Cache Sketches
Using Bloom Filters and Web Caching Against Slow Load Times

Felix Gessert, Florian Bücklers
{fg, fb}@baqend.com
@baqendcom
Who we are

- Felix Gessert, Florian Bücklers

Research Project since 2010

Backend-as-a-Service Startup since 2014
Introduction

Web Performance: State of the Art

Main Part

Cache Sketch: Research Approach

Conclusions

Using Web Caching in Applications
Presentation is loading
Why performance matters

1000ms

Loading…

Average: 9.3s

1000ms

Conversions

Traffic

Visitors

Revenue

-1% Revenue

-4% Visitors

-20% Traffic

-7% Conversions
An Average Website
Some Statistics
If perceived speed is such an important factor... what causes slow page load times?
The Problem
Three Bottlenecks: Latency, Backend & Frontend

High Latency

Frontend

Backend
Frontend Performance
Break-down of the Critical Rendering Path

Achieve a fast render of the page by:
- Reducing the **critical resources** needed
- Reducing the **critical byte**s which must be transferred
- Loading JS, CSS and HTML templates **asynchronously**
- Rendering the page **progressively**
- **Minifying** & **Concatenating** CSS, JS and images

Google Developers, Web Fundamentals
Frontend Performance
Tools to improve your page load

- Well known problem & good tooling:
  - Optimizing CSS (*postcss*)
  - Concatenating CSS and JS (*processhtml*)
  - Minification and Compression (*cssmin*, *UglifyJS*, *Google Closure*, *imagemin*)
  - Inline the critical CSS (*addyosmani/critical*)
  - Hash assets to make them cacheable (*gulp-rev-all*)
Network Performance
Break down of a single resource load

- **DNS Lookup**
  - Every domain has its own DNS lookup

- **Initial connection**
  - TCP makes a three way handshake $\rightarrow$ 2 roundtrips
  - SSL connections have a more complex handshake $\rightarrow$ +2 roundtrips

- **Time to First Byte**
  - Depends heavily on the distance between client and the backend
  - Includes the time the backend needs to render the page
    $\rightarrow$ Session lookups, Database Queries, Template rendering ...

- **Content Download**
  - Files have a high transfer time on new connections, since the initial congestion window is small $\rightarrow$ many roundtrips
Network Performance

Common Tuning Knobs

- Persistent connections, if possible HTTP/2
- Avoid redirects
- Explicit caching headers (no heuristic caching)

Content Delivery Networks

- To reduce the distance between client and server
- To cache images, CSS, JS
- To terminate SSL early and optimized

Single Page Apps:

- Small initial page that loads additional parts asynchronously
- Cacheable HTML templates + load dynamic data
- Only update sections of the page during navigation
Network Latency: Impact

Network Latency: Impact

2× Bandwidth = Same Load Time

½ Latency ≈ ½ Load Time
Backend Performance

Scaling your backend

- Load Balancing
- Auto-scaling
- Failover

- Stateless session handling
- Minimize shared state
- Efficient Code & IO

- Horizontally scalable databases (e.g. “NoSQL”)
  - Replication
  - Sharding
  - Failover
Research Approaches

Two Examples

Polaris:

Idea: construct graph that captures real read/write and write/write JS/CSS dependencies

Improvement: ~30% depending on RTT and bandwidth

Limitation: cannot deal with non-determinism, requires server to generate a dependency graph for each client view

Research Approaches

Two Examples

Shandian:

Client → Proxy

**Idea:** Proxy is more powerful than browser, especially mobile
→ evaluate page on proxy

**Improvement:** ~50% for slow Android device

**Limitation:** needs modified browser, only useful for slow devices

---

Other Research Approaches

Two Examples

Shandian:

Many good ideas in current research, but:

- Only applicable to **very few use cases**
- Mostly require **modified browsers**
- **Small** performance improvements

Idea: Proxy is more powerful than browser especially mobile -> evaluate page on proxy

