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

- **Backend-as-a-Service**
  - Managed RDBMS
    - Heroku Pos.
    - SQL Azure
    - Amazon RDS

- **Managed NoSQL Databases**
  - Orestes
  - Cloudant

- **Cloud-only DBaaS-Systems**
  - Compose
  - DynamoDB
  - Google F1
  - BigQuery
  - EMR

- **Database-as-a-Service**

- **Platform-as-a-Service**

- **Infrastructure-as-a-Service**

- **Analytics-as-a-Service**

**Cloud-Deployment of DBMSs**
Typical Data Architecture:

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

**IBM DB2**
RDBMS
General-purpose
ACID transactions

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

**VoltDB**
NewSQL
High throughput
relational OLTP

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

**mongoDB**
Document Store
Deeply nested
data models

**riak**
Key-Value Store
Large-scale
session storage

**Neo4j**
Graph Database
Graph algorithms & queries

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

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

**Realtime BaaS**
Communication and collaboration

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

**Firebase**
Managed NoSQL
Full-Text Search

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

**Azure ElastiCache**
Managed Cache
Caching and transient storage

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

**bonsai**
Managed NoSQL
Full-Text Search

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

**Amazon Elastic MapReduce**
Hadoop-as-a-Service
Big Data Analytics

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

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

- unstructured
  - NoSQL machine image
  - Managed NoSQL
  - RDBMS machine image
  - Managed RDBMS/DWH

- schema-free
  - Analytics machine image
  - Analytics-as-a-Service
  - RDBMS/Service

- relational
  - Analytics/ML APIs
  - NoSQL Service
  - RDBMS/DWH Service

Deployment Model

- unmanaged
  - cloud-deployed (IaaS/PaaS)
  - Proprietary DB & Cloud

- managed
  - Managed (cloud-hosted)
Cloud-Deployed Database
Database-image provisioned in IaaS/PaaS-cloud

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

Clustrix

EDB

SQL Azure

Heroku Postgres

NoSQL DB

Cloud SQL

mongoHQ

mongolab

redis

bonsai

Iris Couch

IaaS-Cloud

DWH

Amazon Redshift

Google Cloud SQL

Elasticache

instaclustr
Proprietary Cloud Database
Designed for and deployed in vendor-specific cloud environment

Black-box system

Managed by Cloud Provider

Cloud

Provider’s API

Amazon SimpleDB
Amazon DynamoDB
Google Cloud Datastore
Azure Tables
Database.com

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

Azure Blob Storage
Openstack Swift
Amazon S3
Google Cloud Storage

Object Store

Database
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**
- **Parameter**: 2 instances + X GB storage

Account:
- Payment: End of month

Usage:
- Payment: End of month

Examples:
- e.g. **Compose**
- e.g. **DynamoDB**
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

# 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>X</td>
<td>★★★★★</td>
<td>★★★☆</td>
<td>★★★★★</td>
</tr>
</tbody>
</table>

# Authentication & Authorization

Checking Permissions and Identity

## Internal Schemes
- e.g. Amazon IAM

## External Identity Provider
- e.g. OpenID

## Federated Identity (Single Sign On)
- e.g. SAML

---

**User-based Access Control**
- e.g. Amazon S3 ACLs

**Role-based Access Control**
- e.g. Amazon IAM

**Policies**
- e.g. XACML

---

[Diagram Image]
Service Level Agreements (SLAs)
Specification of Application/Tenant Requirements

**SLA**

**Technical Part**
1. SLO
2. SLO
3. SLO

**Legal Part**
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:

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

Typical approach:

Maximize: utility

response time
**Goal:** minimize penalty and resource costs

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

**Overprovisioning:**
- SLAs met
- Excess Capacities

**Underprovisioning:**
- SLAs violated
- Usage maximized

### SLAs in the wild

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

<table>
<thead>
<tr>
<th></th>
<th>Model</th>
<th>CAP</th>
<th>SLAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleDB</td>
<td>Table-Store (<em>NoSQL Service</em>)</td>
<td>CP</td>
<td>☠️</td>
</tr>
<tr>
<td>Dynamo-DB</td>
<td>Table-Store (<em>NoSQL Service</em>)</td>
<td>CP</td>
<td>☠️</td>
</tr>
<tr>
<td>Azure Tables</td>
<td>Table-Store (<em>NoSQL Service</em>)</td>
<td>CP</td>
<td>99.9% uptime</td>
</tr>
<tr>
<td>AE/Cloud DataStore</td>
<td>Entity-Group Store (<em>NoSQL Service</em>)</td>
<td>CP</td>
<td>☠️</td>
</tr>
<tr>
<td>S3, Az. Blob, GCS</td>
<td>Object-Store (<em>NoSQL Service</em>)</td>
<td>AP</td>
<td>99.9% uptime (S3)</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

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

Average: 9.3s

1000ms

Loading...

