Scalaris – Methods for a Globally Distributed Key-Value Store with Strong Consistency

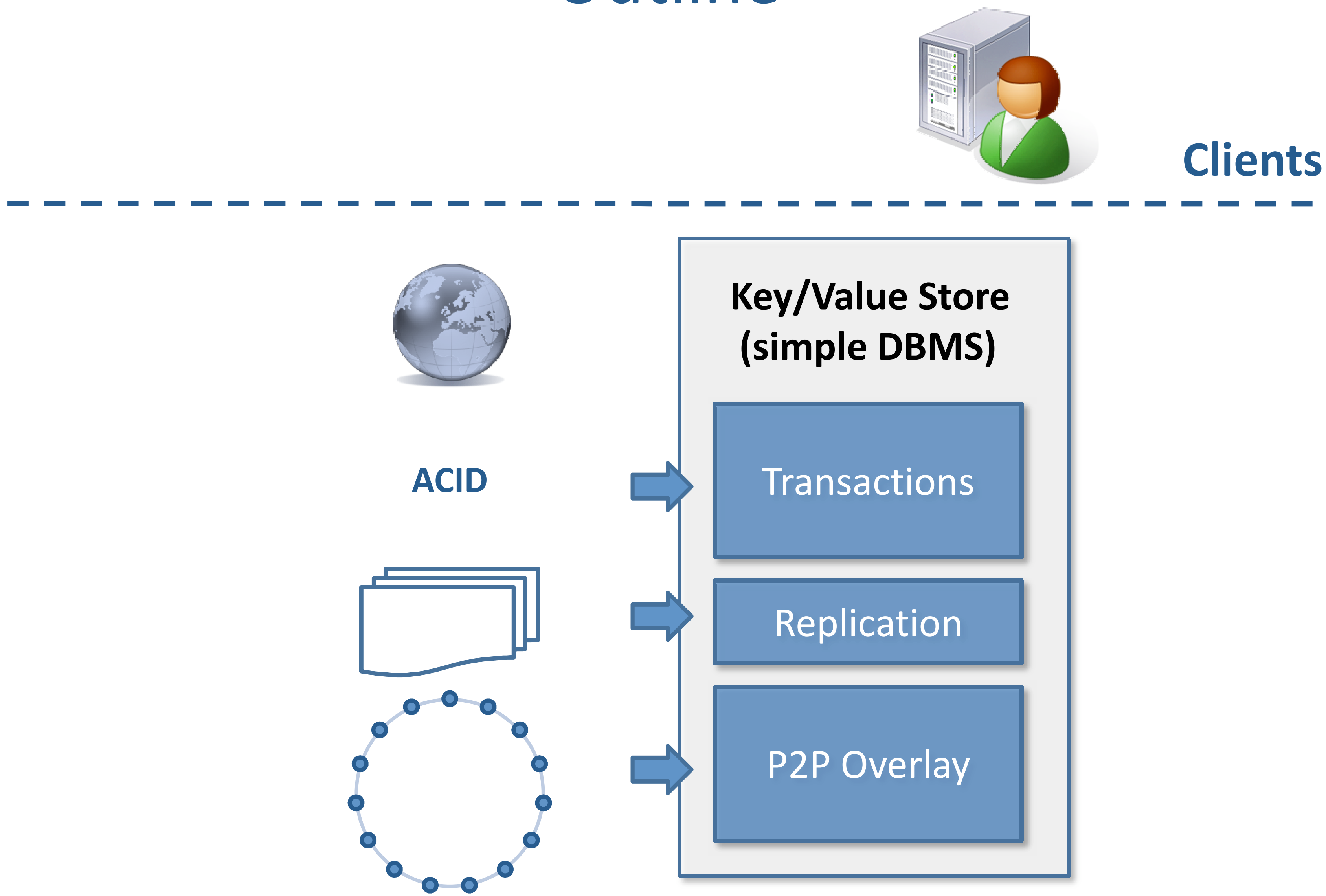
Florian Schintke
Zuse Institute Berlin

June 2009, Workshop on Data Aware Distributed Computing
(Invited Talk)

It should be clear that the DBMS community is in transition from "the old" to "the new". The next decade should be a period of vibrant activity in our field.

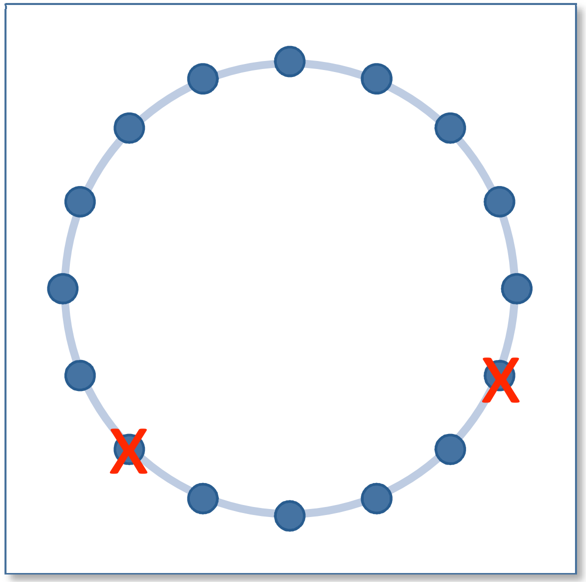
Michael Stonebraker, "One Size Fits All: An Idea Whose Time has Come and Gone", CACM 12/2008

Outline


Clients
Key/Value Store
(simple DBMS)
ACID
Transactions
Replication
P2P Overlay

scalability and self-management

P2P LAYER



P2P Layer

- implements a primitive key/value store
  - synonyms: “key/value store” = “dictionary” = “map”, = …
- just 3 ops
  - `insert(key,value)`
  - `delete(key)`
  - `lookup(key)`

Key	Value
Backus	1977
Hoare	1980
Karp	1985
Knuth	1974
Wirth	1984
. . .	. . .

