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

- Managed NoSQL Databases
  - Orestes
  - Compose
  - Cloudant
  - DynamoDB
  - Google F1
- Cloud-only DBaaS-Systems
  - BigQuery
  - EMR
- Backend-as-a-Service
  - SQL Azure
  - Amazon RDS
  - Heroku Pos.
- Managed RDBMS
- Database-as-a-Service
- Platform-as-a-Service
- Infrastructure-as-a-Service
- Analytics-as-a-Service

Cloud-Deployment of DBMSs
Architecture

Typical Data Architecture:

- Analytics
- Reporting
- Data Mining

The era of one-size-fits-all database systems is over → Specialized cloud databases
Database Sweetspots

**RDBMS**
General-purpose ACID transactions

**HBase**
Wide-Column Store
Long scans over structured data

**Greenplum**
Parallel DWH
Aggregations/OLAP for massive data amounts

**Neo4j**
The graph database
Graph algorithms & queries

**MongoDB**
Document Store
Deeply nested data models

**VoltDB**
NewSQL
High throughput relational OLTP

**Redis**
In-Memory KV-Store
Counting & statistics

**Cassandra**
Wide-Column Store
Massive user-generated content
Cloud-Database Sweetspots

**Firebase**
- Realtime BaaS
- Communication and collaboration

**Azure Tables**
- Wide-Column Store
- Very large tables

**bonsai**
- Managed NoSQL
- Full-Text Search

**Amazon RDS**
- Managed RDBMS
- General-purpose ACID transactions

**Amazon DynamoDB**
- Wide-Column Store
- Massive user-generated content

**Google Cloud Storage**
- Object Store
- Massive File Storage

**Amazon ElastiCache**
- Managed Cache
- Caching and transient storage

**Parse**
- Backend-as-a-Service
- Small Websites and Apps

**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:

- Protocols, APIs, Caching
- Load distribution, Auto-Scaling, SLAs
- Workload Management, Metering
- Replication, Partitioning, Transactions, Indexing
- Application architecture, Data Models
- Multi-Tenancy, Consistency, Availability, Query Processing, Security
Cloud-Database Models

Data Model

- **structured**
  - relational
  - RDBMS machine image
  - Managed RDBMS/DWH
  - RDBMS/DWH Service
- **unstructured**
  - schema-free
  - NoSQL machine image
  - Managed NoSQL
  - NoSQL Service

Deployment Model

- cloud-deployed (IaaS/PaaS)
- Managed (cloud-hosted)
- Proprietary DB & Cloud
- managed

Database-as-a-Service
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

DBaaS-Provider

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

RDBMS

Amazon RDS

SQL Azure

Clustrix

EDB

Heroku Postgres

NoSQL DB

MongoDB

ElastiCache

mongolab

redis

bonsai

Iris Couch

DWH

Amazon Redshift

Google Cloud SQL

Cloudant

instaclustr
Proprietary Cloud Database

Designed for and deployed in vendor-specific cloud environment

Black-box system

Managed by Cloud Provider

Cloud

Provider’s API

Database

Amazon SimpleDB

Google Cloud Datastore

Database.com

BigTable, Megastore, Spanner, F1, Dynamo, PNuts, Relational Cloud, …

Azure Tables

Openstack Swift

Amazon S3

Google Cloud Storage

Object Store

Database Object Store
Analytics-as-a-Service
Analytic frameworks and machine learning with service APIs

Analytics Cluster
Provisioning, Data Ingest

Cloud

Analytics
Amazon Elastic MapReduce
Azure HDInsight

ML
Google BigQuery
Google Prediction API
Backend-as-a-Service

DBaaS with embedded custom and predefined application logic

Authentication, Users, Validation, etc.
Maps to (different) databases

Backend API
Data API
Service-Layer
IaaS-Cloud

(mobile) BaaS

Firebase
GoInstant
Meteor PREVIEW
Parse
AppCelerator Cloud

BaQend
Build faster Apps faster
Pricing Models
Pay-per-use and plan-based

Plan-based
Parameters: Allocated Plan (e.g., 2 instances + X GB storage)
Storage, CPU, Requests, etc.
Payment: Pre-Paid, Post-Paid
Variants: On-Demand, Auction, Reserved