- Conversions
- % Traffic
- % Visitors
- Revenue
If perceived speed is such an important 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

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

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

Database Metrics

Polyglot Persistence Mediator

Annotated Schema

Latency < 30ms
The Big Picture
Implementation in ORESTES

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

Unified REST API
Standard HTTP Caching

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

Desktop
Mobile
Tablet

Content-Delivery-Network

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

Client Cache Sketch

Server Cache Sketch

Browser Caches, Forward Proxies, ISP Caches

Content Delivery Networks, Reverse Proxies

Non-expired Record Keys

Counting Bloom Filter

Report Expirations and Writes

Needs Revalidation?

Needs Invalidation?

Periodic every $\Delta$ seconds

at connect

at transaction begin

1

2

3
The End-to-End Path of Requests

The Caching Hierarchy

Hit: Return Object
Miss: Forward Request

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

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

DBposts.get(id)

Flash-Crowd Protection

Updated by Cache Sketch
Updated by the server

Dynamic Web App

Low Latency

Reduced Database Load

Flash-Crowd Protection

Higher Availability

Orestes
The Client Cache Sketch

- 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:

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

Solution: $\Delta$-Bounded Staleness
- Clients refresh the Cache Sketch so its age never exceeds $\Delta$

→ Consistency guarantee: $\Delta$-atomicity

Diagram:
- Query Cache Sketch
- Cache Sketch $c_t$
- Fresh records
- Cache Hits
- Revalidate record & Refresh Cache Sketch
- Fresh record & new Cache Sketch
- Time $t$
- Time $t + \Delta$
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 $\gg$ 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
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 fast database access be guaranteed?

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 I - 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
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

### Provider

- **Capabilities for available DBs**
- **Either**: Refuse or Provision new DB

### 1. Find optimal
- **RANK**(schema_root, DBs) through recursive descent using annotated schema and metrics

### 2a. If unsatisfiable
- Either: Refuse or Provision new DB

### 2b. Generates routing model

**Routing Model**

Route schema_element → db
- transform db-independent to db-specific operations
Step II - Resolution

Ranking algorithm by example

Annotations
- Linearizability
- Availability

Schema
- ECommerceDB
database
- Customers Table
- ShoppingBasket
  List<String>
- String
- Read latency

RANK Algorithm

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

Continuous requirement
∀ databases calculate
\[ db \rightarrow f_{utility}(db.\text{latency}) \]

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

2. Recursive descent
Step II - Resolution

Ranking algorithm by example

Annotations
- Linearizability
- Availability

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

DB
<table>
<thead>
<tr>
<th>DB</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>MongoDB</td>
<td>0.9</td>
</tr>
<tr>
<td>Redis</td>
<td>0.525</td>
</tr>
<tr>
<td>MySQL</td>
<td>0.12</td>
</tr>
<tr>
<td>HBase</td>
<td>0.5</td>
</tr>
</tbody>
</table>

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

![Diagram of Mediation Process]

**Application**
- 1. CRUD, queries, transactions, etc.
- **Polyglot Persistence Mediator**
  - Uses Routing Model
  - Triggers periodic materialization

Report metrics

1. db1
2. db2
3. db3
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

---

![Article Counter](image)
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

![Diagram showing document structure: Article ID, Title, ... Imp. ID, Imp.]

Found Resolution

Graph showing latency in ms vs. actual throughput in OPS with Orestes with PPM, Orestes without PPM, and Varnish.
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

Create
Update
Delete

ORESTES

Pub-Sub

Polyglot Views
Fresh Caches
Fresh Cache Sketch

Continuous Queries (Websockets)
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
Page-Load Times
What impact does the Cache Sketch have?

+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.
Thank you

ritter,gessert@informatik.uni-hamburg.de

orestes.info, baqend.com