Quaestor: Query Web Caching for Database-as-a-Service Providers

Felix Gessert*
Baqend
fg@baqend.com

Michael Schaarschmidt*
University of Cambridge
michael.schaarschmidt@cl.cam.ac.uk

Wolfram Wingerath
University of Hamburg
wingerath@informatik.uni-hamburg.de

Erik Witt
Baqend
ew@baqend.com

Eiko Yoneki
University of Cambridge
eiko.yoneki@cl.cam.ac.uk

Norbert Ritter
University of Hamburg
ritter@informatik.uni-hamburg.de

ABSTRACT

Today, web performance is primarily governed by round-trip latencies between end devices and cloud services. To improve performance, services need to minimize the delay of accessing data. In this paper, we propose a novel approach to low latency that relies on existing content delivery and web caching infrastructure. The main idea is to enable application-independent caching of query results and records with tunable consistency guarantees, in particular bounded staleness. QUAESTOR (Query Store) employs two key concepts to incorporate both expiration-based and invalidation-based web caches: (1) an Expiring Bloom Filter data structure to indicate potentially stale data, and (2) statistically derived cache expiration times to maximize cache hit rates. Through a distributed query invalidation pipeline, changes to cached query results are detected in real-time. The proposed caching algorithms offer a new means for data-centric cloud services to trade latency against staleness bounds, e.g. in a database-as-a-service. QUAESTOR is the core technology of the backend-as-a-service platform Baqend, a cloud service for low-latency websites. We provide empirical evidence for QUAESTOR's scalability and performance through both simulation and experiments. The results indicate that for read-heavy workloads, up to tenfold speed-ups can be achieved through QUAESTOR's caching.

1. INTRODUCTION

In the web and online industry, page load times strongly affect user satisfaction and central business metrics such as revenue, time-on-site, conversion and bounce rates. Various studies by large web and e-commerce companies have quantified this effect. For instance, Amazon has found that 100 ms of additional latency decrease revenue by 1%. Google measured that 500 ms of additional page load time decrease traffic by 20% [30].

Slow page load times have three sources. When a website is requested, the first source of loading time is the **backend**. It consists of application servers and database systems and assembles the website. The latency of individual OLTP queries and the processing time for rendering HTML slow down the delivery of the site.

The **frontend**, i.e. the website displayed and executed in the browser, is the second source of delay. Parsing of HTML, CSS and JavaScript as well as the execution of JavaScript that can block other parts of the rendering pipeline contribute to the overall waiting time.

As of 2017, loading an average website requires more than 100 HTTP requests [2] that need to be transferred over the **network**. This requires numerous round-trips that are subject to physical network latency. This third source of delay typically has the most significant impact on page load time in practice [30].

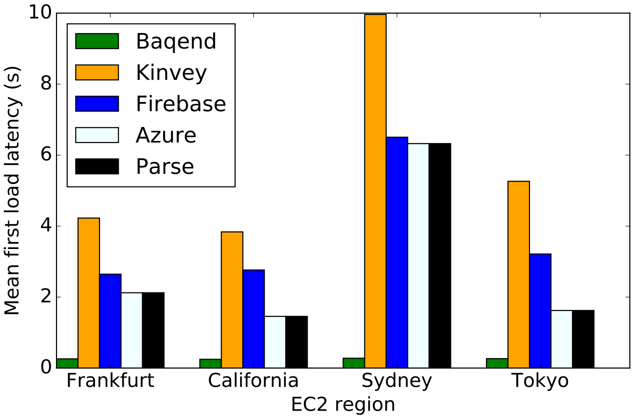


Figure 1: Page load time comparison for different Backend-as-a-Service providers.

Network bandwidth, client resources, computing power and database technology have improved significantly in recent years. Nonetheless, latency is still bound by physical network round-trip times and can hence only be solved by bringing data closer to clients. Baqend develops a cloud platform based on this idea, to help programmers build fast websites with a novel approach to web caching. The central idea is to leverage all available web caches to not only cache immutable data but also cache database records and volatile files. This is made possible through a Database-as-a-

*These authors contributed equally.

Proceedings of the VLDB Endowment, Vol. 10, No. 12
Copyright 2017 VLDB Endowment 2150-8097/17/08.

Service (DBaaS) API that detects changes to objects and files and triggers invalidations [28]. The practical effects are illustrated in Figure 1 using the example of a simple news website loaded from different geographical locations with a cold browser cache and a warm CDN cache. The comparison between our approach and a number of popular, commercial DBaaS providers (Firebase, Parse, Kinvey, Azure Mobile Services) is open-source and can be validated in a web browser[1]. The implementations only use simple key-based access (CRUD) to render a data-driven website in the client. The work presented in this paper extends these results to caching complete query results.

Today, to the best of our knowledge, no other DBaaS system is capable of exploiting the expiration-based HTTP caching model and its globally distributed content-delivery infrastructure. In this paper, we extend the idea of DBaaS web caching from simple key-based access [28] to query results. **Quaestor** (Query Store) is a comprehensive DBaaS system for automatic query result caching that Baqend employs for data storage in its high-performance Backend-as-a-Service. Quaestor completely relies on standard web caching to provide low-latency data access with rich consistency guarantees. We specifically discuss and implement Quaestor for aggregate-oriented NoSQL databases that Baqend is based on (MongoDB and Redis). The approach however is applicable to any system serving dynamic data over a REST/HTTP-based interface.

Motivation. To minimize the effects of backed and frontend latency, web applications are experiencing a sustained shift towards client-centric, "serverless" architectures. Web applications now commonly consume persistence and business logic through REST APIs of database/backend-as-a-service (DBaaS) systems. While this widely adopted *single-page application* model improves usability and developer productivity, web performance now is governed almost exclusively by request latency [30]. Caching is a well-studied technique for minimizing distance and latency between clients and data. The problem addressed in this paper is maintaining consistency while caching dynamic database queries and records. The key idea we exploit is the involvement of the client in a cache coherence protocol for bounded staleness by providing precise staleness information in a summary data structure.

The expiration-based web caching model gained little attention for data management in the past, as its static expirations (time-to-live) were considered irreconcilable with query results that change unpredictably. In this paper, we propose a solution to this apparent contradiction by showing that cache coherence can transparently be maintained by clients. To this end, we introduce the *Expiring Bloom Filter* data structure that captures potentially stale query results and records. By piggybacking the filter at load time, clients can determine which query results and records can safely be fetched from caches or trigger a revalidation if needed. At the same time, the DBaaS pro-actively purges stale results from invalidation-based caches (e.g. content-delivery networks and reverse-proxy caches). This is achieved by a streaming invalidation system that detects changes to cached query results in real-time.

Example. As an example, consider a social blogging application. To retrieve posts on a particular topic, the client queries the DBaaS:

```
SELECT * FROM posts
WHERE tags CONTAINS 'example'
```

This (pseudocode) query is posed as an HTTP GET request. The web's infrastructure consisting of caches, load balancers, routers, firewalls and other middleboxes handles the query similar to any other request issued by web sites. In particular, any expiration-based caches (browser caches and ISP caches) as well as invalidation-based caches (content-delivery networks (CDNs) and reverse-proxy caches) are allowed to answer the query from their local cache, if the DBaaS previously provided a time-to-live (TTL) indicating cacheability for a defined time span.

Challenges. In order to make this caching scheme applicable, Quaestor solves three problems:

1. **Invalidation detection.** Does a given update operation change the result set of cached queries?
2. **Cache Coherence.** How can caches be kept consistent when they cannot be invalidated by the DBaaS?
3. **Cacheability.** Which queries and records are cacheable and what is their optimal TTL?

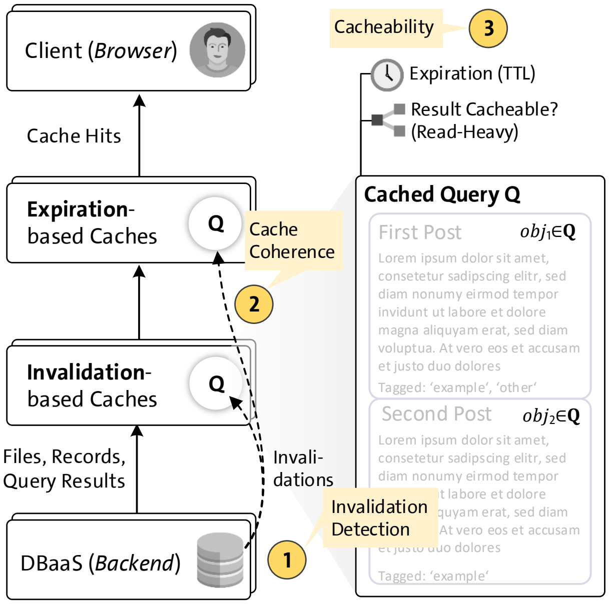

Client (Browser)
Cache Hits
Expiration-based Caches
Q
Cache Coherence
2
Invalidation-based Caches
Q
Files, Records, Query Results
Invalidations
Invalidation Detection
DBaaS (Backend)
1
Cacheability
3
Expiration (TTL)
Result Cacheable? (Read-Heavy)
Cached Query Q
First Post
$obj_1 \in Q$
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores
Tagged: 'example', 'other'
Second Post
$obj_2 \in Q$
Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor
Tagged: 'example'

Figure 2: The three central challenges of query web caching.