Example:
A Key/value store with all
Turing award winners

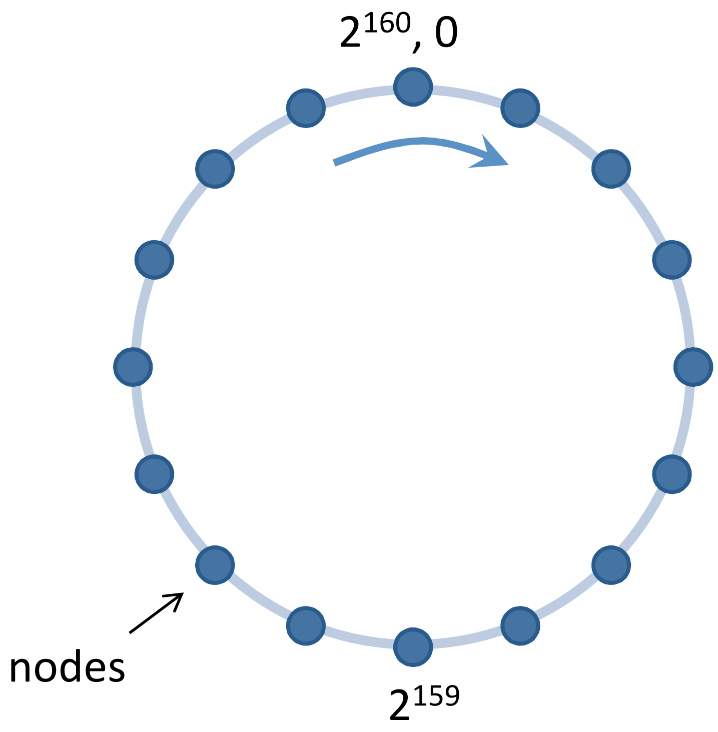
Nodes Maintain a Ring Structure

Keys

- define positions in the ring, e.g. $0 – 2^{160}$ or strings

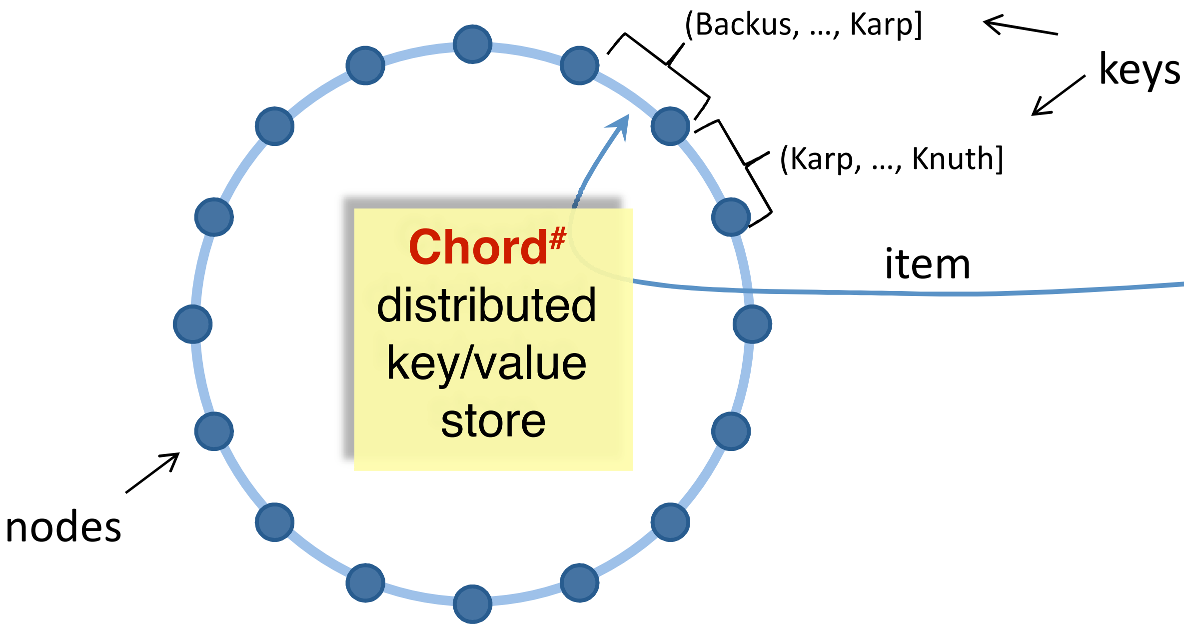
Nodes

- may join, leave or fail (churn)
- know several successors
- know their predecessor
- have random position in the ring



P2P Layer with Chord#

- Chord# uses keys directly as addresses in the ring
  - no hashing, thereby order-preserving → enables range queries
  - just need a total order on items
- The next node in the ring (clockwise) is responsible for a key



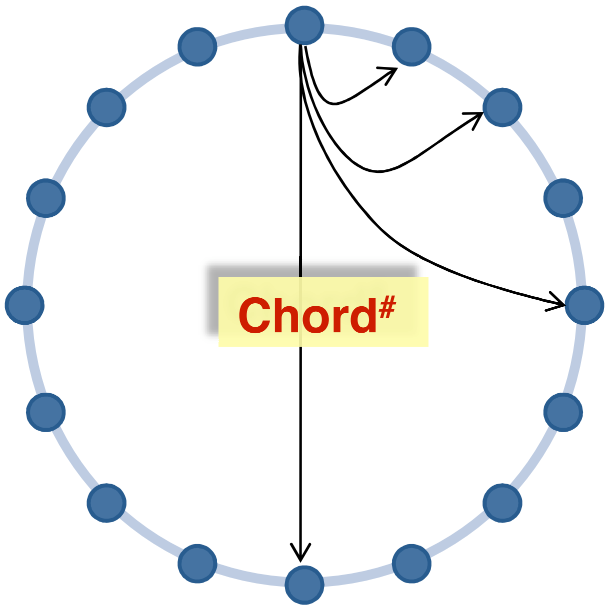
Key	Value
Backus	1977
Hoare	1980
Karp	1985
Knuth	1974
Wirth	1984
. . .	. . .

