Tutorial on Scalable Cloud-Databases in Research and Practice

Norbert Ritter, Felix Gessert

{ritter,gessert}@informatik.uni-hamburg.de

Outline

Motivation

ORESTES: a Cloud-Database Middleware

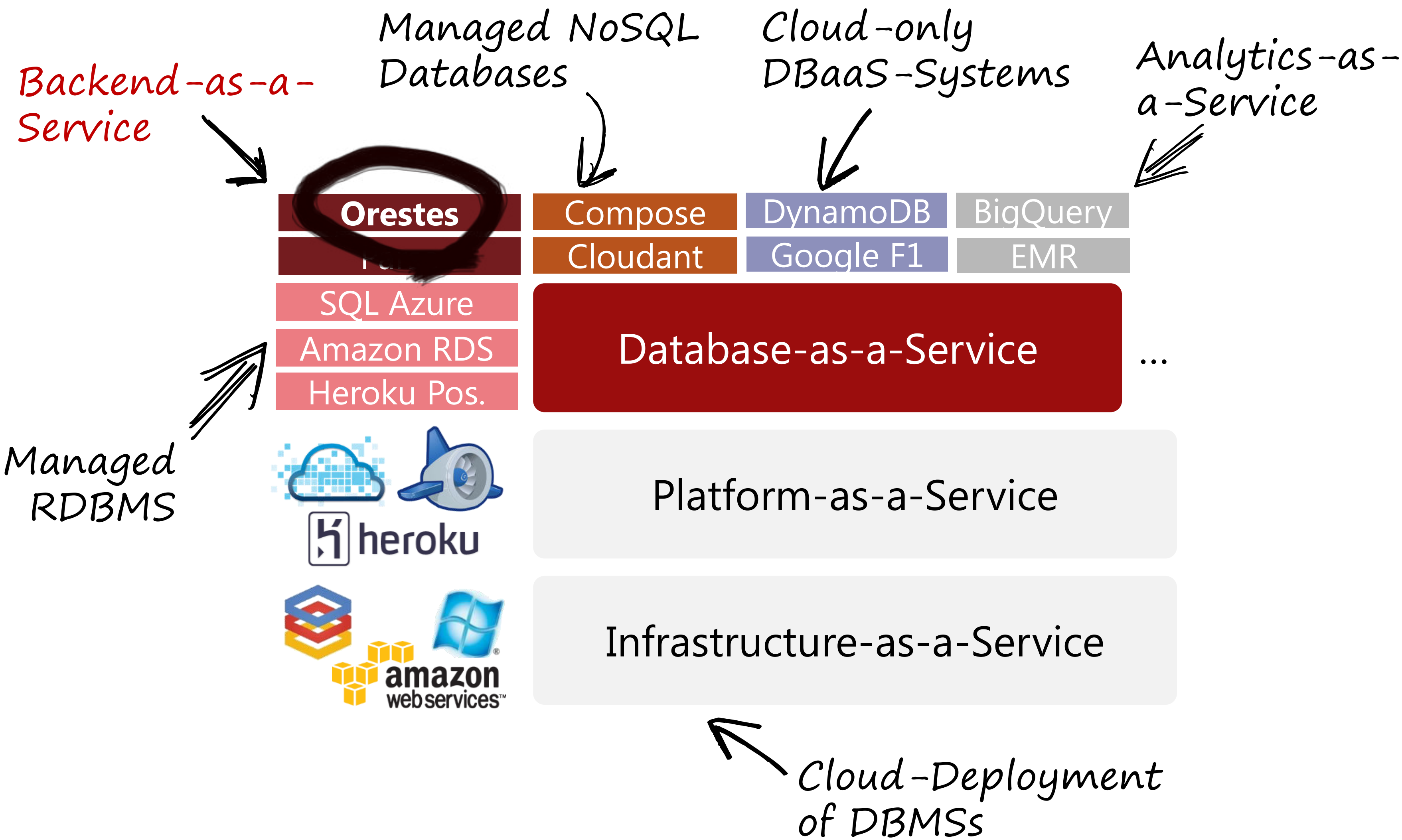
Solving Latency and Polyglot Storage

Wrap-up

- Overview
- The New Field *Cloud Data Management*
- Cloud Database Models
- Research Challenges

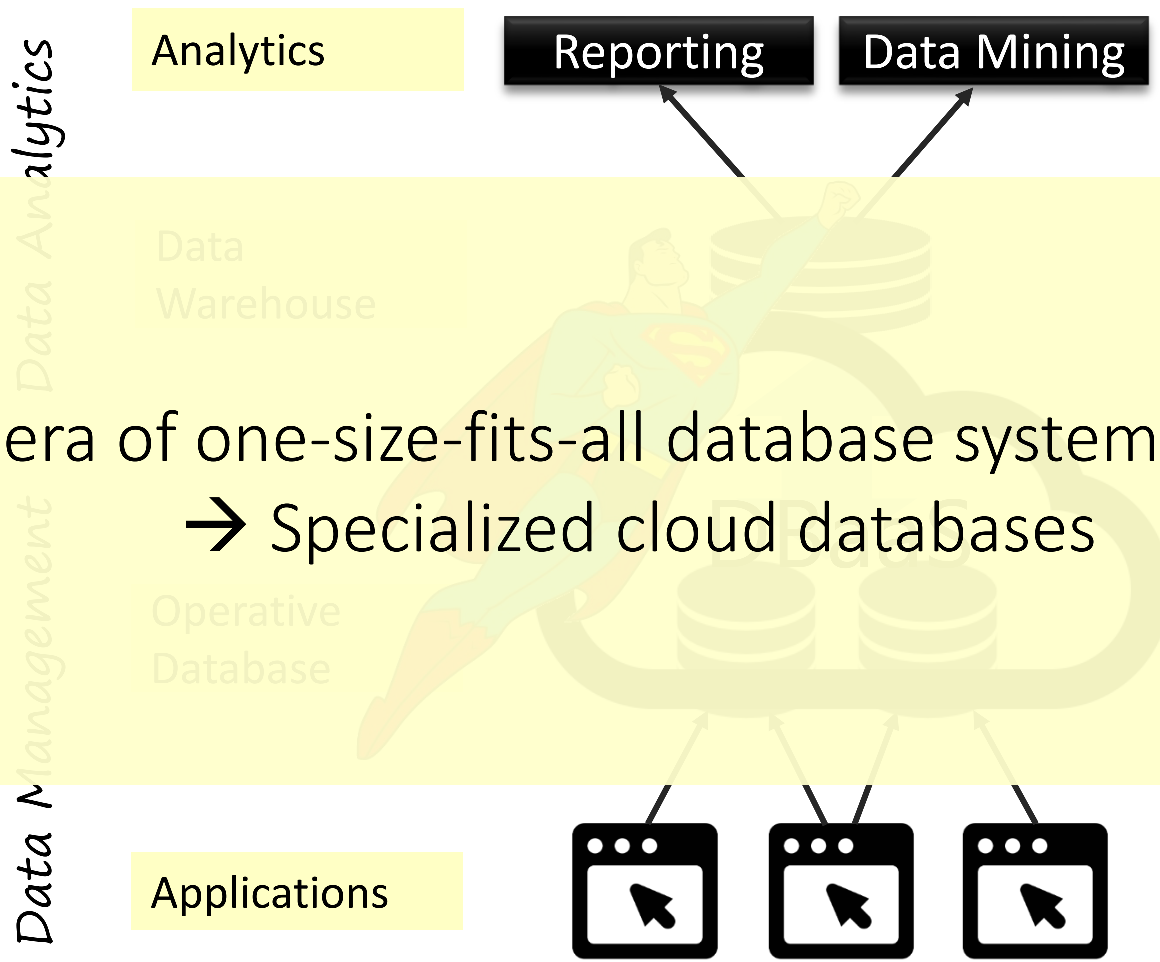
Introduction: Which classes of cloud databases are there?

Cloud Databases



Architecture

Typical Data Architecture:



The era of one-size-fits-all database systems is over

→ Specialized cloud databases

Database Sweetspots

RDBMS

General-purpose ACID transactions

Parallel DWH

Aggregations/OLAP for massive data amounts

NewSQL

High throughput relational OLTP

Wide-Column Store

Long scans over structured data

Document Store

Deeply nested data models

Key-Value Store

Large-scale session storage

Graph Database

Graph algorithms & queries

In-Memory KV-Store

Counting & statistics

Wide-Column Store

Massive user-generated content

Cloud-Database Sweetspots

Firebase

Realtime BaaS

Communication and collaboration

Amazon RDS

Managed RDBMS

General-purpose ACID transactions

Amazon ElastiCache

Managed Cache

Caching and transient storage

Azure Tables

Wide-Column Store

Very large tables

Amazon DynamoDB

Wide-Column Store

Massive user-generated content

Parse

Backend-as-a-Service

Small Websites and Apps

bonsai

Managed NoSQL

Full-Text Search

Google Cloud Storage

Object Store

Massive File Storage

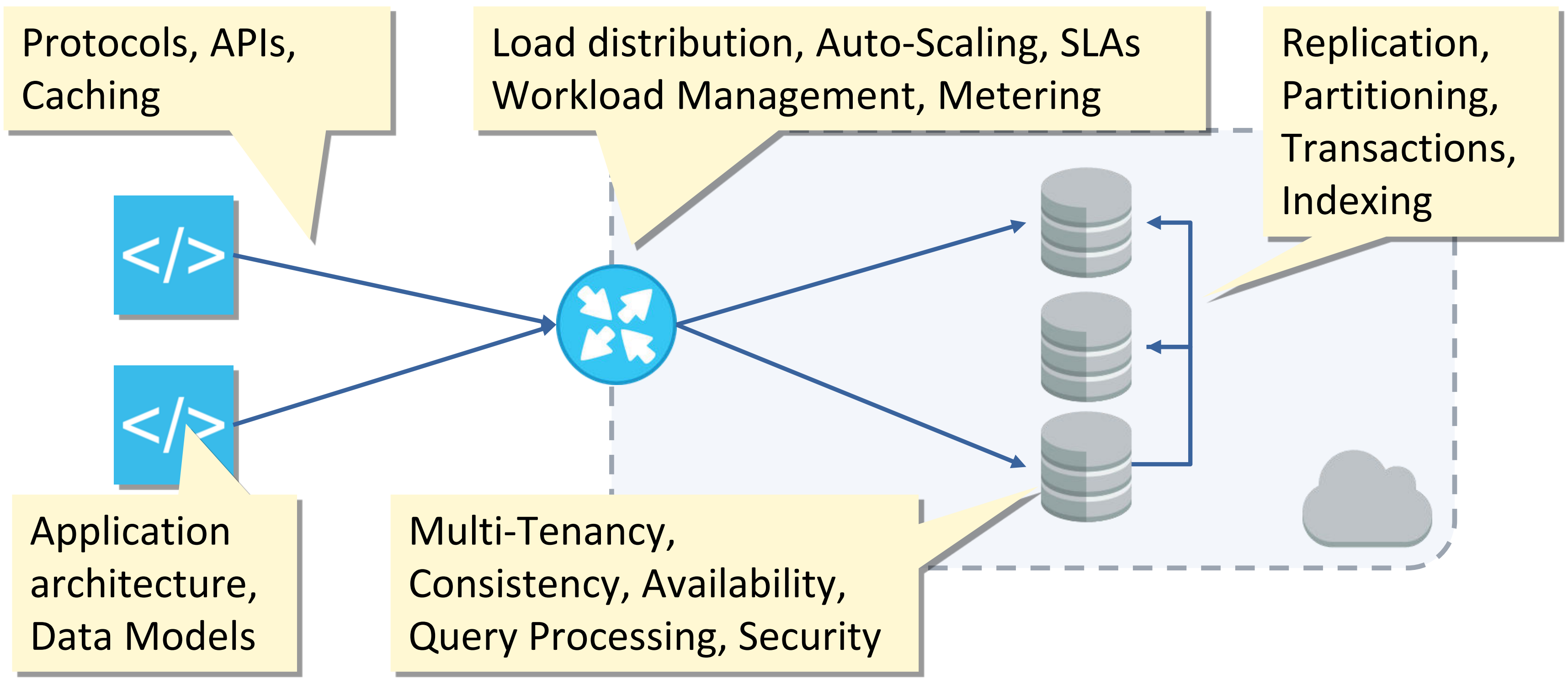
Amazon Elastic MapReduce

Hadoop-as-a-Service

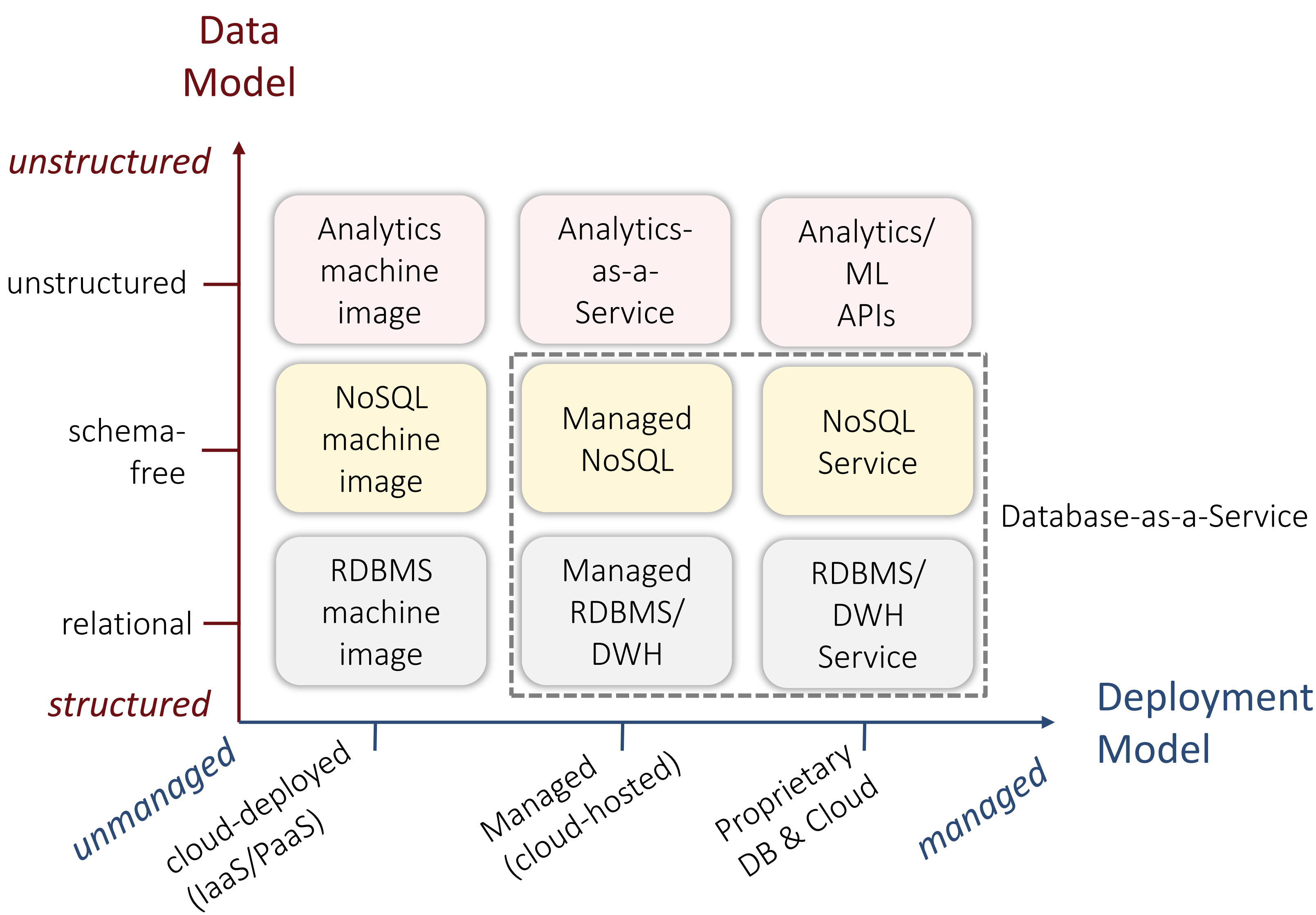
Big Data Analytics

Cloud Data Management

- New field tackling the *design*, *implementation*, *evaluation* and *application implications* of **database systems in cloud environments**:

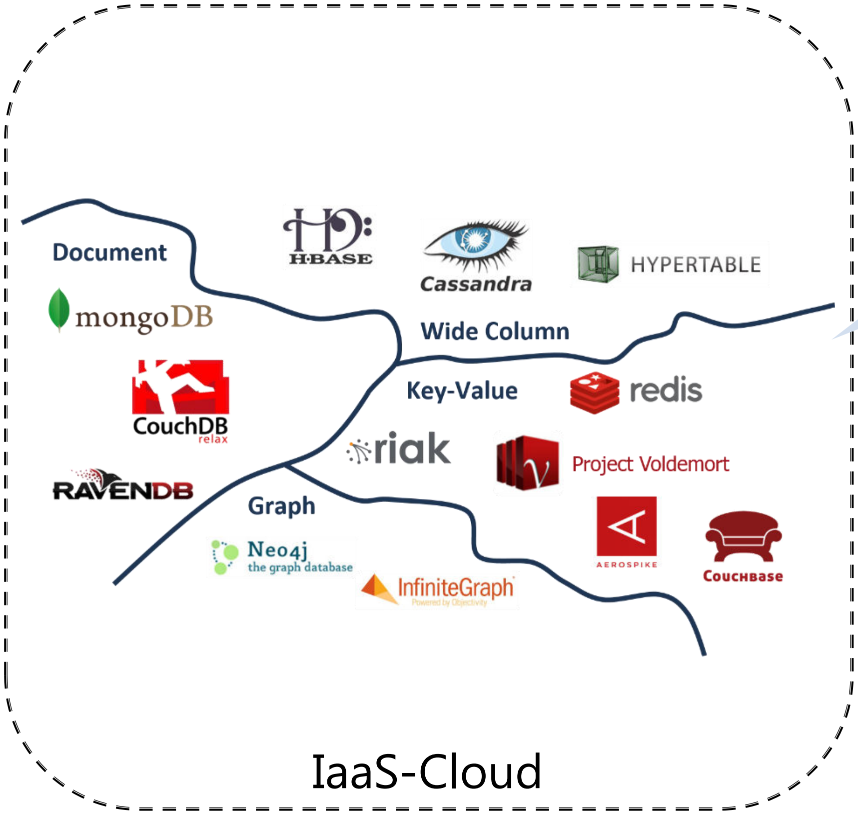


Cloud-Database Models



Cloud-Deployed Database

Database-image provisioned in IaaS/PaaS-cloud



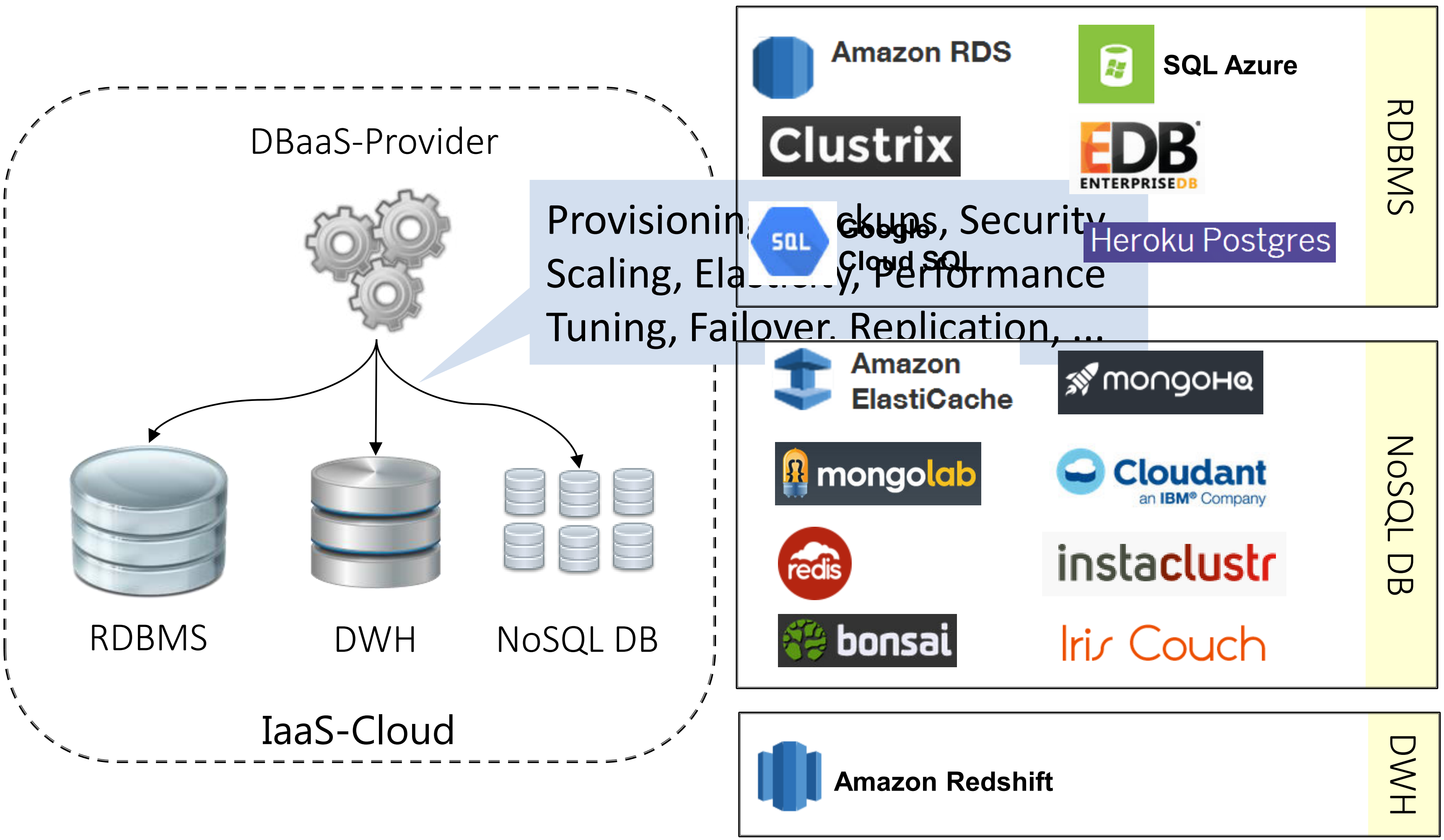
IaaS/PaaS deployment of database system

Does not solve:

Provisioning, Backups, Security, Scaling, Elasticity, Performance Tuning, Failover, Replication, ...

