WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Object oriented Development of Distributed Applications (Slide-set 5)

Gustaf Neumann

Information Systems and New Media

Learning Objectives (Slide-set 5)

- Learn the challenges of large scale data persistence ("Big-Data")
- Asses how distributed data stores overcome the limits of vertical scaling
- Understand the practical implication of the CAP theorem
- Learn Conceptual differences between NoSQL databases
- Asses the effects of different integrity policies
- Learn how to develop highly scalable dynamic web-sites based on Open-Source technology

Overview (Slide-set 5)

- Large Scale Data Persistence
- Vertical and Horizontal Scalability
- CAP Theorem
- NoSQL Database Concepts
- Redis, Cassandra, MongoDB
- Consistency Models of NoSQL Database Systems
- NoSQL Data Models and Dynamic Languages
- Build Web-Applications with NaviServer and NX based on the Business Informer Data Model

Large-scale Data Persistence

Classical Databases

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

- Relational Databases, SQL
- ACID Properties:
  - Atomicity: every (maybe complex) transaction is executed completely (*undividable*) or not (rollback)
  - Consistency: Database transactions move database from one valid state to the next; no constraints are violated, cascading operations are completed, …
  - Isolation: independence of concurrent transactions, serializability (no dirty reads, no non-repeatable reads, no phantoms from other transactions)
  - Durability: when a commit is performed, the data stays persistently stored (crash-safety; after commit, data has to be saved at the storage medium)

Scaling and Relational Databases

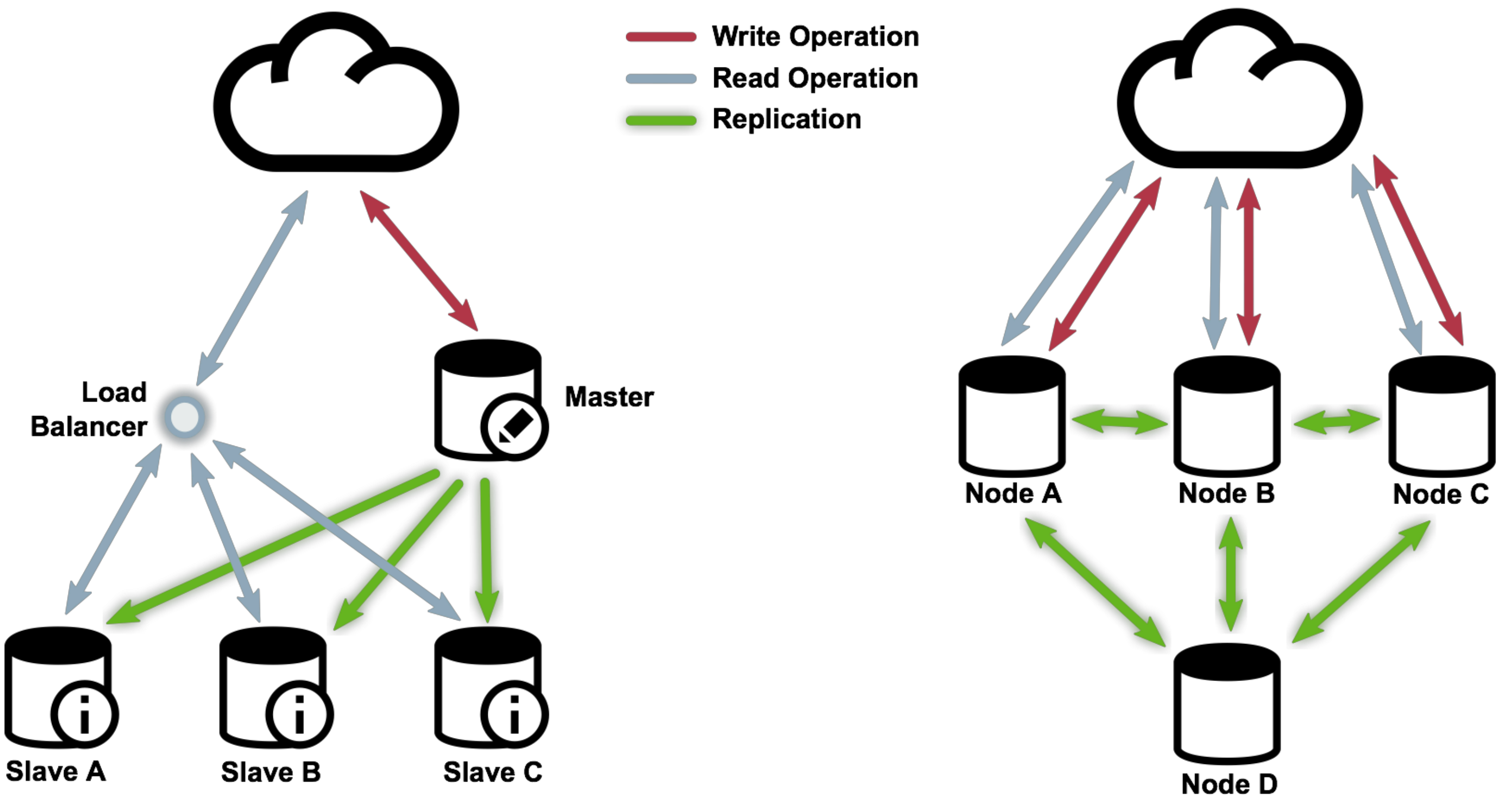
- Core asset of relational databases:
  - ACID Properties
  - Strong Consistency
- State-of-the-Art since many years for single-nodes

- Problems:
  - What if a single node cannot keep all data?
  - How to achieve high availability?