These challenges are illustrated in Figure 2. For every database operation, the DBaaS has to determine whether it invalidates any cached queries or records **(1)**. This is enabled by *InvaliDB*, a scalable subsystem for detecting invalidations of cached query results in real-time. In the above example, an invalidation would be triggered if a blog post contained in the query result is changed or a previously non-matching post adds a tag that matches the query predicate. Cache coherence **(2)** of expiration-based caches is based on an *Expiring Bloom Filter* (EBF), which declares any potentially stale content. Clients check the EBF before each query to decide whether cached results are permittable or a revalidation should instead be performed in order to proactively update any stale caches. Through different refresh strategies, the EBF guarantees Δ-atomicity [29] as a consistency guarantee with clients freely being able to choose the staleness bound Δ. The cache hit rates are maximized by statistically deriving expiration estimates (*TTL Estimator*) and deciding which query results are worth caching **(3)**.

Contributions. The proposed model is a good match for common web workloads that are mostly read-heavy with many clients accessing the same data before it is updated [32]. To the best of our knowledge, Quaestor is the first approach that provides fresh query results served over the web caching infrastructure and can thus improve performance and scalability of database services

[1] http://benchmark.baqend.com/

without requiring additional server infrastructure. QUAESTOR's achieved consistency guarantees are similar to those of Pileus [49], but we employ widespread web caches instead of custom replication sites and extend the purely object-based approach to queries.

The contributions of this paper are threefold:

- We propose a comprehensive, service-independent approach for caching dynamic query results and records with rich default consistency guarantees (bounded staleness, monotonic reads and writes, read-your-writes).
- We introduce a scalable middleware infrastructure for maintaining cache coherence through a query matching pipeline and Expiring Bloom Filters.
- We provide empirical evidence that for web-typical, read-heavy workloads, tremendous latency improvements can be achieved with arbitrarily bounded staleness.

This paper is structured as follows. Section 2 gives some background on the challenges of caching. Sections 3, 4, and 5 present the key techniques used to make query caching feasible on web caches: a cache coherence mechanism, a query invalidation system and a model for dynamic TTL estimation. In Section 6, an in-depth evaluation of QUAESTOR is given. Section 7 summarizes related work and Section 8 concludes.

2. BACKGROUND

The two primary causes for page load times are the physical network latency required to transfer assets and dynamic (Ajax) content of the web application and the processing overhead imposed by fetching data from databases [30]. Caching has previously been used to address both problems separately. Database caching [5, 37, 11] can reduce query latency at the server side, which however only constitutes a small part of the overall end-to-end latency. Web caching is frequently employed to cache immutable files [32, 24, 12, 33, 52, 23] and is hence only applicable to a small subset of data and explicitly excludes all dynamic data modern applications are composed of. We seek to address the issues of both approaches by combining them. As the most general form in which data can be served to end devices are views and queries, QUAESTOR's goal is to cache them while giving consistency guarantees.

Web Caching. To use web caching for data management, the limited capabilities and guarantees of the HTTP web caching model have to be respected. Its two primary mechanisms are *revalidation* and *expiration* [30]. For any resource, the server assigns an explicit time-to-live (TTL). Every cache is allowed to store the resource for the defined TTL and then expires it. Clients and caches can revalidate (presumably) stale resources and thus bypass cached copies to have the origin server confirm freshness based on a version (`Etag`) or modification date (`Last-Modified`). In addition to this, *invalidation-based* caches support dedicated TTLs specific to invalidation-based caches and asynchronous invalidations from the server that purge stale content [42].

Challenge. Traditional web caching does not give any guarantees on freshness when expiration-based caches, e.g. browsers, are involved. Furthermore, web caches cannot execute any application-specific logic, instead they only serve non-expired resources by their unique URL. For cache coherence, expiration-based caches can only be updated through client-triggered revalidations. Hence a mechanism is required that timely and efficiently enables clients to revalidate stale content. Though many DBaaS systems are accessed through HTTP-based REST APIs, to the best of our knowledge, none of them employs web caching.

Geo-Replication. QUAESTOR is orthogonal to geo-replication which can also decrease query latency. Many geo-replicated systems, however, either sacrifice consistency guarantees [17, 22, 19] or induce high write latencies of typically two to three wide-area round-trips [9, 44, 35, 38, 21, 45]. Furthermore, in contrast to related systems such as Pileus [49], QUAESTOR does not require special replication infrastructure, but instead leverages unmodified and readily available web caches. In the best case, queries are answered by standard browser caches of end devices, while requests to geo-replicated database systems require at least the round-trip to the nearest replica site.

Application model. A database-as-a-service targeted at serving queries and data directly to end users is often referred to as a backend-as-a-service. Baqend and QUAESTOR adopt this model in order to achieve latency reductions in an application-independent fashion. In contrast to traditional 3-tier-applications, we assume JavaScript-based web or mobile applications accessing data through QUAESTOR's REST API to load the application (e.g. JavaScript, HTML, images) and dynamic data (queries and records). QUAESTOR therefore provides DBaaS functionality such as query processing, authorization and schema management. It is agnostic of its underlying database system and directly answers HTTP requests from browsers for CRUD (create, read, update, delete) operations, queries and file downloads to make them cacheable as described in the following sections.

In summary, we identify the following problem: In order to empower the simplistic web caching model to cache queries and records, cache coherence has to be implemented using both client-side revalidations and server-side invalidations. In the following sections, we present how QUAESTOR achieves this for the context of a document-oriented database service. We assume records to be rich nested *documents* that are contained in *tables* as well as queries that express any boolean expression over *predicates* on documents within a single table. As a concrete representative, we employ the popular MongoDB query language [40].

3. CACHE COHERENCE

To illustrate the value of a cache coherence mechanism, consider an intuitive straw-man solution for query caching.

Static TTL: The server assigns a constant, application-defined TTL to each query, so that any web cache may serve the query and staleness is bounded by the TTL. This does not require any query invalidation logic in the client or server, as the regular expiration-based semantics of HTTP web caching are used. The problem of this naive solution is that either many stale reads will occur when the TTL is too high or cache hit ratios will suffer when the TTL is too low. The first step to improving this scheme is adapting the purely static TTLs to the actual frequency of changes for each query. However, even for a stochastic TTL estimation (described in Section 3.3), stale reads occur for each deviation from the estimate. To address this, clients require a lightweight representation of stale queries that can be updated frequently. This allows individual staleness bounds for each client without reducing cacheability of queries.

3.1 Expiring Bloom Filters

The purpose of the Expiring Bloom Filter (EBF) is to answer the question whether a given query or record is potentially stale. This information allows clients to "correct" the TTL of objects and queries that change before their TTL expires by periodically refreshing the EBF. The EBF exploits the idea that false positives are safe in the sense that they only cause an unnecessary revalidation – i.e. increased latency – but do not affect consistency. By allowing

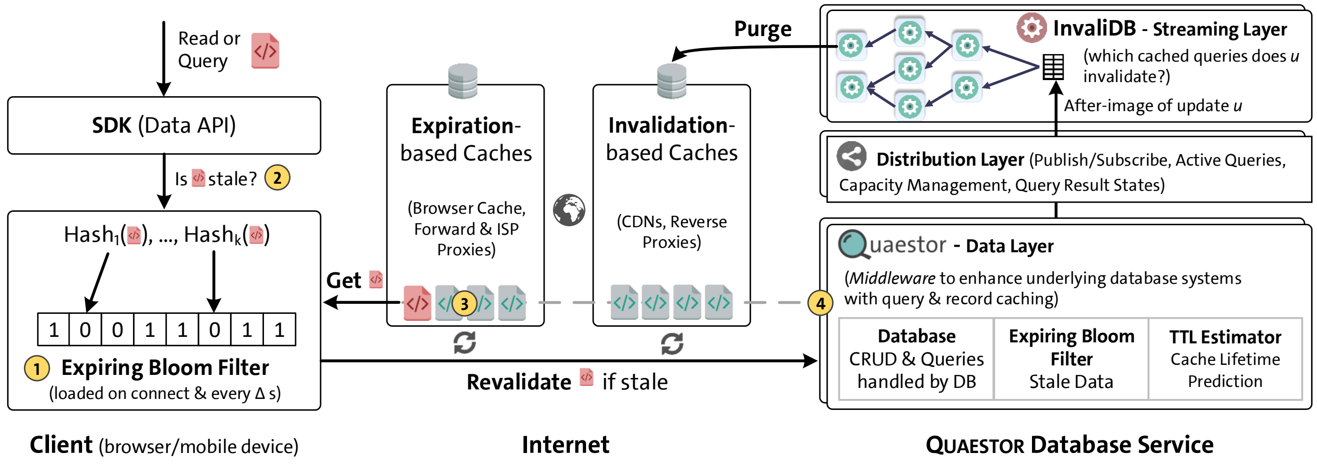

Read or Query
SDK (Data API)
Is stale? 2
Hash1(), ..., Hashk()
1 0 0 1 1 0 1 1
1 Expiring Bloom Filter
(loaded on connect & every Δ s)
Get
Expiration-based Caches
(Browser Cache, Forward & ISP Proxies)
3
Invalidation-based Caches
(CDNs, Reverse Proxies)
Purge
InvaliDB - Streaming Layer
(which cached queries does u invalidate?)
After-image of update u
Distribution Layer (Publish/Subscribe, Active Queries, Capacity Management, Query Result States)
4
Quaestor - Data Layer
(Middleware to enhance underlying database systems with query & record caching)
Database
CRUD & Queries handled by DB
Expiring Bloom Filter
Stale Data
TTL Estimator
Cache Lifetime Prediction
Revalidate if stale
Client (browser/mobile device)
Internet
QUAESTOR Database Service

