Bioinformatics Resources
- NoSQL 2-

Lecture & Exercises
Prof. B. Rost, Dr. L. Richter, J. Reeb
Institut für Informatik I12
## Preliminary Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apr. 28&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Intro, General Overview (1. sh.)</td>
</tr>
<tr>
<td>May 5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Sequence Databases (2. sh.)</td>
</tr>
<tr>
<td>May 12&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Sequence Databases (3. sh.)</td>
</tr>
<tr>
<td>May 19&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Structure Databases (4. sh.)</td>
</tr>
<tr>
<td>May 26&lt;sup&gt;th&lt;/sup&gt;</td>
<td>No Lecture</td>
</tr>
<tr>
<td>Jun 2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>SQL (5. sh.)</td>
</tr>
<tr>
<td>Jun 9&lt;sup&gt;th&lt;/sup&gt;</td>
<td>SQL, NoSql (6. sh)</td>
</tr>
<tr>
<td>Jun 16&lt;sup&gt;th&lt;/sup&gt;</td>
<td>No Lecture</td>
</tr>
<tr>
<td>Jun 23&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>NoSql 2 (7.sh.)</td>
</tr>
<tr>
<td>Jun 30&lt;sup&gt;th&lt;/sup&gt;</td>
<td>MongoDB, JavaScript (8.sh.)</td>
</tr>
<tr>
<td>Jul 7&lt;sup&gt;th&lt;/sup&gt;</td>
<td>PredictProtein (9.sh.)</td>
</tr>
<tr>
<td>Jul 14&lt;sup&gt;st&lt;/sup&gt;</td>
<td>JavaScript/Node.js Applications</td>
</tr>
<tr>
<td>Jul 21&lt;sup&gt;st&lt;/sup&gt;</td>
<td>Wrap Up, Q&amp;A</td>
</tr>
<tr>
<td>Jul 28&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Exam</td>
</tr>
</tbody>
</table>

* These exercises can earn you a bonus
Evaluation

- Lectures are evaluated between June 19\textsuperscript{th} and 30\textsuperscript{th}
- Please take 15 min to complete the survey
- The necessary information was sent to the students registered for the lecture
- This lecture is: 0000002112 Bioinformatische Ressourcen (IN2321) Lecturers: (Dr. Richter, M.Sc. Reeb)
Orga - Exam Date

- Exam scheduled for Friday, Jul 28th
- Time: 16:30-18:00
- Room: MW 0350 Egbert-von-Hoyer Lecture Hall (Mechanical Engineering Building)
- Registration is MANDATORY
- so far 13 students registered
Short SQL Recap

- schema
- typed data
- tables
- defined layout
- space consumption is computable
Short SQL Recap

- well defined theory
- relational algebra
- ACID principle
- standardized query language
- fast access with indices
- well supported by software vendors
No SQL

- in principle known for a long time
- Ken Thompson 1978: Key/Value system
- big push in 2000: Web 2.0
- Map/Reduce, BigTable databases
- data volume in the range of TB and PB
- growing relational databases more and more difficult on commodity hardware
- http://www.w3resource.com/mongodb/nosql.php
Definition

- non relational data model
- enables distributed and horizontal scalability
- open source
- no or simple schema
- support for simple data replication
- simple API
- different consistency model
Issues with Relational DB

- is the schema bad, the query also is
- based on strings, susceptible for typos
- errors are not detected at compile time
- cannot be refactored
Categories of No SQL Systems

- Wide Column Stores/Column Family Systems
- Document Stores
- Key/Values/Tuple Stores
- Graph Databases
Key/Value Systems

- at least very simple schema: key and value
- keys can be grouped in namespaces and databases
- values can be complex besides simple strings there are:
  - hashes
  - set
  - lists
- queries mostly limited to API

BioinfRes SoSe 17
Column Family

- keys can point to an arbitrary number of key/value pairs
- nested key/value pairs
- nested columns
Document Stores

- works not on “actual” documents
- structured data like:
  - JSON
  - YAML
  - RDF
Graph Databases

- bases on graph or tree structures to connect elements
- property graph:
  - nodes to reflect items
  - edges to reflect relations
- very suitable for traversing
Theoretical Concepts

- Map/Reduce
- CAP-Theorem/ Eventually Consistent
- Consistent Hashing
- MVCC-Protocol
- Vector Clock
- Paxos
- REST
Map/Reduce