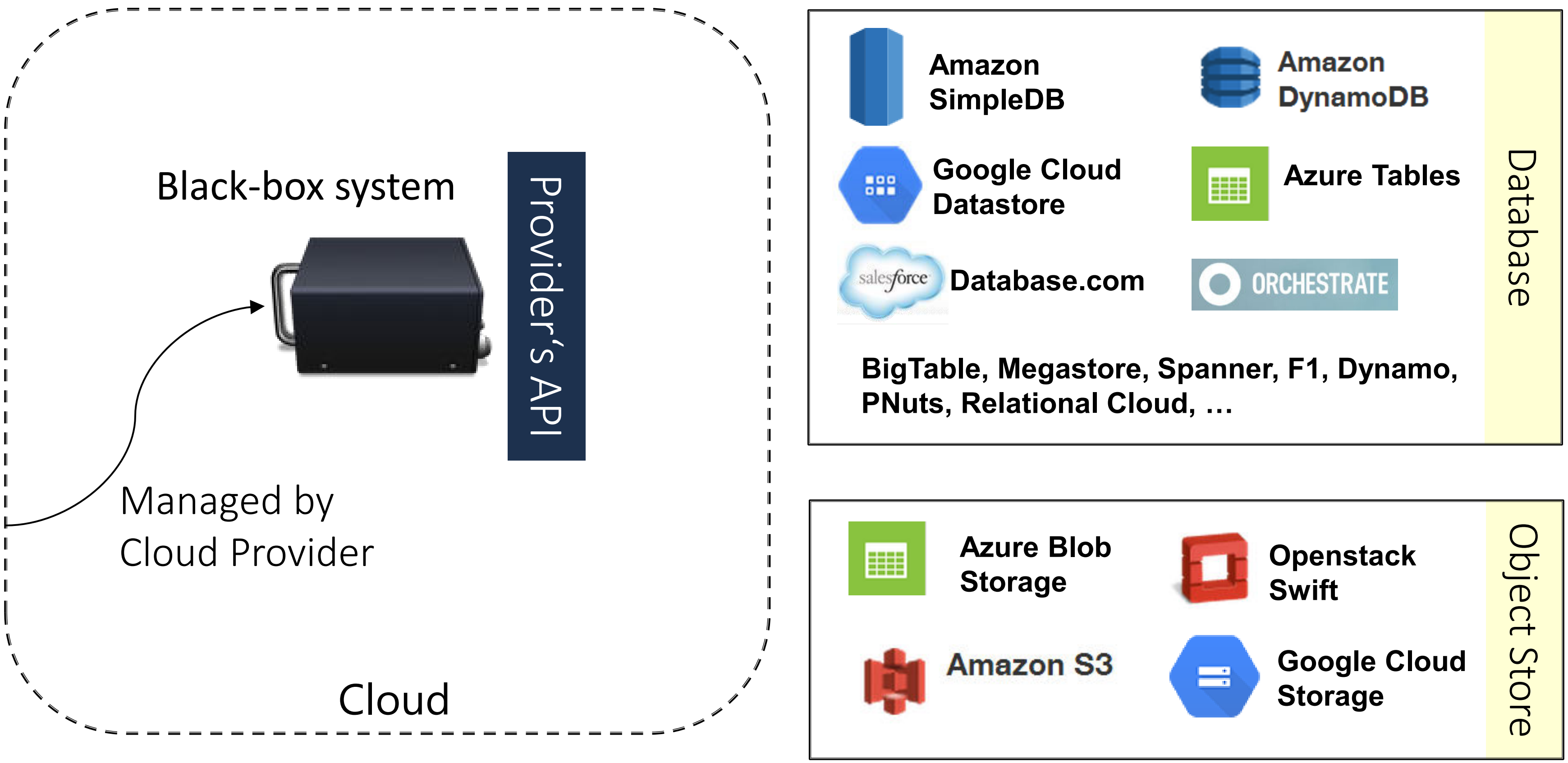
Managed RDBMS/DWH/NoSQL DB

Cloud-hosted database



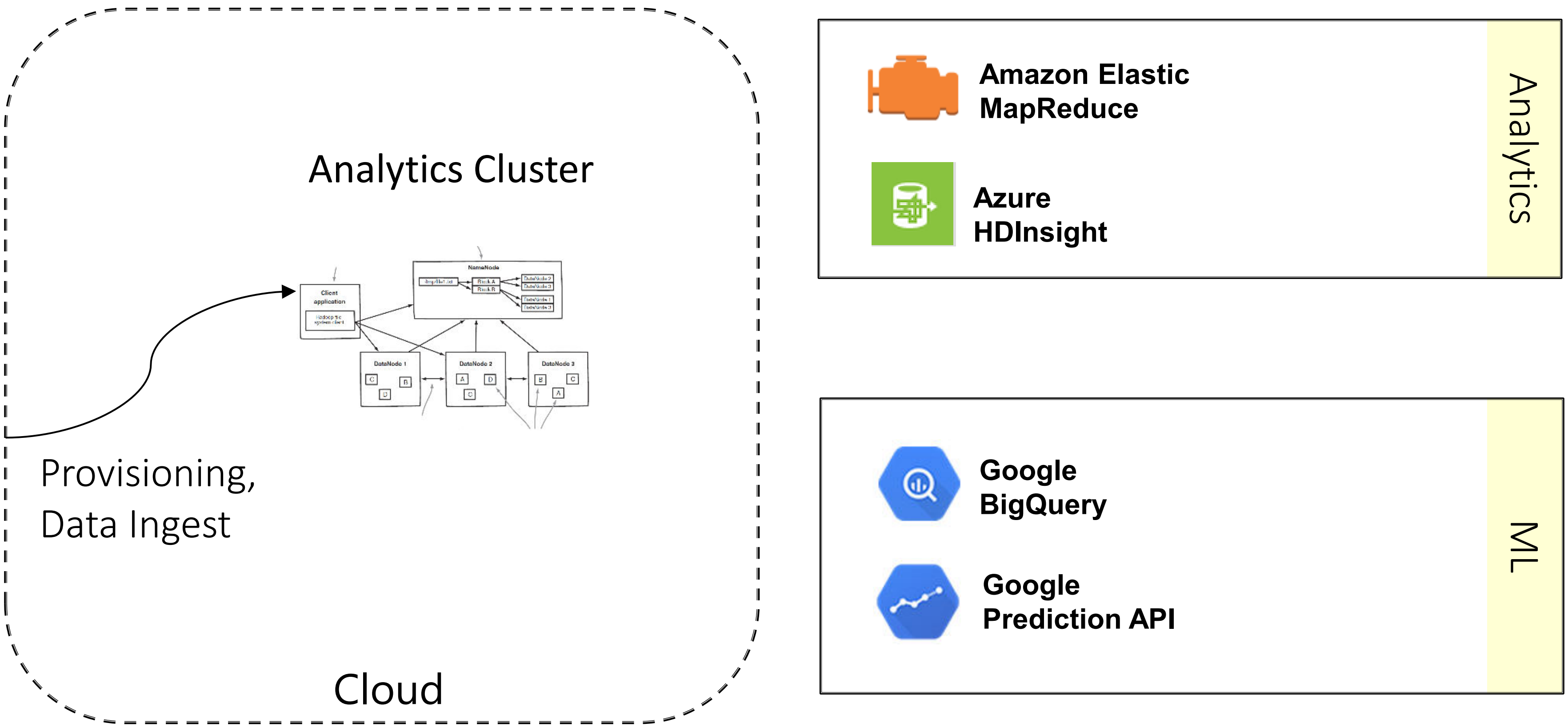
Proprietary Cloud Database

Designed for and deployed in vendor-specific cloud environment



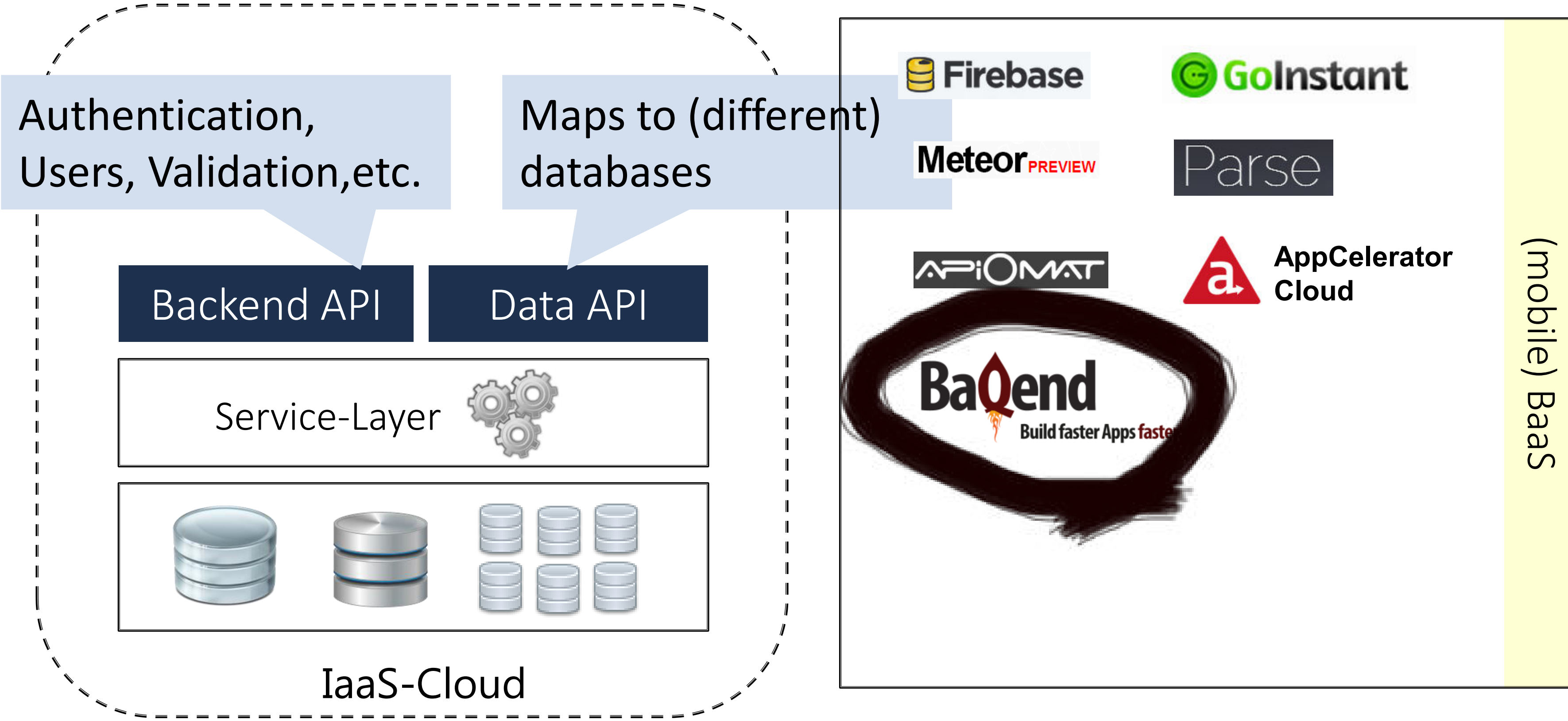
Analytics-as-a-Service

Analytic frameworks and machine learning with service APIs



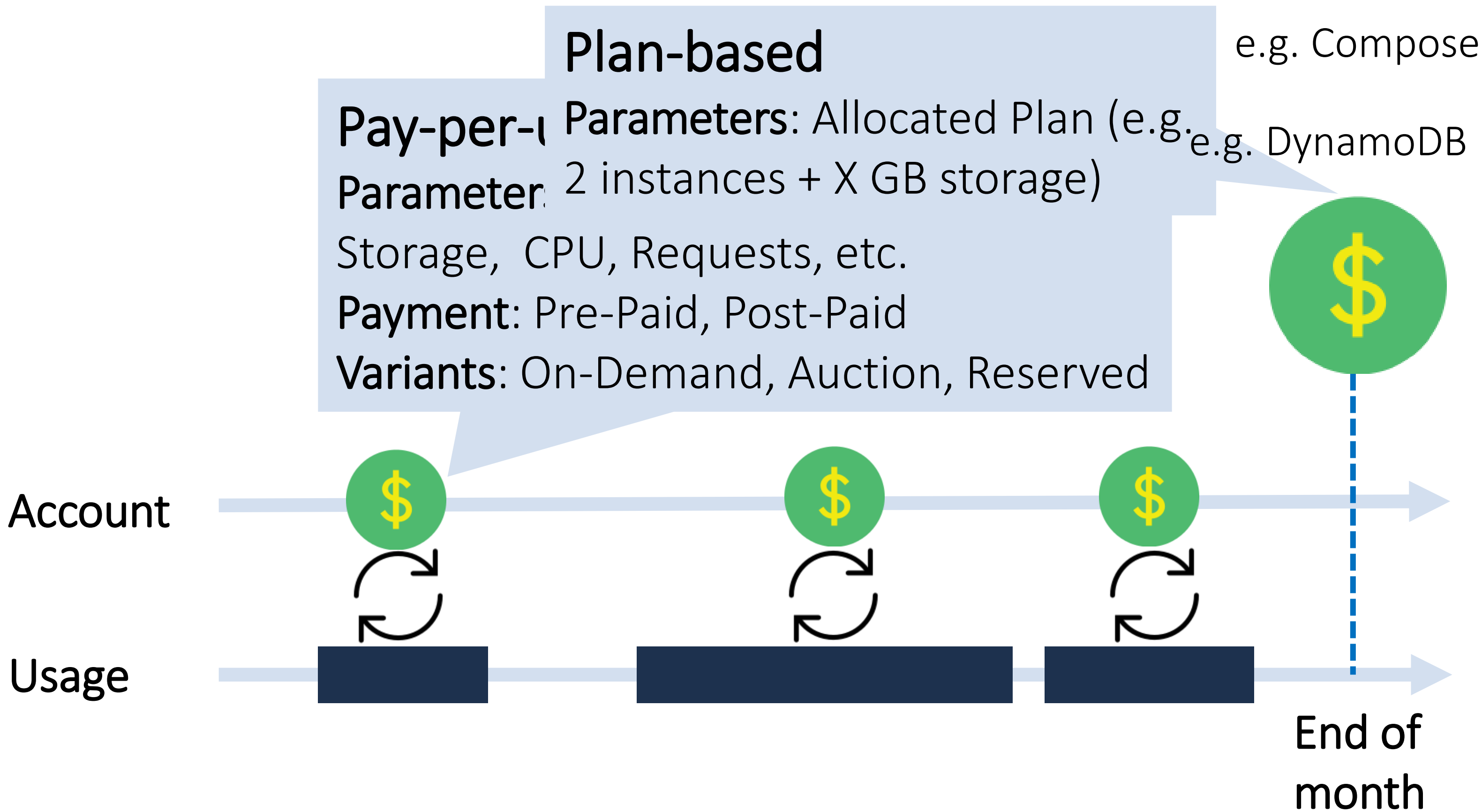
Backend-as-a-Service

DBaaS with embedded custom and predefined application logic



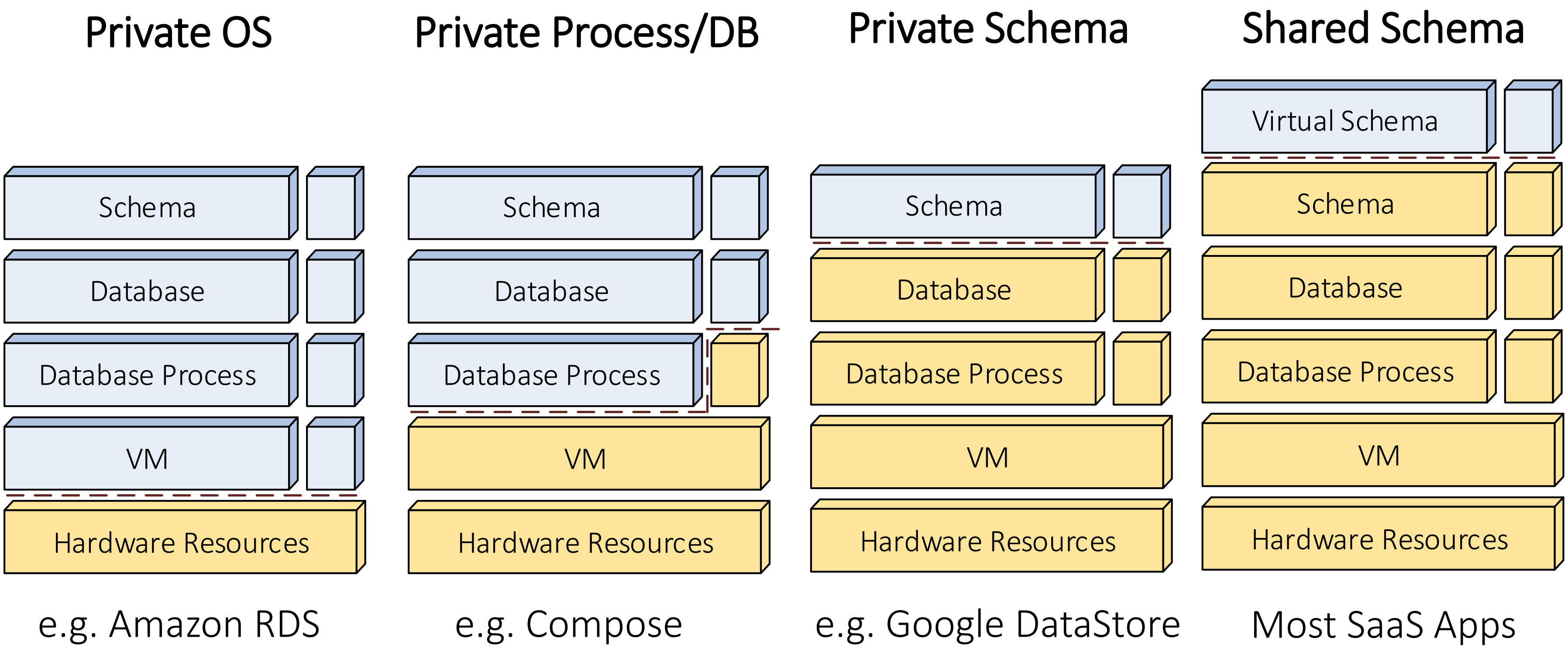
Pricing Models

Pay-per-use and plan-based



Database-as-a-Service

Approaches to Multi-Tenancy



T. Kiefer, W. Lehner "Private table database virtualization for dbaas"
UCC, 2011

Multi-Tenancy: Trade-Offs

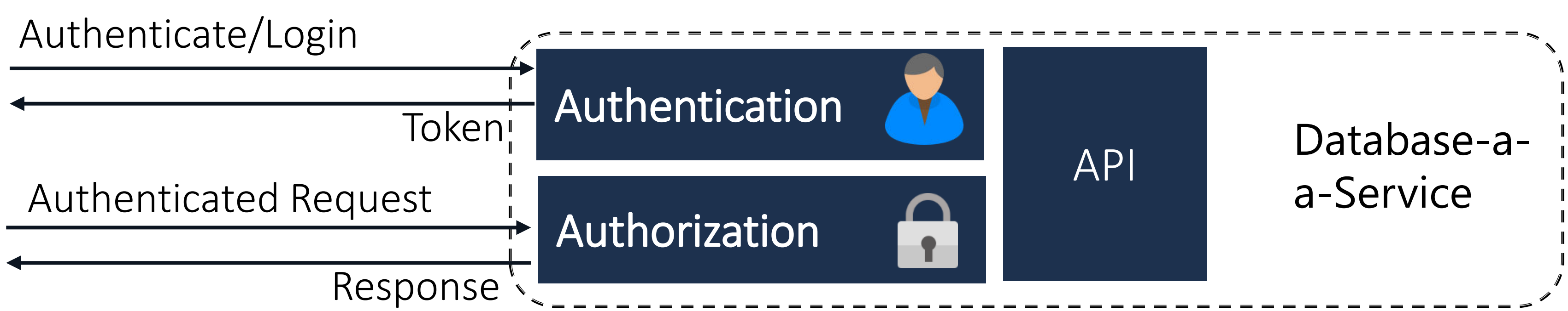
	App. indep.	Ressource Util.	Isolation	Maintenance, Provisioning
Private OS	✔	☆☆☆	★★★	★☆☆
Private Process/DB	✔	★☆☆	★★★	★☆☆
Private Schema	✔	★★☆	★★☆	★★★
Shared Schema	✘	★★★	★☆☆	★★★

W. Lehner, U. Sattler "Web-scale Data Management for the Cloud"
Springer, 2013

Authentication & Authorization

Checking Permissions and Indentity

Internal Schemes	External Identity Provider	Federated Identity (Single Sign On)
e.g. Amazon IAM	e.g. OpenID	e.g. SAML



User-based Access Control	Role-based Access Control	Policies
e.g. Amazon S3 ACLs	e.g. Amazon IAM	e.g. XACML

Service Level Agreements (SLAs)

Specification of Application/Tenant Requirements

SLA

<u>Technical Part</u>

1. SLO
2. SLO
3. SLO

<u>Legal Part</u>

1. Fees
2. Penalties

Service Level Objectives:

- Availability
- Durability
- Consistency/Staleness
- Query Response Time

Service Level Agreements

Expressing application requirements

Functional Service Level Objectives

- Guarantee a „feature“
- Determined by database system
- *Examples*: transactions, join

Non-Functional Service Level Objectives

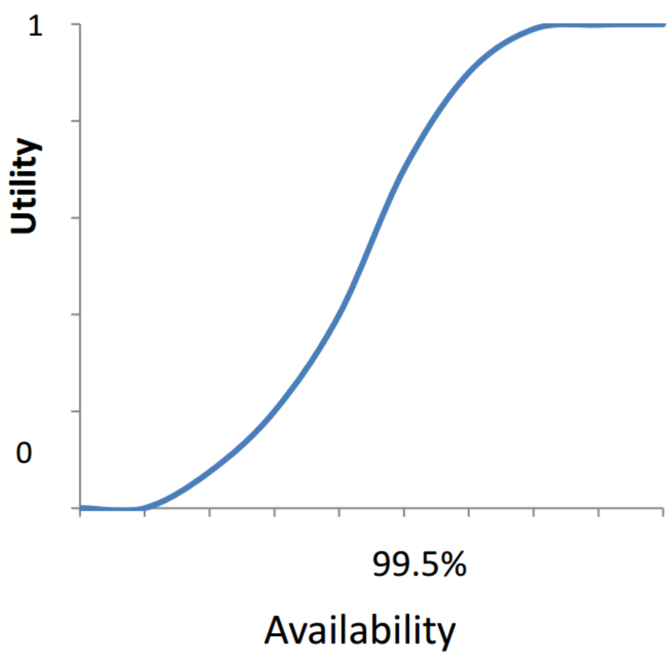
- Guarantee a certain *quality of service* (QoS)
- Determined by database system and service provider
- *Examples*:
  - **Continuous**: response time (latency), throughput
  - **Binary**: Elasticity, Read-your-writes

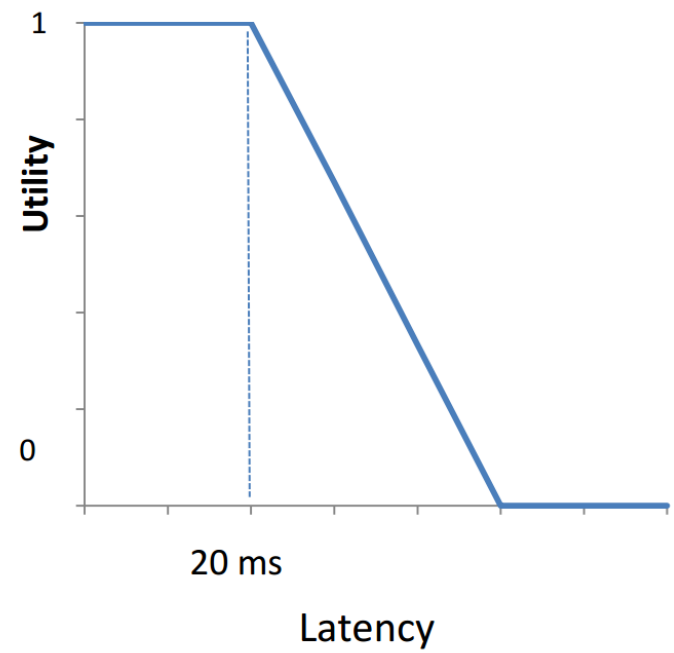
Service Level Objects

Making SLOs measurable through utilities

Utility expresses „value“ of a continuous non-functional requirement:

$$f_{utility}(metric) \rightarrow [0,1]$$



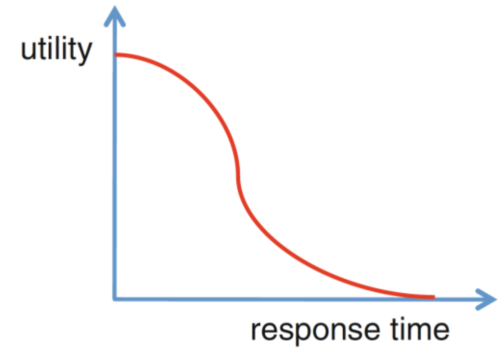


Workload Management

Guaranteeing SLAs

Typical approach:

Maximize:



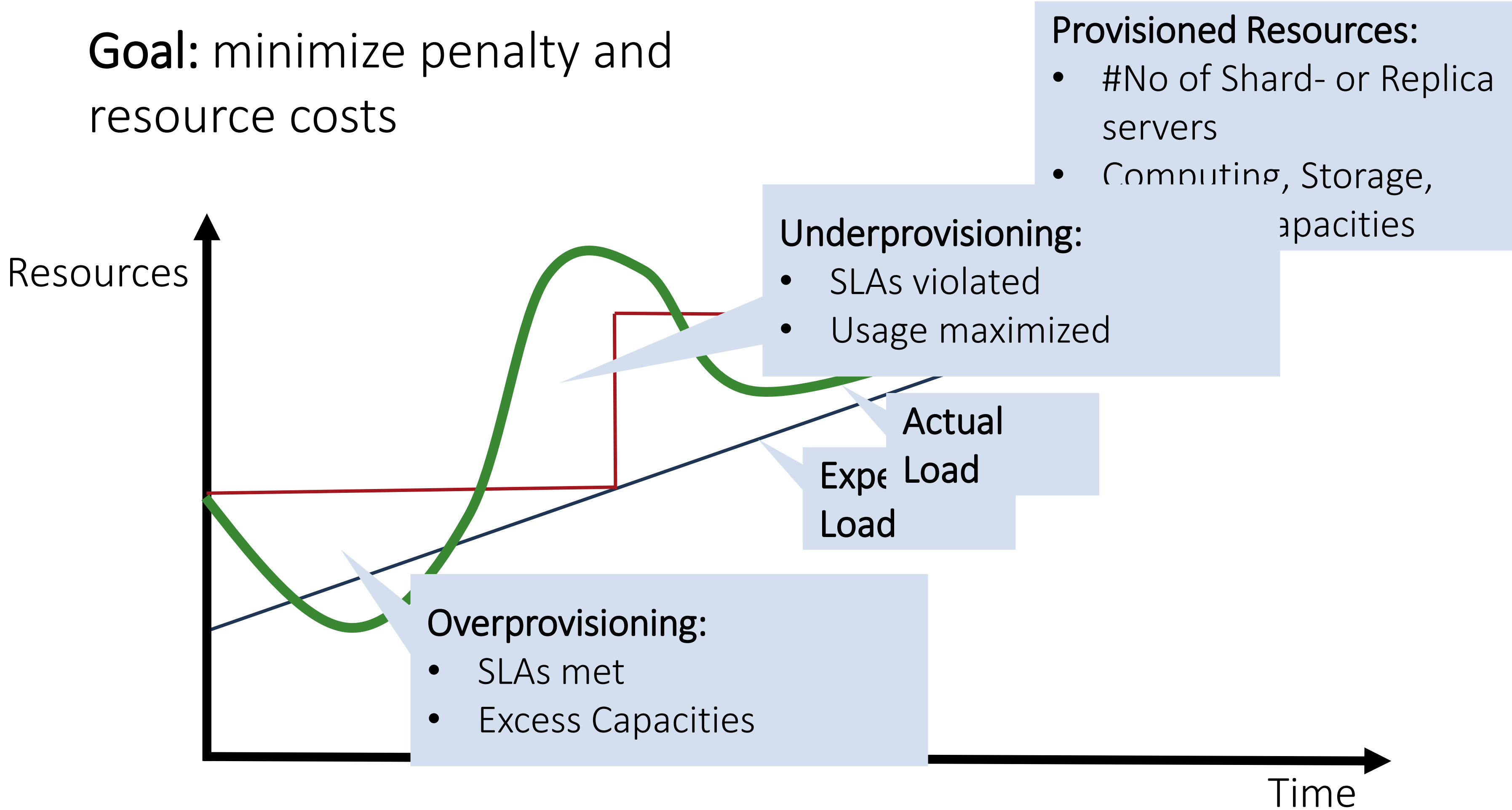
response time

W. Lehner, U. Sattler "Web-scale Data Management for the Cloud"
Springer, 2013

Resource & Capacity Planning

From a DBaaS provider's perspective

Goal: minimize penalty and resource costs



T. Lorido-Botran, J. Miguel-Alonso et al.: "Auto-scaling Techniques for Elastic Applications in Cloud Environments". Technical Report, 2013