Pay-per-use

Account

Usage

End of month
Database-as-a-Service
Approaches to Multi-Tenancy

Private OS
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  e.g. Amazon RDS

Private Process/DB
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  e.g. Compose

Private Schema
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  e.g. Google DataStore

Shared Schema
- Virtual Schema
- Schema
- Database
- Database Process
- VM
- Hardware Resources
  Most SaaS Apps

T. Kiefer, W. Lehner “Private table database virtualization for dbaas”
UCC, 2011
## Multi-Tenancy: Trade-Offs

<table>
<thead>
<tr>
<th></th>
<th>App. indep.</th>
<th>Ressource Util.</th>
<th>Isolation</th>
<th>Maintenance, Provisioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private OS</td>
<td>✅</td>
<td>★★</td>
<td>★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Private Process/DB</td>
<td>✅</td>
<td>★☆</td>
<td>★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Private Schema</td>
<td>✅</td>
<td>★★★</td>
<td>★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Shared Schema</td>
<td>✗</td>
<td>★★★</td>
<td>★☆</td>
<td>★★★</td>
</tr>
</tbody>
</table>

*W. Lehner, U. Sattler “Web-scale Data Management for the Cloud” Springer, 2013*
# Authentication & Authorization

**Checking Permissions and Identity**

<table>
<thead>
<tr>
<th>Internal Schemes</th>
<th>External Identity Provider</th>
<th>Federated Identity (Single Sign On)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Amazon IAM</td>
<td>e.g. OpenID</td>
<td>e.g. SAML</td>
</tr>
</tbody>
</table>

---

## Diagram:

- **Authenticate/Login**
  - **Token**
  - **Authenticated Request**
    - **Authentication**
    - **Authorization**
      - **API**
        - Database-a-a-Service

## Tables:

<table>
<thead>
<tr>
<th>User-based Access Control</th>
<th>Role-based Access Control</th>
<th>Policies</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. Amazon S3 ACLs</td>
<td>e.g. Amazon IAM</td>
<td>e.g. XACML</td>
</tr>
</tbody>
</table>
Service Level Agreements (SLAs)
Specification of Application/Tenant Requirements

Service Level Objectives:
- Availability
- Durability
- Consistency/Staleness
- Query Response Time

Technical Part
1. SLO
2. SLO
3. SLO

Legal Part
1. Fees
2. Penalties
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:

\[ f_{utility}(\text{metric}) \rightarrow [0,1] \]
Workload Management

Guaranteeing SLAs

Typical approach:

Maximize:

utility

response time

response time

Resource & Capacity Planning
From a DBaaS provider’s perspective

Goal: minimize penalty and resource costs

Provisioned Resources:
- #No of Shard- or Replica servers
- Computing, Storage, Network Capacities

Underprovisioning:
- SLAs violated
- Usage maximized

Overprovisioning:
- SLAs met
- Excess Capacities

Resources vs. Time

### SLAs in the wild

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

<table>
<thead>
<tr>
<th>Model</th>
<th>CAP</th>
<th>SLAs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SimpleDB</strong></td>
<td>Table-Store <em>(NoSQL Service)</em></td>
<td>CP</td>
</tr>
<tr>
<td><strong>Dynamo-DB</strong></td>
<td>Table-Store <em>(NoSQL Service)</em></td>
<td>CP</td>
</tr>
<tr>
<td><strong>Azure Tables</strong></td>
<td>Table-Store <em>(NoSQL Service)</em></td>
<td>CP</td>
</tr>
<tr>
<td><strong>AE/Cloud DataStore</strong></td>
<td>Entity-Group Store <em>(NoSQL Service)</em></td>
<td>CP</td>
</tr>
<tr>
<td><strong>S3, Az. Blob, GCS</strong></td>
<td>Object-Store <em>(NoSQL Service)</em></td>
<td>AP</td>
</tr>
</tbody>
</table>
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?
3rd Workshop on Scalable Cloud Data Management

Co-located with the IEEE BigData Conference. Santa Clara, CA, October 29th 2015.