Figure 3: QUAESTOR: client-server architecture and components.

occasional false positives with probability f, the EBF achieves a very small size that is provably space-optimal within a constant factor (1.44) and allows $O(1)$ lookups [13]. It can therefore efficiently be transferred from the server to the client and is much smaller than the actual set of stale queries and records for most applications. Even if the set of actual small queries is small and therefore could be represented as a list, the EBF does not impose considerable overhead as it will be sparse and therefore well-compressible through HTTP with Gzip [13]. The presented scheme is an extension our previous Cache Sketch [28] work to arbitrary query results, records and files.

Overview of Request Flow. Figure 3 gives a high-level overview of QUAESTOR's architecture and the role of the Expiring Bloom Filter. From the perspective of a client performing a query, the request flow is as follows:

1. Upon connection, the client gets a piggybacked EBF (a flat Bloom filter) containing freshness information.

2. Before issuing a query, the EBF is queried by the SDK to decide between a normal cached load and a revalidation.

3. The caches either serve their cached copy or forward the query upstream.

4. For cache misses and revalidations, the server returns the query result using an appropriate TTL, while registering the query in InvaliDB to detect future query invalidations. If the query result later changes before the TTL is expired, the query is added to the EBF and purged from invalidation-based caches.

Expiring Bloom Filter Structure. A query or read is performed by querying the Bloom filter b_t that was generated at time t. The key (i.e. the normalized query string or record id) is hashed using k independent uniformly distributed hash functions that map from the key domain to $[1,m]$, where m is the bit array size of b_t. If all bits $h_1(key),\dots,h_k(key)$ equal 1, the record is contained and considered stale. Theorem 1 derives the guarantees of the EBF using the time-based consistency property Δ-atomicity [29]. Δ-atomic semantics assert that every update becomes visible during the first Δ time units after the acknowledgement of its write.

Definition 1 *Let b_{t_1} be the Expiring Bloom Filter generated at time t_1. It contains the query string q of every $result(q)$ that became stale before it expired in all caches. Formally, this is every q for which holds that $\exists r(q,t_2,TTL), w(x,t_w) : t_r + TTL > t_1 > t_w > t_r$ where $r(x,t_2,TTL)$ is a query of q at time t_2 with a TTL and $w(x,t_w)$ is a write happening at t_w on a record x so that $result_{t_r}(q)$ is invalidated (see notification events add, change, remove in Section 3.2).*

Theorem 1 *A query q performed at time t_2 using b_{t_1} satisfies Δ-atomicity with $\Delta = t_2 - t_1$, i.e. it is guaranteed to see only query results that are at most Δ time units stale.*

PROOF. Consider a query issued at time t_2 using b_{t_1} and returning $result(q)$ that was stale for $\Delta > t_2 - t_1$. This implies that q must have been invalidated at a time $t_w < t$ as otherwise $t_2 - t_w < \Delta$ (Δ-atomicity not violated). Hence, there must have been an earlier query $r(q,t_r,TTL)$ with $t_r + TTL > t_2$ so that $result(q)$ is still cached. However, by the construction of b_{t_1}, the query is contained in b_{t_1} until $t_r + TTL$ and hence stale for at most $\Delta = t_2 - t_1$ (proof by contradiction). □

The EBF hence is a Bloom filter that contains all stale query for one point in time, i.e. queries that became invalid while still being stored in some cache. Theorem 1 also subsumes record caching, if q is substituted by the record id and $result(q)$ by the record. In the following, we will refer to queries and imply that the same holds true for records. **Freshness Policies.** The achieved freshness is reflected by the age of the Expiring Bloom Filter. The basic way of utilising the EBF is to fetch it at page load to use it for the initial resources of the application, e.g. stylesheets and images (*cached initialization*). To maintain *bounded staleness*, the EBF is refreshed in configurable intervals. This is achieved in a non-disruptive fashion by promoting the first query after Δ seconds to a revalidation that additionally piggybacks an up-to-date EBF. Clients can therefore precisely control the desired level of consistency. This polling approach for the EBF resembles Pileus' [49] method, where clients poll timestamps from all replication sites to determine which replica can satisfy the demanded consistency level. However, the EBF is much more scalable as the freshness information is already aggregated and does not have to be assembled by clients from different caches or replicas.

3.2 Consistency

Consistency Guarantees. The consistency levels provided by QUAESTOR are summarized in Figure 4. The central consistency level enabled by the EBF is Δ-*atomicity* with the application and clients being able to choose Δ. Several additional session consistency guarantees are achieved. *Monotonic writes*, i.e. a global order of all writes from one client session, are assumed to be given by the database (e.g. MongoDB) and are not impeded by the EBF. *Read-your-writes* consistency is obtained by having the client cache its own writes within a session: after a write, the client is able to read her writes from the local cache. *Monotonic read* consistency guarantees that a client will only see monotonically increasing versions of data within a session. QUAESTOR achieves this by having clients cache the most recently seen versions and comparing any subsequent reads to the highest seen version. If a read returns an older version (e.g. from a different cache), the client resorts to the cached version if it is not contained in the EBF or triggers a revalidation otherwise.

If QUAESTOR exposes an eventually consistent data store, its inconsistency window Δ_{DB} lowers the Δ-atomicity guarantee. The same holds true if invalidations are performed asynchronously. However, as the probability that this violates consistency is low [8], it is a very common choice to accept $(\Delta + \Delta_{DB} + \Delta_{Invalidation})$-atomicity. By choosing a lower Δ, users can easily compensate both effects. In practice, adjusting Δ to $\Delta - \Delta_{Invalidation}$ allows revalidation requests to be answered by invalidation-based caches instead of the origin servers. This optimization significantly offloads the backend.

Opt-in Consistency Guarantees. By allowing additional cache misses, *causal consistency* and even *strong consistency* are possible as an opt-in by the client. With causal consistency, any causally related operations are observed in the same order by all clients [7]. With caching, causal consistency can be violated if of two causally dependent writes one is observed in the latest version and the other is served by a cache. Using the EBF, any causal dependency younger than the EBF is observed by each client, as the EBF acts a staleness barrier for the moment in time it was generated. However, if a read is newer than the EBF, causal consistency might be violated on a subsequent second read. Therefore the client has two options to maintain causal consistency after a read newer than the EBF. 1) The EBF can be refreshed to reflect recent updates. 2) Every read happening before the next EBF refresh is turned into a revalidation. For strong consistency within a client session *every* read within that session is performed as a revalidation.

The strongest semantics QUAESTOR can provide are ACID transactions. These optimistic transactions exploit the fact that caching reduces transaction durations and can thereby achieve low abort rates with a variant of backwards-oriented optimistic concurrency control [27]. We omit details for space reasons, but the key idea is to collect read sets of transactions in the client and validate them at commit time to detect both violations serializability and stale reads. The scheme is similar to externally consistent optimistic transactions in F1 and Spanner [21, 44] but can leverage caching and the EBF to decrease transaction duration for clients connected via wide-area networks.

Additionally, clients can directly subscribe to websocket-based query result change streams that are otherwise only used for the construction of the EBF. Through this synchronization scheme, the application can define its critical data set through queries and keep it up-to-date in real-time. For applications with a well-defined scope of queries this approach is preferable, while complex web applications will profit from using the EBF due to lower latency for the initial page load and lower resource usage in the backend.

3.3 Usage and implementation

Server-side EBF Maintenance. Besides the Bloom filter, the server-side EBF also tracks a separate mapping of queries to their respective TTLs. In this way, only non-expired queries are added to the Bloom filter upon invalidation. After their TTL is expired, queries are removed from the Bloom filter. As a normal Bloom filter does not allow removals, the EBF is maintained as a *Counting Bloom filter* [13] which allows to discard queries once they are no longer stale. As it is inefficient to generate the non-counting Bloom filter for each request, the server-side EBF efficiently updates the flat Bloom filter (i.e. all non-zero counters) upon changes.