Improvement: ~50% for slow Android device

Limitation: needs *modified browser*, only useful for slow devices

Performance: State of the Art
Summarized

Frontend
- Doable with the right set of best practices
- Good support through build tools

Latency
- Caching and CDNs help, but a considerable effort and only for static content

Backend
- Many frameworks and platforms
- Horizontal scalability is very difficult
Performance: State of the Art Summarized

Good Resources:

- [High Performance Browser Networking](https://chimera.labs.oreilly.com/books/12300000000545)
- [Designing for Performance: Working with Aesthetics and Speed](https://shop.oreilly.com/product/0636920033578.do)

Good Tools:

- PageSpeed Insights: [https://developers.google.com/speed/pagespeed/](https://developers.google.com/speed/pagespeed/)
- GTmetrix: [https://gtmetrix.com](https://gtmetrix.com)
- [WebPageTest](http://www.webpagetest.org/)
Performance: State of the Art
Summarized

**Frontend**
- Doable with the right set of best practices
- Good support through build tools

**Latency**
- Caching and CDNs help, but large effort and only for static content

**Backend**
- Many frameworks and platforms
- Horizontal scalability is very difficult

How to cache & scale **dynamic** content?
Introduction

Web Performance: State of the Art

Main Part

Cache Sketch: Research Approach

Conclusions

Using Web Caching in Applications
Goal: Low-Latency for Dynamic Content
By Serving Data from Ubiquitous Web Caches

Low Latency

Less Processing
In a nutshell

Problem: changes cause stale data
In a nutshell

Solution: Proactively Revalidate Data

Cache Sketch (Bloom filter)

Is still fresh?

update
Innovation

Solution: Proactively Revalidate Data


Web Caching Concepts
Invalidation- and expiration-based caches

Expiration-based Caches:
- An object $x$ is considered fresh for $TTL_x$ seconds
- The server assigns TTLs for each object

Invalidation-based Caches:
- Expose object eviction operation to the server
Classic Web Caching: Example

A tiny image resizer

Desktop

Mobile

Tablet

Cached and delivered many times

Resized once
The „Bloom filter principle“: 

“Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.”

- Bit array of length $m$
- $k$ independent hash functions
- `insert(obj)`: add to set
- `contains(obj)`: 
  - Always returns true if the element was inserted
  - Might return true even though it was not inserted (false positive)

Bloom filter Concepts
Visualized

Empty Bloom Filter

Insert x

Insert y

Query x
Bloom filter Concepts
False Positives

The false positive rate depends on the bits $m$ and the inserted elements $n$:

$$f \approx (1 - e^{-\ln(2)})^k \approx 0.6185 \frac{m}{n}$$

For $f=1\%$ the required bits per element are: $2.081 \ln(1/0.01) = 9.5$
Our Bloom filter
Open Source Implementation

Library of different Bloom filters in Java with optional Redis-backing, counting and many hashing options. — Edit
Our Bloom filters
Example: Redis-backed Counting Bloom Filter

- Redis-backed Bloom filters:
  - Can be **shared** by many servers
  - Highly **efficient** through Redis’ bitwise operations
  - Tunable **persistence**

- Counting Bloom Filters: use counters instead of bits to also allow **removals**
  - Stores the materialized Bloom filter for fast retrieval

```
| BITS   | 0 1 0 0 1 0 1 0 1 1 |
| COUNTS | 0 2 0 0 1 0 3 0 1 1 |
```
The Cache Sketch approach
Caching Dynamic Data

Idea: use standard *HTTP Caching* for query results and records

Problems:

- How to keep the **browser cache** up-to-date?
- When is data **cacheable** and for **how long** approximately?
- How to automatically cache dynamic data in a **CDN**?
Orestes Architecture

Infrastructure

Backend-as-a-Service Middleware:
- Caching, Transactions, Schemas, Invalidation Detection
- Storage

