL discovered that all users in the 10%-quantile of fastest users visit 7.5 pages, slower users only 5. When increasing load time of search results at Google by 500ms, traffic decreases by 20%. See: http://velocityconf.com/velocity2009

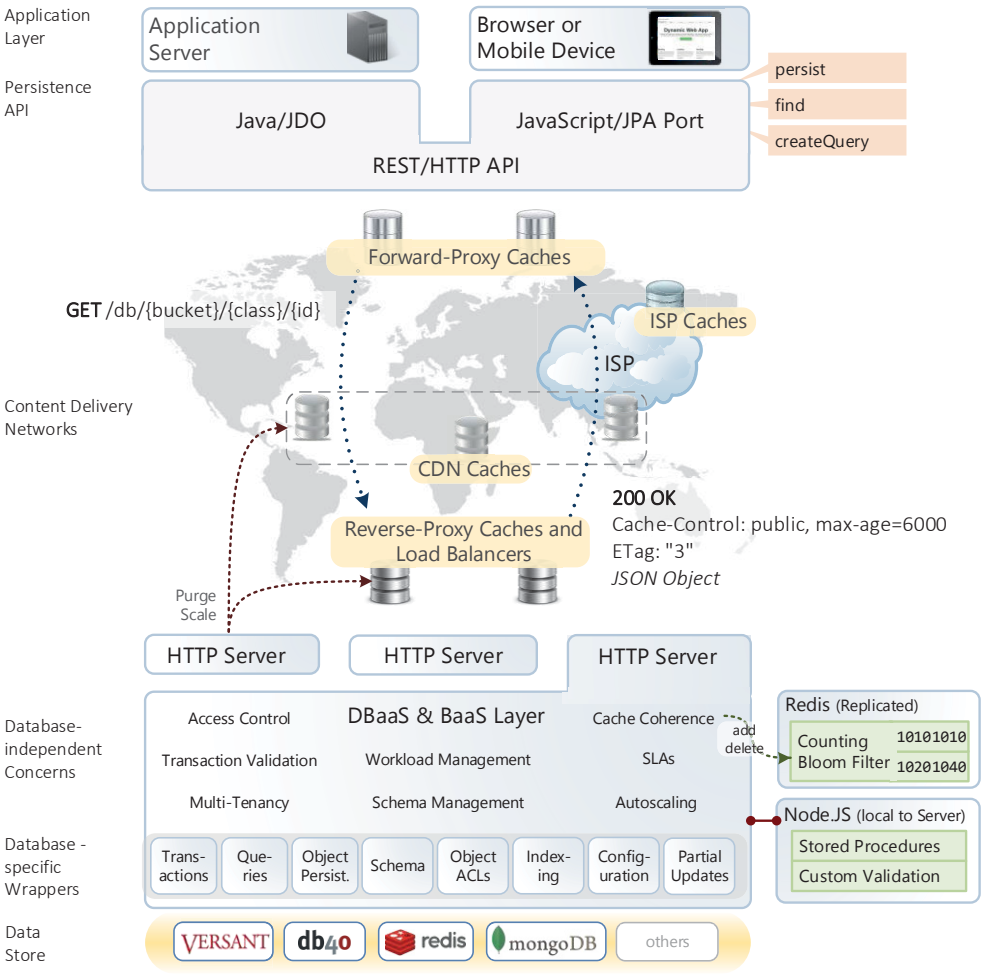
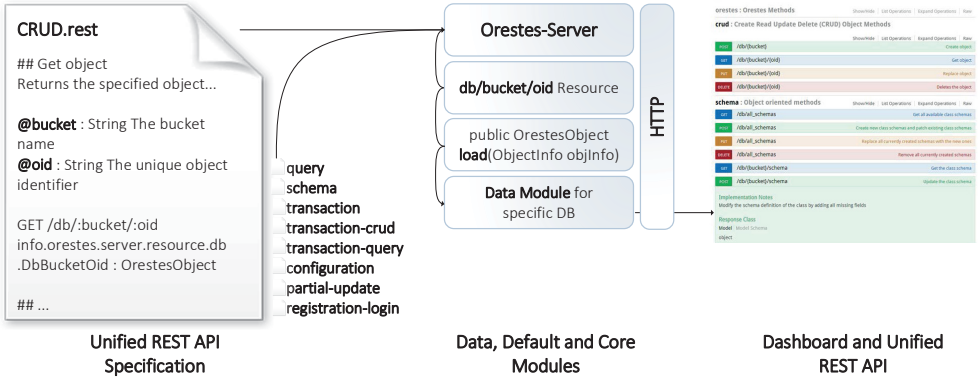