Consistency Level	How	
Δ-atomicity (staleness never exceeds Δ seconds)	Controlled by age (i.e. refresh interval) of EBF	✓ Always
Monotonic Writes	Guaranteed by database	
Read-Your-Writes and **Monotonic Reads**	Cache written data and most recent read-versions in client	
Causal Consistency	Given if read timestamp is older than EBF, else revalidate	+ Opt-in
Strong Consistency (Linearizability)	Explicit revalidation (cache miss at all levels)	

Figure 4: Consistency Levels provided by QUAESTOR: Δ-atomicity, Monotonic Writes, Read-Your-Writes, Monotonic Reads are given by default, Causal Consistency and Strong Consistency can be chosen per operation (with a performance penalty).

Client-side EBF Usage. Clients receive a flat, immutable copy of the EBF, i.e. a normal Bloom filter. As the server has no knowledge of data in individual caches, it is not client-specific. A stale query is contained in the EBF until the highest TTL that the server previously issued for that query has expired. While contained, the query always causes a cache miss. QUAESTOR's client SDK abstracts from this by transparently performing the EBF lookup for each query executing the freshness policy in the background. As discrepancies between actual and estimated TTLs can cause extended periods for which queries are contained in the EBF and considered stale, clients perform a *differential whitelisting*: every query and record that has been revalidated since the last EBF update is added to a whitelist and considered fresh until the next EBF renewal.

The false positive rate f depends on the Bloom filter size m in bits. When the size matches the initial congestion window of TCP with $m \approx 10 \cdot 1460\,byte = 14.6\,KB$ it is always transferred in one round-trip. With these parameters, the Bloom filter has a false positive rate of 6% when containing 20,000 distinct stale queries.

Scalability. The EBF is able to scale both reads and writes. *Read scalability* is achieved by replicating the complete EBF and balancing loads of the Bloom filter over the replicas. Write scalability is reached through *per-table partitioning*: each table has its own EBF instance. This horizontally distributes Bloom filter modifications and expiration tracking. At read time, the aggregated EBF is constructed by a union over the EBF partitions through a bitwise OR-operation over the Bloom filter bit vectors. Alternatively, clients can also exploit the table-specific EBFs to decrease the total false positive rate at the expense of loading more individual EBFs. We omit detailed evaluation results for brevity, but the Redis-based implementation of the Expiring Bloom Filter provides sufficient performance to sustain a throughput of >150 K queries or invalidations per second for each Redis instance.

Implementation. The EBF is in the critical request path: cachable queries and reads lead to an EBF write for the respective record and any thereby invalidated query result. To scale to high throughputs, we implemented two EBF variants available as open-source[2]. The *in-memory* implementation targets single-server setups while the *distributed* implementation is capable of sharing the state of the EBF across machines. In the distributed case, all DBaaS servers communicate with the in-memory key-value store Redis [3], which holds the counting Bloom Filter and the tracked expirations.

In summary, QUAESTOR maintains an Expiring Bloom Filter (EBF) of potentially stale queries and records, so that expiration-based caches can be leveraged while guaranteeing tunable Δ-atomicity. To maintain the EBF, changes to query results have to be detected and added in real-time, as described in the following section.

4. INVALIDATIONS AND EXPIRATIONS

4.1 Invalidation Detection

To provide server-side query invalidations, QUAESTOR registers all cached queries in *InvaliDB* which in turn notifies QUAESTOR as soon as a query result becomes stale. While we use SQL for the sake of clarity in our illustrations, InvaliDB supports MongoDB's query language.

The invalidation pipeline (*InvaliDB*) matches change operations to cached queries. For each cached query, it determines whether an update changes the result set. The invalidator then outputs a set of queries with stale cached query results to QUAESTOR, which sends out invalidations to reverse proxy caches and CDNs. This check is performed by re-evaluating queries on after-images of the relevant database partition in a distributed stream processing system (Apache Storm) co-located with QUAESTOR. The throughput of the invalidation pipeline is the limiting constraint of query caching and determines how many queries can be cached at the same time. Through a *capacity management model* only queries that are sufficiently cachable are admitted and prioritized based on the costs of maintaining them.

Notification Events. InvaliDB continuously matches record after-images provided with each incoming write operation (insert, update, delete) against all registered queries. QUAESTOR can subscribe to an arbitrary combination of the following events, each of which triggers a notification: `add` (an object enters a result set) , `remove` (an object leaves a result set), `change` (an object already contained in a result set is updated without altering its query). To illustrate these different events, consider the query in Figure 5 which selects blog posts tagged with the keyword `example`. First, a new blog post is created which is yet untagged and therefore not contained in the result set (box). When an update operation adds the `example` tag to the blog post, it enters the result set which triggers an `add` notification. Later, another tag is added which does not affect the matching condition and therefore only changes the object's state, thus entailing a `change` notification. When the `example` tag is finally removed from the blog post, the matching condition does not hold anymore and the object leaves the result set, causing a `remove` notification to be sent.

With respect to query invalidation, only two combinations of event notifications are useful: When the cached query result contains the IDs of the matching objects (id-list), an invalidation is only required on result set membership changes (`add`/`remove`). Caching full data objects (object-list) on the other hand also requires an invalidation as soon as any object in the result set changes its state (`add`/`remove`/`change`).

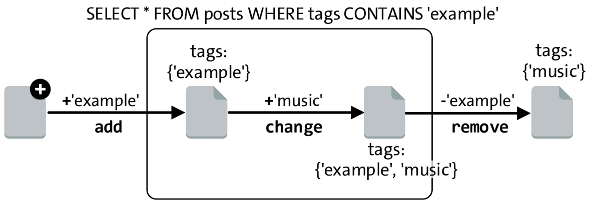


Figure 5: Notifications as an object gets updated.

Workload Distribution. InvaliDB relies on three tasks to provide scalable real-time notifications: query ingestion (registration of new queries), changestream ingestion (distribution of record after-images) and matching (invalidation detection), each of which is distributed over the nodes in a cluster using the Apache Storm real-time computation framework [1]. The matching workload is distributed by hash-partitioning both the stream of incoming data objects and the set of active queries orthogonally to one another, so that every instance of the matching task is responsible for only a subset of all queries (query partitioning) and only a fraction of their result sets (datastream partitioning). The ingestion workload, in contrast, is not partitioned, but scattered across task instances. Every instance of the query and changestream ingestion tasks transactionally pulls newly arrived data items (query activations/deactivations or update operations, respectively) from the source and forwards them according to the partitioning scheme.

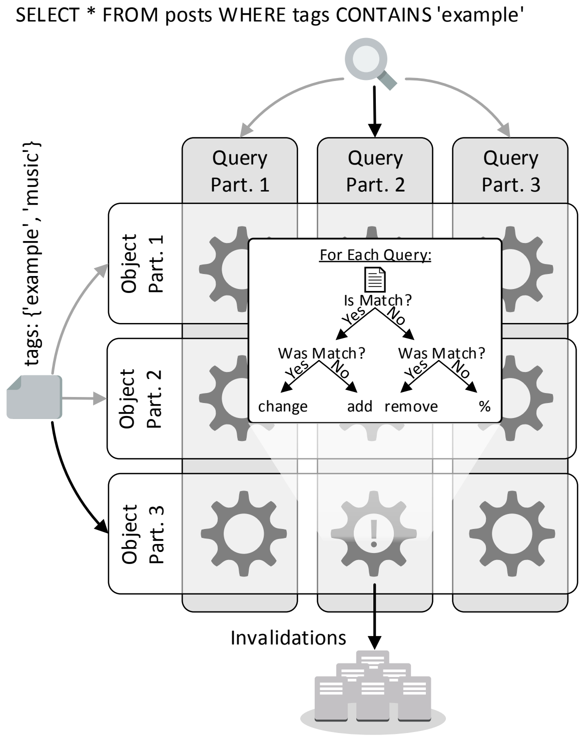


Figure 6: InvaliDB workload distribution: Every node is only assigned a subset of all queries and a fraction of all incoming updates.

Figure 6 illustrates workload distribution in a 9-node cluster with three object partitions (lightly shaded rows) and three query partitions (strongly shaded columns). Please note that we omit the parallelism of the data ingestion tasks here in favour of simplicity and only make the distribution of the matching task explicit.

When a query is received by one instance of the query ingestion

[2] https://github.com/Baqend/Orestes-Bloomfilter

task, it is forwarded to all matching task instances in its respective query partition (e.g. query partition 2). Since InvaliDB has to be aware of the result sets of all newly added queries in order to maintain their correct state, every new query is initially evaluated on QUAESTOR and then sent to InvaliDB together with the initial result set. To rule out the possibility of missing updates in the timeframe between the initial query evaluation (on QUAESTOR) and the successful query activation (on all responsible InvaliDB nodes), all recently received objects are replayed for a query when it is installed. When an update operation is registered by one of the changestream ingestion task instances, this operation and its corresponding after-image are forwarded to all matching task instances in the respective object partition (e.g. object partition 3). In the example, the one that is responsible for query partition 2 and object partition 3 detects a new match for the example query and therefore sends out an `add` notification. To prevent ingestion and matching task instances from competing for resources, we do not colocate them on the same nodes.

