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

Orestes

Backend-as-a-Service Startup since 2014
Introduction

Main Part

Conclusions

Web Performance: State of the Art

Cache Sketch: Research Approach

Using Web Caching in Applications
Presentation is loading
Why performance matters

100 ms

Loading...

-1% Revenue

Average: 9.3s
Why performance matters

400 ms

Average: 9,3s

Loading…

-9% Visitors

-1% Revenue
### Why performance matters

<table>
<thead>
<tr>
<th>Loading…</th>
<th>Average: 9,3s</th>
</tr>
</thead>
<tbody>
<tr>
<td>500 ms</td>
<td></td>
</tr>
</tbody>
</table>

- **-20%** Traffic
- **-9%** Visitors
- **-1%** Revenue
Why performance matters

Average: 9.3s

1s

Loading…

-7% Conversions

-20% Traffic

-9% Visitors

-1% Revenue
An Average Website
Some Statistics

Average Bytes per Page by Content Type

- Images - 1463 kB
- Stylesheets - 78 kB
- HTML - 66 kB
- Fonts - 124 kB
- Video - 200 kB
- Other - 4 kB
- Scripts - 360 kB

Total 2301 kB

http://httparchive.org/
An Average Website

Some Statistics

Total Requests per Page

- 1-25 requests: 11%
- 26-50 requests: 16%
- 51-75 requests: 19%
- 76-100 requests: 15%
- 101-125 requests: 12%
- 126-150 requests: 8%
- 151-175 requests: 6%
- 176-200 requests: 4%
- 201-225 requests: 2%
- 226-250 requests: 2%
- 251-275 requests: 1%

http://httparchive.org/
An Average Website
Some Statistics

Total Transfer Size & Total Requests

[Graph showing total transfer size and total requests over time]

http://httparchive.org/
If perceived speed is such an important factor...

...what causes slow page load times?
The Problem
Three Bottlenecks: Latency, Backend & Frontend

High Latency
Achieve a fast render of the page by:

- Reducing the **critical resources** needed
- Reducing the **critical bytes** which must be transferred
- Loading JS, CSS and HTML templates **asynchronously**
- Rendering the page **progressively**
- **Minifying & Concatenating** CSS, JS and images
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 → 2 roundtrips
  - SSL connections have a more complex handshake → +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
    → Session lookups, Database Queries, Template rendering ...

- **Content Download**
  - Files have a high transfer time on new connections, since the initial congestion window is small → 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

Page Load Time as bandwidth increases:
1 Mbps: 3500 ms
2 Mbps: 2000 ms
3 Mbps: 1500 ms
4 Mbps: 1500 ms
5 Mbps: 1500 ms
6 Mbps: 1500 ms
7 Mbps: 1500 ms
8 Mbps: 1500 ms
9 Mbps: 1500 ms
10 Mbps: 1500 ms

Page Load Time as latency decreases:
200 ms: 3500 ms
180 ms: 3000 ms
160 ms: 2500 ms
140 ms: 2000 ms
120 ms: 1500 ms
100 ms: 1000 ms
80 ms: 500 ms
60 ms: 500 ms
40 ms: 500 ms
20 ms: 500 ms

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:

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:

- https://developers.google.com/web/fundamentals/performance/?hl=en
- https://chimera.labs.oreilly.com/books/12300000000545
- https://shop.oreilly.com/product/0636920033578.do

Good Tools:

- PageSpeed Insights
  - http://www.baqend.com/
- GTmetrix
  - 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
In a nutshell

Problem: changes cause stale data
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 $\text{TTL}_x$ 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
Bloom filter Concepts
Compact Probabilistic Sets