SLAs in the wild

Most DBaaS systems offer no SLAs, or only a a simple uptime guarantee

	Model	CAP	SLAs
SimpleDB	Table-Store (*NoSQL Service*)	CP	✗
Dynamo-DB	Table-Store (*NoSQL Service*)	CP	✗
Azure Tables	Table-Store (*NoSQL Service*)	CP	99.9% uptime
AE/Cloud DataStore	Entity-Group Store (*NoSQL Service*)	CP	✗
S3, Az. Blob, GCS	Object-Store (*NoSQL Service*)	AP	99.9% uptime (S3)

Open Research Questions
in Cloud Data Management

- Service-Level Agreements
  - How can SLAs be guaranteed in a virtualized, multi-tenant cloud environment?
- Consistency
  - Which consistency guarantees can be provided in a geo-replicated system without sacrificing availability?
- Performance & Latency
  - How can a DBaaS deliver low latency in face of distributed storage and application tiers?
- Transactions
  - Can ACID transactions be aligned with NoSQL and scalability?

www.scdm2015.com

Location : **Santa Clara**

Submission Deadline: **August 30**

Outline

- Motivation
- **ORESTES: a Cloud-Database Middleware**
- Solving Latency and Polyglot Storage
- Wrap-up

- Two problems:
  - Latency
  - Polyglot Storage
- Vision: Orestes Middleware

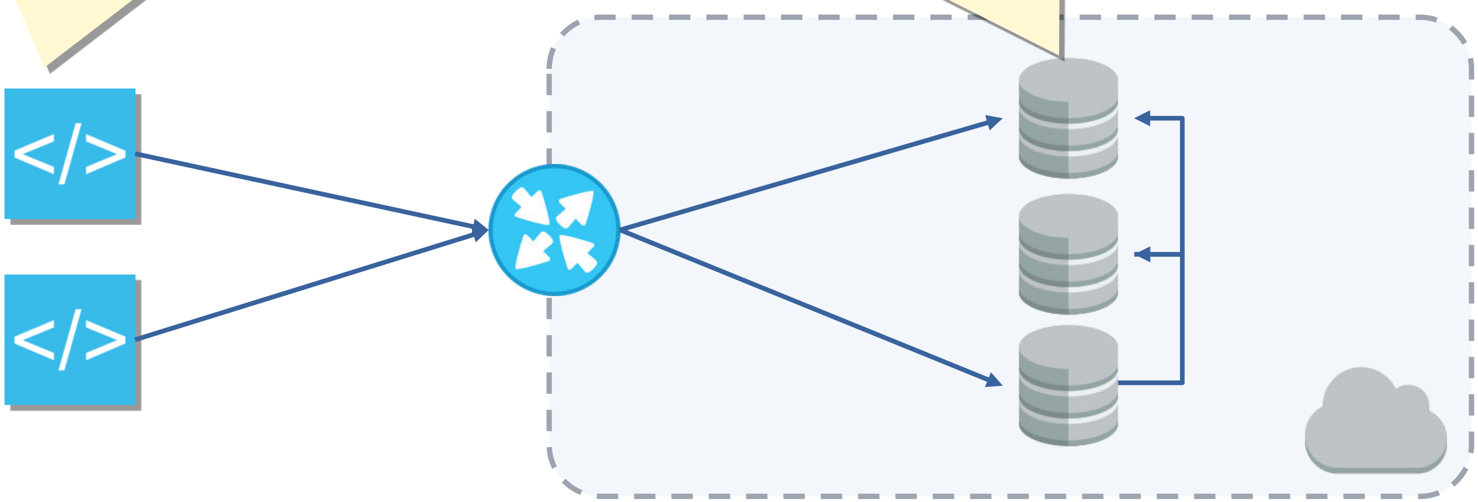
Latency & Polyglot Storage

Two central problems

- Goal of ORESTES: Solve both problems through a scalable **cloud-database middleware**

If the application is *geographically distributed*, how can we guarantee **fast database access**?

If one size *doesn't* fit all – how can **polyglot persistence** be leveraged on a declarative, automated basis?



Problem I: Latency

100ms 400ms 500ms 1s

Average: 9,3s

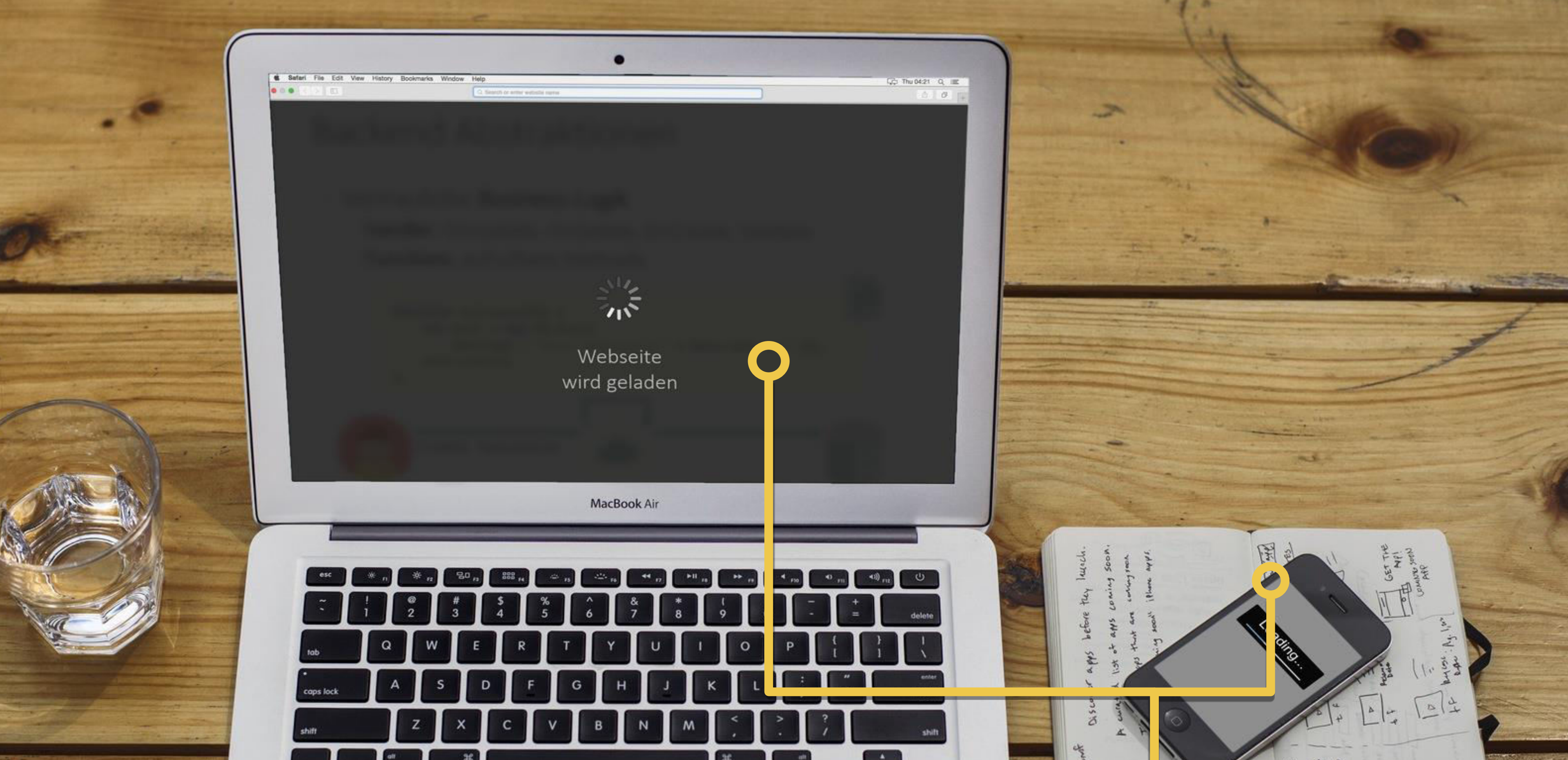
Loading…

Conversions

% Traffic

% Visitors

% Revenue

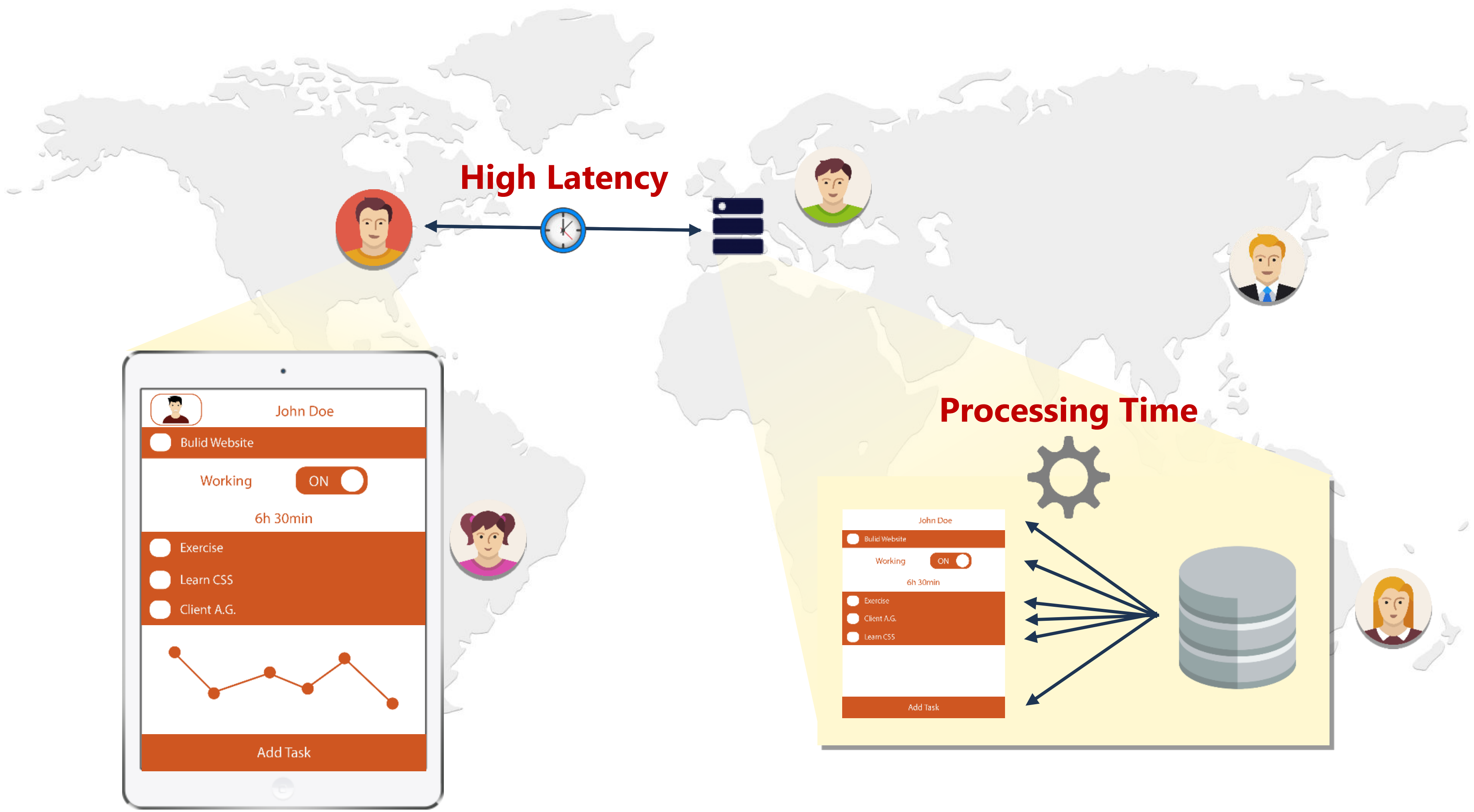


If perceived speed is such an import factor

...what causes slow page load times?

