Bioinformatics Resources
- NoSQL -

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

BioinfRes SoSe 16
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
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 reflects 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));
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 function
- 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 16
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 $T_{x_{Alice}}$

$v_{latest} = v_0$

$v_0$ $v_1$ $v_0$ $v_1$

read $v_0$ write $v_0 \rightarrow v_1$ read $v_0$

$v_{latest} = v_0$

Bob

$t_0$ $t_1$ $t$

Alice

read $v_0$ read $v_1$
MVCC – Conflict Case

transaction $Tx_{\text{Alice}}$

Alice

$t_0$

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

$t_1$

read $v_0$

write $v_0 \rightarrow v_{1a}$

$v_{1a}$

$t_2$

write $v_0 \rightarrow v_{1b}$

$v_{1b}$

$t_3$

Bob

Conflict!

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

transaction $Tx_{\text{Bob}}$
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 to 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 $e_2$

- 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

BioinfRes SoSe 16
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) 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, 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 16
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 performance 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{align*}
V1 \rightarrow & v2 \rightarrow v3 \rightarrow v3 \\
V2 \rightarrow & v2 \rightarrow v3 \rightarrow v4 \\
V3 \rightarrow & v2 \\
V3 & 
\end{align*}

\begin{tabular}{c|cccc}
 & v1 & v2 & v3 & v4 \\
\hline
v1 & 0 & 0 & 0 & 0 \\
v2 & 1 & 1 & 1 & 0 \\
v3 & 2 & 1 & 0 & 0 \\
v4 & 0 & 1 & 0 & 0 \\
\end{tabular}

\begin{tabular}{c|cccc}
 & e1 & e2 & e3 & e4 \\
\hline
v1 & 1 & 1 & 1 & 0 \\
v2 & 0 & 0 & -1 & 2 \\
v3 & -1 & -1 & 0 & 0 \\
v4 & 0 & 0 & 0 & -1 \\
\end{tabular}
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");
  ```
Graph Creation

```java
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);
```

BioinfRes SoSe 16
Graph Creation

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
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
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