Unified REST API

Standard HTTP Caching

Orestes

Desktop

Mobile

Tablet

Content-Delivery-Network

Elasticsearch

MongoDB

Redis

Orestes Servers

Reverse-Proxy Caches

Node.js

User-defined Business Logic

Node.js

User-defined Business Logic

Node.js

User-defined Business Logic

Node.js

User-defined Business Logic

Node.js

User-defined Business Logic

Invalidation Detection

TTL Estimator

Cache Lifetime Prediction

Expiring Bloom Filter

Stale Data

Into

Orestes

Content Delivery

Network
Baqend Architecture

Infrastructure

CDN on fastly

IaaS-Cloud on Amazon Web Services
The Cache Sketch approach
Letting the client handle cache coherence

Client
Expiration-based Caches
Invalidation-based Caches
Request Path
Server/DB
Cache
Hits
Browser Caches, Forward Proxies, ISP Caches
Content Delivery Networks, Reverse Proxies

Client
Cache Sketch
10101010 Bloom filter

Needs Revalidation?

Server Cache Sketch
10201040
10101010
Counting Bloom Filter
Non-expired Record Keys
Report Expirations and Writes

Report Expirations and Writes

Non-expired Record Keys
10101010
10201040
Counting Bloom Filter

Periodic every Δ seconds at connect
at transaction begin

Staleness-Minimization
Invalidation-Minimization

Server Cache Sketch

Bloom filter

10101010

Staleness-Minimization
Invalidation-Minimization

10101010

Client Cache Sketch

Bloom filter

10101010

10101010

Client Cache Sketch

10101010

Bloom filter

10101010

Bloom filter

10101010

Bloom filter

10101010

Bloom filter
The End to End Path of Requests

The Caching Hierarchy

DB.posts.get GET /db/posts/{id}

Cache-Hit: Return Object
Cache-Miss or Revalidation
Request

Orestes

Updated by Cache Sketch

Updated by the server
The Client Cache Sketch

Let $c_t$ be the client Cache Sketch generated at time $t$, containing the key $\text{key}_x$ of every record $x$ that was written before it expired in all caches, i.e. every $x$ for which holds:

$$\exists r(x, t_r, \text{TTL}), w(x, t_w) : t_r + \text{TTL} > t > t_w > t_r$$

**JavaScript Bloomfilter:**

- ~100 LOCs
- ~1M lookups per second

**Guarantee:** data is never stale for more than the age of the Cache Sketch
The Server Cache Sketch
Scalable Implementation

Add $key_x$ if $x$ unexpired and write occurred

Remove $x$ from Bloom filter when expired

Load Bloom filter

Performance > 200k ops per second:
Faster Page Loads

- Clients load the Cache Sketch at connection
- Every non-stale cached record can be reused without degraded consistency
Faster CRUD Performance

Solution: **Δ-Bounded Staleness**

- Clients refresh the Cache Sketch so its age never exceeds $\Delta$

$\rightarrow$ *Consistency guarantee*: $\Delta$-atomicity
Scalable ACID Transactions

Solution: Conflict-Avoidant Optimistic Transactions
- Cache Sketch fetched with transaction begin
- Cached reads → Shorter transaction duration → less aborts
Scalable ACID Transactions

**Novelty:** ACID transactions on sharded DBs like MongoDB

Current Work: **DESY** and **dCache** building a scalable namespace for their file system on this
TTL Estimation
Determining the best TTL and cacheability

- **Problem**: if TTL $\gg$ time to next write, then it is contained in Cache Sketch unnecessarily long
- **TTL Estimator**: finds „best“ TTL
- **Trade-Off**:

  **Shorter TTLs**
  - less invalidations
  - less stale reads

  **Longer TTLs**
  - Higher cache-hit rates
  - more invalidations
Idea:
1. Estimate average time to next write $E[T_w]$ for each record
2. Weight $E[T_w]$ using the cache miss rate
End-to-End Example

Client Cache Sketch
- $b = \emptyset$

Browser
- $b = \{x_2\}$
  - Query $x_3$
  - Response $x_3$
  - Read $x_2$
  - Response $x_2$