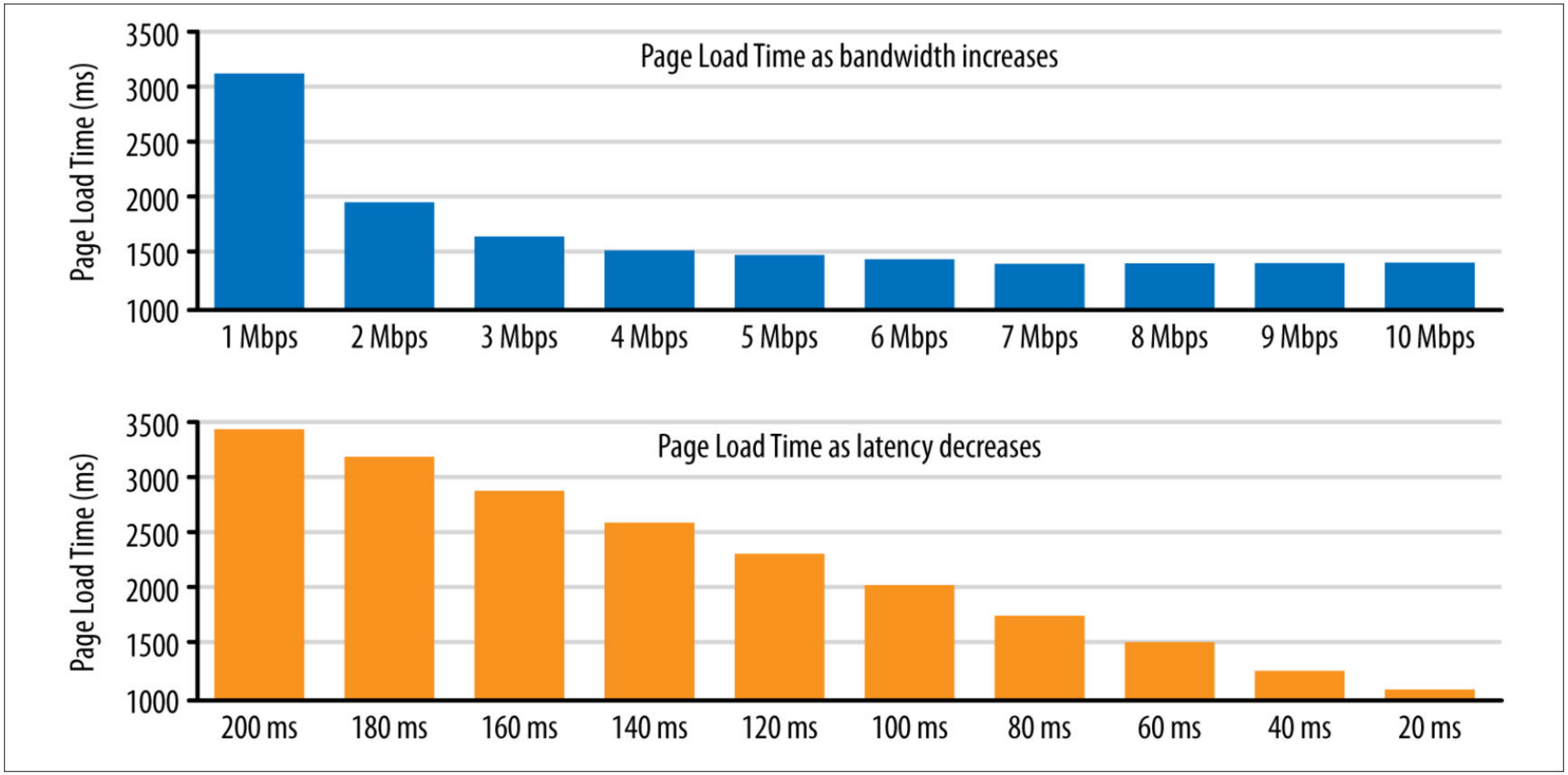
State of the art

Two bottlenecks: latency und processing



Network Latency

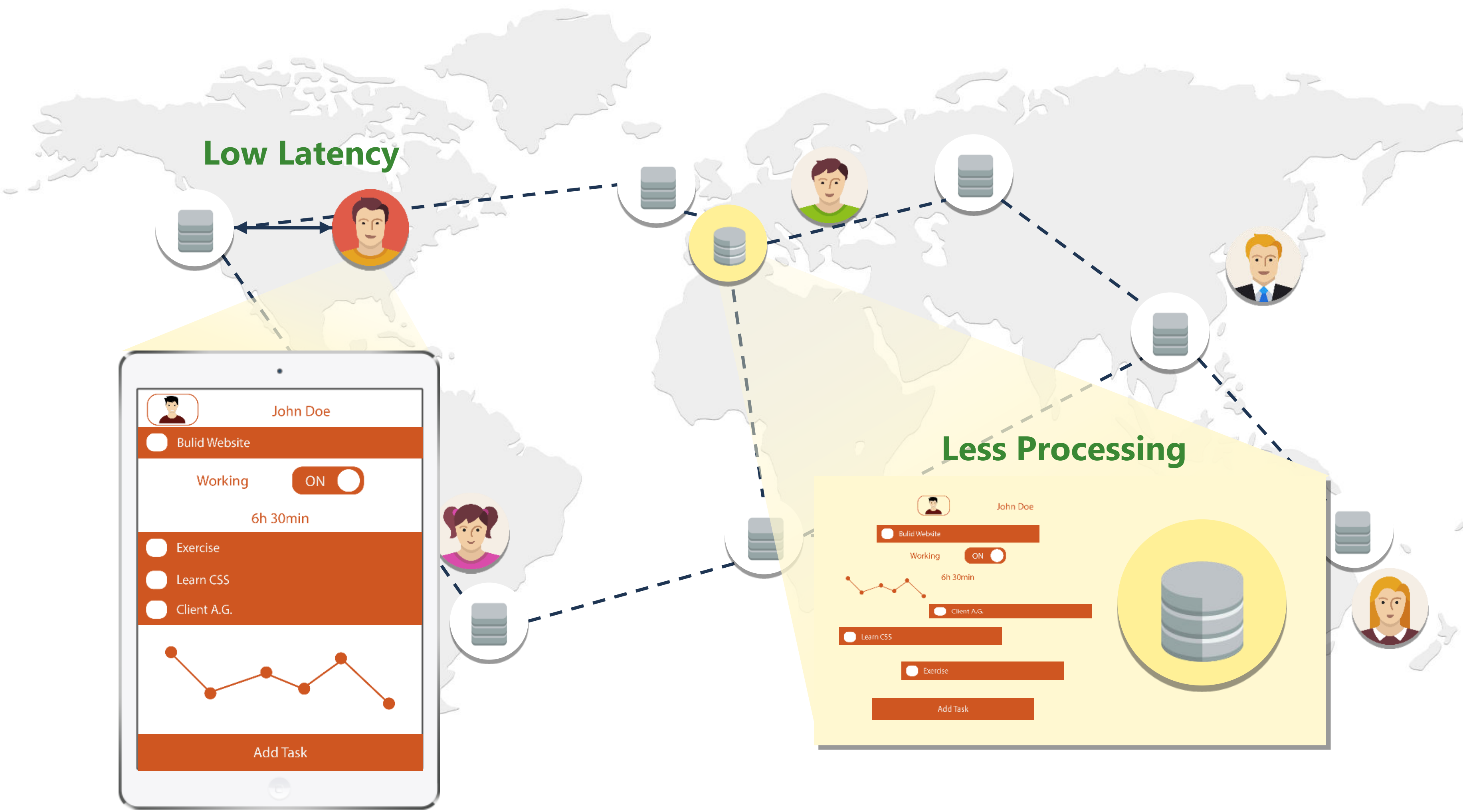
The underlying problem of high page load times



I. Grigorik, High performance browser networking. O'Reilly Media, 2013.

The low-latency vision

Data is served by ubiquitous web-caches



The web's caching model

Staleness as a consequence of scalability

Expiration-based
Every object has a defined Time-To-Live (TTL)

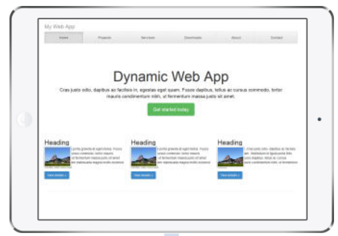
Revalidations
Allow clients and caches to check freshness at the server

Stale Data

Research Question:
Can database services leverage the web caching infrastructure for low latency with rich **consistency guarantees**?

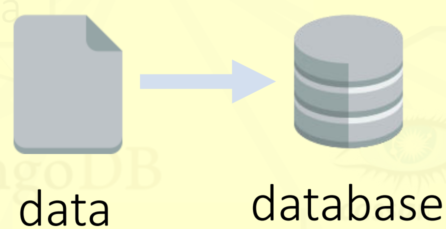
Problem II: Polyglot Persistence

Current best practice



Research Question:

Can we automate the

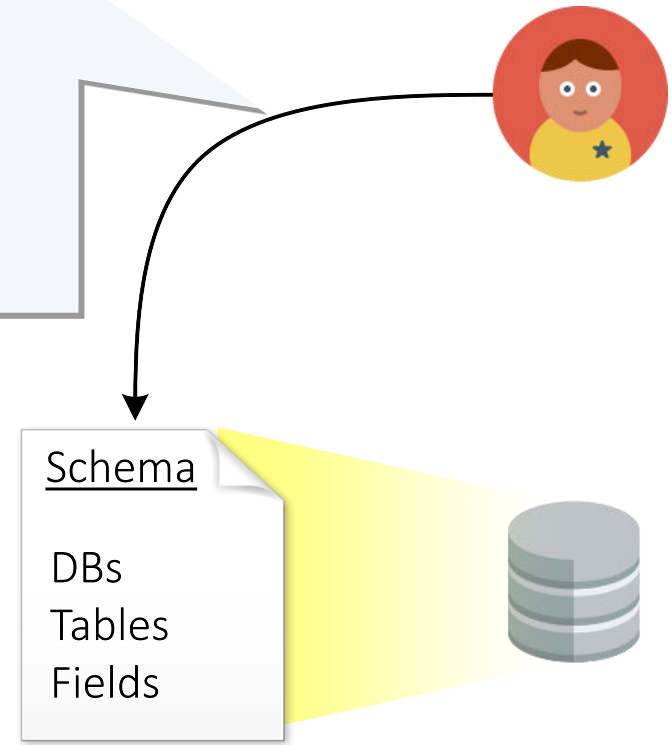


mapping problem?

Vision

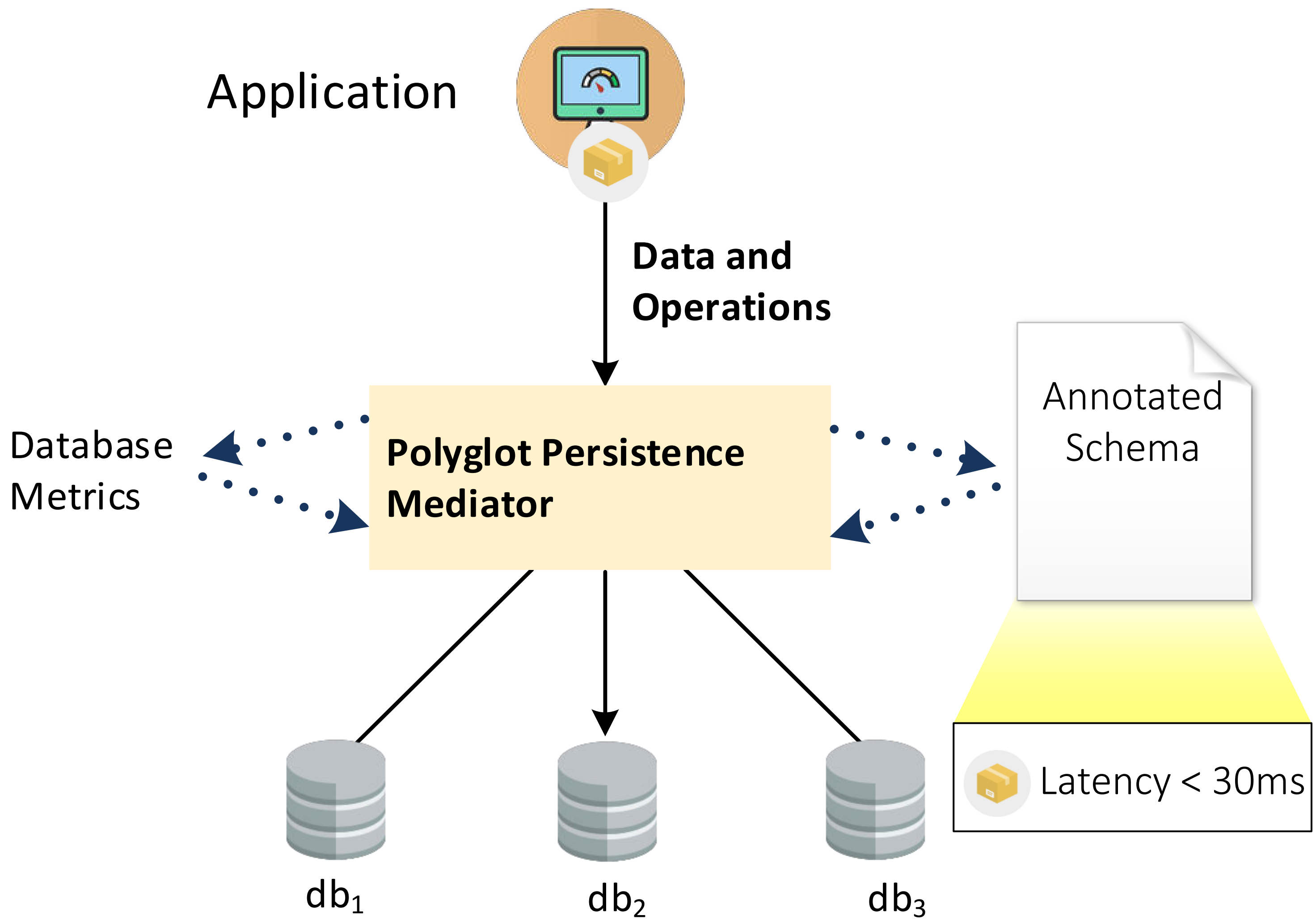
Schemas can be annotated with requirements

- Write Throughput > 10,000 RPS
- Read Availability > 99.9999%
- Scans = true
- Full-Text-Search = true
- Monotonic Read = true



Vision

The Polyglot Persistence Mediator chooses the database



The Big Picture

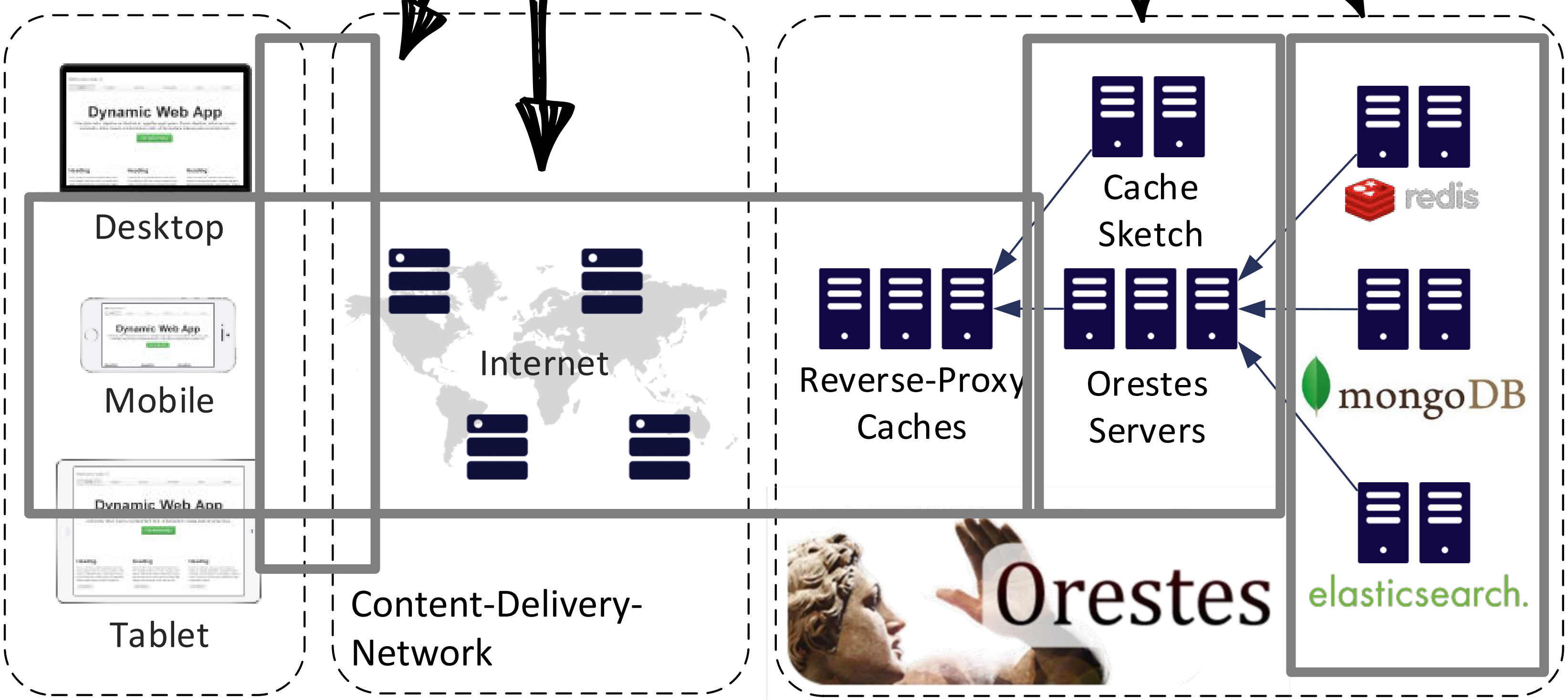
Implementation in ORESTES

- Polyglot Storage and Low Latency are the central goals of ORESTES

Database-as-a-Service Middleware:
Caching, Transactions, Schemas, Polyglot Storage,
Authorization, Multi-Tenancy

Unified REST API

Standard HTTP Caching



Outline

- Motivation
- ORESTES: a Cloud-Database Middleware
- **Solving Latency and Polyglot Storage**
- Wrap-up

- Cache Sketch Approach
  - Caching Arbitrary Data
  - Predicting TTLs
- Polyglot Persistence Mediator
  - SLA-Approach
  - Database Selection

Web Caching Concepts

Invalidation- and expiration-based caches

Expiration-based Caches:

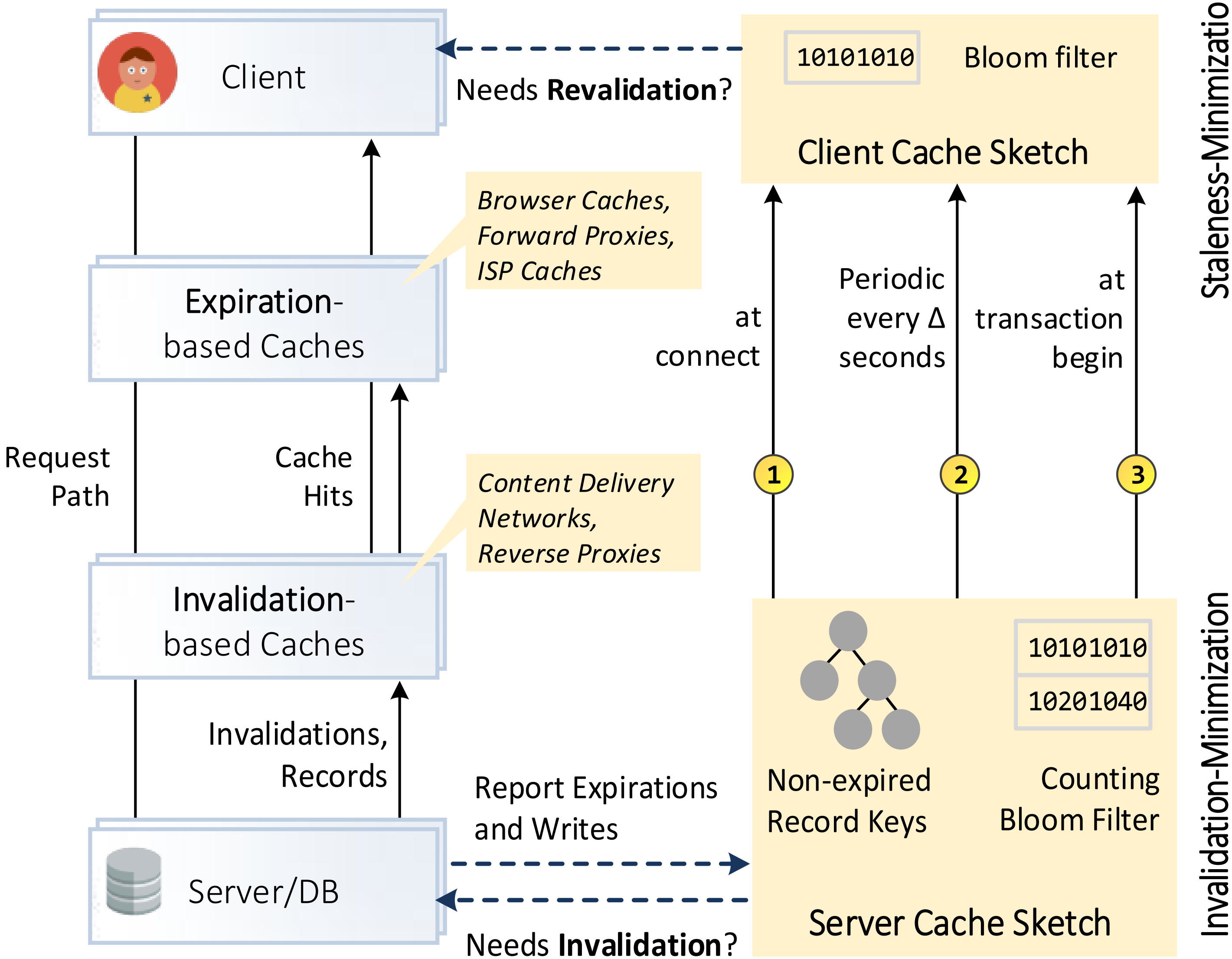
- An object x is considered fresh for TTL_x seconds
- The server assigns TTLs for each object