Figure 2: ORESTES middleware architecture with an exemplary object request.

guages in MVC web frameworks (e.g. Play) but enhanced to also capture descriptions and types of parameters and return values. It describes the effects of HTTP methods on resources identified by URI patterns. The REST specification is loaded by the ORESTES HTTP servers for validation, conversion (e.g. from JSON to a schema object) and to generate an interactive REST API documentation on the server's dashboard. The operations declared in the REST specification correspond with one or more methods in the data and default modules that in turn communicate with the underlying database via its specific drivers and protocols. For an end-to-end example, consider a web application in which a user loads his profile. The call is done through the JavaScript JSPA persistence API: `entityManager.find(id)`. It employs the unified REST API: `GET`

`db/profiles/id`. Suppose the ORESTES middleware wraps a MongoDB cluster. After parsing and checking the request, the server will call the CRUD data module's `load` method for MongoDB. The data module will issue a `db.find()` query to MongoDB that returns the document. Based on the schema defined for `Profile`, the server returns the requested object. The fully-typed profile object returned by JSPA can afterwards be displayed, for instance by feeding it into a template of an SPA framework like Angular.js or Backbone. Figure 3b provides examples of other REST API methods.




(a) REST specification lanaguage and its interpretation.

| Request | Response | Explanation |
|---|---|---|
| `POST /db/:bucket`<br>`JSON-Object` | Created object including assigned object id (oid) and version number | Creates a new object. |
| `PUT /db/:bucket/:oid`<br>`JSON-Object` | Created/replaced object | Replaces or creates an object using an object id. The request can be conditioned on a version. |
| `GET /db/:bucket/:oid` | Database object | Fetches or revalidates an object. Request can be answered by web caches. |
| `GET /db/:bucket?query`<br>`&start=0&count=-1` | List of matching ids | Executes a DB-specific ad-hoc query. |
| `GET /db/all_schemas` | All schemas for all classes | Retrieves all schemas. |
| `POST /transaction` | ID and URL of the transaction | Starts a new transaction. |
| `POST /db/:bucket/:oid/:field` | Success or validation failure | Performs a partial update (e.g. counter increase). |

(b) Example requests from different modules.

Figure 3: Composition of the unified REST API through modules.

**Schema Management.** ORESTES employs explicit object-oriented schemas as a default module. Schema-free databases (Redis, MongoDB) thus get "bolt-on" rich schemas, whereas schema-aware databases (db4o, Versant) can expose their own schemas. We think that explicit schemas are an advantage as they allow type-checking and validation to prevent data corruption. An ORESTES-schema is a mapping of field names to types. A type can be *primitive* (String, Boolean, Integer, Float, Date, GeoPoint), a *reference* (object Id), a *collection* (Set, Map, List), an *embedded object* (defined by a schema) or *JSON* (Array, Object). Through the JSON types the schema grants all freedoms of schema-free data stores. However, this lack of structure is usually unnecessary as schemas can be asynchronously evolved. Every ORESTES server holds all schemas in memory and broad-

casts client-initiated schema changes to all other servers. There are two kinds of schema changes: *Safe Changes* (adding fields, changing field types to a parent type) are commutative, associative and idempotent and thus can be safely transferred asynchronously. As safe changes are non-destructive, they can be lazily applied. *Unsafe Changes* (deleting and renaming fields, changing field types to a non-parent type) have side-effects and need to be broadcasted in a coordinated, blocking fashion using 2PC.