Call  Submit Paper

Location : Santa Clara
Submission Deadline: August 30
Outline

• Motivation

**ORESTES: a Cloud-Database Middleware**
• Two problems:
  • Latency
  • Polyglot Storage
• Vision: Orestes Middleware

• Solving Latency and Polyglot Storage

• Wrap-up
The Slide Image Text:

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

Average: 9,3s

Loading…

Revenue

-1% Revenue
100 ms
-9% Visitors
400 ms 500 ms
-20% Traffic
1s
-7% Conversions

Conversions

% Traffic

% Visitors

Revenue

Aberdeen Group

Google

Yahoo!

Amazon.com
If perceived speed is such an important factor...what causes slow page load times?
State of the art
Two bottlenecks: latency and processing
Network Latency
The underlying problem of high page load times

---

The low-latency vision
Data is served by ubiquitous web-caches

Low Latency

Less Processing
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

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

Application

Data and Operations

Polyglot Persistence Mediator

Database Metrics

Annotated Schema

Latency < 30ms
Polyglot Storage and Low Latency are the central goals of ORESTES.

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

The Big Picture

Implementation in ORESTES

Desktop
Mobile
Tablet

Content-Delivery-Network
Internet
Reverse-Proxy Caches
Orestes Servers
Cache Sketch

Unified REST API
Standard HTTP Caching

Polyglot Storage and Low Latency are the central goals of ORESTES.
Outline

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

Right:

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

Wrap-up
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
The Cache Sketch approach
Letting the client handle cache coherence

Client

Expiration-based Caches

Request Path

Cache Hits

Invalidation-based Caches

Invalidate Request

Needs Revalidation?

Client Cache Sketch

Bloom filter

Periodic every $\Delta$ seconds

at connect

Content Delivery Networks, Reverse Proxies

Non-expired Record Keys

Counting Bloom Filter

10101010

Server Cache Sketch

Needs Invalidation?

Server/DB

Invalidations, Records

Report Expirations and Writes

10101010

10201040

Staleness-Minimization

Invalidation-Minimization

Browser Caches, Forward Proxies, ISP Caches

at transaction begin

Counting Bloom Filter

10201040

Client

Expiration-based Caches

Invalidation-based Caches

Server/DB
The End-to-End Path of Requests

The Caching Hierarchy

- Low Latency
- Reduced Database Load
- Flash-Crowd Protection
- Higher Availability

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

Updated by Cache Sketch
Updated by the server

Hit: Return Object
Miss: Forward Request to DB with caching TTL

Low Latency
Reduced Database Load
Flash-Crowd Protection
Higher Availability
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:

$$ \exists r(x, t_r, TTL), w(x, t_w) : t_r + TTL > t > t_w > t_r $$
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
Solution: **Δ-Bounded Staleness**
- Clients refresh the Cache Sketch so its age never exceeds Δ
  → *Consistency guarantee*: Δ-atomicity
High Abort Rates in OCC

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

1. Begin Transaction
2. Bloom Filter
3. Cache Sketch fetched with transaction begin
4. Writes
5. Cache Sketch fetched with transaction begin
6. Commit: read- & write-set versions
7. Committed OR aborted + stale objects
TTL Estimation
Determining the cache expiration

- **Problem**: if \( \text{TTL} \gg \text{time to next write} \), 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):

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
Low Latency

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

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 heterogeneous (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 1 - Requirements

Expressing the application’s needs

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

Annotations
- Continuous non-functional
  e.g. write latency < 15ms
- Binary functional
  e.g. Atomic updates
- Binary non-functional
  e.g. Read-your-writes

Tenant

1. Define schema
2. Annotate

Database

Table

Field Field Field Field

Inherits continuous annotations

annotated
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

![Diagram](attachment:image.png)

The Provider resolves the requirements through recursive descent using annotated schema and metrics.