Invalidation-based Caches:

- Expose object eviction operation to the server

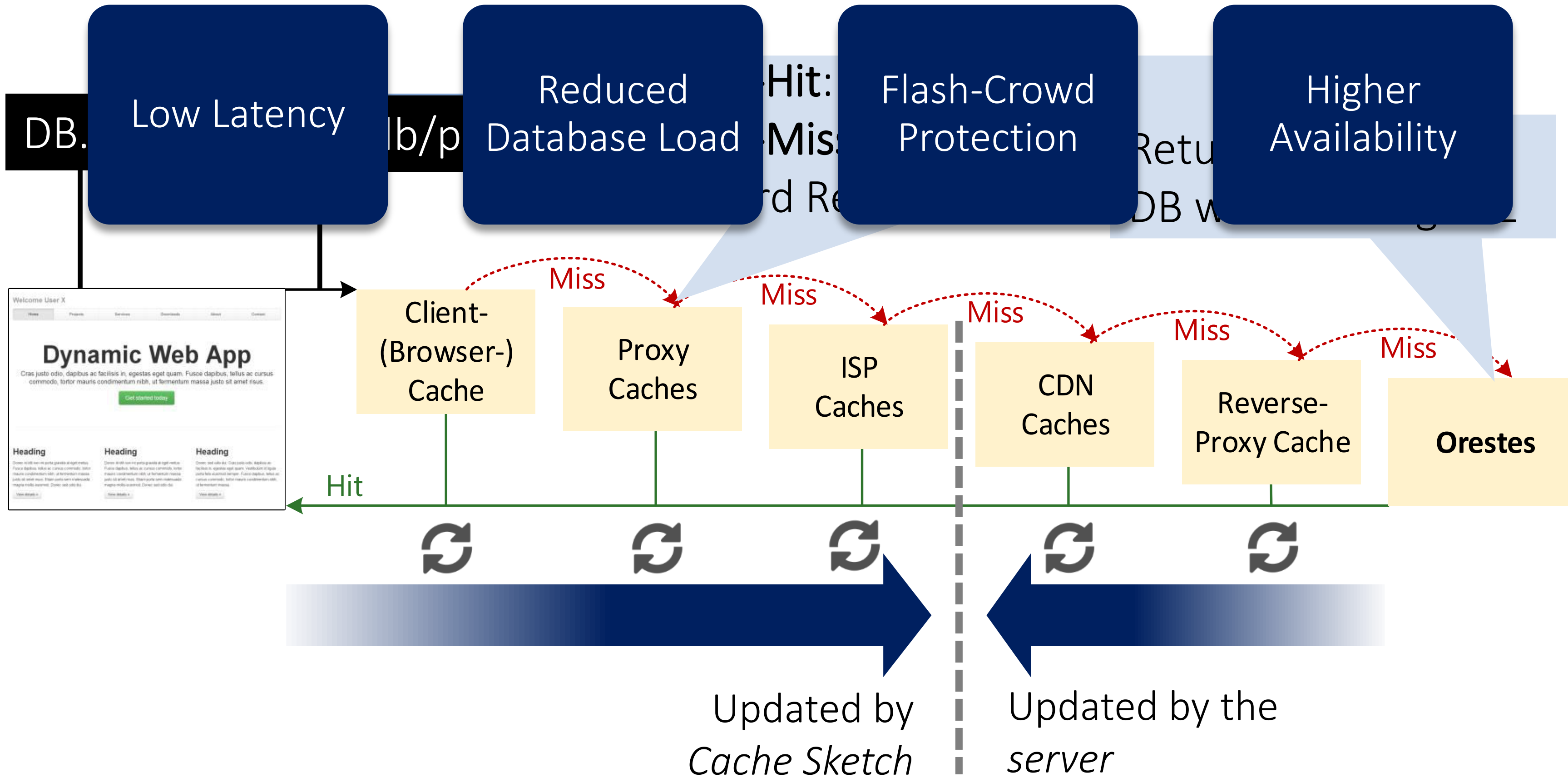
The Cache Sketch approach

Letting the client handle cache coherence



The End-to-End Path of Requests

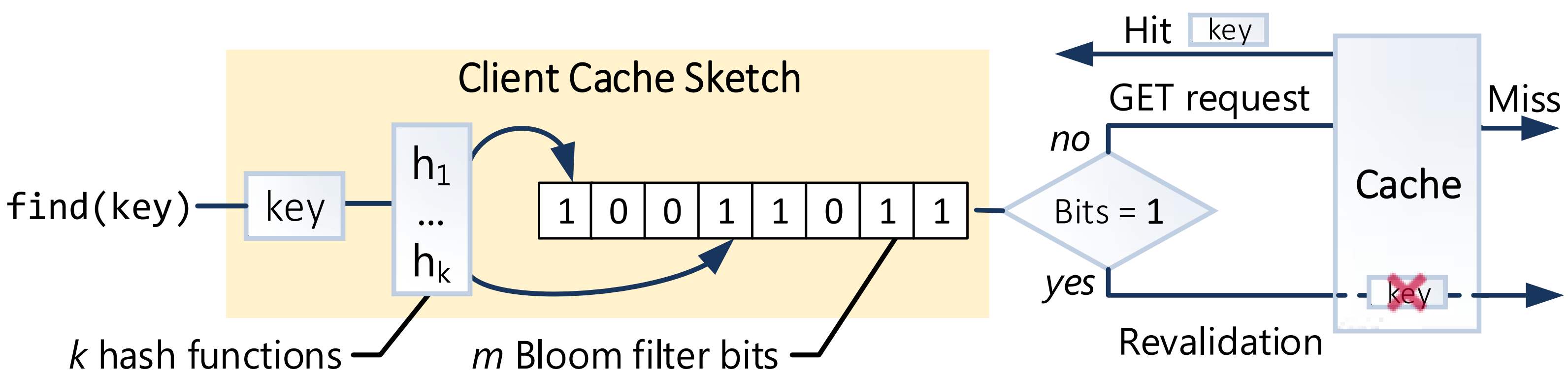
The Caching Hierarchy



The Client Cache Sketch

- *Let* $\boldsymbol{c_t}$ *be the client Cache Sketch generated at time* $\boldsymbol{t}$*, containing the key* $\boldsymbol{key_x}$ *of every record* $\boldsymbol{x}$ *that was <u>written before it expired</u> in all caches, i.e. every* $\boldsymbol{x}$ *for which holds:*

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



1 Slow initial page loads

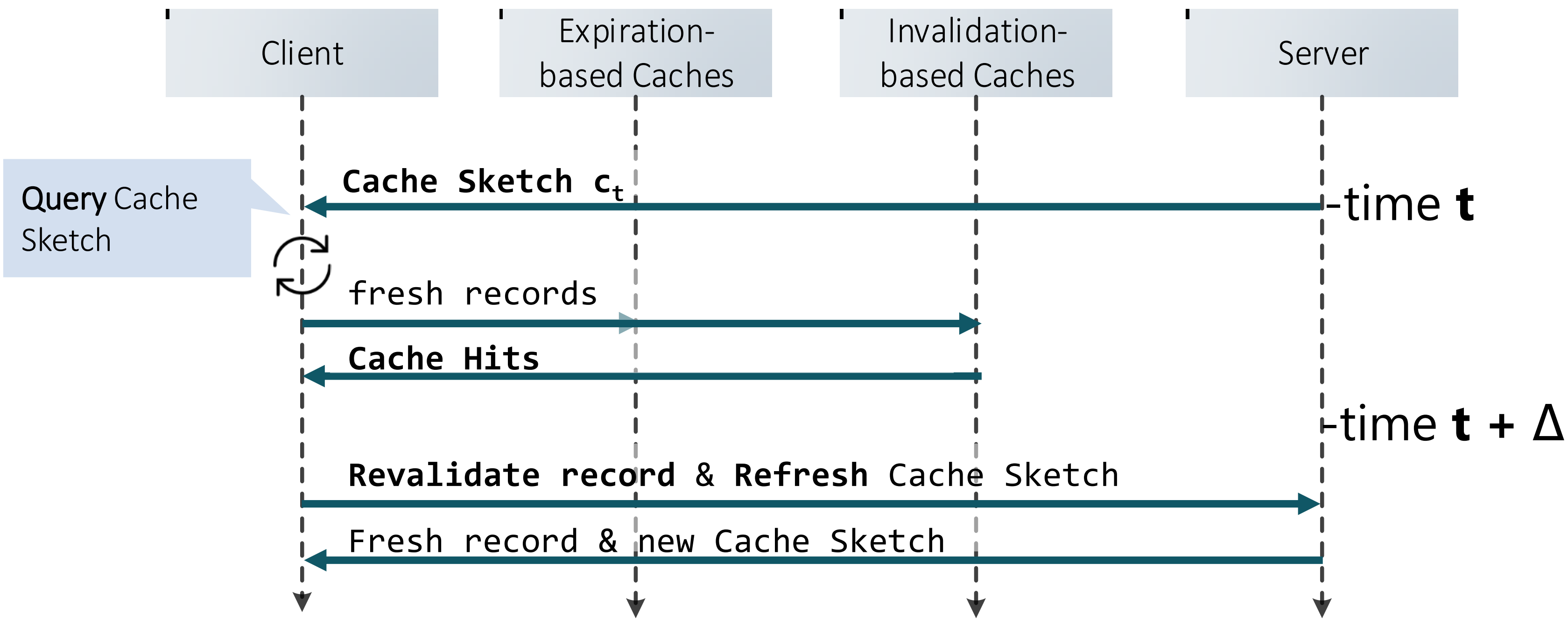
- Solution: **Cached Initialization**
  - Clients load the Cache Sketch at connection
  - Every non-stale cached record can be reused without degraded consistency

l)

2 Slow CRUD performance

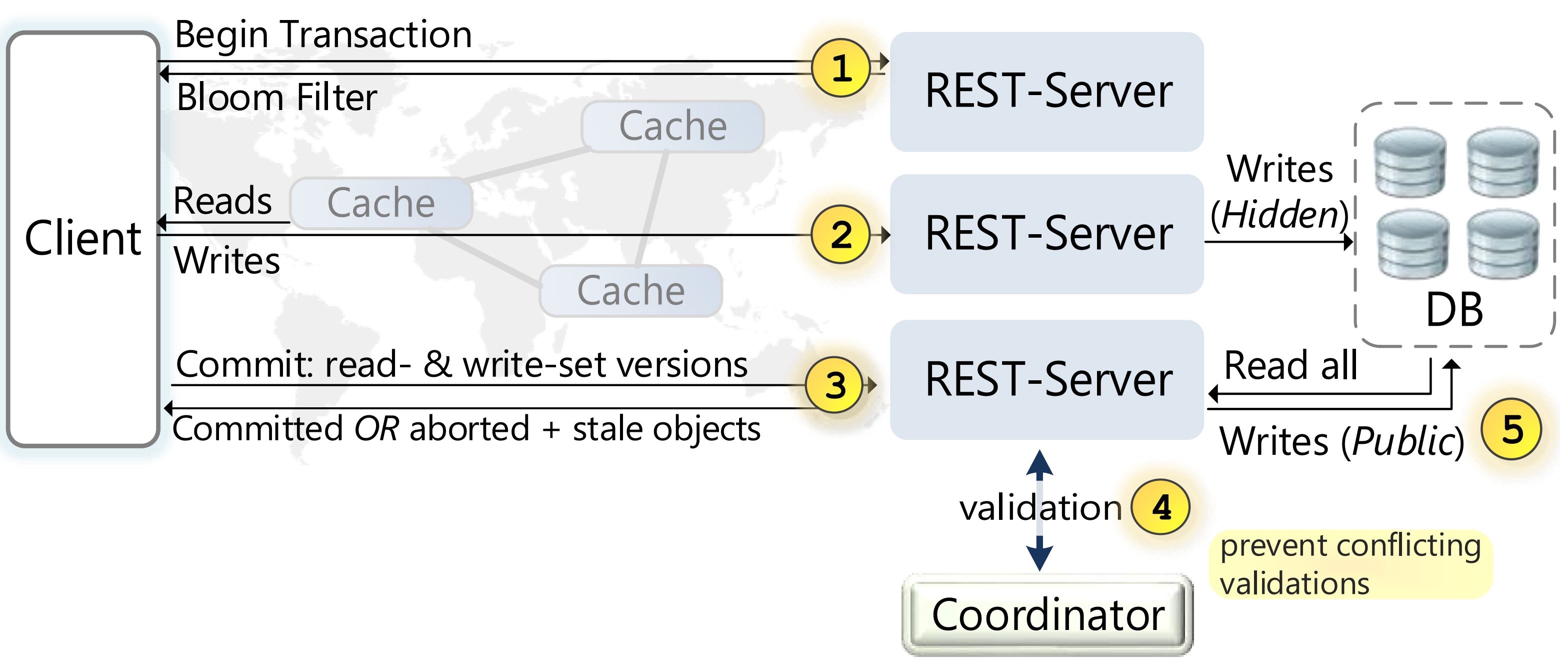
- Solution: **Δ-Bounded Staleness**
  - Clients refresh the Cache Sketch so its age never exceeds Δ

→ *Consistency guarantee*: Δ-atomicity



3 High Abort Rates in OCC

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



TTL Estimation

Determining the cache expiration

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

Shorter TTLs

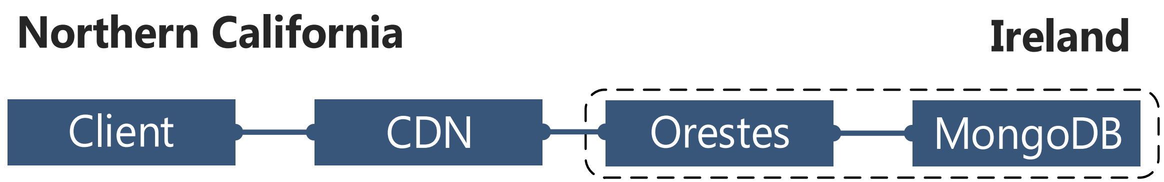
- less invalidations
- less stale reads

Longer TTLs

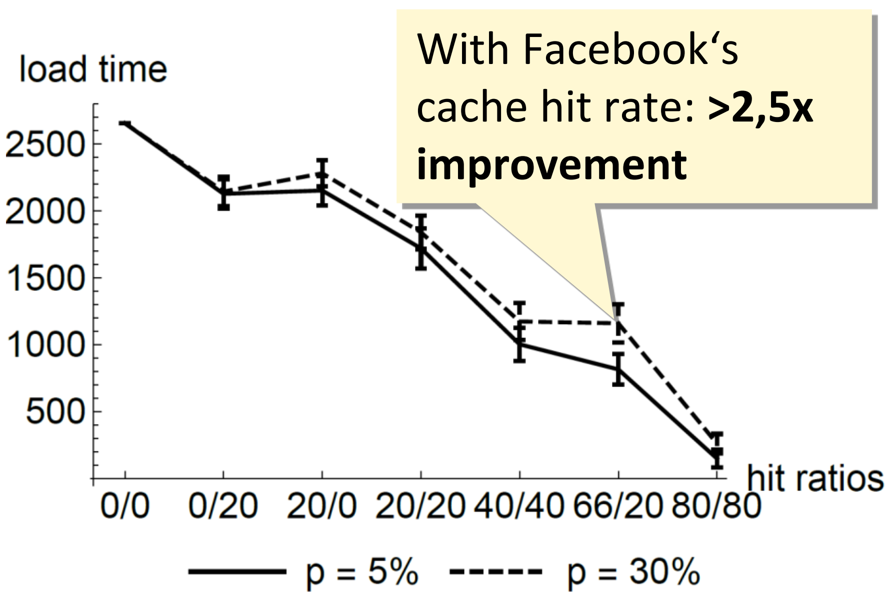
- Higher cache-hit rates
- more invalidations