- require a (map/reduce) framework
- designed for efficient handling of data in the order of Tera or Peta bytes
- developed by Google
- patented since 2010
Map/Reduce Details

- originates from functional programming
- parallel processing
- no side effects
- no deadlocks
- no race conditions
- initial datastructure is not altered
- new copy with every level
Map/Reduce Details

- functions like in math:
  - a set of transformation definitions
  - no control structures
  - recursion
  - functions can be used as argument or return value: higher order functions
Map/Reduce Details

- two functions: map, reduce/fold
- used alternating (two phase approach)
- map (in parallel):
  - applied to all elements of list
  - returns a modified list
- reduce:
  - aggregate the return values from map into one result
Map/Reduce Details

- user has to provide:
  - map function
  - reduce function

- framework provides:
  - automatic parallelization and distribution
  - fault tolerance mechanisms for hard- and software failure
  - I/O scheduling
  - status and control information
Pseudocode Example

map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result += ParseInt(v);
   Emit(AsString(result));

BioinfRes SoSe 17
Characteristics of a Map/Reduce System

- commodity hardware
- Ethernet network
- large number of nodes (>100)
- distributed file system, data is stored in chunks and redundant
- data are local to processing node
CAP and Eventually Consistent

- horizontal scaling of relational databases insufficient
  - too much time to extend database to more computers
  - frequently modification of source code required
- mostly due to implementation of ACID principle
CAP Theorem

- Consistency, availability and partition tolerance cannot all completely satisfied at the same time
- only two of these criteria can be satisfied at the same time, here: availability and partition tolerance is the important combination
- consistency is reduced
Consistency

- after a transaction the database is consistent, i.e.
  - all replicating nodes of database system have the same state after an transaction; changes are propagated to all nodes
  - read access to any node returns the same result
  - this require to wait for the completion of the propagation
Availability

- acceptable response time
- depends on the specific business case
- a certain response time is guaranteed up to a specified load level
Partition Tolerance

- if a node or a connection fails the system remains to be responsive
- in large computer centers those failures are frequent
BASE Consistency Model

- Basically available
- Soft state
- Eventually consistent
Characteristics

- focus on availability
- consistency is less important
- BASE is optimistic about consistency and defines it as a transition process and not as a defined state after a transaction
  - -> Eventually Consistency
- consistent at some point in time
- interpretation different between systems
Levels of Consistency

- Causal Consistency
- Read-your-write Consistency
- Session Consistency
- Monotonic Read Consistency
- Monotonic Write Consistency
Consistent Hashing

- belongs to the family of hashing functions
- maps elements of (potentially) very large source set to a hash value from a typically much smaller value set
- advantage: constant time
- applications:
  - check sums
  - securing against manipulations
  - fast search in data structures
Consistent Hashing

- here: find a constant place memory for an object
- minimize object movements on addition or removal of nodes
- minimize object movements upon insertions
- distribute equally among resources
- circular hash space
- servers and data object are integrated (clockwise
- upon insertion or removal only neighbors are affected

BioinfRes SoSe 17
Multiversion Concurrency Control (MVCC)

- Data objects are versioned
- Represents change timeline
- Every write access creates a new version
- Contains reference to the least recent version
- Conflict resolution through explicit version comparison
Multiversion Concurrency Control (MVCC)

- disadvantage of conventional locks:
  - complete tables are locked
  - inefficient if communication time is high because of long cache pipeline or network traffic
  - not 100% guaranteed in distributed systems
  - parallel access are blocked
Multiversion Concurrency Control (MVCC)

- data objects are versioned
- represents change timeline
- every write access creates a new version
- contains reference to the least recent version
- conflict resolution through explicit version comparison
MVCC – No Conflict

transaction $Tx_{\text{Alice}}$

$v_{\text{latest}} = v_0$

Alice

$t_0$

$v_0$

read $v_0$

write $v_0 \rightarrow v_1$

$v_1$

read $v_1$

$v_0$

$t_1$

$v_0$

$v_0$

$v_1$

Bob

$v_{\text{latest}} = v_1$
MVCC – Conflict Case

Transaction $T_{x_{Alice}}$

- $v_{latest} = v_0$
- Read $v_0$
- Write $v_0 \rightarrow v_{1a}$
- $v_{latest} = v_{1a}$

Transaction $T_{x_{Bob}}$

- $v_{latest} = v_{1b}$
- Read $v_0$
- Write $v_0 \rightarrow v_{1b}$

Conflict!