Scalability. Since InvaliDB partitions both the change stream *and* the set of all active queries, single-node performance does not limit overall system performance: As long as every query can be handled by a single node, changestream partitioning is not required and the load can be spread across the cluster by simply assigning every node a fair share of all active queries. However, additional changestream partitioning allows distributing responsibility for a single query over several machines and guarantees low latency even when the resources required for handling individual queries exceed single-node capacity. Thus, overall performance is neither bounded by update throughput nor by the number of active queries nor by query selectivity or result set size and scales linearly with the number of cluster nodes (see Section 6.3).

Managing Query State. Simple static matching conditions such as `WHERE tags CONTAINS 'example'` are stateless, meaning that no additional information is required to determine whether a given after-image satisfies them. As a consequence, the only state required for providing `add`, `remove` or `change` notifications to stateless queries is the former matching status on a per-record basis. This state can be partitioned by record id and thus can be easily distributed, just like the computation itself.

With additional `ORDER BY`, `LIMIT` or `OFFSET` clauses, however, a formerly stateless query becomes stateful in the sense that the matching status of a given record becomes dependent on the matching status of other objects. For sorted queries, InvaliDB is consequently required to keep the result ordered and maintain additional information such as the entirety of all items in the offset. To capture result permutations, `changeIndex` events are emitted that represent positional changes within the result. Our current implementation maintains order-related state in a separate processing layer partitioned by query.

Implementation. All components of InvaliDB are written in Java and executed on Apache Storm. The query engine is pluggable and supports any stateless predicates. Communication between QUAESTOR and InvaliDB is handled through Redis message queues.

Scope. InvaliDB does not yet support joins and aggregations. Since QUAESTOR is designed for aggregate-oriented, denormalized NoSQL databases, the capability to pose predicates on nested documents is sufficient to reflect 1:1 and 1:n relationships. Aggregations with groupings are ongoing work and therefore currently uncached.

In summary, InvaliDB provides a scalable stream processing system for detecting query invalidations. Its central trade-off lies in the partitioning of both queries and changes, which renders joins infeasible but enables linear scalability and low latency.

4.2 Statistical TTL Estimation

The TTL Estimator provides stochastic estimations of cache expiration times for query results and cached records. Our mechanism is based on the insight that any cached record should ideally expire right before its next update occurs, thus achieving maximum cache hit rates while avoiding unnecessary invalidations. The discrepancy between the actual and the estimated TTL directly determines the amount of data considered stale and hence affects the false positive rate of the EBF. High cache hit rates and an effective EBF size thus require reliable TTL estimates.

We use a dual strategy for estimating expirations for query results and records. Initially, TTLs are estimated through the stochastic process of incoming updates. Poisson processes count the occurrences of events in a time interval t characterised by an arrival rate λ and are an established model for web workloads [50]. For a Poisson process, the inter-arrival times of events have an exponential cumulative distribution function (CDF), i.e. each of the identically and independently distributed random variables X_i has the cumulative density $F(x;\lambda) = 1 - e^{(-\lambda x)}\ for\ x \geq 0$ and mean $1/\lambda$. For each database record, QUAESTOR can estimate (through sampling) the rate of incoming writes λ_w in some time window t.

The result set Q of a query of cardinality n can then be regarded as a set of independent exponentially distributed random variables $X_i, \ldots, X_n$ with different write-rates $\lambda_{w1}, \ldots, \lambda_{wn}$. Estimating the TTL for the next change of the result set requires a distribution that models the minimum time to the next write, i.e. $min\{X_1, \ldots, X_n\}$, which is again exponentially distributed with $\lambda_{min} = \lambda_{w1} + \ldots + \lambda_{wn}$. The quantile function then provides estimates that have a probability p of seeing a write before expiration:

$$F^{-1}(p, \lambda_{min}) = \frac{-ln(1-p)}{\lambda_{min}}. \quad (1)$$

By varying the quantile, higher/lower TTLs and thus cache hit rates can be traded off against more or fewer invalidations. Alternatively, the TTL can be estimated using the expected time until the next write. This results in always using the observed mean TTL, but in turn does not allow fine-grained adjustments.

For individual records, we always use an estimate based on the approximated write-rates. For queries, the Poisson estimate based on the write-rates on the keys of the result set is only used as an initial estimate. If a query result is invalidated, the *actual TTL* of the result was the difference between the invalidation time stamp and the previous read time stamp. We can hence update our old estimate according to an exponentially weighted moving average (EWMA) closer towards the true TTL:

$$TTL_{query} = \alpha \times TTL_{old} + (1-\alpha) \times TTL_{actual}. \quad (2)$$

The current TTL estimate for a query is kept in a shared partitioned data structure called the *active list*, which is accessed by all QUAESTOR nodes. The key idea of the TTL estimation model is to make an educated guess about the initial TTL which should then move towards the "true" TTL with some lag after invalidations. TTL estimation is used for queries and records in both expiration- and invalidation-based caches. Note that this does not require clock synchronization, as only relative time spans are used.

Representing Query Results. A cached query can either be served as a list of record URLs (*id-list*) or as a full result set (*object-list*). Id-lists are more space-efficient and yield higher per-record cache hit rates but require more round-trips to assemble the result – the decision which representation to use cannot be made by the cache. QUAESTOR employs a cost-based decision model in order to

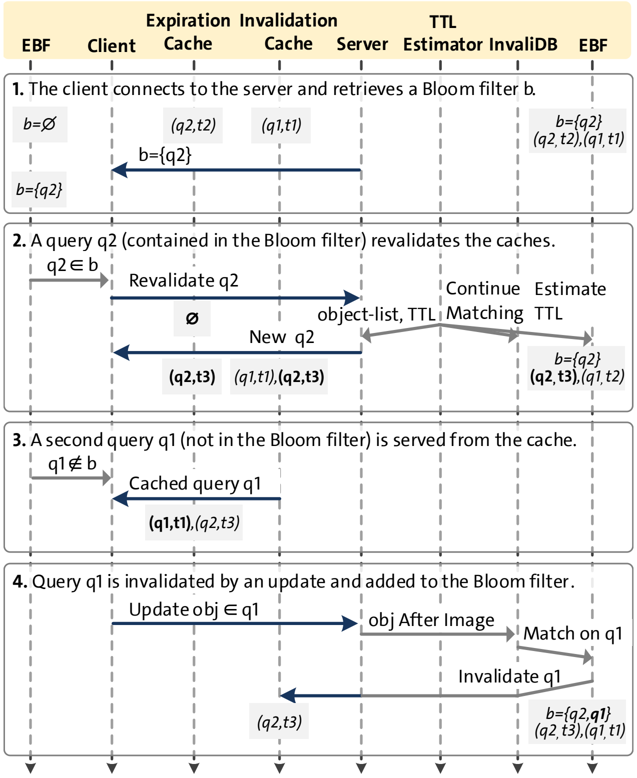


Figure 7: End-to-end example of query caching.

weigh fewer invalidations against fewer round-trips when choosing object-lists or id-lists (omitted due to space constraints).

5. END-TO-END EXAMPLE

Figure 7 gives an end-to-end example of the steps involved in serving cachable queries. In the depicted setting, the client begins by fetching the EBF containing a stale query (q_2) still cached in the client (1). Therefore when loading the query, the client triggers a revalidation that refreshes the client cache and causes a miss at the invalidation-based cache. Using the active list, the server passes the query to InvaliDB for future change detection, while estimating the TTL and deciding between an id-list and object-list representation (2). Before returning the result, the query is reported to the EBF, so that every subsequent invalidation within the newly estimated TTL makes the cached query stale. The returned result is cached in both caches using the new expiration. When the client performs a query that is not stale (q_1), the cache can serve the result (3). A change operation to a record contained in that query result is forwarded to the database and the respective after-image passed to InvaliDB (4). InvaliDB detects the change to the query and reports the invalidation to the EBF. As the query still has a non-expired TTL, the EBF adds the stale query and triggers an invalidation to prevent stale reads of the old query result.

6. EVALUATION

In this section, we demonstrate that QUAESTOR's scalability is only limited by the write throughput of the underlying database system. We evaluate QUAESTOR with regard to latency, throughput and staleness (and hence the effectivenes of the TTL estimator) compared to a baseline of just using a CDN, only using a client cache and no caching at all. We further demonstrate the linear scalability of InvaliDB and the high throughput of the Expiring Bloom Filter.

6.1 Experimental setup

Setup. Our experimental design is based on the YCSB benchmark [20]. YCSB defines a set of common workloads to evaluate the performance of cloud databases. We implemented a YCSB-style framework that extends the widely-used original benchmark in two aspects: a multi-threading model for massive connection parallelism and a multi-client model to scale the client tier [25]. As a baseline to our experiments, we used the Orestes DBaaS [27] with uncached communication, which we deem representative (aside from static latency overhead) for state-of-the-art database services that do not use web caching. The Orestes architecture also provides the foundation for Baqend's cloud services.