Performance

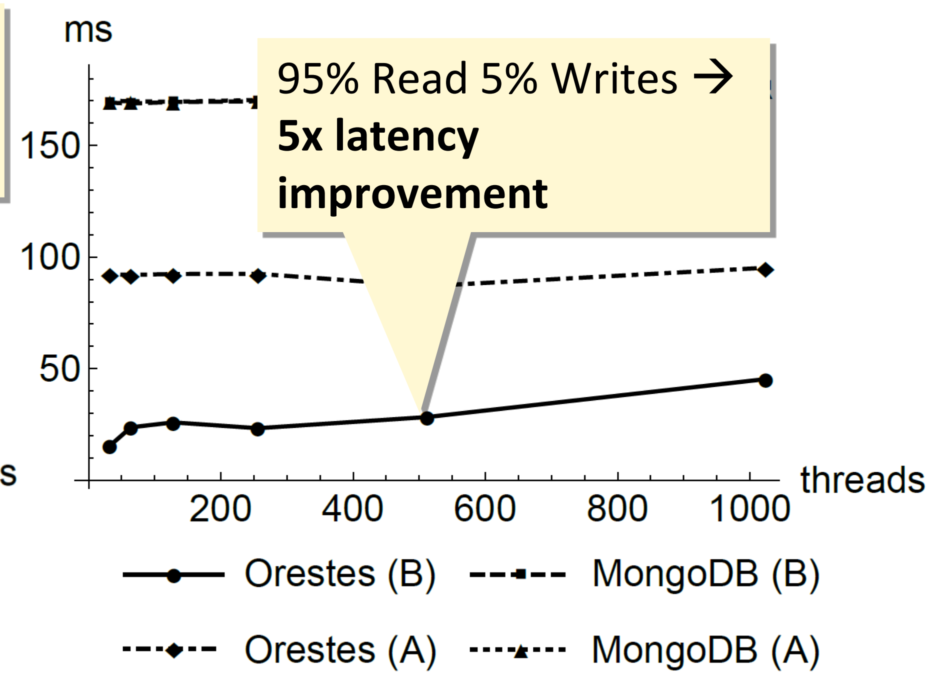
Setup:



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



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

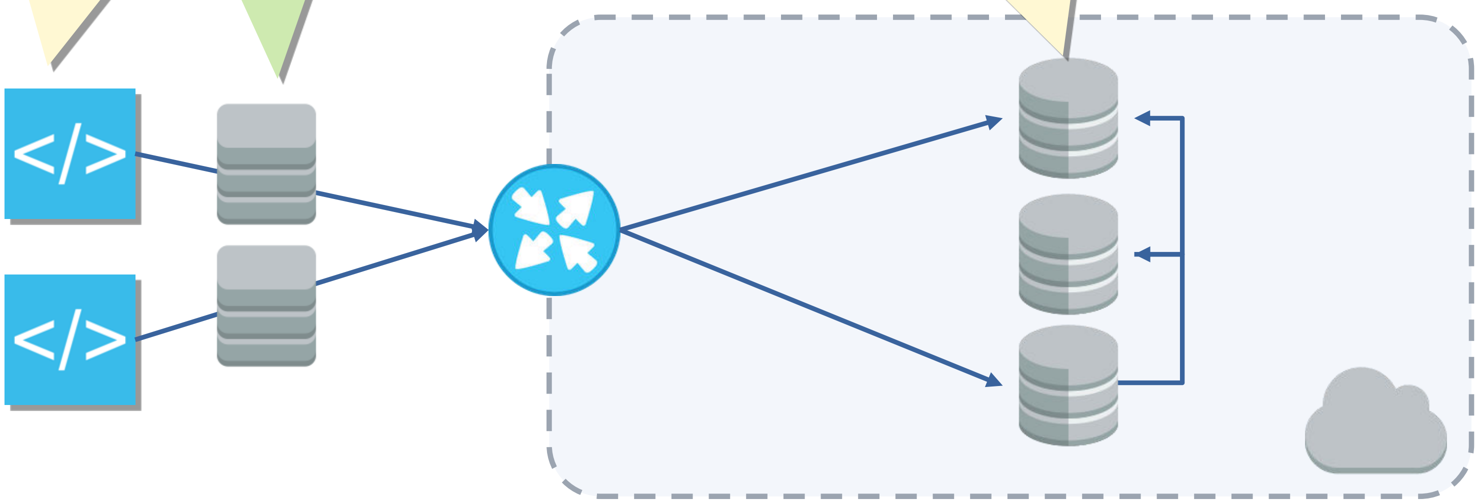


Low Latency

If the appl
distributed
fast datab

Transparent **end-to-end caching** using the Cache Sketch.

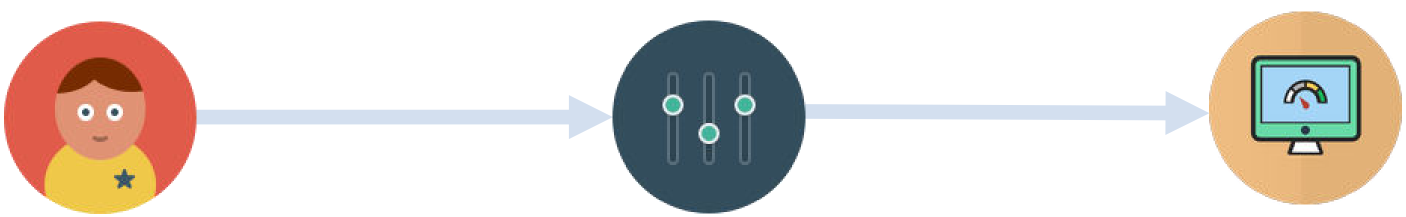
If one size *doesn't* fit all – how can **polyglot persistence** be leveraged on a declarative, automated basis?



Towards Automated Polyglot Persistence

Necessary steps

- **Goal**:
  - Extend classic workload management to *polyglot persistence*
  - Leverage hetereogeneous (NoSQL) databases



1. Requirements

Tenant specifies requirements as Service-Level-Agreements

2. Resolution

Find or provision a suitable combination of databases

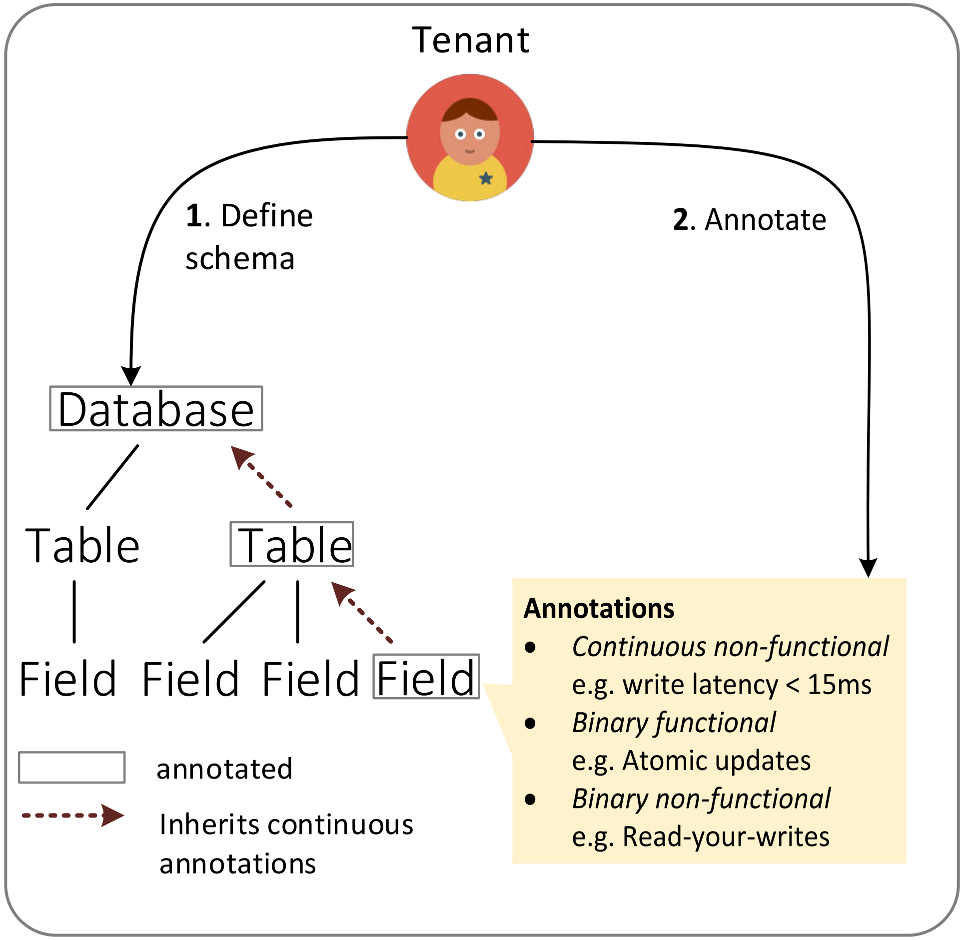
3. Mediation

Mediate data and database operations

Step I - Requirements

Expressing the application's needs

Annotation	Type	Annotated at
Read Availability	Continuous	*
Write Availability	Continuous	*
Read Latency	Continuous	*
Write Latency	Continuous	*
Write Throughput	Continuous	*
Data Vol. Scalability	Non-Functional	Field/Class/DB
Write Scalability	Non-Functional	Field/Class/DB
Read Scalabilty	Non-Functional	Field/Class/DB
Elasticity	Non-Functional	Field/Class/DB
Durability	Non-Functional	Field/Class/DB
Replicated	Non-Functional	Field/Class/DB
Linearizability	Non-Functional	Field/Class
Read-your-Writes	Non-Functional	Field/Class
Causal Consistency	Non-Functional	Field/Class
Writes follow reads	Non-Functional	Field/Class
Monotonic Read	Non-Functional	Field/Class
Monotonic Write	Non-Functional	Field/Class
Scans	Functional	Field
Sorting	Functional	Field
Range Queries	Functional	Field
Point Lookups	Functional	Field
ACID Transactions	Functional	Class/DB
Conditional Updates	Functional	Field
Joins	Functional	Class/DB
Analytics Integration	Functional	Field/Class/DB
Fulltext Search	Functional	Field
Atomic Updates	Functional	Field/Class

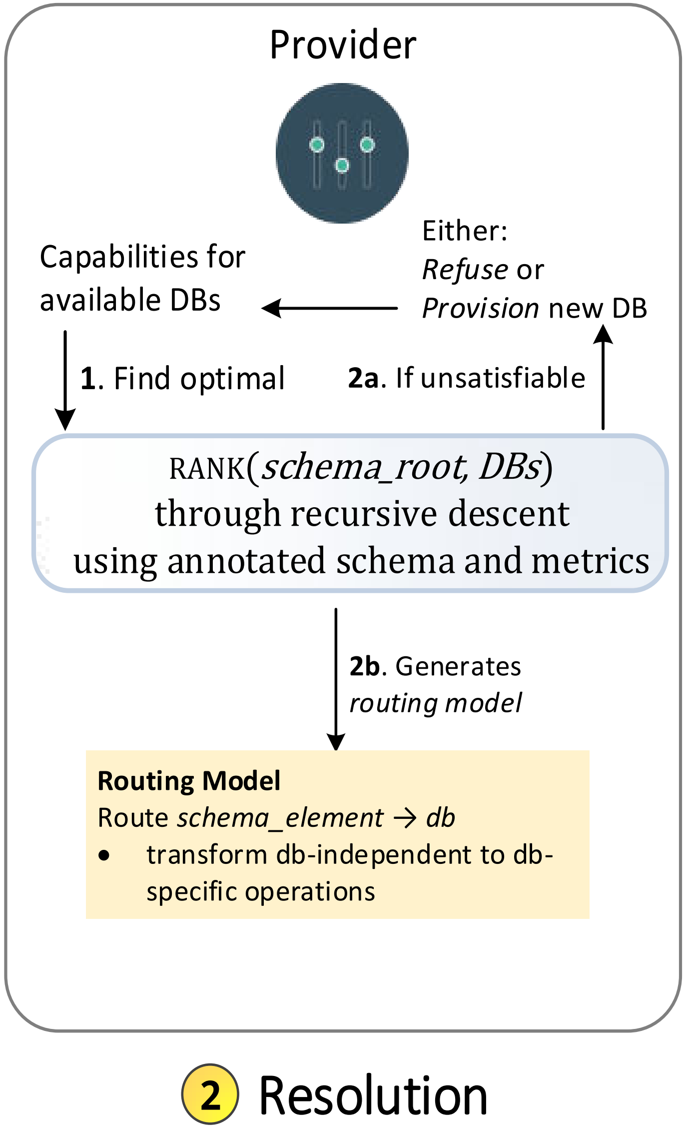


1 Requirements

Step II - Resolution

Finding the best database

- The Provider resolves the requirements
- RANK: scores available database systems
- **Routing Model**: defines the optimal mapping from schema elements to databases



Step II - Resolution

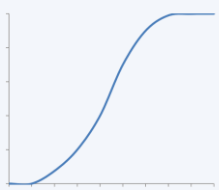
Ranking algorithm by example

DBs = { *MongoDB, Riak, Cassandra, CouchDB, Redis, MySQL, S3, Hbase* }

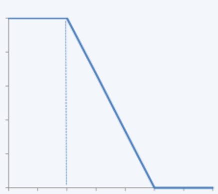
Annotations

Lineariza-
bility

Availability



Read latency



Schema

ECommerceDB
database

Customers
Table

ShoppingBasket
List<String>

String

Rank Algorithm

DBs = { *MongoDB, ~~Riak~~, ~~Cassandra~~, ~~CouchDB~~, Redis, MySQL, ~~S3~~, Hbase* }

Databa		
Mongo		
Redis	95%→0.05	1ms→1
MySQL	94%→ 0.04	40ms→0.2
HBase	99.9%→0.9	50ms→0.1

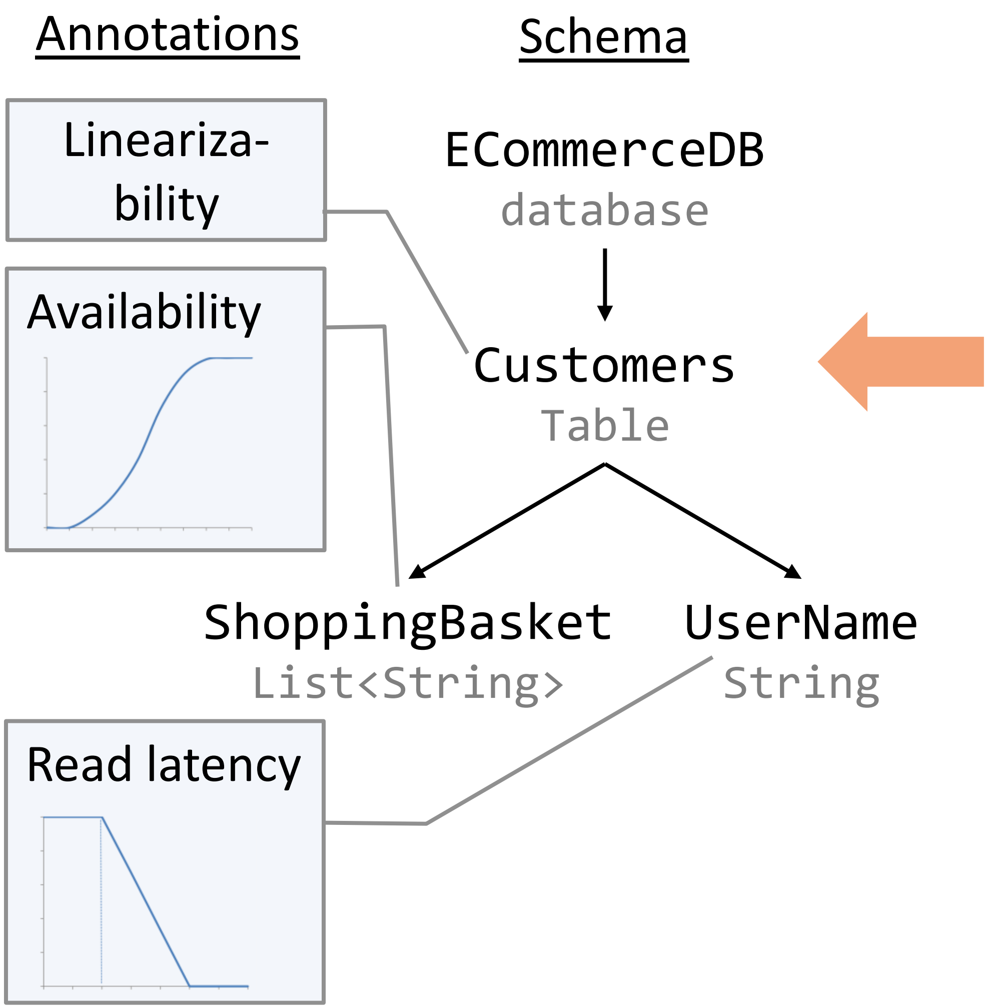
2. Recursive descent

Continuous requirement →
∀ databases calculate

$$db \rightarrow f_{utility}(db.availability)$$

Step II - Resolution

Ranking algorithm by example



DB	Score
MongoDB	0.9
Redis	0.525
MySQL	0.12
HBase	0.5

Binary requirement →

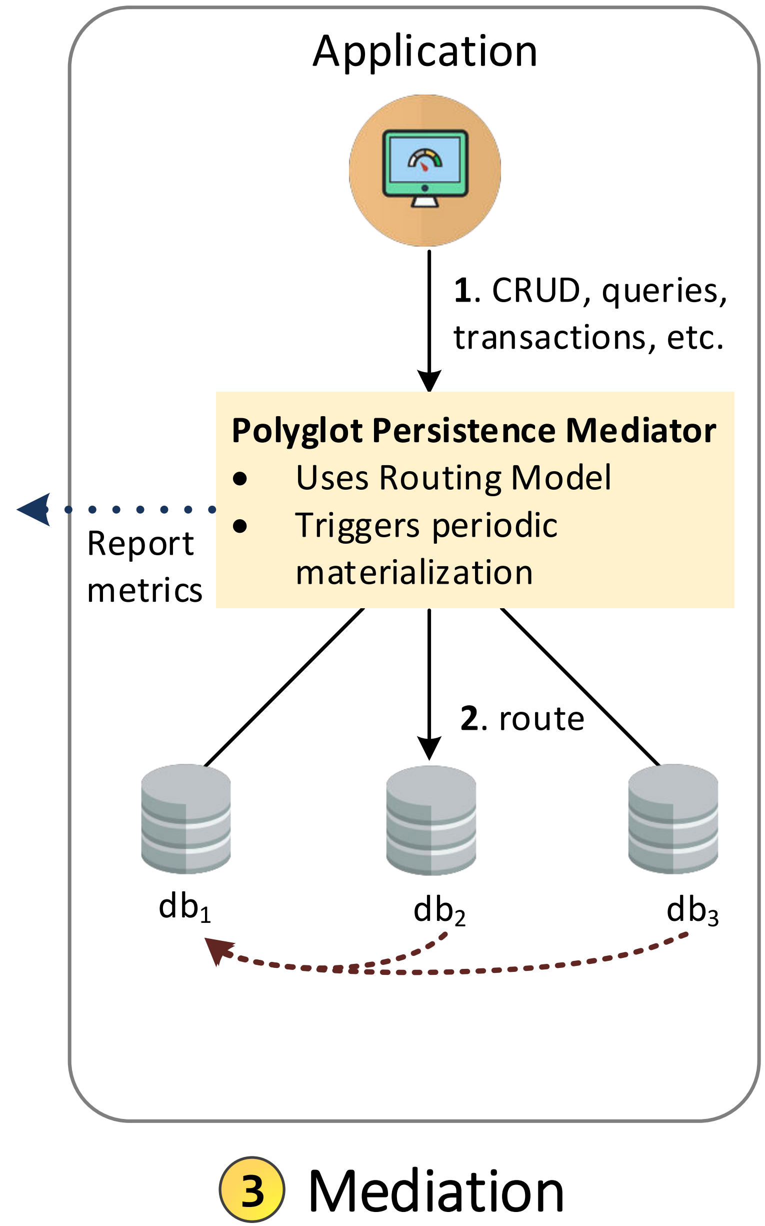
1. Exclude DBs that do not support it
2. Recursive descent
3. Pick DB with best total score and add it to routing model