Routing Table and Data Lookup

Routing Table

- table contains $\log_2 N$ pointers
- pointers are exponentially spaced

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1} \,.\, pointer_{i-1} & : i \neq 0 \end{cases}$$

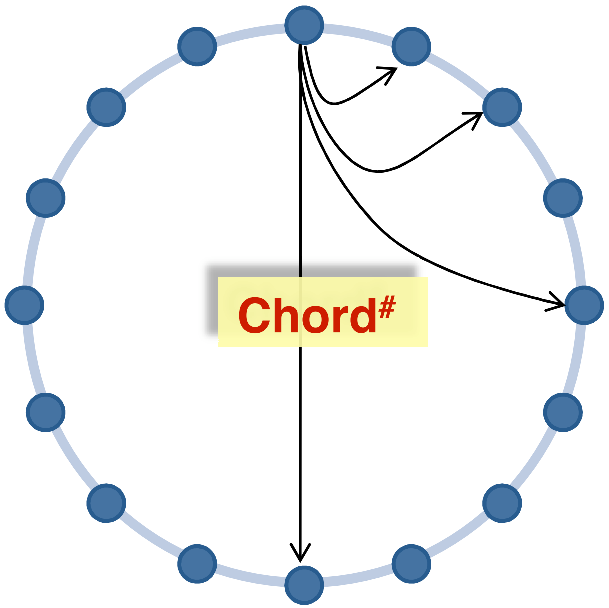


Routing Table and Data Lookup

Routing Table

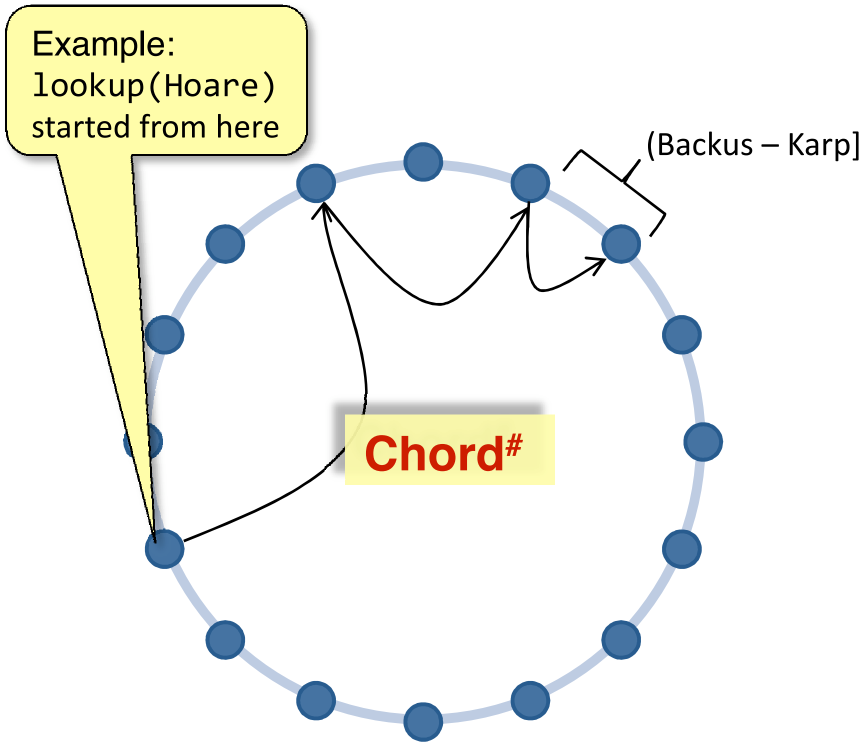
- table contains $\log_2 N$ pointers
- pointers are exponentially spaced

$$pointer_i = \begin{cases} successor & : i = 0 \\ pointer_{i-1} \cdot pointer_{i-1} & : i \neq 0 \end{cases}$$



