RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
Motivation (NoSQL)

"One size doesn't fit all"

Stonebraker

Reinefeld
Design Goals

- Key/Value store
- Scalability: many concurrent write accesses
- Strong data consistency
- Evaluate on a real-world web app
  - Wikipedia
- Implemented in Erlang
- Java API
Motivation (Consistency)
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
High Level Overview

Erlang implementation of a distributed key-value store that has majority based transactions on top of replication on top of a structured peer to peer overlay network
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
Architecture - P2P Layer

Scalaris: Key/Value Store (= simple database)

- implements ACID
- improves availability at the cost of consistency
- implements - scalability
  - eventual consistency

unreliable, distributed nodes

Figure 1. Scalaris system architecture.
Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.
Architecture - Chord - Properties

- Load balancing
  - consistent hashing

- Logarithmic routing
  - finger tables

- Scalability

- Availability

- Elasticity
Architecture - Chord # - Properties

- No consistent hashing
- Keys are ordered lexicographically
- Efficient range queries
- Load balancing
  - must be done periodically if the keys are not randomly distributed
Chord #

Node space: $N_0 - N_{15}$

Key space: 0 - 255
Figure 1. Scalaris system architecture.
Replication Layer

- Symmetric replication
- Replicated to $r$ nodes
- Operations performed on a majority of replicas
Replication Layer

- Can tolerate at most \((r - 1) / 2\) failures
- Objects have version numbers
- Return the object with the highest version number from a majority of votes
Architecture - Transaction Layer

**Figure 1.** Scalaris system architecture.
Transaction Layer

- Writes use the adapted Paxos commit protocol
- Non-blocking protocol
- Strong consistency
  - Update all replicas of a key consistently
- Atomicity
  - Multiple keys transactions.
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
Data Model

- Key - Value Store
- Keys are represented as strings
- Values are represented as binary large objects
- In-memory
- Persistence is difficult with quorum algorithms
- Snapshot mechanism is best option for persistence
- Database back ends provide storage beyond RAM & Swap
Data Model

- The dictionary has three operators
  - `insert(key, value)`
  - `delete(key)`
  - `lookup(key)`
- Scalaris implements a distributed dictionary

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clarke</td>
<td>2007</td>
</tr>
<tr>
<td>Allen</td>
<td>2006</td>
</tr>
<tr>
<td>Bachman</td>
<td>1973</td>
</tr>
<tr>
<td>Thompson</td>
<td>1983</td>
</tr>
<tr>
<td>Knuth</td>
<td>1974</td>
</tr>
<tr>
<td>Codd</td>
<td>1981</td>
</tr>
</tbody>
</table>

Each node only stores part of the data.
Distributed Dictionary on Chord #

Items are stored on their clockwise successor.
Adapted Paxos Commit

- Middle Layer of Scalaris
- Ensures that all replicas of a single key are updated consistently
- Used for implementing transactions over multiple keys
- Realizes ACID
Adapted Paxos Commit

1. Step: \(O(\log N)\) hops

- Leader: \texttt{InitTM}
- Replicated Transaction Managers (TMs): \texttt{RegisterRTM}
- Items at Transaction Participants (TPs): \texttt{GetTP}, \texttt{RegisterTP}

2.-6. Step: \(O(1)\) hops

- Leader: \texttt{TMs and TPs}
- Replicated Transaction Managers (TMs): \texttt{Prepare}
- Items at Transaction Participants (TPs): \texttt{Prepare/Abort}

- Leader: \texttt{Ack Prepared/Abort}
- Replicated Transaction Managers (TMs): \texttt{Commit/Abort}
- Items at Transaction Participants (TPs): \texttt{Commit/Abort}

After majority

1. Step
2. Step
3. Step
4. Step
5. Step
6. Step
Replica Management

- All key/value pairs over $r$ nodes using symmetric replication

- Read and write operations are performed on a majority of the replicas, thereby tolerating the unavailability of up to $\lfloor (r - 1)/2 \rfloor$ nodes

- A single read operation accesses $\lceil (r + 1)/2 \rceil$ nodes, which is done in parallel.
Failure Management

- **Self-Healing**
  - Continuously monitors the system
  - Nodes can crash
    - If they announce the system handles gracefully
    - Unresponsive nodes lead to false positives
      - Failure detector reduces FP to .001
  - When a node crashes, the overlay network is immediately rebuilt

- **Crash Stop**
  - Assumption is that a majority of replicas are available
  - If a majority of replicas are not available, the data is lost
Consistency Model

- Strict consistency between replicas
  - adapted Paxos protocol
  - atomic transactions
ACID Properties

- Atomicity, Consistency and Isolation
  - majority based distributed transactions
  - Paxos protocol

- Durability
  - replication
  - no disk persistence
  - Scalaxis: branch version, adds disk persistence
Elasticity

- Implemented at the p2p layer level
- Transparent addition and removal of nodes in Chord
  - failures
  - replication
  - automatic load distribution
- Self-organization
- Low maintenance
Load Balancing

- Based on p2p system properties
- Chord: consistent hashing
- Chord #: explicit load balancing
- Efficient adaptation to heterogeneous hardware and item popularity
Optimizing for Latency

- Multiple datacenters
  - Only one overlay network

- Symmetric replication
  - Store replicas at consecutive nodes
  - i.e. same datacenter

- Chord # supports explicit load balancing
  - Place replicas to minimize latency to majority of clients
    - e.g. German pages of Wikipedia in European datacenters
Optimizing for Latency