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

=1? contained

y contained
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

<table>
<thead>
<tr>
<th>Source</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gradle/wrapper</td>
<td>cleanup build</td>
<td></td>
</tr>
<tr>
<td>src</td>
<td>-Implemented and tested Bloom filter for JS</td>
<td></td>
</tr>
<tr>
<td>.gitignore</td>
<td>.gitignore.iml</td>
<td></td>
</tr>
<tr>
<td>CHANGELOG.md</td>
<td>Update CHANGELOG.md</td>
<td>8 months ago</td>
</tr>
<tr>
<td>LICENSE</td>
<td>Added Tutorial steps</td>
<td>3 years ago</td>
</tr>
<tr>
<td>README.md</td>
<td>Update README.md</td>
<td>8 months ago</td>
</tr>
<tr>
<td>bloom-filter.iml</td>
<td>-Updated GSON</td>
<td>2 months ago</td>
</tr>
<tr>
<td>build.gradle</td>
<td>-Updated GSON</td>
<td>2 months ago</td>
</tr>
<tr>
<td>gradle.properties</td>
<td>[ci skip] new version commit: '1.1.8-SNAPSHOT'</td>
<td>14 days ago</td>
</tr>
</tbody>
</table>
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
The Cache Sketch approach
Caching Dynamic Data

- **Idea**: use standard *HTTP Caching* for query results and records
- **Problems**:
  - When is data **cacheable** and for **how long** approximately?
  - How to keep the **browser cache** up-to-date?
  - How to automatically cache dynamic data in a **CDN**?
Orestes Architecture

Infrastructure

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

desktop

mobile

tablet

internet

content-delivery-network

backend

as

service

middleware:
caching,
transactions,
schemas,
invalidation

detection,
...
Orestes Architecture

Infrastructure

Standard HTTP Caching

Diagram showing various components of the Orestes architecture, including desktop, mobile, and tablet devices, Internet, Reverse-Proxy Caches, Orestes Servers, and databases such as redis, mongoDB, and elasticsearch.
Orestes Architecture
Infrastructure

Unified REST API

Desktop
Mobile
Tablet

Internet
Content-Delivery-Network

InvaliDB
Streaming Queries

TTL Estimator
Cache Lifetime Prediction

Expiring Bloom Filter
Stale Data

Node.js
User-defined Business Logic

Reverse-Proxy Caches

Orestes Servers

redis
mongoDB
elasticsearch
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

Request Path

Cache Hits

Browser Caches, Forward Proxies, ISP Caches

Invalidation-based Caches

Invalidations, Records

Content Delivery Networks, Reverse Proxies

Server/DB

Needs Revalidation?

10101010 Bloom filter

Client Cache Sketch

Periodic every $\Delta$ seconds at connect

Needs Invalidation?

10101010

Server Cache Sketch

Non-expired Record Keys

Report Expirations and Writes

Counting Bloom Filter

10201040
The End to End Path of Requests
The Caching Hierarchy

DB.posts.get(id) → JavaScript

Client-(Browser-)
Cache → Miss → Proxy
Caches → Miss → ISP
Caches → Miss → CDN
Caches → Miss → Reverse-
Proxy Cache → Miss → Orestes

Welcome User!
Dynamic Web App
Javascript
The End to End Path of Requests
The Caching Hierarchy

GET /db/posts/{id} HTTP

Dynamic Web App

Client-(Browser)-Cache -> Proxy Caches -> ISP Caches -> CDN Caches
Miss Miss Miss Miss

Orestes
The End to End Path of Requests

The Caching Hierarchy

Cache-Hit: Return Object
Cache-Miss or Revalidation: Forward Request
The End to End Path of Requests

The Caching Hierarchy

- Client-(Browser-) Cache
- Proxy Caches
- ISP Caches
- CDN Caches
- Reverse-Proxy Cache

Return record from DB with caching TTL
The End to End Path of Requests
The Caching Hierarchy
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:

\[
\exists r(x, t_r, TTL), w(x, t_w) : t_r + TTL > t > t_w > t_r
\]
Let $c_t$ be the client Cache Sketch generated at time $t$, containing the key $key_x$ of every record $x$ that was written before it expired in all caches, i.e. every $x$ for which holds:

$$\exists r(x, t_r, TTL), w(x, t_w) : t_r + TTL > t > t_w > t_r$$
Let $c_t$ be the client Cache Sketch generated at time $t$, containing the key $key_x$ of every record $x$ that was written before it expired in all caches, i.e. every $x$ for which holds:

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

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:
1 Faster Page Loads

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

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

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

- Clients load the Cache Sketch at connection
- Every non-stale cached record can be reused without degraded consistency
1. Faster Page Loads

- Clients load the Cache Sketch at connection.
- Every non-stale cached record can be reused without degraded consistency.
1. Faster Page Loads