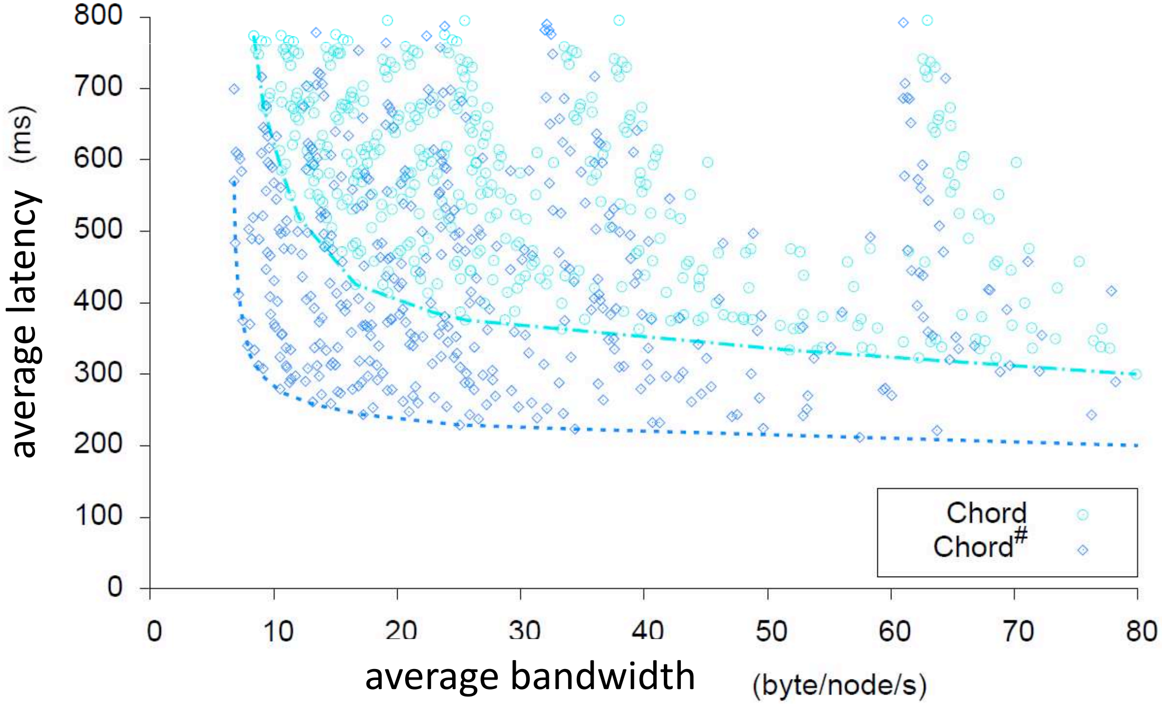
Retrieving Items

- $\leq \log_2 N$ hops



P2P Layer with Chord#

- Chord# features
  - fully decentralized, operations require only local knowledge
  - self-organizes as nodes join, leave, and fail
  - easy routing table maintenance
  - ≤ log (N) hops
  - range queries



Failure detector

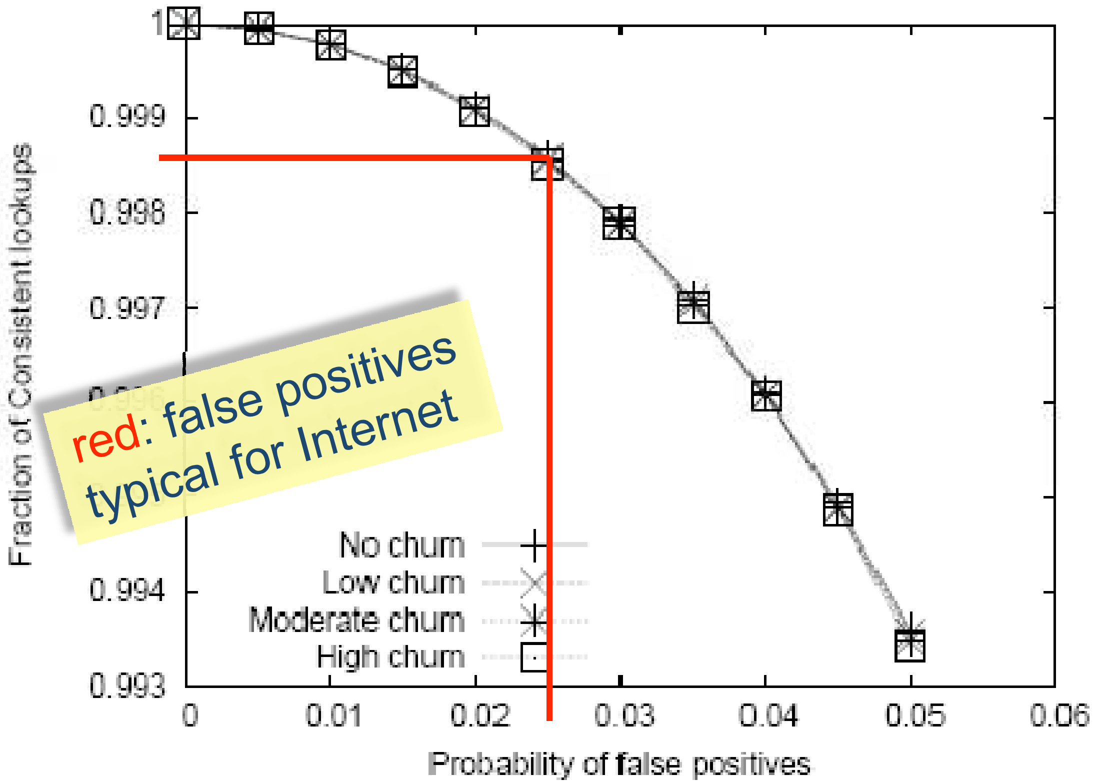
- Need a **failure detector** to check for aliveness of nodes.

- But failure detector may be wrong.
  - Node dead? Or just slow?
  - **Even without churn, inconsistencies may occur!**

- Two types of inconsistency
  - responsibility inconsistency
  - lookup inconsistency

How often does this occur?

- We simulated nodes with imperfect failure detectors
  (A node detects another alive node as dead probabilistically)