Figure 3. Symmetric replication and multi-datacenter scenario. By assigning the majority of the ‘de’-, ‘nl’-, and ‘se’-replicas to nodes in Europe, latencies can be reduced.
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
Implementation

- 19,000 lines of code of Erlang
  - 2,400 lines of code for the transactional layer
  - 16,500 for the rest of the system
- 8,000 lines of code of the Java API
- 1,700 lines of code for the Python API

Each Scalaris node runs the following processes:
- Failure Detector
- Configuration
- Key Holder
- Statistics Collector
- Chord # Node
- Database
Implementation
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion

50,000 requests per second
- 48,000 handled by proxy
- 2,000 hit the DB cluster

Proxies and web servers were "embarrassingly parallel and trivia to scale"

Focus therefore was implementing the data layer
Translating the Wikipedia Data Model

Wikipedia
- SQL DB

Chord#
- Key-Value Store

CREATE TABLE /*$wgDBprefix*/page (  
  page_id int unsigned NOT NULL NULL auto_increment,  
  page_namespace int NOT NULL,  
  ...  

Map Relations to Key-Value Pairs
- (Title, List of Versions)
- (CategoryName, List of Titles)
- (Title, List of Titles) //Backlinks

<table>
<thead>
<tr>
<th>MySQL</th>
<th>Scalaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master/Slave setup</td>
<td>Chord# setup</td>
</tr>
<tr>
<td>200 servers</td>
<td>16 servers</td>
</tr>
<tr>
<td>2,000 requests</td>
<td>2,500 requests per second</td>
</tr>
<tr>
<td>Scaling is an issue</td>
<td>Scales almost linearly</td>
</tr>
<tr>
<td></td>
<td>All updates are handled in transactions</td>
</tr>
<tr>
<td></td>
<td>Replica synchronization is handled automatically</td>
</tr>
</tbody>
</table>
RoadMap

- Motivation
- Overview
- Architecture
- Features
- Implementation
- Benchmarks
- API
- Users
- Demo
- Conclusion
API - Erlang interface

\[
F = \text{fun} (\text{TransLog}) \to \\
\{X, TL1\} = \text{read}(\text{TransLog}, \text{"Account A"}), \\
\{Y, TL2\} = \text{read}(\text{TL1}, \text{"Account B"}), \\
\text{if} \\
\quad X > 100 \to \\
\quad \quad \text{TL3} = \text{write}(\text{TL2}, \text{"Account A"}, X - 100), \\
\quad \quad \text{TL4} = \text{write}(\text{TL3}, \text{"Account B"}, Y + 100) \\
\quad \quad \{\text{ok}, \text{TL4}\}; \\
\text{true} \to \\
\quad \{\text{ok}, \text{TL2}\}; \\
\text{end} \\
\text{end}, \\
\text{transaction:do_transaction}(F, \ldots).
\]
// new Transaction object
Transaction transaction = new Transaction();

// start new transaction
transaction.start();
// read account A
int accountA =
    new Integer(transaction.read("accountA")).intValue();
// read account B
int accountB =
    new Integer(transaction.read("accountB")).intValue();

// remove 100$ from accountA
transaction.write("accountA", new Integer(accountA - 100).toString());
// add 100$ to account B
transaction.write("accountB", new Integer(accountB + 100).toString());

transaction.commit();
API - Erlang

TFun = fun(TransLog) ->
    Key = "Increment",
    {Result, TransLog1} = transaction_api:read(Key, TransLog),
    {Result2, TransLog2} =
        if Result == fail ->
            Value = 1, % new key
            transaction_api:write(Key, Value, TransLog);
            true ->
                {value, Val} = Result, % existing key
                Value = Val + 1,
                transaction_api:write(Key, Value, TransLog1)
        end,
        % error handling
        if Result2 == ok ->
            {{ok, Value}, TransLog2};
            true -> {{fail, abort}, TransLog2}
        end
    end,
    % error handling
    if Result2 == ok ->
        {{ok, Value}, TransLog2};
        true -> {{fail, abort}, TransLog2}
    end
end,
SuccessFun = fun(X) -> {success, X} end,
FailureFun =
    fun(Reason) -> {failure, "test increment failed", Reason} end,
% trigger transaction
transaction:do_transaction(State, TFun, SuccessFun, FailureFun, Source_Pid).
Users

- Mostly an academic project
  - Actively developed by Zuse Institute
- onScale
  - Zuse spin-off
  - Scalarix
    - DB snapshotting
    - multi-datacenter optimization
- Eonblast
  - Scalaris fork
  - Scalaxis
    - Disk Persistence
    - External Interface, Atomic Operations, Query Extensions, more
Demo
Conclusions

- Scalable key/value store
- Strong data consistency
- Good performance
  - Wikipedia
- Implemented in Erlang
- Java API
Opinions

Joe Armstrong (Ericsson):

“So my take on this is that this is one of the sexiest applications I've seen in many a year. I've been waiting for this to happen for a long while. The work is backed by quadzillion Ph.D's and is really good believe me. “

Richard Jones (lastfm):

"Scalaris is probably the most face-meltingly awesome thing you could build in Erlang. CouchDB, Ejabberd and RabbitMQ are cool, but Scalaris packs by far the most impressive collection of sexy technologies."
Discussion

● Do we need strict consistency?
Discussion

- Does it affect performance?
Discussion

- Does it make implementation more complex?
Discussion

- Is Scalaris a practical system?