- Clients load the Cache Sketch at connection.
- Every non-stale cached record can be reused without degraded consistency.
1 Faster Page Loads

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

![Diagram showing the process of faster page loads with hashA(oid) and hashB(oid) as well as the sequence 0 1 1 1 1 for browsers and 0 3 1 4 1 for CDNs.](image_url)
1 Faster Page Loads

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

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

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

False-Positive Rate:

\[ f \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \]

Hash-Functions:

\[ k = \left\lfloor \ln(2) \cdot \left(\frac{n}{m}\right) \right\rfloor \]

With 20,000 distinct updates and 5% error rate: 11 KByte
2 Faster CRUD Performance

- Solution: **Δ-Bounded Staleness**
  - Clients refresh the Cache Sketch so its age never exceeds Δ
  -> *Consistency guarantee*: Δ-atomicity

![Diagram](image-url)
Scalable ACID Transactions

- Solution: **Conflict-Avoidant Optimistic Transactions**
  - Cache Sketch fetched with transaction begin
  - **Cached reads → Shorter transaction duration → less aborts**
3 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
**TTL Estimation**  
Determining the best TTL

**Idea:**
1. Estimate average time to next write $E[T_w]$ for each record
2. Weight $E[T_w]$ using the cache miss rate

Client

Reads

Caches

Server

Write rate $\sim$ Poisson

Misses

Write rate $\sim$ Poisson

\[ \lambda_m: \text{Miss Rate per record} \]

\[ \lambda_w: \text{Write Rate per record} \]

**TTL Estimator**

Objective:
- maximize Cache Hits
- minimize Purges
- minimize Stale Reads
- bound Cache Sketch
- bound Cache Sketch

false positive rate

Write rate [ops/time unit]

miss rate [ops/time unit]

No Caching

Maximum TTL

\[ \lambda_m \lambda_w \]
TTL Estimation
Determining the best TTL

Idea:
1. Estimate average time to next write $E[T_w]$ for each record
2. Weight $E[T_w]$ using the cache miss rate

Good TTLs $\rightarrow$ small Bloom filter
TTL $< \text{TTL}_{\text{min}}$ $\rightarrow$ no caching of write-heavy objects
**End-to-End Example**

Client Cache Sketch

- $b = \emptyset$

Browser

- $b = \{x_2\}$

- **Query**
  - $x_3$

- **Response**
  - $false$

- **Query**
  - $x_2$

- **Response**
  - $true$

- **Query**
  - $x_3$

- **Response**
  - $c = \{(x_2, t_2), (x_3, t_3)\}$

- **Response**
  - $c = \{(x_1, t_1)\}$

- **Response**
  - $c = \{(x_3, t_3)\}$

Server Cache Sketch

- $b = \{x_2\}$

- **Connect**
  - $t = \{(x_2, t_2), (x_3, t_3), (x_1, t_1)\}$

- **Report Read**
  - $x_2, t_4$

- **Report Write**
  - $x_1$

- **Response**
  - $false$

- **Response**
  - $true$

- **Response**
  - $ok$

- **Response**
  - $inv = true$

- **Response**
  - $x_1$
## 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><strong>Montonic Writes</strong></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 versions</td>
</tr>
<tr>
<td><strong>Causal Consistency</strong></td>
<td>If timestamp older than Cache Sketch it is given, else revalidate</td>
</tr>
<tr>
<td><strong>Strong Consistency (Linearizability)</strong></td>
<td>Explicit Revalidation (Cache Miss)</td>
</tr>
</tbody>
</table>

**Always**

**Opt-in**
Performance

Setup:

Northern California
Client → CDN → Orestes → MongoDB
Ireland

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

- Average latency for YCSB Workloads A and B (real):
  - With Facebook’s cache hit rate: >2.5x improvement
  - 95% Read 5% Writes $\rightarrow$ 5x latency improvement
Varnish and Fastly
What we do on the edge

- Cache all GET requests
- Validate & renew session tokens of users
- Authorize the user on protected resources
- Collect access logs & report failures
- Handle CORS pre-flight requests
- Reject rate limited users
The Cache Sketch