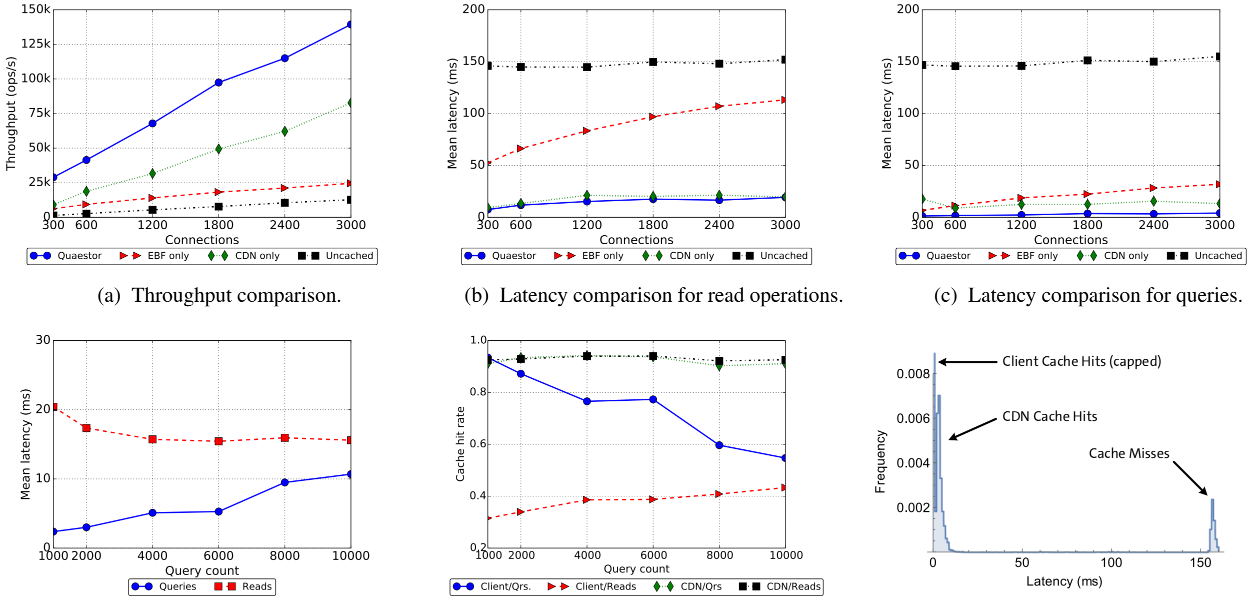
We evaluated QUAESTOR on the following EC2 setup: MongoDB was configured in a cluster setting with 3 `m3.xlarge` (4 vCPUs, 15 GB RAM, 2x40 GB SSDs) instances with 2 shard servers and 1 configuration server. Documents were sharded through their hashed primary key. The Expiring Bloom Filter as well as the Redis-backed active list were hosted on one `m3.xlarge` instance, respectively. Further, we used 3 Quaestor servers and a varying number of workload-generating client instances (all `m3.xlarge`). To demonstrate the full impact of geographic round trip latency, QUAESTOR, MongoDB and InvaliDB were hosted in a virtual private cloud in the EC2 Ireland region, with workloads being generated from the Northern California region. In the setups using a CDN, Fastly was used (round-trip latency 4 ms). Cache misses at CDN edge servers were forwarded to QUAESTOR nodes in a round-robin manner.

Workloads. Workloads were specified by defining a discrete distribution of operations (reads, queries, inserts, partial updates and deletes). TCP connections were pre-warmed for 30 seconds on a dummy table. Load was generated using asynchronous requests with 300 HTTP connections per client instance. Each data point was created under 5 minutes of load, which was sufficient to achieve stable and reproducible results. Requests were generated by first sampling a request type and then sampling the key/query and table to use (using a Zipfian distribution). For the workloads we analyzed, 10 database tables, each with 10,000 documents, were generated for each run. Further, 100 distinct queries per table were generated to initially return on average 10 documents.

Monte Carlo simulation. We also implemented a Monte Carlo simulation framework of our caching model that simulates interactions of concurrent clients with client and CDN node as well as QUAESTOR. Simulation is the most reliable method to analyze properties like staleness as it provides globally ordered event time stamps for each operation and does not rely on error-prone clock synchronization. Further, the simulation enables detailed analysis optimization of various workload parameters such as latency distributions, TTL estimation models and capacity configurations.

6.2 Quaestor

To demonstrate the effectiveness of QUAESTOR, we vary typical workload parameters such as incoming connections, the number of queries and documents, and update rates. We study QUAESTOR's scalability and performance under high throughput and extend the analysis to more clients and measured staleness using simulation. We do not compare QUAESTOR to geo-replicated systems (e.g. Pileus) as our main point is to show that commodity web caching highly improves latency with very little staleness and no additional



(a) Throughput comparison.

(b) Latency comparison for read operations.

(c) Latency comparison for queries.

(d) Mean request latency for reads and queries.

(e) Cache hit rates at client and CDN.

(f) Query latency histogram.

Figure 8: Cloud-based evaluation of QUAESTOR.

servers. Geo-replication schemes tuned towards one specific geographical setup will likely still outperform QUAESTOR.

Read-heavy workload. We begin evaluating QUAESTOR on a read-heavy workload with 99% queries and reads (equally weighted) and 1% writes. Figure 8a demonstrates QUAESTOR's throughput scalability against a baseline without dynamic caching, a CDN with InvaliDB, and the EBF-based client cache only. At maximum load (3000 asynchronous connections delivered by 10 client instances), QUAESTOR achieves an 11-fold speed-up versus an uncached baseline, a 5-fold improvement over the EBF-based client cache (EBF only) and a 69.5% improvement over a CDN with InvaliDB. Using a CDN with InvaliDB yields superior performance to only using client caches since clients rely on the CDN to fill up their caches quickly. Client-side Bloom filters were refreshed every second to ensure minimal staleness. Figure 8f illustrates the latency distribution where most queries are client cache hits with no latency, CDN hits induce an average latency of 4 ms and cache misses 150 ms. Note that linear scalability is not possible since an increasing number of clients increases the number of updates and thus reduces cacheability.

Mean round-trip latency between client instances and QUAESTOR was 145 ms with a variance of 1 ms between runs (error bars omitted due to scale). Figures 8b and 8c show read and query latency for the same setup. For 3000 connections, QUAESTOR achieved a mean query latency of 3.2 ms and a mean read latency of 17.5 ms. As there are 100× more records than queries, cache hit rates for queries are higher and latencies lower. Note that the latency of the client-cache only (EBF only) variant increases due to more overhead at the database. In contrast, CDN latency for queries improves initially and remains constant afterwards because separate clients access the same CDN edge.

Varying query count. Scalability with regard to query count is governed by the provided InvaliDB configuration (which scales linearly, as shown in Section 6.3). We demonstrate the effect of increasing query counts with regard to average request latency and cache hit rates for the same InvaliDB configuration used in the read-heavy workload (8 InvaliDB matching nodes). Figure 8d shows how both read and query request latencies are affected by increasing query count. Read latency improves because a larger portion of keys is part of a cached query result. All records in an result are inserted to the cache as individual entries, thus causing read cache hits by side effect. This improves read latency from initially 20 ms to a mean read latency of 15 ms. Query latency increases with query count due to decreasing cache hit rates at the client, as shown in Figure 8e. Cache hit rates at the CDN are comparably stable, since the concurrent client instances cause sufficient cache hits by side effect for each other. Ultimately, QUAESTOR's performance for increasing query counts depends more on the popularity of individual queries and the update rate than on the total number of queries.

Varying write rates. Read-dominant workloads naturally lend themselves to caching since they allow higher consistency, longer TTLs, fewer invalidations and less database load. With increasing update rates, throughout is limited by the database. We demonstrate how cache hit rates degrade by increasing update rates (keeping equal read and query rates) in Figure 9. Only 1200 connections were used in order to avoid being limited by the write throughput of the MongoDB cluster. Client cache hit rates for both records and queries decrease predictably with increasing update rate. Figure 9 shows how staleness (EBF refresh interval) can be used to mitigate performance degradation in write-heavy scenarios. Notably, the refresh interval has only little impact on cache hit rate degradation. There is no linear correlation between increasing refresh rate and lower latency on higher write rates. This is because increasing write rates also leads to lower TTLs. Hence, increasing EBF refresh rates above a certain threshold only leads to more staleness without improved client performance.

Varying document count. Finally, we investigate QUAESTOR's performance for varying document counts. Table 1 compares latencies for different database sizes, which was achieved by changing the number of database collections. Each collection contains 10,000 documents and is accessed by 100 distinct queries. We increased experiment durations to 600 s and changed the Zipf constant to 0.99 to account for the fact that with increasing document

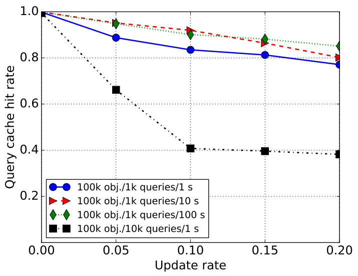


Figure 9: Client cache hit rates for queries with varying update rates for different EBF refresh intervals.