- **RANK** \((schema\_root, DBs)\) for available DBs
- Either: Refuse or Provision new DB
- 1. Find optimal
- 2a. If unsatisfiable
- 2b. Generates routing model

Routing Model
Route \(schema\_element \rightarrow db\)
- transform db-independent to db-specific operations
**Step II - Resolution**

**Ranking algorithm by example**

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Schema</th>
<th>RANK Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linearizability</td>
<td>ECommerceDB</td>
<td>DBs = { MongoDB, Riak, Cassandra, CouchDB, Redis, MySQL, S3, Hbase }</td>
</tr>
<tr>
<td>Availability</td>
<td>database</td>
<td></td>
</tr>
<tr>
<td>Customers</td>
<td>Table</td>
<td></td>
</tr>
<tr>
<td>ShoppingBasket</td>
<td>List&lt;String&gt;</td>
<td></td>
</tr>
<tr>
<td>Read latency</td>
<td>String</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Database</th>
<th>Linearizability</th>
<th>Availability</th>
<th>Read latency</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>99%</td>
<td>0.8</td>
<td>10ms</td>
<td>1</td>
</tr>
<tr>
<td>Redis</td>
<td>95%</td>
<td>0.05</td>
<td>1ms</td>
<td>1</td>
</tr>
<tr>
<td>MySQL</td>
<td>94%</td>
<td>0.04</td>
<td>40ms</td>
<td>0.2</td>
</tr>
<tr>
<td>HBase</td>
<td>99.9%</td>
<td>0.9</td>
<td>50ms</td>
<td>0.1</td>
</tr>
</tbody>
</table>

1. Recursive descent

2. Continuous requirement

∀ databases calculate

\[ db \rightarrow f_{utility}(db.\text{linearizability}) \]
Step II - Resolution
Ranking algorithm by example

Annotations
Linearizability
Availability

Schema
ECommerceDB
database
Customers
Table
ShoppingBasket
List<String>
UserName
String

Read latency

DB
Score
MongoDB
0.9
Redis
0.525
MySQL
0.12
HBase
0.5

Routing Model:
Customers \(ightarrow\) MongoDB

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 \(\rightarrow\) 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

---

**Polyglot Persistence Mediator**
- Uses Routing Model
- Triggers periodic materialization

1. CRUD, queries, transactions, etc.

2. route

Report metrics

3. Mediation
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

![Hacker News Article](image)

*Announcing MongoDB 3.0 (mongodb.com)*

196 points by meghan 142 days ago | 144 comments | in pocket speaker

read by 1.344.222
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

Mediator: 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

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

ORESTES

Create
Update
Delete

Pub-Sub

Fresh Cache Sketch
Continuous Queries (Websockets)

Polyglot Views
Fresh Caches
Fresh Cache Sketch

Pub-Sub
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?**

<table>
<thead>
<tr>
<th>Location</th>
<th>Average Load Times (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>California</td>
<td>0.5s, 1.8s, 2.8s, 3.6s, 3.4s</td>
</tr>
<tr>
<td>Frankfurt</td>
<td>0.6s, 3.0s, 7.2s, 5.0s, 5.7s</td>
</tr>
<tr>
<td>Sydney</td>
<td>0.5s, 2.4s, 4.0s, 5.7s, 4.7s</td>
</tr>
</tbody>
</table>

**+156%**
Updating and deleting data

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:

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

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.

```javascript
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();
      });
    }
  };
}());
```
Thank you

ritter,gessert@informatik.uni-hamburg.de

orestes.info, baqend.com