Summary

**Static Data**

Immutability ideal for static web caching:

max-age=31557600

**Mutable Objects**

```
{
    "id": "/db/Todo/b5d9bef9-6c1f-46a5-...",
    "version": 1,
    "acl": null,
    "listId": "7b92c069-...",
    "name": "Test",
    "activities": [],
    "active": true,
    "done": false
}
```

**Queries/Aggregates**

SELECT TOP 4, WHERE tag="x"

Cache Sketch for browser cache, proxies and ISP caches

Invalidations for CDNs and reverse proxies

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

Continuous Queries (Websockets)

Polyglot Views

Fresh Caches

Fresh Cache Sketch

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>
</table>
| \([id_1, id_2, id_3]\)  | \([\{id: 1, \text{val: 'a'}\}, \{id: 2, \text{val: 'b'}\},\)\]
|                         | \(\{id: 3, \text{val: 'c'}\}\]                   |

- **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

Main Part

Conclusions

Web Performance: State of the Art

Cache Sketch: Research Approach

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

BaQend
Build faster Apps faster.
## Page-Load Times

What impact does caching have in practice?

<table>
<thead>
<tr>
<th>Politik</th>
<th>Wirtschaft</th>
<th>Kultur</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deutsche Rentenversicherung</td>
<td>Guter Rat zur Geldanlage ist selten</td>
<td>NICOLAUS HARMONCOURT</td>
</tr>
<tr>
<td>Renten könnten 2015 um zwei Prozent steigen</td>
<td>Guter Rat zur Geldanlage ist selten</td>
<td>Mozarts Triptychon</td>
</tr>
<tr>
<td>Europäischer Gerichtshof</td>
<td>Der berühmteste Wohlträger Chinas – nach eigenen Angaben</td>
<td>HANS MAGNUS ENZENBERGER</td>
</tr>
<tr>
<td>Deutschland darf EU-Ausländern Hartz IV verweigern</td>
<td>Der chinesische Unternehmer Chen Guangbao wurde ausgerechnet mit Bauschutt sehr reich. Jetzt hat er Wände aus Geldbündeln und zentrifugiert öffentlich Luxusuhrs.</td>
<td>Der Unerschütterliche</td>
</tr>
<tr>
<td>APEC-Gipfeltreffen</td>
<td>China steckt in der Wachstumsschlucht</td>
<td>DDR-DESIGN</td>
</tr>
<tr>
<td>Obama besänftigt China</td>
<td>Jahrhundert hat China die Welt mit hohen, oft zweistelligen Wachstumsraten beeindruckt. Doch diese Zeiten sind vorbei, wie unsere Grafik des Tages zeigt.</td>
<td>Sandmännchen und Stasi-Mikrofone</td>
</tr>
<tr>
<td>ISRAEL</td>
<td>Russlands Zentralbank lässt Rubel frei handeln</td>
<td>AZEALIA BANKS</td>
</tr>
<tr>
<td>Keiner will von Intifada sprechen</td>
<td>Russlands Zentralbank lässt Rubel frei handeln</td>
<td>Klare Ansage aus Harlem</td>
</tr>
<tr>
<td>Messerattacken auf Israelis, Krawalle auf dem Tempelberg, Scharmützel im Gassengewirr</td>
<td>Russlands Zentralbank lässt Rubel frei handeln</td>
<td>Erst galt Azealia Banks als großes Raptalent, dann als strebsüchtig und selbstverliebt. Ihr erstes Album erwartetes Debüt zeigt jetzt, wie gut das eine zum anderen passt.</td>
</tr>
</tbody>
</table>
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

App Status

- Running

- bbq

3,889 Requests
19,1 MB Outgoing Data
4,1 MB DB Space
9.8 % CDN Cache Hit ratio

Pay as you go
0 €
Set Limit

Current Plan
50 €
Medium
Change Plan

- td

2,405 Requests
2,9 MB Outgoing Data
114,8 KB DB Space
43.8 % CDN Cache Hit ratio

Pay as you go
0 €
Set Limit

Current Plan
500 €
Business
Change Plan

Live Demo: Using Caching in Practice
Want to try Baqend?

- Free **Baqend Cloud** instance
- Download **Community Edition**
Thank you

Questions?

baqend.com

fg@baqend.com

Twitter: @baqendcom