**Access Control.** To extend a DBaaS to a BaaS, user and access control are required as clients cannot be trusted. In ORESTES this is achieved by role-based access-control (RBAC) and default schemas for common BaaS use cases: users, groups, installations, messages, social posts, in-app purchases etc. These BaaS schemas have special semantics. For instance, a user instance automatically gets created when a user registers via OAuth or OpenId and logins are checked against the user database object. Registration and login can thus be provided as database-independent default modules. On successful login, an access token is generated, which uses an enhanced scheme of Fu et al. [FSSF01]: $expires = t \,\&\, data = u \,\&\, digest = HMAC_k(exp = t \,\&\, data = u)$ where $u = id(user) \,\&\, \{id(role) \mid user \in role\}$ and $k = tenant\_id \,\&\, server\_secret$, i.e. the token contains the user's id and his roles and a signature (a HMAC for performance reasons). Based on this token the servers check *schema-level ACLs* and *object-level ACLs*. An ACL grants or denies access to users and roles for certain operations: *read* and *write* on object-level and *read*, *query*, *create*, *delete*, *extend schema*, *subclass schema* on schema-level. In contrast to schema-level ACLs, object-level ACLs need a specific data module for mapping. In MongoDB, for instance, object level ACLs are enforced by conditioning data operations over a per-object, indexed ACL field containing allowed and denied users and groups. As the access token is self-contained, no shared server state is required. Even reverse-proxy caches and CDNs can perform ACL checks for read operations if they support sufficiently expressive configuration languages (e.g. Varnish VCLs). To enable this, a merged view of schema- and object-level ACLs is attached to objects, which are used by these caches and stripped from objects before sending them to clients. To execute critical business logic, server-side JSPA scripts can be registered as stored procedures, before- or after-handlers. Before- and after-handlers are called with the operation and schema they are defined for (e.g. to validate user input), whereas stored procedures are explicitly invoked over the REST API (e.g. to submit an order). Through Node.js, JSPA can likewise be used on the client and server. REST API calls are handled via inter-process communication between the Java-based ORESTES servers and the scripts in Node.js.

**Polyglot Persistence.** Making use of the loose coupling provided by the unified REST API, we introduce a *Polyglot Persistence Mediator* (PPM), which automatically routes data to different backend data stores. Decision-making is based on a per-schema SLA, which combines different functional and nonfunctional requirements. The prototype supports a decision between MongoDB and Redis with MongoDB as the default principal storage facility. The SLA allows for simple annotations like saving an attribute in a certain datastructure (e.g. priority queue or hash-set) for fast updates and special queries (e.g. top-k) as well as for more refined, parameter based requirements. For instance, a hotspot-field could be annotated with a certain maximum write latency. The persistence mediator takes the schema annotations to make a runtime decision for the appropriate backend, based

on a scoring of available backends for the demanded SLA. Similarly, fields can be annotated with particular boundaries on latency, availability and replication factors. Cross-data store materialization is handled through parallel background processes in each ORESTES server, which guarantee a strict upper time limit for each objects materialization. This new approach enables applications to make use of polyglot persistence on a declarative basis. Though this is ongoing work, we present some promising preliminary results in Section 3.

**Consistent Caching.** If read access is not restricted by ACLs, objects are returned as publicly cachable, i.e. carrying a `Cache-Control:public,max-age=ttl` header, so intermediate caches save object copies for `ttl` seconds. We define 4 consistency levels:

**Read-Any (RA)** clients may receive any previously cached version of an object; staleness is only bounded by the expiration time `ttl`.

**Read-Newest (RN)** explicitly fetches the newest object version using an HTTP cache revalidation request, i.e. through refreshing intermediaries.

**Transactional (TA)** clients read-sets and write-sets are validated and checked for stale reads at commit time. This approach is cache-compatible and conceptually similar to the strategy employed for optimistic transactions in Google's F1 [SVS+13] and the Omid system [GJK+14] and have been described for ORESTES in [GBR14].

**Bloom-Filter-Bounded Staleness (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* has the strongest latency benefit – in the best case, the object is resident in the client cache and even a round-trip to a content delivery network (CDN) is cheap (in the order of 20ms). *Read-Newest* guarantees freshness at the cost of a full round-trip to the cloud data-center. As shown in Figure 2, web caches are transparently leveraged at HTTP level. While client-, ISP- and forward-proxy caches cannot be invalidated by the server, CDNs and reverse-proxies support explicit invalidations. ORESTES uses a plug-in mechanism to propagate updates in their respective purging protocols. SLAs provided by CDNs differ significantly, for instance Amazon Cloudfront rate-limits total invalidations and takes more than 10 minutes to apply them, whereas other CDNs (e.g. Fastly) grant arbitrary invalidation frequencies and instant application. This has to be taken into account when coupling the REST API to a CDN as it strongly effects cache-hit ratios and staleness.