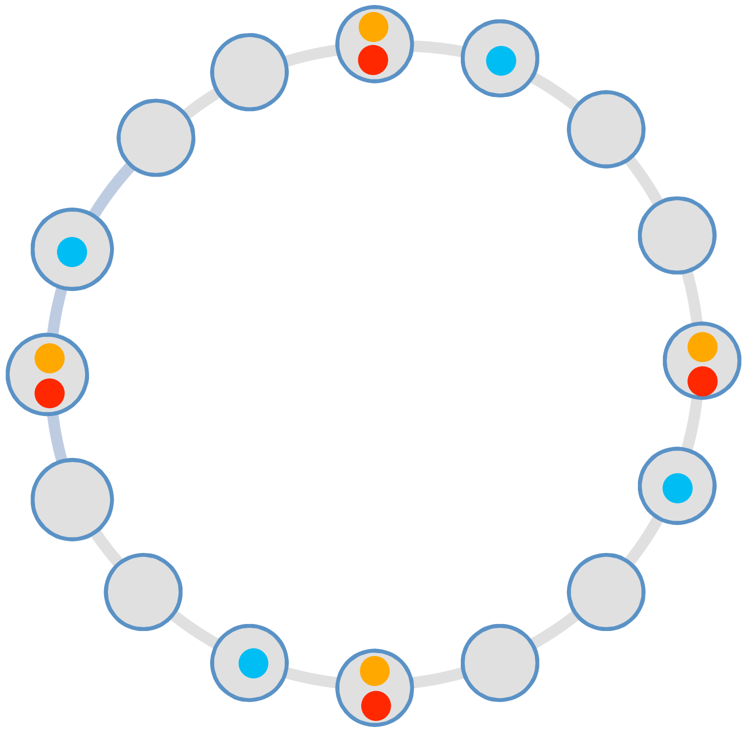


providing data availability

REPLICATION LAYER

Replication

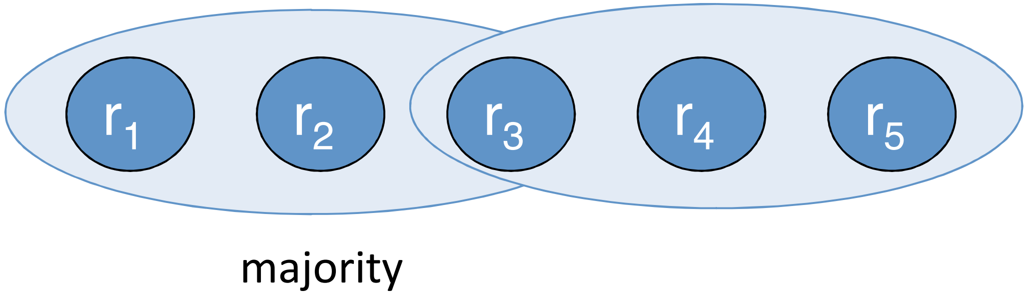
- We use symmetric replication
  - Use a globally known function to determine a set of keys under which the data is stored →



- Must ensure data consistency
  - need quorum-based methods

Replication and Quorum-based Algorithms

- Only read/write on majorities:

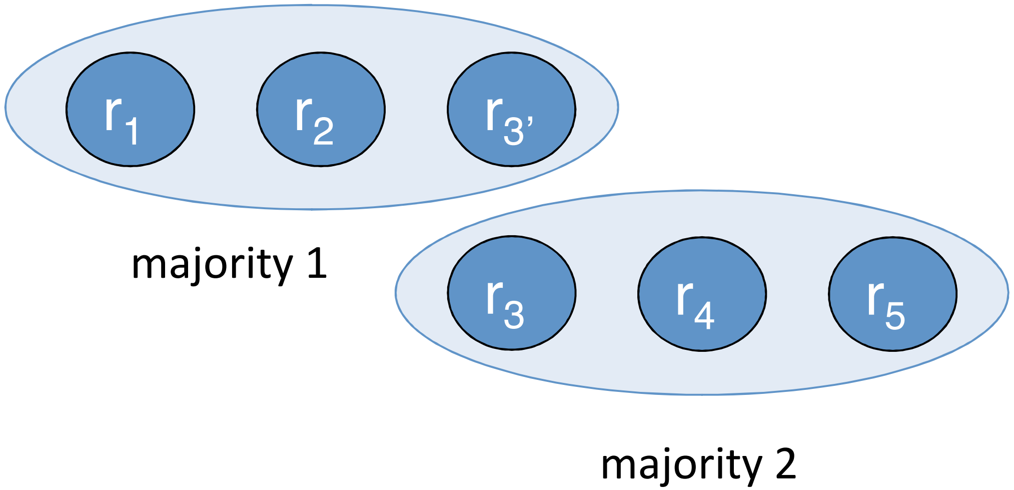


- Concurrent operations have overlapping majorities
  => conflict detection

- Comes at the cost of increased latency
  - but latency can be avoided by intelligent distribution over datacenters

Replication and Quorum-based Algorithms

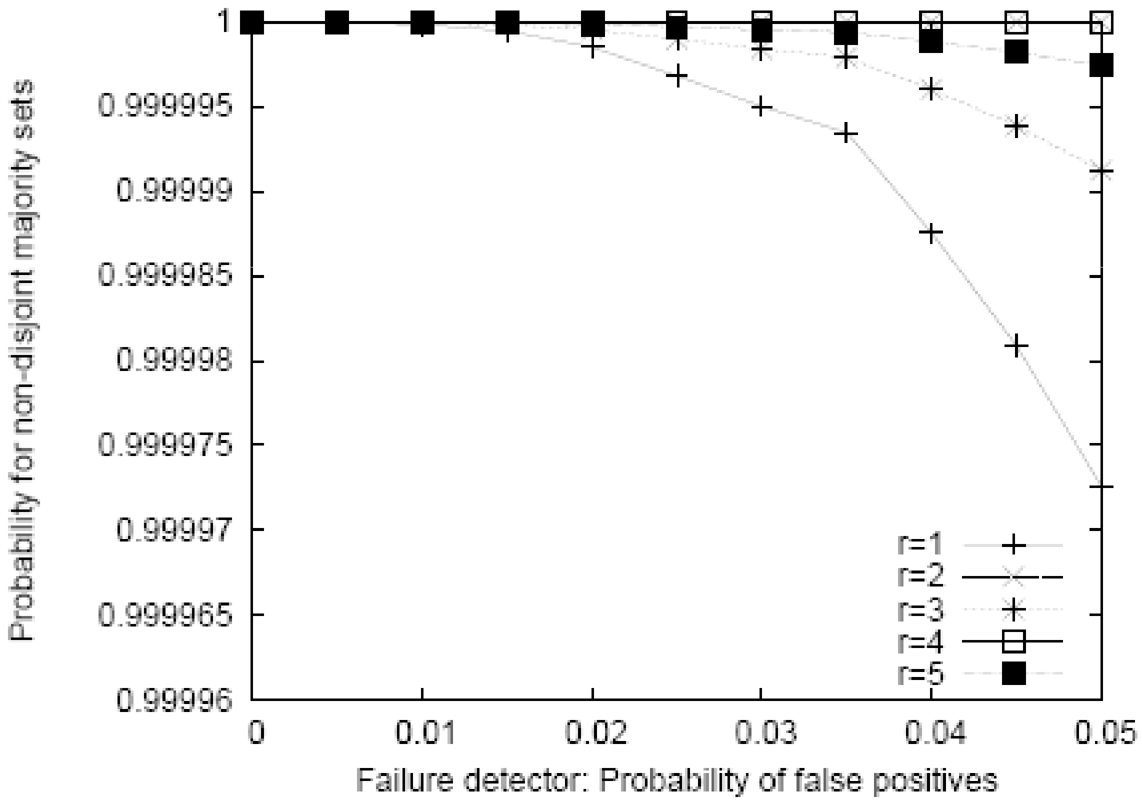
- Lookup inconsistency may result in more than f replicas
- Then, two (or more) non-overlapping majorities exist:



=> Must ensure that number of replicas is always $\leq$ f when using *simple* majority access

- relaxed with stronger majorities

More consistent accesses with replication and quorum access



coping with concurrency

TRANSACTION LAYER

Challenges for Transactions in SONs

- churn
  - nodes may leave, join, or crash at any time
    → changing responsibilities

- "crash stop" fault model

- no perfect "failure detector"
  - never know whether a node crashed or just slow network

Goal: Strong Consistency

- What is it?
  - When a write is finished, all following reads will return the new value.

- How to implement?
  - Always read/write majority $\lfloor f/2 \rfloor + 1$ of f replicas.

    => **Latest version** is always in the read/write set

Goal: Atomicity

- What is it?
  - Make **all** or **no** changes!
  - Either ‘commit’ or ‘abort’.

- How to implement?
  - 2PC? Blocks if the transaction manager fails.
  - We use a variant of **Paxos Commit**
    - non blocking, because of *multiple acceptors*

Transactions + Replicas

BOT	BOT
	`debit (`a_1`, 100);`
`debit (a, 100);`	`debit (`a_2`, 100);`
	`debit (`a_3`, 100);`
	`deposit (`b_1`, 100);`
`deposit (b, 100);`	`deposit (`b_2`, 100);`
	`deposit (`b_3`, 100);`
EOT	EOT

Scalaris Transactions in Erlang

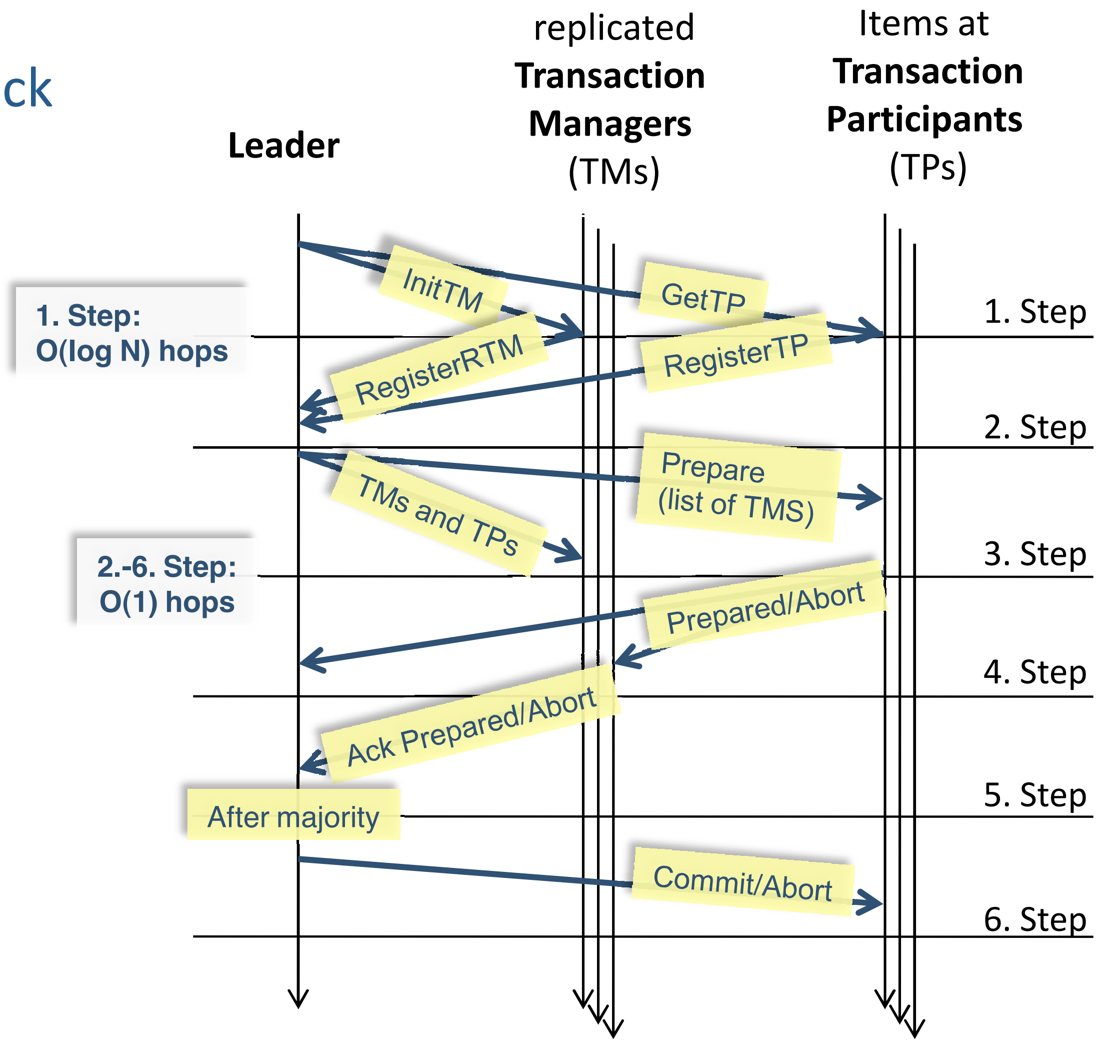
```
F = fun (TransLog) ->
     {X, TL1} = scalaris:read(TransLog, "Account A"),
     {Y, TL2} = scalaris:read(TL1, "Account B"),
      if
         X > 100 ->
              TL3 = scalaris:write(TL2, "Account A", X - 100),
              TL4 = scalaris:write(TL3, "Account B", Y + 100),
              {ok, TL4};
         true ->
              {ok, TL2};
      end
end,
MyTransLog = F(EmptyTransLog),


scalaris:commit(MyTransLog).
```

Build translog with quorum reads:
for a read: (value, version)
for a write: (value, quorum read version + 1)
+ infos on read/write locks

Validate and commit transaction using Paxos commit.