Browser Cache
- $c = \{(x_2, t_2), (x_3, t_3)\}$

CDN Cache
- $c = \{(x_1, t_1)\}$

Server
- Connect
  - $b_{t0} = \{x_2\}$

Server Cache Sketch
- $b = \{x_2\}$
  - $t = \{(x_2, t_2), (x_3, t_3), (x_1, t_1)\}$

Client Cache Sketch
- $b = \{x_1\}$
- $c = \{(x_3, t_3)\}$

Browser
- $b = \{x_2\}$
  - Query $x_2$
  - Response $x_2$

Browser Cache
- $c = \{(x_3, t_3)\}$

CDN Cache
- $c = \{(x_2, t_4)\}$

Server
- Report Read
  - $x_2, t_4$

Server Cache Sketch
- $b = \{x_2\}$
  - $t = \{(x_2, t_4), (x_3, t_3), (x_1, t_1)\}$

Client Cache Sketch
- $b = \{x_1, x_2\}$
- $c = \{(x_1, t_4)\}$

Browser
- $b = \{x_1, x_2\}$
  - Query $x_1$
  - Response $x_1$

Browser Cache
- $c = \{(x_1, t_4)\}$

CDN Cache
- $c = \{(x_2, t_4)\}$

Server
- Report Write
  - $x_1$

Server Cache Sketch
- $b = \{x_1, x_2\}$
  - $t = \{(x_2, t_4), (x_3, t_3), (x_1, t_1)\}$

Client Cache Sketch
- $b = \{x_1\}$
- $c = \{(x_1, t_4)\}$

Browser
- $b = \{x_1\}$
  - Query $x_1$
  - Response $x_1$

Browser Cache
- $c = \{(x_1, t_4)\}$

CDN Cache
- $c = \{(x_2, t_4)\}$

Server
- Invalidate
  - $x_1$

Server Cache Sketch
- $b = \{x_1, x_2\}$
  - $t = \{(x_2, t_4), (x_3, t_3), (x_1, t_1)\}$

Client Cache Sketch
- $b = \{x_1\}$
- $c = \{(x_1, t_4)\}$

Browser
- $b = \{x_1\}$
  - Query $x_1$
  - Response $x_1$

Browser Cache
- $c = \{(x_1, t_4)\}$

CDN Cache
- $c = \{(x_2, t_4)\}$

Server
- Report Read
  - $x_2, t_4$

Server Cache Sketch
- $b = \{x_2\}$
  - $t = \{(x_2, t_4), (x_3, t_3), (x_1, t_1)\}$
## Consistency

### What are the guarantees?

<table>
<thead>
<tr>
<th>Consistency Level</th>
<th>How</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Δ-atomicity</strong> (staleness never exceeds Δ seconds)</td>
<td>Controlled by age of Cache Sketch</td>
</tr>
<tr>
<td>Montonic Writes</td>
<td>Guaranteed by database</td>
</tr>
<tr>
<td><strong>Read-Your-Writes and Montonic Reads</strong></td>
<td>Cache written data and most recent read-versions in client</td>
</tr>
<tr>
<td><strong>Causal Consistency</strong></td>
<td>If read timestamp is older than Cache Sketch it is given, else revalidate</td>
</tr>
<tr>
<td><strong>Strong Consistency</strong> (Linearizability)</td>
<td>Explicit revalidation (cache miss at all levels)</td>
</tr>
</tbody>
</table>
**Performance**

Setup:

Northern California: Client - CDN - Orestes - MongoDB

Ireland

Page load times with **cached initialization** (simulation):

With Facebook’s cache hit rate: >2,5x improvement

Average Latency for YCSB Workloads A and B (real):

95% Read 5% Writes → 5x latency improvement
Varnish and Fastly
What we do on the edge

Reject rate limited users

Cache all GET requests

Collect access logs & report failures

Validate & renew session tokens of users

Authorize the user on protected resources

Handle CORS pre-flight requests

Authorize the user on protected resources

Access-Control-*
The Cache Sketch

Summary

Static Data

Mutable Objects

Queries/Aggregates

Immutability ideal for static web caching: max-age=31557600

Cache Sketch for browser cache, proxies and ISP caches

Invalidations for CDNs and reverse proxies

SELECT TOP 4, WHERE tag="x"