$v_{latest} \neq v_0$
Vector Clocks

- challenge:
  - many instances write data
  - they have to be synchronized and ordered afterwards
- solution: Vector Clocks
  - originated in the field of operating systems
  - Leslie Lamport (1978) describes Timestamps/Clocks
Lamport Timestamps/ Criteria

- weak consistency criterion: if event $e_1$ causes event $e_2$ then the timestamp of $e_1$ has be smaller than the timestamp of $e_2$

- strong consistency criterion (the opposite): if the timestamp of $e_1$ is smaller than the one of $e_2$ then event $e_1$ has been the cause for event $e_2$

- events can be sorted in a partial order
  - every event gets a timestamp which does not reflect real time
  - monotone increasing integer

- Timestamps fulfill only the weak criterion
Version Vector / Vector Clock

- Version Vector: Vector (Tuple) of values/timestamps of an object

- Vector Clock:
  - Each process/database has an counter which is incremented
  - every process remembers the sender and the timestamp
  - every message/version has a vector of id-timestamp pairs attached
Vector Clocks in NoSQL

- so the Vector Clock is a list of ID x Time tuples
- this enable the client to sort and figure out the different versions if multiple clients update and replicate records at the same time
- we demonstrate this with a simple example:
  - three people, denoted by their initial want to agree on a sports activity
Story to the Example

- Laura, Anna and Paul (replacing nodes) want to agree on sports (have consistent data)
  - nodes can request the current version of a record and they can update each other
  - simultaneous broadcast creates confusion
  - goal: consistent information (→ consensus protocols)
Story to the Example / Solution

- Laura starts, suggesting to go jogging: jogging, \([L:1]\) (jogging is the data to store, \(L:1\) the Vector Clock) and sends/replicates this to Anna and Paul.

- Paul is becoming active and suggest to go surfing: surfing, \([L:1,P:1]\) and sends this to Anna and Paul.

- Because of network problem Anna does not receive the message, Laura receives it.
Story to the Example / Solution

- Laura agrees to Paul and return the surfing suggestion, incrementing her counter: surfing, [P:1, L:2]

- Anna becomes concerned and agrees to jogging, based on Laura’s suggestion: jogging[L:1, A:1] and sends it to Paul

- Paul has to (and can) detect the conflict: jogging could had a majority (Laura & Anna), **BUT** Laura also already agreed on surfing (Laura & Paul)
Story to the Example / Solution

- surfing[A:0, P:1, L:2]
- jogging[P:0, L:1, A:1]
  not yet known counters are listed with 0

- Paul can detect that Anna’s message was not a response to his suggestion since P:0. There are two possible resolutions:
  - jogging, because initially both girls wanted to
  - surfing, because Laura changed her mind
Story to the Example / Solution

- Paul decides to go on with surfing and communicates this to Anna and Laura: surfing, [L:2, A:1, P:2]

- the discussion could still go on now, but this way the Vector Clocks help to make reasonable decisions and to check causal dependencies
Paxos

- goal: ensure data integrity if nodes in cluster with replicated data fail
- belongs to Quorum-Consensus algorithms
- leads to an agreement between participating nodes
- superior to classical Two-Phase-Commit (2PC)
- tolerant for:
  - minority of the nodes fails
  - a transaction crashes
  - message loss
Basic Paxos Consensus Algorithm

- based on voting:
  - one *client* suggests a values
  - the other *acceptors* (quorum) vote
  - each ballot has a *leader* (coordinator)
  - proposers support clients, convince acceptors and coordinate conflict resolutions
Basic Paxos – Execution

- Phase 1a (prepare): proposer/leader acquires the current (maximum) ballot number from phase 1 and sends it to the quorum

- Phase 1b (prepare): if the received number is larger than any number received before, a node sends its status to the leader including:
  - largest received number from phase 1a
  - largest number sent in phase 2b
  - no smaller or equal ballot numbers than the current will be accepted
Basic Paxos – Execution

- Phase 2a (accept): if the leader for a ballot received positive 1b messages from a quorum
  - free – no quorum has sent a number larger than 2b and has therefore voted for a value \( v \) (no completed ballot before)
  - forced – a quorum has sent a ballot larger phase 2b, i.e. it has selected an value \( v \)
- if forced leader sends value \( v \), if free leader can send any value
Basic Paxos – Execution

- **Phase 2b (accepted):** If an acceptor gets a 2a message for which he agreed before with a 1b message, the value is accepted and it sends a phase 2b message with v and ballot to the leader.