Routing Model:
Customers → MongoDB

Step III - Mediation

Routing data and operations

- The PPM routes data
- **Operation Rewriting:** translates from abstract to database-specific operations
- **Runtime Metrics**: Latency, availability, etc. are reported to the resolver
- **Primary Database Option**: All data periodically gets materialized to designated database



Evaluation: News Article

Prototype of Polyglot Persistence Mediator in ORESTES

Scenario: news articles with impression counts
Objectives: low-latency top-k queries, high-throughput counts, article-queries

Evaluation: News Article

Prototype built on ORESTES

Scenario: news articles with impression counts
Objectives: low-latency top-k queries, high-throughput counts, article-queries

Counter updates kill performance

Evaluation: News Article

Prototype built on ORESTES

Scenario: news articles with impression counts

Objectives: low-latency top-k queries, high-throughput counts, article-queries

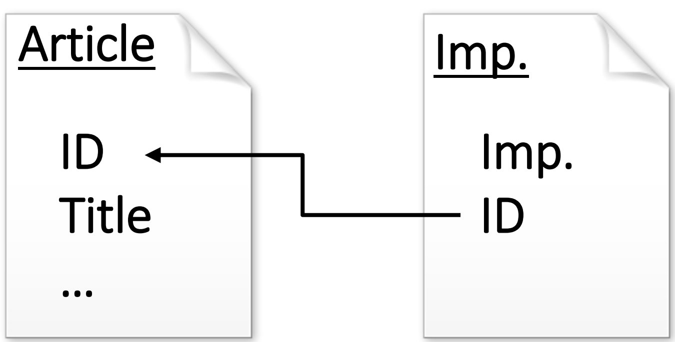
No powerful queries

Evaluation: News Article

Prototype built on ORESTES

Scenario: news articles with impression counts

Objectives: low-latency top-k queries, high-throughput counts, article-queries



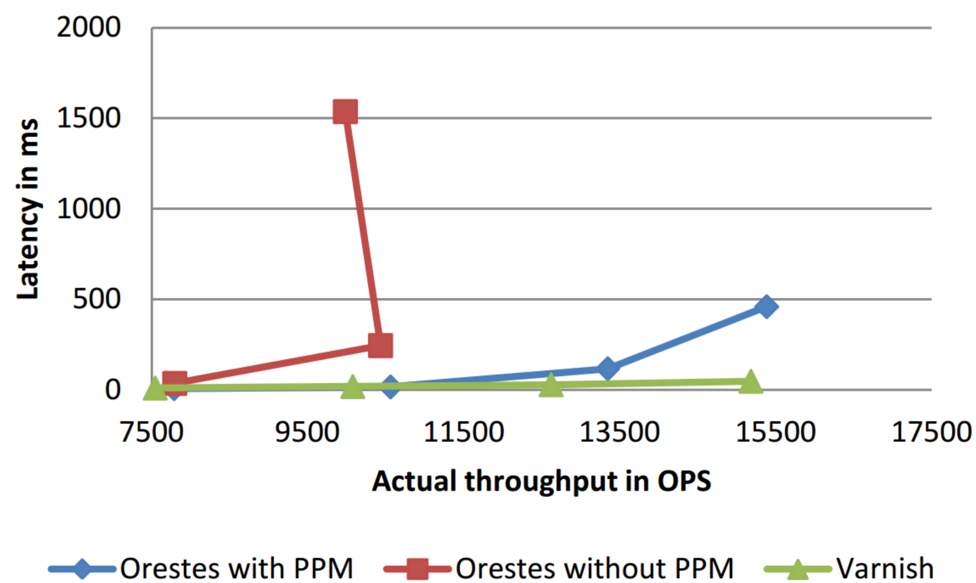
Document

Sorted Set

mongoDB

redis

Found Resolution

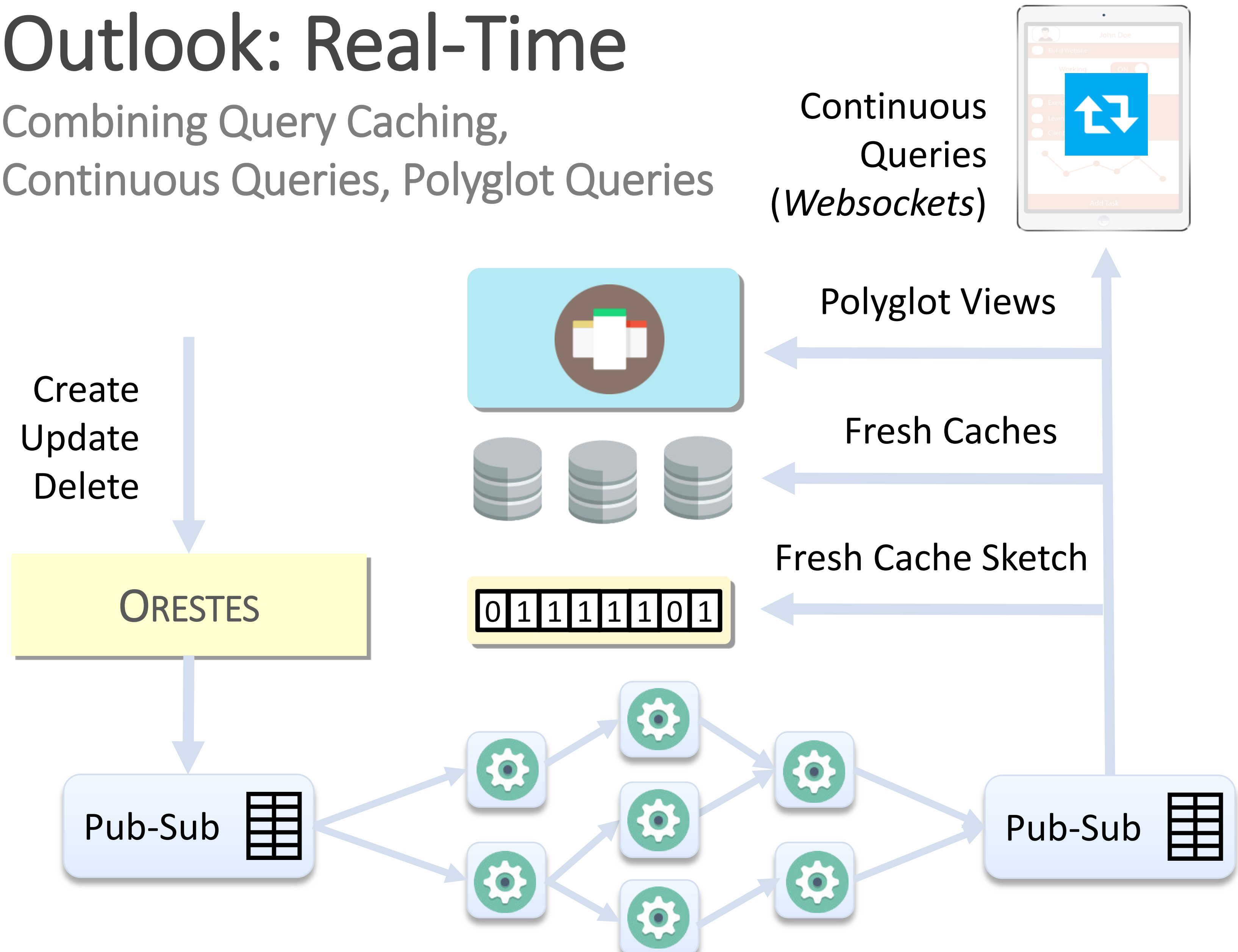


Outline

- Motivation
- ORESTES: a Cloud-Database Middleware
- Solving Latency and Polyglot Storage
- **Wrap-up**
  - Current/Future Work
  - Summary
  - Putting ORESTES into practice

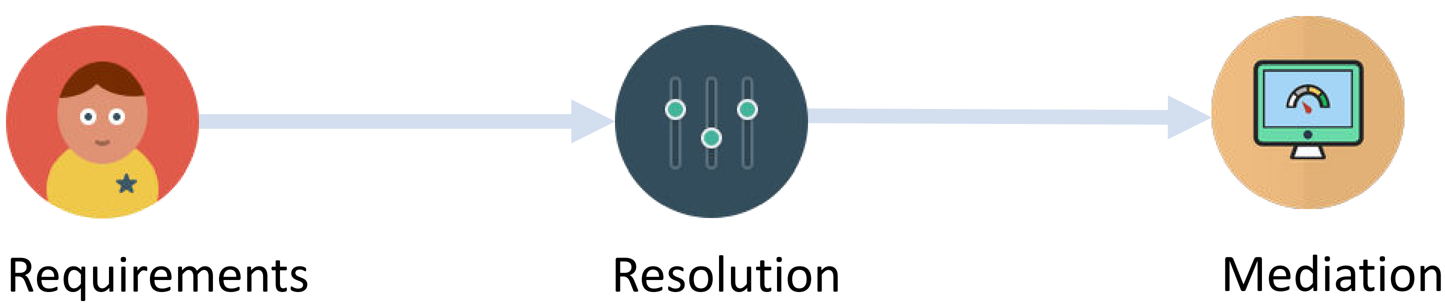
Outlook: Real-Time

Combining Query Caching,
Continuous Queries, Polyglot Queries



Summary

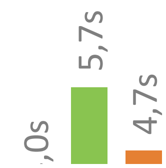
- **Cache Sketch**: web caching for database services
  - Consistent (Δ-atomic) *expiration-based* caching
  - *Invalidation-based* caching with minimal purges
  - **Bloom filter** of stale objects & **TTL Estimation**
- **Polyglot Persistence Mediator**:
  1. SLA-annotated **Schemas**
  2. **Score** DBs and choose best
  3. **Route** data and operations

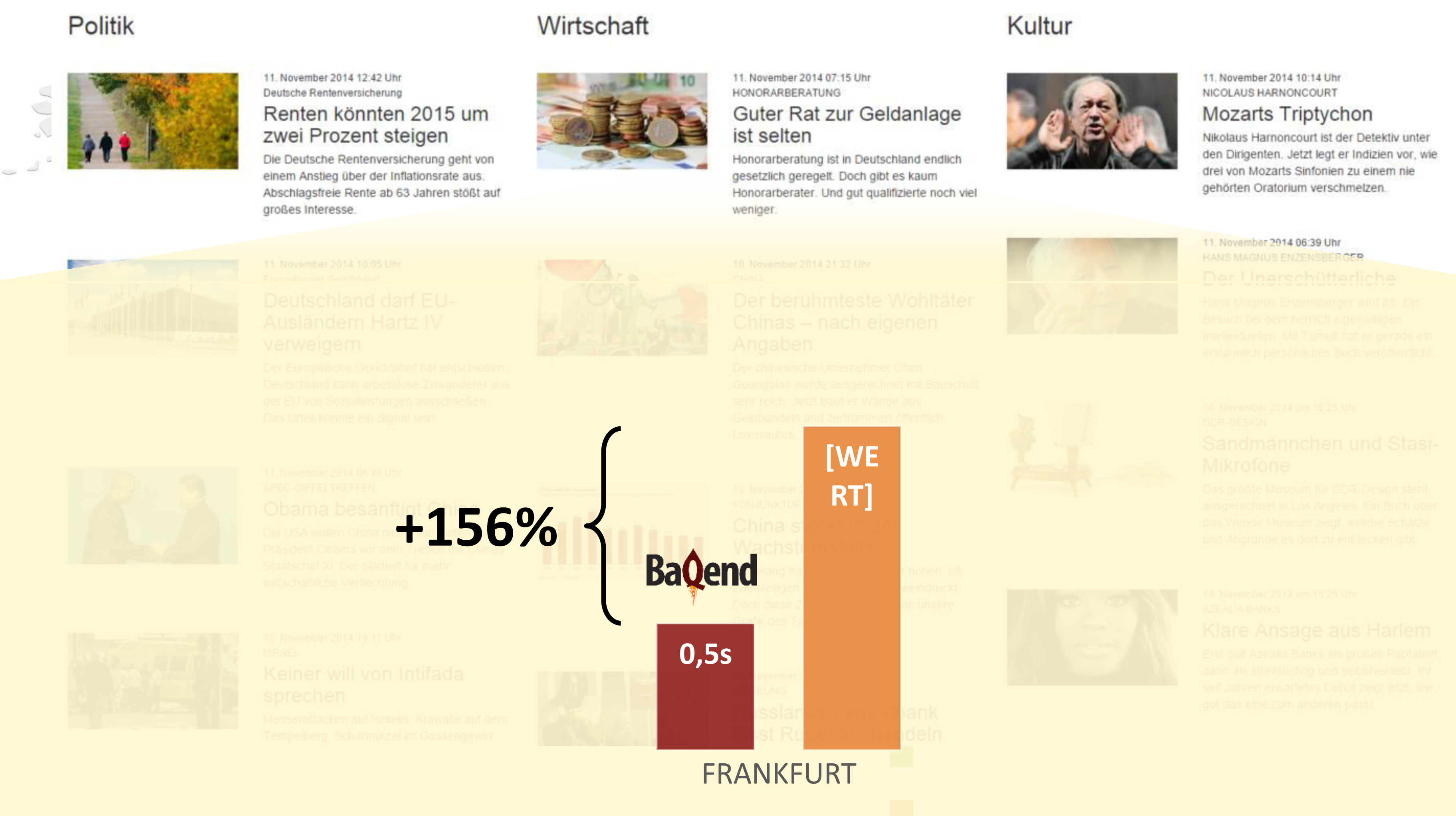



BaQend
Build faster Apps faster.

Page-Load Times

What impact does the Cache Sketch have?





Backend-as-a-Service

Tutorial on the BaaS paradigm from app perspective

BaQend Overview Tutorial Download Architecture Docs

Updating and deleting data

1 2 3 4 5 6 7 8 9

Both updates and inserts are performed by calling `save` on an object.

If at the time of the update the local copy was outdated, the operation will result in an error. In the error callback we could either refresh the object with `myTodo.load()` and retry the update or decide to overwrite the newer version at the server with our older version:

```
myTodo.save({
  force : true //Overwrite even if our copy is outdated
});
```

Deleted and saved! >

When objects reference each other, we can control, up to which depth referenced objects should also be persisted using the `depth` parameter (*persistence-by-reachability*).

If we want to get rid of an object, we do a `myTodo.delete()`. `delete` has the same options as `save` and behaves similarly.

Edit   HTML CSS JS OUTPUT

```
//The TodoService handles access to persistent Todo items
var listId = "tutorial-list" + Math.random();
var TodoService = (function() {
  return {
     //save a Todo
    save: function(todo) {
      return todo.save();
    },
    //delete a Todo
    delete: function(id) {
      return DB.Todo.load(id).then(function(todo) {
       return todo.delete();
```

Todo List

First Todo

www.baqend.com

Thank you

ritter,gessert@informatik.uni-hamburg.de

orestes.info, baqend.com