How to do this?
Continuous Query Matching
Generalizing the Cache Sketch to query results

Main challenge: when to invalidate?
- **Objects**: for every update and delete
- **Queries**: as soon as the query result changes

How to detect query result changes in real-time?
Query Caching

Example

- Add, Change, Remove all entail an invalidation and addition to the cache sketch.
Architecture

Create
Update
Delete

ORESTES

01111101

Polyglot Views

Fresh Caches

Fresh Cache Sketch

Continuous Queries (Websockets)

Pub-Sub

InvaliDB
Matching on Apache Storm

Apache Storm:
• "Hadoop of Real-Time"
• Low-Latency Stream Processing
• Custom Java-based Topologies

InvaliDB goals:
• Scalability, Elasticity, Low latency, Fault-tolerance
Query Matching Performance
Latency of detecting invalidations

- Latency mostly < 15ms, scales linearly w.r.t. number of servers and number of tables
Learning Representations
Determining Optimal TTLs and Cacheability

**Setting:** query results can either be represented as references (id-list) or full results (object-lists)

<table>
<thead>
<tr>
<th>Id-Lists</th>
<th>Object-Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>([id_1, id_2, id_3])</td>
<td>([{id: 1, val: 'a'}, {id: 2, val: 'b'}, {id: 3, val: 'c'}])</td>
</tr>
</tbody>
</table>

Less Invalidations  
Less Round-Trips

**Approach:** Cost-based decision model that weighs expected round-trips vs expected invalidations

**Ongoing Research:** Reinforcement learning of decisions
What is the impact of query caching?
What is the impact of query caching?

**Insight:**

Query Caching = Real-Time Apps
Continuous Queries
Complementing Cached Queries

- Same streaming architecture can similarly notify the browser about query result changes

**Application Pattern:**

- Initial Page Load using *Cached Queries*
- **Subscribe** tag='b'
- **Insert** tag='b' ...
- Critical data declaratively specified and proactively pushed via websockets
Continuous Query API
Subscribing to database updates

```javascript
var stream = DB.News.find().stream();
stream.on("add", onNews);
stream.on("remove", onRemove);
```
Summary

- **Orestes**: DB-independent Backend-as-a-Service
- **Cache Sketch Approach**:
  - Client decides when to `revalidate`, server `invalidates` CDN
  - Cache Sketch = **Bloom filter** of stale IDs
  - Compatible with end-to-end ACID transactions
  - Query change detection in **real-time**
Introduction

Web Performance: State of the Art

Main Part

Cache Sketch: Research Approach

Conclusions

Using Web Caching in Applications
Orestes Caching Technology as a Backend-as-a-Service
Page-Load Times

What impact does caching have in practice?
Welcome to Baqend Cloud

Your Baqend account has been created!
Have a look at these resources to help you get started quickly.

| Tutorial | Getting Started | Documentation | API Docs |

| bbq | Running |

<table>
<thead>
<tr>
<th>Requests</th>
<th>Outgoing Data</th>
<th>DB Space</th>
<th>CDN Cache Hit ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,889</td>
<td>19.1 MB</td>
<td>4.1 MB</td>
<td>9.8%</td>
</tr>
</tbody>
</table>

App Status:

- Pay as you go:
  - Current Plan: 500€ (Business)

- Set Limit:

<table>
<thead>
<tr>
<th>td</th>
<th>Running</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests</td>
<td>Outgoing Data</td>
</tr>
<tr>
<td>2,905</td>
<td>2.9 MB</td>
</tr>
</tbody>
</table>

App Status:

- Pay as you go:
  - Current Plan: 500€ (Business)
Want to try Baqend?

- Free Baqend Cloud instance
- Download Community Edition
Thank you

Questions?

Twitter: @baqendcom

fg@baqend.com

baqend.com