Adapted Paxos Commit

- Optimistic CC with fallback
- Write
  - 3 rounds
  - non-blocking (fallback)
- Read even faster
  - reads majority of replicas
  - just 1 round
- succeeds when
  >f/2 nodes alive



scaling without borders

MULTI DATACENTER DEPLOYMENT

Requirements

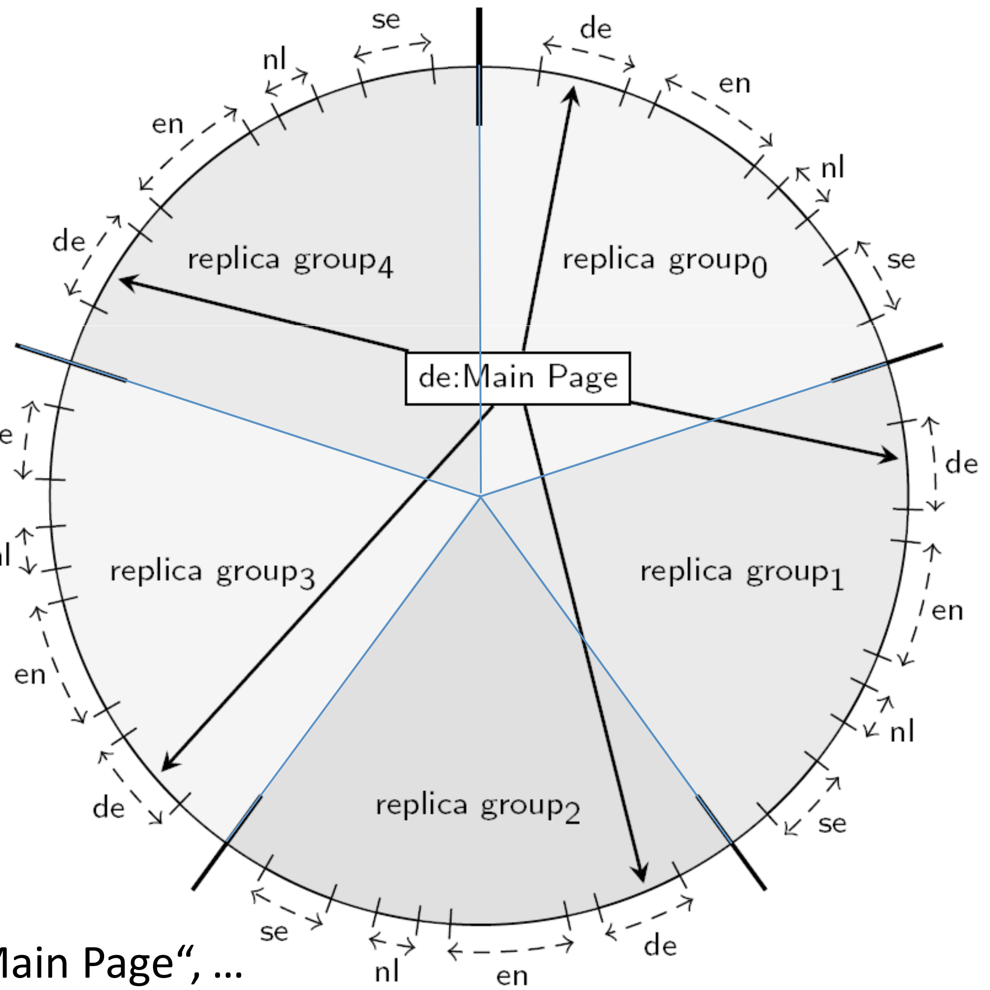
- Minimize inter datacenter traffic
- Minimize query latency
- Tolerate network partitioning
- Handle multiple applications with *one* overlay (reduced operation cost)

Multi Data-Center Scenario

- Multi-Data-Center Scenarios
  - Optimize for Latency
  - Increase Availability

- Prefix Articles with
  - Language
  - Replicas Number

- E.g. „de:Main Page“
  - 5 replicas
  - 2 in Germany
  - 1 in UK
  - 1 in USA
  - 1 in Asia



„0de:Main Page“, „1de:Main Page“, „2de:Main Page“, …

Tolerate Network Partitioning

Overlay

- Hierarchical Overlay
- Reconciliation
  - Gossip based ring maintenance
  - (T-MAN)