- **Phase 3:** If the leader get a phase 2b message for v and ballot from a quorum, it knows that v was accepted and communicates this to all interested processes.
Graph Databases

- graphs allow to represent connected information very intuitively by using vertices and edges
- useful for current problems like, a.o.:
  - internet routing
  - contacts in social networks
  - recommender systems
  - fraud detection
  - regulatory networks
  - semantic web
  - ...

BioinfRes SoSe 17
Graph Lingo

- graphs are represented by a pair (tuple) of two sets, $V$ (vertices) and $E$ (edges)
- vertices are nodes, representing a kind of fact
- edges are the connections/relations between vertices and can be directed or undirected
- $G = (V,E)$, $V = \{1,2\}$, $E = V \times V = \{(1,2)\}$
Property Graph Model

- directed, multi-relational graph
- labeled/(typed) edges
- vertices and labels have properties
- properties are key/value pairs of type `<String, Object>` like:
  Name: Alice or Age: 30
Property Graph Model

- strong typing of vertices and edges possible ('Type'/'_Type', depends on the support of the system)
  - useful to a semantic meaning
  - support of automatic handling
  - allows for definition of consistency criteria and indices
  - make partition of graphs easier

- bidirectional edges are realized by two unidirectional edges
Property Graph Model

- multi-edges require different labels
- vertices and edges have an unique identity: ‘Id’, '_Id'
- used for ids: integers, string, URIs
- extension: multi value property, which allow lists or sets of values
- special case for edge label values: edge weights
- another extension: higher order relations with hyper edges and hyper vertices

BioinfRes SoSe 17
Property Graph Model / Extensions

- higher order relations with hyper edges and hyper nodes
  - hyper edge: connects more than two nodes
  - hyper vertex: combination of a set of vertices/nodes, keeps internal edges

- paths: sequence of edges

- subgraphs: a defined combination of nodes and edges into a single node

- version information allows to represent the graph evolution and/or concurrency
Graph Representations

- different representations available for persistence and memory
- difficult to match a good performance of persistence and a good support for a variety of graph algorithms at the same time
Adjacency Matrix

- square matrix/table
- all n nodes are listed horizontally and vertically
- if an edge exists between nodes u and v, there is an entry in the table at position [u,v]
- test for the connection of two nodes u and v can be done very quick
Adjacency Matrix / Problems

- disadvantage: huge space consumption even with sparse matrices, i.e. graphs with many nodes but only a few edges
- it is difficult to identify the connecting edges for a given node
- to identify neighbors you always have to read a complete row or column
- hypergraphs can not be represented
Incidency Matrix

- A matrix with nodes on one axis and edges on the other axis
- Much more space efficient for very weakly connected edges than the adjacency matrix
- In more connected graphs it needs more space than the adjacency matrix
- Can represent hypergraphs
Adjacency List

- extension of edge list
- edges are sorted according to their start node
- for every node the connecting edges are stored
- time consumption depends only to connectivity of the node, not on the complete graph size
Edge List

- nodes and edges are stored separately
- insertion and deletion of single edges is very efficient
- identification of connecting edges given a node is inefficient, since the whole edge list has to be searched
Example Graph

\[ \begin{array}{c|cccc}
  v1 & v2 & v3 & v4 \\
  v1 & 0 & 0 & 0 & 0 \\
  v2 & 1 & 1 & 1 & 0 \\
  v3 & 2 & 1 & 0 & 0 \\
  v4 & 0 & 1 & 0 & 0 \\
\end{array} \]

\[ \begin{array}{c|cccccc}
  e1 & e2 & e3 & e4 & e5 & e6 \\
  v1 & 1 & 1 & 1 & 0 & 0 & 0 \\
  v2 & 0 & 0 & -1 & 2 & 1 & 1 \\
  v3 & -1 & -1 & 0 & 0 & 0 & 1 \\
  v4 & 0 & 0 & 0 & 0 & -1 & 0 \\
\end{array} \]

