Bioinformatics Resources
- NoSQL -

Lecture & Exercises
Prof. B. Rost, Dr. L. Richter, J. Reeb

Institut für Informatik I12
Orga - Exam

- Change confirmed:
  Wednesday, Jul 22\textsuperscript{nd}, 15-17 o’clock
- Interimshörsaal 2
- Please register in TUMonline before Jun 30\textsuperscript{th}
- Lecture evaluation
- Information event about 5./6. semester at 12.00
  i.e. today’s lecture will close earlier
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.
BASE Consistency Model

- Basically available
- Soft state
- Eventually consistent
Characteristics

- focus on availability
- consistency is less important
- BASE is optimistic about consistency and defines is 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: given a replicated/multiple node database
  - two processes A and B, A writes X, B reads X and writes Y
  - A and B are causal dependent
  - even if A and B are close in time it is guaranteed on all nodes that B can read X before Y is written
Levels of Consistency

- **Read-your-write Consistency:**
  - special case of causal consistency
  - a process which modified a data object will never get an older version of the object than the one it created for every subsequent access

- **Session Consistency**
  - this a more implementation oriented approach using sessions
  - during a session a process is guaranteed to have read-your-write consistency
Levels of Consistency

- **Monotonic Read Consistency**
  - if a process reads the value of data object it will never get an older version of that data object in a subsequent access

- **Monotonic Write Consistency**
  - all write operations of a process are serialized
  - write operations to the data objects happen in the same order the process initiated them
Multiversion Concurrency Control (MVCC)

- disadvantage of conventional table locks:
  - complete tables are locked
  - efficient if not too long and too often
  - inefficient if communication time is high because of long cache pipeline or network traffic
  - not 100% guaranteed in distributed systems
  - parallel access is 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 $T_{x_{Alice}}$

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

Alice

$\text{read } v_0$

$t_0$

$\text{write } v_0 \rightarrow v_1$

$v_1$

$v_0$

$t_1$

$\text{read } v_0$

$v_1$

$\text{read } v_1$

Bob

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

Transaction $Tx_{Alice}$
- Read $v_0$
- Write $v_0 \rightarrow v_{1a}$

Transaction $Tx_{Bob}$
- 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 e1 causes event e2 then the timestamp of e1 has be smaller than the timestamp of e2

- strong consistency criterion (the opposite): if the timestamp of e1 is smaller than the one of e2 then event e1 has been the cause for event e2

- event 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 IDxTime 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:
  - four 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) to 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 sind 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: ensures 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 it 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 SS 15
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
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 performing 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

![Graph Diagram]

<table>
<thead>
<tr>
<th></th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>v3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>e1</th>
<th>e2</th>
<th>e3</th>
<th>e4</th>
<th>e5</th>
<th>e6</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v2</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>v3</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>v4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
</tbody>
</table>

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
// 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");
```
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"));
[...]
Traversals 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
//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 nods via index query
start Person = (Person-index, Name, "Alice")
match (Person)-[:knows]->()[:knows]->(FriendofFriend)
where not(FriendofFriend.Age < 17)
return FriendofFriend.Name
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 run-time 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
MongoDB

- own query syntax
- Map/Reduce implementation to handle large data
- supports journaling: with a write ahead log, it can recover after a crash by reverting all incomplete transactions
Data Model

- scheme free, i.e. no predefined scheme
- scheme is created upon insertion
- documents are organized in collections
- the documents in a collection can differ in structure
- but: documents in a collection should be similar to enhance indexing
- collection can be limited in size (capped collections): maintained on first-in first-put basis
MongoDB Components

- MongoDB database server
- MongoDB JavaScript shell
- backup and recovery tools
- import and export tools
- GridFS tool
- MongoDB sharding dispatcher
- administration tool
Access

- via JavaScript shell:
  ```
  > db.foo.insert( { Key : 'Value' } )
  > db.foo.find()
  { "_id" : ObjectId("4d8f8fbdd5af48929c713c223"),
    "Key": "Value" }
  ```

- dedicated Java API to support the CRUD operations (package com.mongeddb)
import java.net.UnknownHostException;
import com.mongodb.*;

public class FirstMongoDBClient {
    public static void main(String[] args) {
        try {
            Mongo m = new Mongo("localhost", 27017);
            DB db = m.getDB("TestDB");
            DBCollection col = db.getCollection("testCollection");
            BasicDBObject doc = new BasicDBObject();
            doc.put("MyKey", 555);
            doc.put("foo", "bar");
            col.insert(doc);
        }
        catch (UnknownHostException e) {} 
        catch (MongoException e) {} 
    }
}
import java.net.UnknownHostException;
import com.mongodb.*;
public class FirstMongoDBClient {
    public static void main(String[] args) {
        try {
            Mongo m = new Mongo("localhost",27017);
            DB db = m.getDB("TestDB");
            DBCollection col = db.getCollection("testCollection");
            DBObj myDoc = col.findOne();
            System.out.println(myDoc);

            DBCursor cur = col.find();
            while(cur.hasNext()) System.out.println(cur.next());
            BasicDBObject query = new BasicDBObject("MyKey",555);
            myDoc = col.findOne(query);
            System.out.println(myDoc);
            DBCursor cur = col.find(query);
            while(cur.hasNext()) System.out.println(cur.next());
        }
        catch (UnknownHostException e) {} }
        catch (MongoException e) {} }
}