*BFB* solves the following fundamental problem: if objects are cached for a predefined duration `ttl`, ad-hoc changes will result in stale cache reads. BFB introduces the new idea of transferring the cache invalidation task from the server to the client. To achieve this, the server tracks all objects that have been updated in the last `ttl` second in a shared Counting Bloom Filter. It represents the set $S$ of potentially stale objects $o_i$, i.e. all objects whose logical write timestamp $WTS(o_i)$ lies less than `ttl` seconds in the past and is not equal to their creation timestamp $CTS(o_i)$: $S = \{o_i \in db | now() - ttl > UTS(o_i) \wedge WTS(o_i) \neq CTS(o_i)\}$. Upon connection, the flat Bloom filter is piggybacked to the client. Before retrieving an object over the REST API, the client performs an $O(1)$ hash-lookup in the Bloom filter to decide whether the object should be fetched normally, i.e. allowing cache-hits or if a HTTP revalidation should be attached to the request, i.e. letting caches validate the freshness by the server through an `If-None-Match:version-number` condition. The benefit of this scheme is that any HTTP-compliant cache is implicitly used by

the unified REST API, yielding the largest conceivable geo-replication system without any up-front deployment and with a strong guarantee – the age of the Bloom filter is a strict lower time bound for staleness.

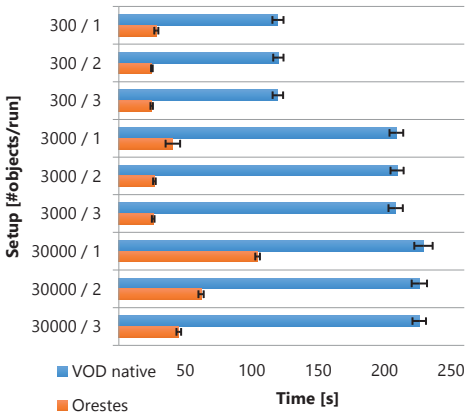
The Bloom filter is very efficient for representing the objects in $S$, since due to its probabilistic nature it is very small. A Bloom filter with $m$ bits using $k$ hash functions to store $n$ distinct objects, has a false-positive rate $f \approx (1 - exp(-kn/m))^k$. This allows to trade off false-positives (revalidations instead of cached reads) against size, i.e. the overhead of fetching the Bloom filter. For instance, with $f = 0.01$, each object requires 9.6 bits. So if 1000 individual objects were changed in the last `ttl` seconds, only 1.17 KB would need to be transferred. At the server-side, a Counting Bloom filter is employed to support object removal. As every update operation leads to an addition in the filter, it has to support high update rates as well as fast retrieval of the flat Bloom filter. As shown in Figure 2, this is achieved by storing the Counting Bloom filter as an in-memory bit vector (in Redis) using a scheme where each bit-position $b_i$ is represented by a materialized Bloom filter bit and the respective counter: $b_i = (count > 0, count)$. The materialized bits are stored contiguously, i.e. to fetch the whole Bloom filter it can be directly read from main memory. To remove objects from the filter that have not changed in the last `ttl` seconds, each server maintains a priority queue of objects inserted into the filter. As time advances the top elements from priority queue are successively evicted from the filter. The Counting Bloom filter can also be employed to estimate the update frequency of objects to compute their optimal `ttl` value. The enabling technique, known as *spectral Bloom filters*, performs cardinality estimation of an object using its counter values [BM02]. This estimation divided by the current `ttl` yields the update frequency. Our open-source Bloom filter framework, including a detailed statistical and performance analysis, can be found online[2]. Bloom filter-bounded staleness is ongoing work. We are currently working on a sharded Counting Bloom filter as a *Commutative Replicated Data-Type* (CRDT) to also serve it in a highly available eventually consistent manner, as well as statistical framework for the optimal choice of Bloom filter parameters and caching durations.