V1 → v2 → v3 → v3
V2 → v2 → v3 → v4
V3 → v2
V3
Graph Traversal

- either partial or complete visit of the nodes
- three strategies:
  - breadth-first/depth-first
  - algorithmic traversals
  - random based
Graph Indexing and Partitioning

- Graph indexes are first-class citizens
- can inserted as sub-graphs and attached to specific nodes as specific information
- If Graph gets to big it can be split into partial graphs
- Optimal Partitioning is highly domain and semantics-dependent -> no good standard solution
Tinkerpop Graph Processing Step

- attempt to provide uniform interfaces for Property-Graph based systems
- covers the backend database from the application developer
- consists of several sub-projects:
  - Blueprints: Java interface for Property-Graph models
    - no own persistence yet
  - supports transactions
Graph Creation

// Erzeuge einen neuen Graphen mit Neo4j-Persistenz
Graph graph = new Neo4jGraph("/tmp/my_graph");

// Erzeuge Knoten mit Id "Alice", "Bob" und "Carol"
Vertex alice = graph.addVertex("Alice");
Vertex bob = graph.addVertex("Bob");
Vertex carol = graph.addVertex("Carol");

// Füge die Namen und das Alter als Properties hinzu
alice.setProperty("Name", "Alice");
alice.setProperty("Alter", 18);
bob.setProperty("Name", "Bob");
bob.setProperty("Alter", 22);
carol.setProperty("Name", "Carol");
carol.setProperty("Alter", 20);

// Erstelle die dazugehörigen Kanten...
Edge e1 = graph.addEdge("e1", alice, bob, "kennt");
Edge e2 = graph.addEdge("e2", alice, carol, "kennt");
Edge e3 = graph.addEdge("e3", carol, bob, "kennt");

// ...und setzte das Kanten-Property "seit"
e1.setProperty("seit", "2001/10/03");
e2.setProperty("seit", "2003/12/04");
e3.setProperty("seit", "2001/07/12");

graph.shutdown();

Set<String> indexKeys = new HashSet<String>();
indexKeys.add("Name");

// Indiziere die angegebenen Property-Schlüssel aller Knoten
AutomaticIndex index = graph.createAutomaticIndex(
    "IndexOfName", Vertex.class, indexKeys);

// Bereits vorhandene Knoten müssen neu indiziert werden
AutomaticIndexHelper.reIndexElements(index, graph.getVertices());

// Iteriere über die Ergebnisse der Indexanfrage
for (Vertex vertex : index.get("Name", "Alice")) {
    System.out.println("Vertex: " + vertex);
}
Graph Query Languages

- no common standard yet
- pattern-based: SPARQL, RDF Query language
- navigation-base: Gremlin, sones GQL
- logic-bases: OWL, GraphLog
Neo4j

- one of the oldest NoSQL graph databases (2003)
- full ACID support
- uses own format to store graphs on disc
- Apache Lucene used for indexing
- can run as server as well as embedded
Integration with Java

- easiest integration using Maven (adding to the xml file), then:
  ```java
  GraphDatabaseService graphdb = new EmbeddedGraphDatabase("/var/graphdb");
  ```
Graph Creation

enum Relationships implements Rel { knows }

Transaction tx = graphdb.beginTx();
try {
    Node Alice = graphdb.createNode();
    Node Bob = graphdb.createNode();
    Node Carol = graphdb.createNode();  
    Alice.setProperty("Name", "Alice");
    Bob.setProperty("Name", "Bob");
    Carol.setProperty("Name", "Carol");  
    Alice.setProperty("Age", 18);
    Bob.setProperty("Age", 20);
    Carol.setProperty("Age", 22);
    Relationship Alice_Bob = Alice.createRelationshipTo(Bob, Rel.knows);
    Relationship Alice_Carol = Alice.createRelationshipTo(Carol, Rel.knows);
    Relationship Carol_Bob = Carol.createRelationshipTo(Bob, Rel.knows);
Graph Creation

```java
Alice_Bob.setProperty("since", );
Alice_Carol.setProperty("since", );
Carol_Bob.setProperty("since", );
    tx.success();
} catch (Exception e) {
    tx.failure();
} finally {
    tx.finish();
}
```
Manual Indexing