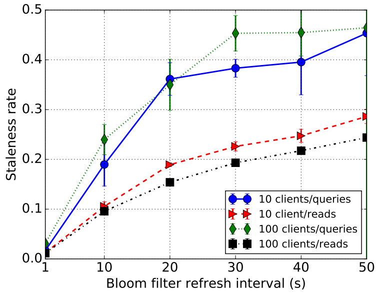


Figure 10: Stale read and query rates for 10/100 clients and different refresh intervals.

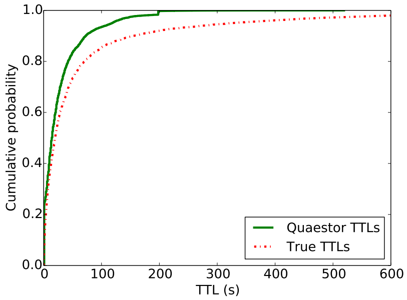


Figure 11: CDF of Quaestor's TTL estimation scheme vs. true CDF.

and query counts, caches take significantly longer to fill up. Results show that for very small databases and distributions with high Zipf constants, reads and writes concentrate on the same few objects and thus limit cache hit rates. For increasing database sizes, caches take longer to fill up and TTLs have to be adjusted upwards, thus limiting performance during experiments. In the following section, we analyze client-side staleness through Monte Carlo simulation.

Table 1: Performance overview for increasing document counts for a request distribution with Zipfian constant 0.99.

Documents	Queries	Queries	Reads
10,000	100	13.8 ms	70 ms
100,000	1000	5.5 ms	40.2 ms
1 million	10,000	11.9 ms	27.2 ms
10 million	100,000	34.8 ms	133 ms

EBF-Bounded Staleness. The EC2-based evaluation showed QUAESTOR under maximum load, using relatively few client instances with many parallel connections. To analyze staleness , we use a more typical configuration of many clients (100) with fewer HTTP connections per client (6, as is in most browsers) in the simulation. The simulation detects staleness (i.e. any violations of linearizability [29]) in the client caches and the CDN. Client-side staleness is bounded by the EBF refresh interval. Upon every EBF renewal, clients revalidate stale cache entries identified by the filter. CDN staleness is primarily governed by invalidation latency. In our experiments, CDN staleness was constantly below 0.1%.

Figure 10 illustrates the relationship between Bloom filter refresh rate and client staleness. Staleness initially increases fast between 1 s and 10 s refresh rate, but is limited by two factors for higher refresh rates. First, every time a client begins an update operation it invalidates the corresponding record from its own cache. Second, client staleness rates are limited by cache hit rates, which were up to 60% for records and up to 95% for queries in the benchmark, thus explaining the difference between record and query staleness.

TTL estimation. We also used the simulator to compare our TTL estimation scheme against the *true TTL* for every query, which we define as the time period a query could have been cached until invalidation. Figure 11 shows the cumulative distribution functions (CDFs) for estimated and true TTLs for a 1% write rate for 10 minutes. While we omit a detailed analysis of per-query errors due to space constraints, the CDF comparison shows the expected result of having a similar distribution for the majority of TTLs and larger errors on the unpredictable long tail of the access distribution.

Production results. Baqend currently hosts a range of production applications and has delivered performance improvements to numerous websites. As an example we report the results of the e-commerce company Thinks. While being featured in a TV show with 3.5 Mio. viewers, the shop had to provide low latency to potential customers. By relying on QUAESTOR to cache all static data (e.g. files) and dynamic query results (e.g. articles with stock counters) the website achieved sub-second loads while being requested by 50,000 concurrent users (>20,000 HTTP requests per second). The business effect was measurable: the shop achieved a conversion rate of 7.8%, which is roughly 3 times above the industry average [16]. Usually such a request volume requires massive scale in the backend. However, since the CDN cache hit rate was 98%, the load could be handled by 2 DBaaS servers and 2 MongoDB shards.

6.3 InvaliDB

Setup. To demonstrate the scalability of our real-time matching approach, we measured sustainable matching throughput and match latency for differently sized InvaliDB deployments on Amazon EC2. Our test setup comprised one client machine, one QUAESTOR server, one Redis server and an InvaliDB cluster. As a baseline, we evaluated our InvaliDB deployment with only a single node for query matching and then doubled both the number of active queries and the number of matching nodes with every subsequent experiment series. Every deployment had a single node dedicated to query and change stream ingestion. The Redis server hosting the message queues for communication between InvaliDB and the QUAESTOR server as well as all InvaliDB nodes were `c3.large` instances with 2 VCPUs (Xeon E5-2680 v2, Ivy Bridge) and 3.75 GB RAM each.

Workload. For every InvaliDB configuration, we performed a series of experiments, each of which consisted of two phases: In the preparation phase, any still-active queries from earlier experiments

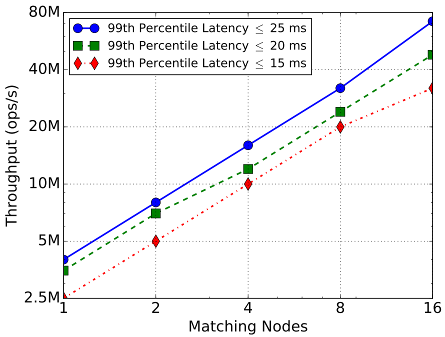


Figure 12: InvaliDB throughput for varying cluster sizes satisfying the given latency bounds.

were removed and queries for the upcoming one were activated. In the subsequent 2-minute measurement phase, the client machine performed 1,000 insert operations per second against the QUAESTOR server and measured notification latency as the difference between the timestamp of notification arrival and of the point in time directly before sending the corresponding insert statement.

We chose the same constant update throughput of 1,000 inserts per second for all experiment series, but varied the number of active queries relatively to the number of matching nodes in each InvaliDB cluster, so that all clusters were exposed to the same relative load. We started each experiment series with 500 active queries per node and increased their number by the same amount until the system was saturated and incoming operations started queueing up. Thus, the cluster with only 1 matching node started with 500 active queries, whereas the 16-node cluster started with 8,000.

Evaluation. To demonstrate the efficiency and scalability of InvaliDB, we measured notification latency under increasing load for 5 InvaliDB clusters employing between 1 and 16 matching nodes. All clusters achieved 99th percentile latencies below 20 ms up to 3 mio. and below 30 ms up to 4 mio. ops/s per node, while huge latency spikes marked system capacity at roughly 5 mio. ops/s per node. Peak latencies never exceeded 100 ms under load of 3 mio. ops/s per node or less. The line plot in Figure 12 illustrates that matching throughput scales linearly with the number of matching nodes even under tight latency bounds.

7. RELATED WORK

Web caching. In contrast to server-side caching solutions (e.g. Memcache, Oracle Result Cache, Data Grids) we aim to provide low *end-to-end* latency by exploiting existing HTTP caching infrastructures. In earlier work [28, 27], we have proposed a comprehensive scheme for leveraging HTTP caching for single database records. We extended this foundation by considering the more realistic setting of a full DBaaS API that includes arbitrary queries and non-stationary workloads. In related work, web caches are either treated as a storage tier for immutable content or as means of content distribution for media that do not require freshness guarantees [32, 24]. Candan et al. [14] first explored automated invalidation-based web caching with the CachePortal system that detects changes of HTML pages by analyzing corresponding SQL queries. Breslau et al. were the first to systematically analyze how Zipf-distributed access patterns ideally lend themselves for limited storage capacities of web caches [12, 33, 52]. This insight is related to our proposed capacity management scheme: even if only a small subset of "hot" queries can be actively matched against update operations, this is sufficient to achieve high cache hit rates. Another example for the use of Bloom filters in caching is Orestes [28], that employs them for staleness checks on cached database records.

Query Caching. Scalable query caching has previously been tackled from different angles. Garrod et al. have proposed Ferdinand, a proxy-based caching architecture forming a distributed hash table [26]. Their consistency management is based on a publish/subscribe invalidation architecture where query templates are mapped to multicast groups. DBCache, DBProxy and MTCache [5, 37, 11] also rely on dedicated database proxies to generate distributed query plans that can efficiently combine cached data with the original database. These systems need built-in tools of the database system for consistency management and are less motivated by latency reduction than by reducing query processing in the database. Blanco et al. investigated query caching in the context of incremental search indices [10]. To achieve cache coherence, they generate a synopsis of invalidated documents in the ingestion pipeline and check it before returning a cached search query. Unlike our evolving EBF, the synopses are immutable, created in batch and only used to predict likely invalidations at server-side caches.