## 3 Evaluation

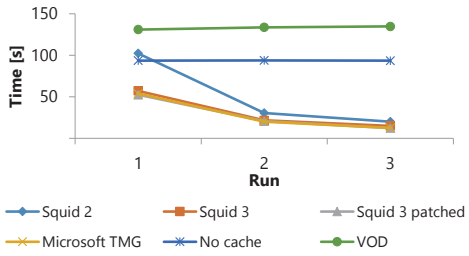
We conducted several experiments to evaluate the performance implications of the REST/ HTTP layer and the effects of caching. Figure 4a shows the results of a setup where 50 client VMs using a forward-proxy cache and ORESTES with VOD are separated by a network latency of $165ms \pm 2ms$ using the Amazon EC2 regions Ireland and California. The clients concurrently execute a microbenchmark modeled after a social networking scenario (500 operations, read/write ratio 90%/10%, navigational access with sporadic queries). There are three consecutive runs for different sizes of the database comparing ORESTES exposing VOD against the native binary VOD protocol. The graph shows that the average execution time is heavily reduced with ORESTES as a consequence of its caching approach. Figure 4b reports the results of the same microbenchmark for a setup of hardware machines using a single client, different caches (both in Hamburg) access-

[2] github.com/DivineTraube/Orestes-Bloomfilter

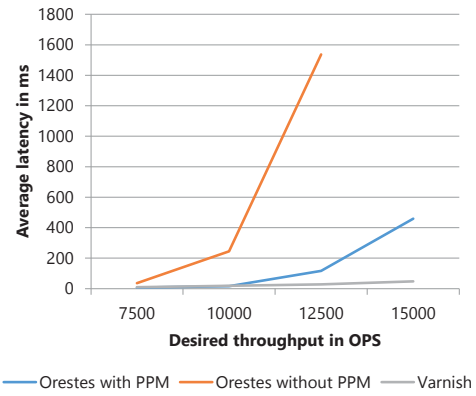
ing ORESTES/VOD (California). The results show that the large performance advantage is consistent across different web caches. This is a consequence of ORESTES relying on standard HTTP caching. Please note that both experiments only consider forward-proxy caching. In our current work, we are tackling the problem of also considering client caches, CDNs and reverse-proxies, which are likely to yield even more drastic results.



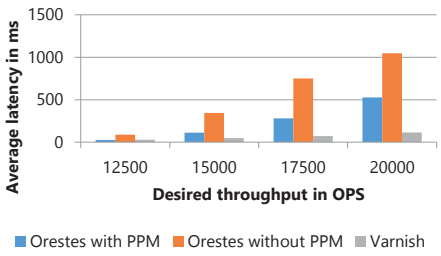

(a) Read-heavy cloud scenario.




(b) Behaviour of different web caches.




(c) Cloud-based PPM Evaluation.




(d) Effects of increased DB performance.

Figure 4: Evaluation of ORESTES, its caching behaviour and the PPM.

We also report preliminary results for the Polylgot Persistence Mediator. The evaluation considers a web site/app displaying top-k elements based on a counting metric in content objects (e.g. impressions for a news article). The schema SLA specifies that counter increases and top-k queries should both be performed with small latency. The PPM in ORESTES, which is backed by MongoDB and Redis, routes counter increases to a Redis *sorted set* mapping counters to object ids. The PPM materializes counters from Redis to MongoDB in 1 minute intervals so they can also be accessed through MongoDB-queries (for other queries than top-k). The experiment shown in Figure 4c is performed on Amazon EC2 using a c3.4xlarge instance for both the client and ORESTES server and a weaker m1.large instance for each Redis and MongoDB. MongoDB is attached to an EBS storage volume with 1000 provisioned IOPS and is accessed using the `acknowledged` write concern that does not guarantee persistence. Figure 4c shows the throughput of performing

100k counter updates comparing ORESTES with PPM to ORESTES with MongoDB. This is compared to the HTTP performance "gold standard" by replacing the ORESTES server by a Varnish reverse-proxy serving a static response from memory. The graph shows that the PPM is able to handle much higher throughput with lower latency by using the SLA to route counter updates to Redis. It is also evident that the overhead of the ORESTES layer is very low – despite additional database communication latency remains comparable to Varnish up to 2/3 of the maximum throughput. Figure 4d shows a consistent result when instance sizes of the database machines are increased to c3.4xlarge.