IndexManager index = graphdb.index();
Index<Node> UserIdx = index.forNodes("User");
RelationshipIndex KnowsIdx = index.forRelationships("knows");
UserIdx.add(Alice, "Name", Alice.getProperty("Name"));
UserIdx.add(Alice, "Age", Alice.getProperty("Age"));
[...]
Traversal Configuration

- besides simple traversals and wildcard searches there are number of sophisticated tweaks:
  - Order: Determines the branching order (DFS/BFS)
  - Uniqueness: how to handle multiple hits of the same nodes
  - Pruning: which branches not to follow
  - Filtering: which hits are considered for the result
  - Relationship expanding: dedicated edge handling
Example Traversal

TraversalDescription td = new TraversalDescriptionImpl();
    td = td.prune(Traversal.pruneAfterDepth(2)).
        filter(Traversal.returnAllButStartNode()).
        relationships(KNOWS);
Traverser tr = td.traverse(startNode);
for ( Path path : tr ) {
    System.out.println( "End Node: " +
        path.endNode().getProperty( NodeProperty.NAME ) );
}
Cypher

- own graph query language since version 1.4
  - developed for pattern recognition
  - declarative
  - implemented in Scala -> parallel enabled

- query structure:
  - starts with a set of nodes
  - match statement (node in (), edges ->)
  - return statement with optional where or sort
Examples

// start nodes via Ids
start Person = (1, 2)
match (Person)-[:knows]->(Friend)
where Friend.Age > 18
return Friend.Name, Friend.Age, Friend.City?
sort by Friend.Name

// starts nodes via index query
start Person = (Person-index, Name, "Alice")
match (Person)-[:knows]->()-[:knows]->(FriendofFriend)
where not(FriendofFriend.Age < 17)
return FriendofFriend.Name
Interfacing NoSql

- specific APIs vary heavily
- most support RESTful interface:
  - REpresentational State Transfer
  - architecture for web applications
  - predominantly implemented using HTTP protocol
CRUD

- minimum set of access functions:
  - Create, Read, Update, Delete

<table>
<thead>
<tr>
<th>CRUD</th>
<th>SQL</th>
<th>HTTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>insert</td>
<td>POST</td>
</tr>
<tr>
<td>Read</td>
<td>select</td>
<td>GET</td>
</tr>
<tr>
<td>Update</td>
<td>update</td>
<td>PUT</td>
</tr>
<tr>
<td>Delete</td>
<td>delete</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
Components

- Resources, Operations, Links
- each request is independent, i.e. it has no state
  -> no need for synchronization
- abstract view of http protocol: nouns and verbs.
  – each request is defined by the application of a
    verb to noun and an optional response
- a request is composed of a header with a
  method and meta data in key/value format and
  an optional body
- a response is like a request but without a method
Resources

- addressable end-point to the system
  - e.g. HTML document, video, a process
- a resource is abstract and can have more than one representation
- the user always interacts with a representation (HTML, a graphics format, XML,...) and may choose the desired one
Operations

- HTTP defines a set of operations with known semantics:
  - GET
  - HEAD
  - PUT
  - POST
  - DELETE
Characteristics for Operations

- operations can be classified according the criteria safe and idempotent which are important for the system’s integrity and caching performance
- safe: no side effects, no responsibility for the user
- idempotent: side effect, but only the first time – upon multiple execution the server state does not change anymore
GET/HEAD

- safe and idempotent
- HEAD: returns only meta information about the resource
- GET: contains in addition to the meta information also a representation of the resource
- a non conforming example: “http://www.example.com/api?action=delete”
PUT

- idempotent
- the referenced resource representation is transmitted to the server (side effect -> not safe)
- only the first execution changes the state of the server
- this can be achieved if a server maintains version numbers for a document which has to be match by the request
PUT – Simple Example

- Q (request): GET doc
- R (response): return doc v=1, doc content
- Q: Put doc v=1, doc content modified
- R: Request v=1 matches server v=1; doc content modified stored; update version v=2
- Q (a second time): Put doc v=1, doc content modified (maybe again)
- R: Request v=1 does not match server v=2; doc content not stored;
DELETE

- idempotent
  - once the resource is removed all subsequent requests fail -> server state remains the same
- not safe
- the referred resource is remove from the server/access blocked
POST

- no guarantees at all
- transmits data for processing
- the processing result can be used to create a new resource, modify an existing one or not at all
- can be used for very complex queries because all parameters can be included in the body – GET had to include it in the URI
LINKS

- HTTP does not represent links
- links are modeled in URIs
- encoding depends on the type of representation
- can contain meta data to support appropriate resource by the user
Example (Stefan Edlich et al. “NoSQL”, 2. Auflage, Hanser Verlag (2011))

POST /api/ HTTP/1.1
Host: cocktails.example.com
Content-Type: application/json

...