Expiration-Based Caching. In the literature, the idea of using a TTL-based model has previously been explored in the context of file and search result caching. Fixed TTL schemes that neither vary in time nor between requested objects/queries lead to a high level of staleness [53]. A popular and widely used TTL estimation strategy is the Alex protocol [31] that originates from the Alex FTP cache [15]. It calculates the TTL as a percentage of the time since the last modification, capped by an upper TTL bound. This is similar to QUAESTOR's TTL update strategy for queries but has the downside of neither converging to the actual TTL nor being able to give estimates for new queries. Alici et al. proposed an adaptive TTL computation scheme for web-search results [4]. In their model, expired queries are compared with their latest cached version. If the result has changed, the TTL is reset to a minimum TTL, otherwise the TTL is augmented by an increment function that can either be static or trained from logs. Though the model is adaptive, it requires offline learning, does not incorporate invalidations and assumes a central cache co-located with the search index.

QUAESTOR separates itself from previous work on query caching in multiple aspects. First, it uses existing HTTP infrastructure and does not require custom caching servers. Employing stochastic models, this work provides a record-level analysis of query results to provide much more fine-grained TTL estimates. Furthermore, the cost-based optimization and flexibility of the EBF yield a tunable trade-off between query latency, consistency and server load by adapting to the workload at runtime.

Geo-replication. Another common approach for latency reduction is geo-replication, which can be combined with QUAESTOR's caching. Instead of storing data on geographically distributed web caches, the database system itself is distributed over multiple geographical replica sites [9, 44, 35, 38, 21, 45, 17, 22, 19, 43]. Web caching as employed in QUAESTOR can be viewed as a form of asynchronous, on-demand geo-replication. Our work is inspired by Pileus [49] that also achieves low latency, single round-trip writes and bounded staleness. In contrast to Pileus, QUAESTOR 1) supports queries 2) relies on web caches instead of custom replicas 3) scales to an arbitrary number of caches, as staleness information is consolidated in one EBF instead of being polled from each replica.

Latency-consistency trade-off. Consistency in replicated storage systems has been studied in both theory [29, 51] and practice [8, 39]. Similar to asynchronously replicated systems [22, 19, 36], QUAESTOR trades consistency against performance by invalidating asynchronously and allowing stale reads. We studied the strict staleness bounds imposed by the EBF through a Monte Carlo simulation that is similar to PBS proposed by Bailis et al. [8].

Web performance. A key finding of performance in modern web applications is that perceived speed and page load times are a result of physical network latency [30]. The HTTP/1.1 protocol that currently forms the basis of the web and REST APIs suffers from inefficiencies that have partly been addressed by HTTP 2 [34]. Once adopted by caches, CDNs and end devices, its push model will allow to simplify the query result representation in QUAESTOR to always favor id-lists without any performance downsides. As operations are furthermore multiplexed through a single connection, any refreshes of the EBF can be performed without causing head-of-line blocking for queries and CRUD requests.

Continuous Query Maintenance. Even though the integration of stored and streaming data has been studied for decades in the context of relational databases [48, 6], their inadequacy to handle real-time data has been widely accepted [46, 47]. Materialized view maintenance [18] and query notifications [41], in particular, are designed for domains where updates are infrequent. InvaliDB, in contrast, scales with write throughput and the number of currently maintained queries through a shared-nothing architecture.

In summary, Quaestor is inspired by the idea of geo-replication and fundamentally based on techniques from related work, such as Bloom filters for compact digests, expiration-based caching for passive replication as well as continuous queries for cache invalidations. We believe, that Quaestor adds a useful design choice for low-latency, data-centric cloud services.

8. CONCLUSION

In this paper, we investigated the applicability of web caching for mutable query results. The contribution of this paper is a novel caching approach for dynamic data to improve loading times in web applications. We rely on three pivotal ideas to make this possible: (1) the Expiring Bloom Filter as a compact client representation for stale data, (2) a scalable real-time invalidation scheme that matches updates to cached query results, and (3) an online TTL estimator. As a result, Quaestor offers a middleware for query caching with client-defined staleness bounds as well as several client-centric consistency guarantees. The presented approach is the central technology of the cloud service Baqend that uses it to provide significant load time improvements for websites. Evaluation results demonstrate Quaestor's effectiveness in reducing latency by up to an order of magnitude while strictly limiting staleness.

Acknowledgements

This work was generously supported by the EPSRC (grant references EP/M508007/1, EP/P004024), Cambridge University GCRF, and a Computer Laboratory Premium Scholarship (Sansom scholarship).

9. REFERENCES

[1] Apache Storm. `http://storm.apache.org/`. Accessed: 2016-07-14.

[2] HTTP Archive. `http://httparchive.org/trends.php`. Accessed: 2016-07-14.

[3] Redis. `http://redis.io/`. Accessed: 2016-07-14.

[4] S. Alici et al. Adaptive time-to-live strategies for query result caching in web search engines. In *Advances in Information Retrieval*. Springer, 2012.

[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In *ICDE*, pages 821–831, 2003.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[7] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *SIGMOD*, pages 761–772. ACM, 2013.

[8] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica. Probabilistically bounded staleness for practical partial quorums. *VLDB*, 2012.

[9] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Lon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234, 2011.

[10] R. Blanco, E. Bortnikov, F. Junqueira, R. Lempel, L. Telloli, and H. Zaragoza. Caching search engine results over incremental indices. In *SIGIR*, 2010.

[11] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering Bulletin*, 2004.

[12] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *INFOCOM*, volume 1, pages 126–134 vol.1, Mar 1999.

[13] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Math.*, 1(4):485–509, 2003.

[14] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In *SIGMOD*, 2001.

[15] V. Cate. Alex-a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, number 7330, pages 1–12. Citeseer, 1992.

[16] D. Chaffey. Ecommerce conversion rates. *smartinsights.com*, 2017. accessed: 2017-05-15.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, et al. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[18] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 2012.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, and P. a. o. Bohannon. Pnuts: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al. Spanner: Google's globally distributed database. *TOCS*, 2013.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, et al. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[23] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM TON*, 8(3):281–293, 2000.

[24] M. J. Freedman. Experiences with coralcdn: A five-year operational view. In *NSDI*, 2010.

[25] S. Friedrich, W. Wingerath, F. Gessert, and N. Ritter. NoSQL OLTP Benchmarking: A Survey. In *DMC*, volume 232 of *LNI*, pages 693–704. GI, 2014.

[26] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *VLDB*, 2008.

[27] F. Gessert, F. Bucklers, and N. Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *ICDE*, 2014.

[28] F. Gessert, M. Schaarschmidt, W. Wingerath, S. Friedrich, and N. Ritter. The cache sketch: Revisiting expiration-based caching in the age of cloud data management. In *BTW*, 2015.

[29] W. Golab, X. Li, and M. A. Shah. Analyzing consistency properties for fun and profit. In *PODC*, pages 197–206. ACM, 2011.

[30] I. Grigorik. *High performance browser networking*. O'Reilly Media, 2013.

[31] J. Gwertzman and M. Seltzer. World wide web cache consistency. In *ATC*, 1996.

[32] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *SOSP*, 2013.

[33] R. T. Hurley and B. Y. Li. A performance investigation of web caching architectures. In *C3S2E*, pages 205–213, 2008.

[34] IETF. Rfc 7540 - hypertext transfer protocol version 2 (http/2). 2015.

[35] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *EuroSys*, pages 113–126. ACM, 2013.

[36] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[37] P.-Å. Larson, J. Goldstein, and J. Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *ICDE*, pages 177–188. IEEE, 2004.

[38] W. Lloyd, M. J. Freedman, M. Kaminsky, et al. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *SOSP*, 2011.

[39] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at facebook. In E. L. Miller and S. Hand, editors, *SOSP*, pages 295–310. ACM, 2015.

[40] MongoDB, Inc. MongoDB. http://www.mongodb.org/.

[41] C. Murray, T. Kyte, et al. Using continuous query notification. In *Oracle Database Advanced Application Developer's Guide, 11g Release 1 (11.1)*. Oracle, 2016.

[42] M. Pathan and R. Buyya. A taxonomy of cdns. In *Content Delivery Networks*. Springer Berlin Heidelberg, 2008.

[43] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman, B. Ghosh, A. Curtis, O. Seeliger, Z. Zhang, A. Auradar, and others. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *SIGMOD*, pages 1135–1146. ACM, 2013.

[44] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, et al. F1: A distributed sql database that scales. *VLDB*, 2013.

[45] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, pages 385–400. ACM, 2011.

[46] M. Stonebraker and U. Çetintemel. "one size fits all": An idea whose time has come and gone. In *ICDE*, 2005.

[47] M. Stonebraker, U. Çetintemel, and S. B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 2005.

[48] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, 1992.

[49] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*. ACM, 2013.

[50] P. Van Mieghem. *Performance analysis of complex networks and systems*. Cambridge University Press, 2014.

[51] P. Viotti and M. Vukolic. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, 2016.

[52] P. Wendell and M. J. Freedman. Going viral: Flash crowds in an open cdn. In *SIGCOMM*, IMC '11, pages 549–558, New York, NY, USA, 2011. ACM.

[53] K. J. Worrell. Invalidation in Large Scale Network Object Caches. 1994.