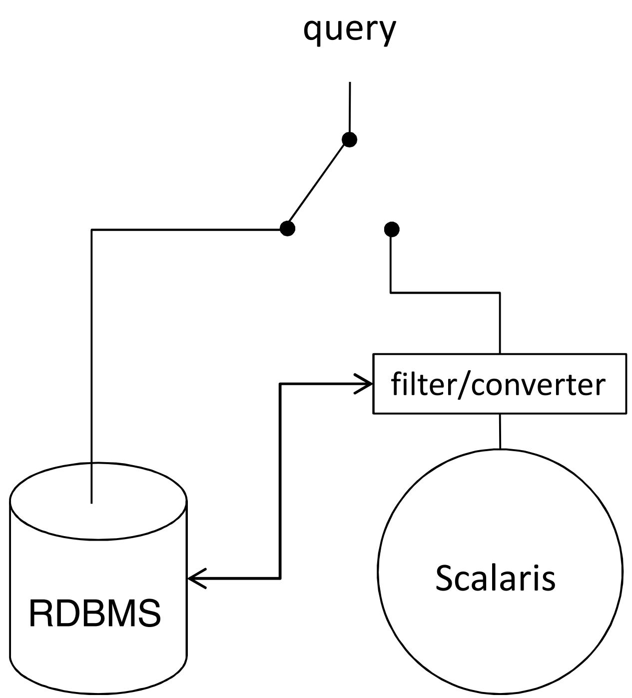
Data

- Transaction Algorithm
  - Majority based

Migration Strategy

- Migrating your SQL data to Scalaris
  a) offline migration
    1. stop SQL
    2. take dump
    3. start Scalaris
    4. replay dump

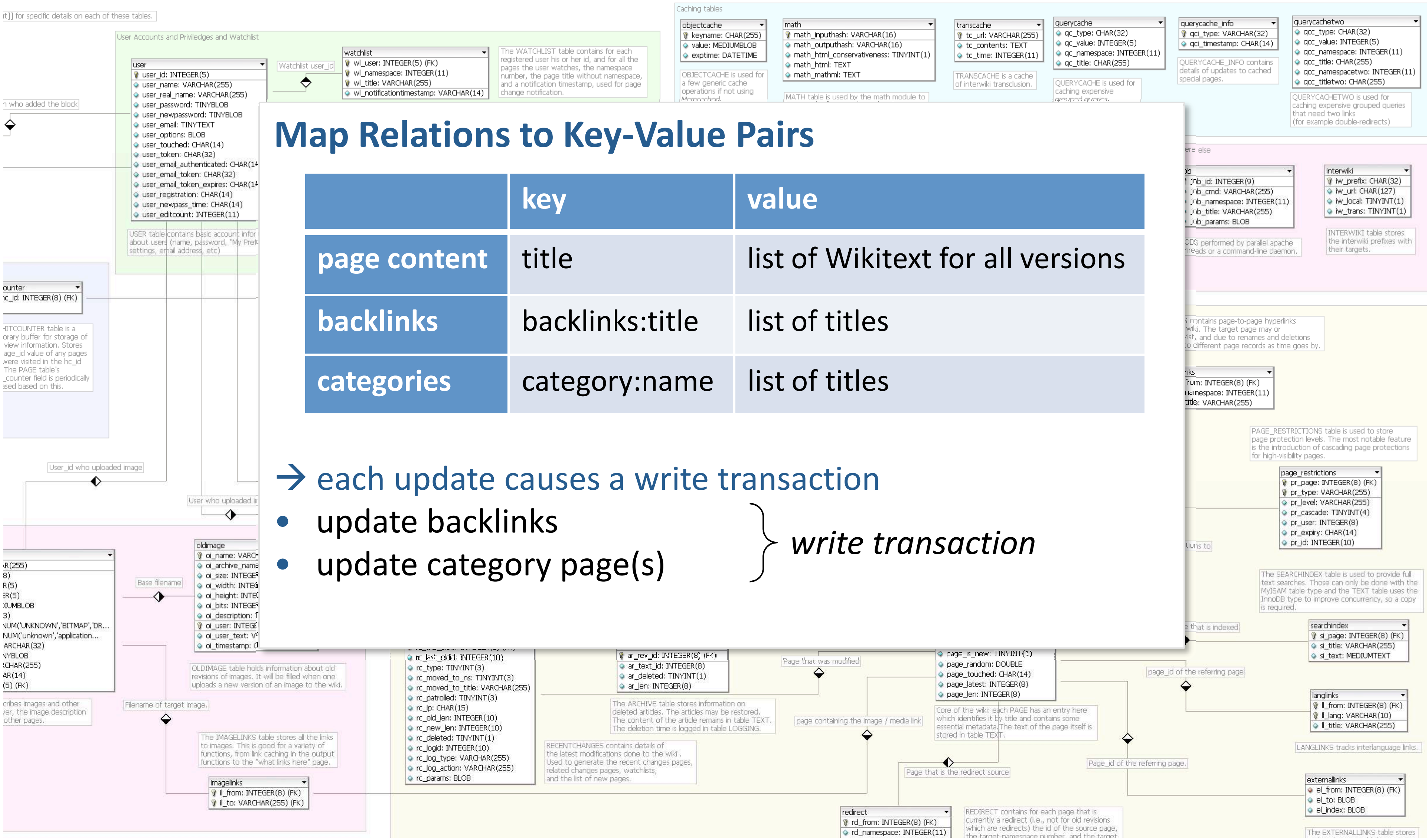
  b) online migration
    - build Scalaris around SQL server →



Proof-of-Concept

WIKIPEDIA

Wikipedia SQL DB → Key/Value Store



Map Relations to Key-Value Pairs

	key	value
page content	title	list of Wikitext for all versions
backlinks	backlinks:title	list of titles
categories	category:name	list of titles

→ each update causes a write transaction

- update backlinks
- update category page(s)

} *write transaction*

Scalaris Implementation

- Scalaris: 9,700 lines of Erlang code
  - 7,000 for Chord# and infrastructure
  - 2,700 for transactions

- Application specific code
  - 1,300 for our Wikipedia code
  - Java for rendering and user interface

Team

- Thorsten Schütt
- Florian Schintke
- Monika Moser
- Stefan Plantikow
- Christian Hennig
- Alexander Reinefeld
- Nico Kruber
- Christian von Prollius
- Seif Haridi (SICS)
- Ali Ghodsi (SICS)
- Tallat Shafaat (SICS)

Summary

- data aware P2P for distributed data
- Scalaris supports consistent, distributed data access

Scalaris = Scalability + Consistency + Availability

scalaris.googlecode.com