{ "name" : "Ipanema",
  "description" : "Eine alkoholfreie Variante für den Caipirinha-Abend",
  "ingredients" : {
    "Limette" : { "amount" : 1, "preparation" : "Achteln" },
    "Brauner Zucker" : { "amount" : 2, "unit" : "TL" },
    ...
  },
  "preparation" : "Limetten und Zucker in einem Glas mörsern, mit crushed ice bedecken und den Flüssigkeiten auffüllen. Mit einem Strohhalm servieren"
}
HTTP/1.1 201 Created
Content-Type: application/json
Location: http://cocktails.example.com/cocktails/1

{ "id": "1" }

GET /cocktails/1 HTTP/1.1
Host: cocktails.example.com

{ "id": "1",
 "name": "Ipanema",
 "description": "Eine alkoholfreie Variante für den Caipirinha-Abend",
 "ingredients": {
   "Limette": { "amount": 1, "preparation": "Achteln" },
   "Brauner Zucker": { "amount": 2, "unit": "TL" }
 },
 "preparation": "Limetten und Zucker in einem Glas mörsern, mit crushed ice bedecken und den Flüssigkeiten auffüllen. Mit einem Strohhalm servieren",
 "links": {
   "linktypes/publish": "http://cocktails.example.com/publish/1",
   "linktypes/edit": "http://cocktails.example.com/cocktails/1",
   "linktypes/delete": "http://cocktails.example.com/cocktails/1"
 } }
PUT /cocktails/1 HTTP/1.1
Host: cocktails.example.com
Content-Type: application/json
...
{
  "tags" : [ "alkoholfrei", "Eis" ],
  ...
}

DELETE /cocktails/1 HTTP/1.1
Host: cocktails.example.com

POST /publish/1 HTTP/1.1
Host: cocktails.example.com
Content-Type: application/json
...
{
  "publish" : true
}

{
  "id" : "1",
  ...
  "links" : {
    "linktypes/delete" : "http://cocktails.example.com/cocktails/1",
    "linktypes/ratings" : "http://cocktails.example.com/ratings/1"
  }
}
Document Stores

- originates to Damien Katz and Lotus Notes, CouchDB
- the responsibility for the schema is moved from the database towards the application:
  - loss of enforcement of normalization and referential integrity
  - gain of flexibility and schema modifications at runtime for the application
- data mostly stored as JSON
MongoDB

- Document Store
- try to close the gap between classic RDBMS and Key/Value stores
- supported by a number of successful internet companies (10gen, ...)
- good integration with programming languages: C++, C#, Java, JavaScript, PHP, Ruby, Perl, Python
JSON Example

```
{
  "id": "1",
  "name": "Ipanema",
  "description": "Eine alkoholfreie Variante für den Caipirinha-Abend",
  "ingredients": {
    "Limette": { "amount": 1, "preparation": "Achteln" },
    "Brauner Zucker": { "amount": 2, "unit": "TL" },
    ...
  },
  "preparation": "Limetten und Zucker in einem Glas mörsern, mit crushed ice bedecken und den Flüssigkeiten auffüllen. Mit einem Strohhalm servieren",
  "links": {
    "linktypes/publish": "http://cocktails.example.com/publish/1",
    "linktypes/edit": "http://cocktails.example.com/cocktails/1",
    "linktypes/delete": "http://cocktails.example.com/cocktails/1"
  }
}
```
JSON in MongoDB

- each document needs a special ID field: _id
- the _id values has to be unique
- can be anything
- automatic default:
  - automatic 12-byte number:
    - 4 byte time stamp
    - 3 byte client machine id
    - 2 byte process id
    - 3 byte counter
Demo

- Check out the command line and python tutorial under: http://api.mongodb.com/python/current/tutorial.html
- get a toy mongodb server for free at: https://mlab.com