**HTTP enhancements.** Though HTTP is widely used for cloud data management, it has restrictions. The evolving HTTP/2.0 standard will address some performance problems of HTTP/1.1 through multiplexing, binary encoding, header compression and server push. However, it leaves HTTP semantics unchanged. We propose two semantic improvements to HTTP for REST-based cloud data management: *variable-precision timestamps* and *multipart caching*. Timestamp-based versioning schemes of underlying databases cannot be exposed as HTTP timestamps – their precision is limited to seconds. This lack of precision forces REST APIs to expose database timestamps as opaque ETags. The cost is that ordering semantics are lost and HTTP conditional requests (e.g. If-Modififed-Since) are not possible. In recent NoSQL databases timestamp-based versioning is common, for instance MongoDB's ids are prefixed with a 32-bit timestamp, HBase uses ms-timestamps for cell-level-versioning and Cassandra generates $\mu$s-timestamps for its last-write-wins concurrency. Therefore we suggest to extend the *HTTP-date* type defined in RFC 2616 [FGM+99] to also allow UTC timestamps (ISO 8601) that support arbitrary second fractions: `HTTP-date = rfc1123-date | rfc850-date | asctime-date |` **`ISO-8601-date`**. A second suggested improvement is *multipart caching*. Multipart messages (RFC 1341) are used in HTTP to send a collection of resources in bulk (e.g. query result sets). However, these resources do not carry individual metadata, only encoding information. We propose a `multipart/resources` type allowing Cache-Control headers as well as canonical URIs. This eliminates additional round-trips and makes GET request profit from previous bulk replies.

**Networking improvements.** We also found that cache-aware REST APIs for cloud databases lead to two problems: *falsely non-persistent HTTP methods* and *temporary TCP deadlocks*. We encountered non-persistent methods in the Squid proxy cache which is arguably the best-researched web cache [Nag04] and heavily used by large-scale web sites such as Wikipedia and Flickr. In Squid, persistent TCP connections [FGM+99] were handled incorrectly as depicted in Figure 5, where the server connection is terminated after just one request. This misbehavior was caused by falsely treating requests containing a body as non-idempotent, instead of only POST requests. Since the default rule for non-idempotent messages was to open a new connection as a precaution, this caused at least one costly additional round-trip (TCP handshake) for POST/PUT/DELETE requests. We reported and corrected this issue which lead to significant performance improvements. A second issue with an even higher performance impact is still present in all current versions of Squid, Microsoft TMG and possibly other caches: a temporary TCP deadlock of 200-500ms (depending on the OS). It is caused by an interference of the *Nagle algorithm* and the *delayed ACK algorithm* which are defined in the TCP protocol standard to increase

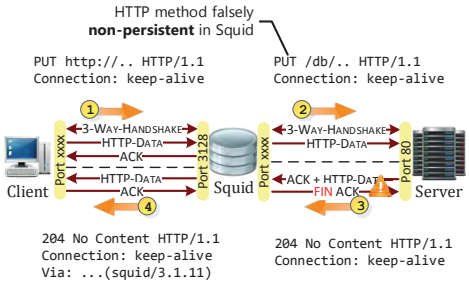



Figure 5: Non-persistent HTTP methods.

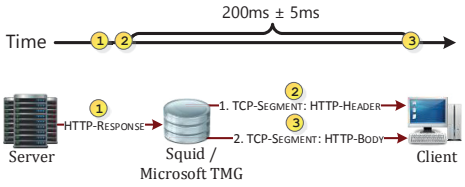



Figure 6: Temporary TCP deadlock.

the effectiveness of TCP buffer management. Figure 6 illustrates the problem: **(1)** When the server returns the HTTP response, the web cache processes the HTTP header and body and passes them to the system's socket interface for forwarding in two consecutive steps. In modern operating systems, the Nagle algorithm is enabled by default and follows the intuition of buffering data until new ACKs arrive or a complete TCP segment can be filled. **(2)** As there is no outstanding ACK from the client, the HTTP header will be sent immediately. The client receives the first packet but does not instantly acknowledge it, as its socket's delayed ACK algorithm tries to send ACKs piggybacked with data or for every second full-sized segment it receives. **(3)** As HTTP is a request/response protocol, the client has no data to send, so the ACK is delayed until a timeout of 200ms (depending on the OS) occurs. The web cache's Nagle algorithm then detects the incoming ACK and sends the queued HTTP body segment. Affected systems (e.g. Squid and TMG) should amend the problem by buffering data in the application or turning off the Nagle algorithm. The temporary deadlock is detrimental to the performance of REST APIs, in particular for latency-sensitive cloud database workloads. Until the problem is addressed in the TCP specification, we suggest a temporary solution: small HTTP messages can be filled with whitespace characters to reach the full segment size that triggers immediate propagation.

## 4 Related Work

REST APIs for DBaaS systems have been tackled from different directions. Google's GData and Microsoft's OData are two approaches for a standard HTTP CRUD API that are used by some of their respective cloud services. Many commercial DBaaS systems offer custom REST APIs tailored for one particular database (e.g. DynamoDB, Cloudant). A first theoretic attempt for a unified DBaaS REST API has been made by Haselman et al. [HTV10] for RDBMSs. Unlike the Orestes REST API, these approaches cannot make use of caching and do not offer a scalable middleware for exposing arbitrary aggregate-oriented data stores, nor can they be used in a BaaS setting. Two older approaches for reducing database access latency are DBCache and DBProxy [APTP03]. Both require a full-fledged RDBMS as a proxy. Bloom filters are widely used for content summaries [BM02] (e.g. in Summary Cache [Nag04]). To our best knowledge, though, they have

never applied to client-driven cache consistency. A different approach to low latency is geo-replication which can be either synchronous (e.g. in MDCC, Spanner, MegaStore [SVS+13]) or asynchronous (e.g. BigTable/HBase, Cassandra, Riak). Unlike the web caching approach, this requires database-specific protocols and deployments.

## 5 Conclusions

In this paper, we present a unified modular REST API that can expose aggregate-oriented data stores as a Database- and Backend-as-a-Service. The ORESTES middleware provides the REST API in a scalable, extensible fashion allowing data stores to easily plug-in and reuse common functionality like schema management, authentication, access control, caching and transactions. To solve the latency problem of the modern web, we proposed the BFB cache consistency scheme that offloads revalidations to clients using an efficiently managed Bloom filter representing a sliding window of recently changed objects. We also present the first steps towards a Polyglot Persistence Mediator that routes data to different storage backends based on declarative SLAs. The evaluation shows that the unified REST/HTTP API achieves high throughput and very low latency.

## References


[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for Web applications. In *Proceedings of the ICDE*, page 821–831, 2003.

[BM02] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Math.*, 2002.

[CJP+11] Carlo Curino, Evan PC Jones, Raluca Ada Popa, Nirmesh Malviya, et al. Relational cloud: A database-as-a-service for the cloud. In *Proc. of CIDR*, 2011.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and other. RFC 2616. 1999.

[FSSF01] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The Dos and Don'ts of Client Authentication on the Web. In *USENIX Security Symposium*, page 251–268, 2001.

[GBR14] Felix Gessert, Florian Bücklers, and Norbert Ritter. ORESTES: a Scalable Database-as-a-Service Architecture for Low Latency. In *CloudDB*, 2014.

[GJK+14] Ferro Daniel Gómez, Flavio Junqueira, Ivan Kelly, Benjamin Reed, et al. Omid: Lock-free Transactional Support for Distributed Data Stores. In *ICDE*, 2014.

[Gri13] Ilya Grigorik. *High performance browser networking*. O'Reilly Media, 2013.

[HTV10] T. Haselmann, G. Thies, and G. Vossen. Looking into a REST-Based Universal API for Database-as-a-Service Systems. In *CEC*, page 17–24, 2010.

[Nag04] S. V. Nagaraj. *Web caching and its applications*, volume 772. Springer, 2004.

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL distilled*. Pearson Education, 2012.

[SVS+13] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.