Vertical and Horizontal Scaling

- Vertical Scaling (scale up):
  - Add more resources to a system (CPUs, cores, memory, disks, ….)
- Horizontal Scaling (scale out):
  - Add more nodes (distributed systems)
  - For databases:
    - **Replication:** multiple copies of the same data
    - **Sharding (partitioning):** partial data on different nodes

Replication Patterns



Master-Slave Replication

Multi-Master Replication

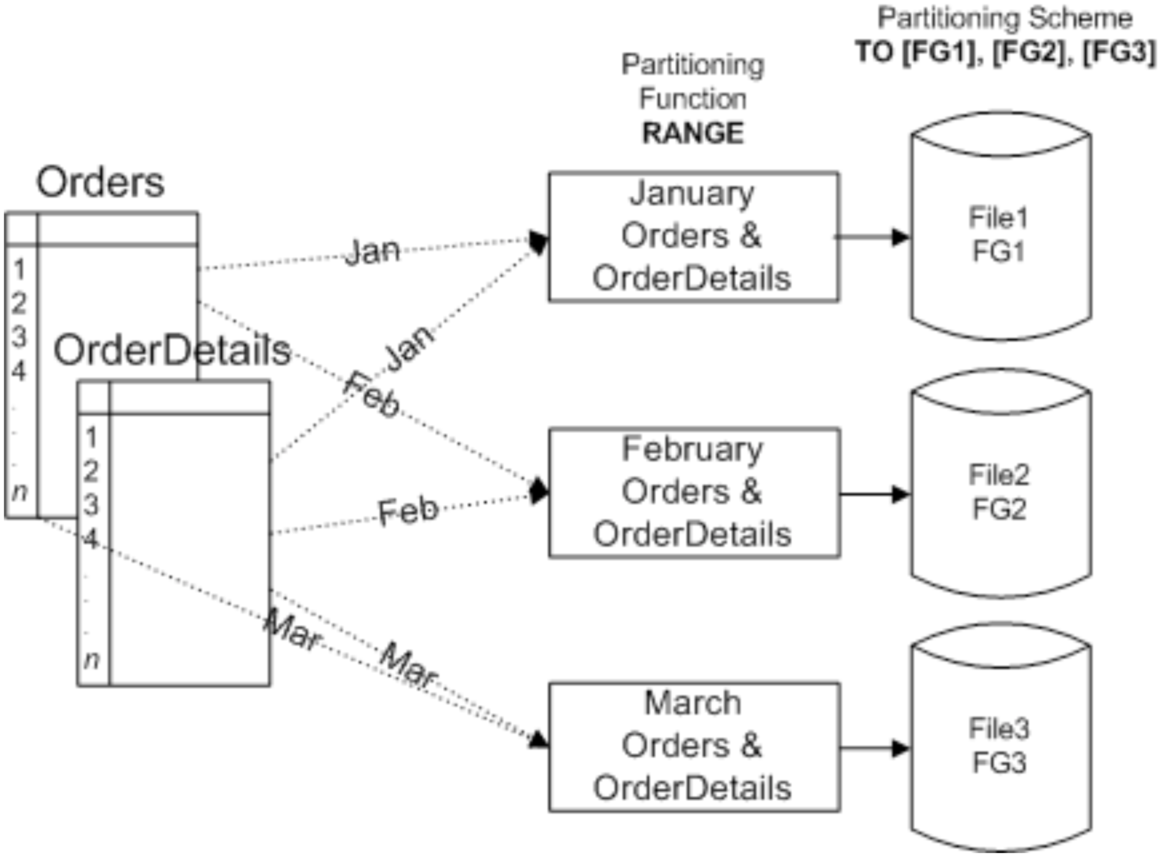
Multi-Master Replication

- Mechanism
  - Write operations to multiple nodes
  - Nodes of a cluster are responsible to propagate data other nodes
- Consequences
  - Write becomes horizontally scalable
  - Needs conflict resolution strategies, otherwise simultaneous writes on the same data may lead to inconsistencies
  - Eager Propagation causes latencies
  - For high number of nodes, conflict resolution tends to become intractable
  - Many systems loose ACID properties in trade for performance

Master-Slave Replication

- Mechanism
  - All write operations are issued to a single node (master)
  - Master node propagates data to all slaves
  - Read operations can be issued to multiple nodes (slaves)
- Consequences
  - Only benefits for read operations
  - Propagation needs synchronization to achieve ACID properties
  - Large of frequent write operations kill performance
  - If master is down, no writes are possible

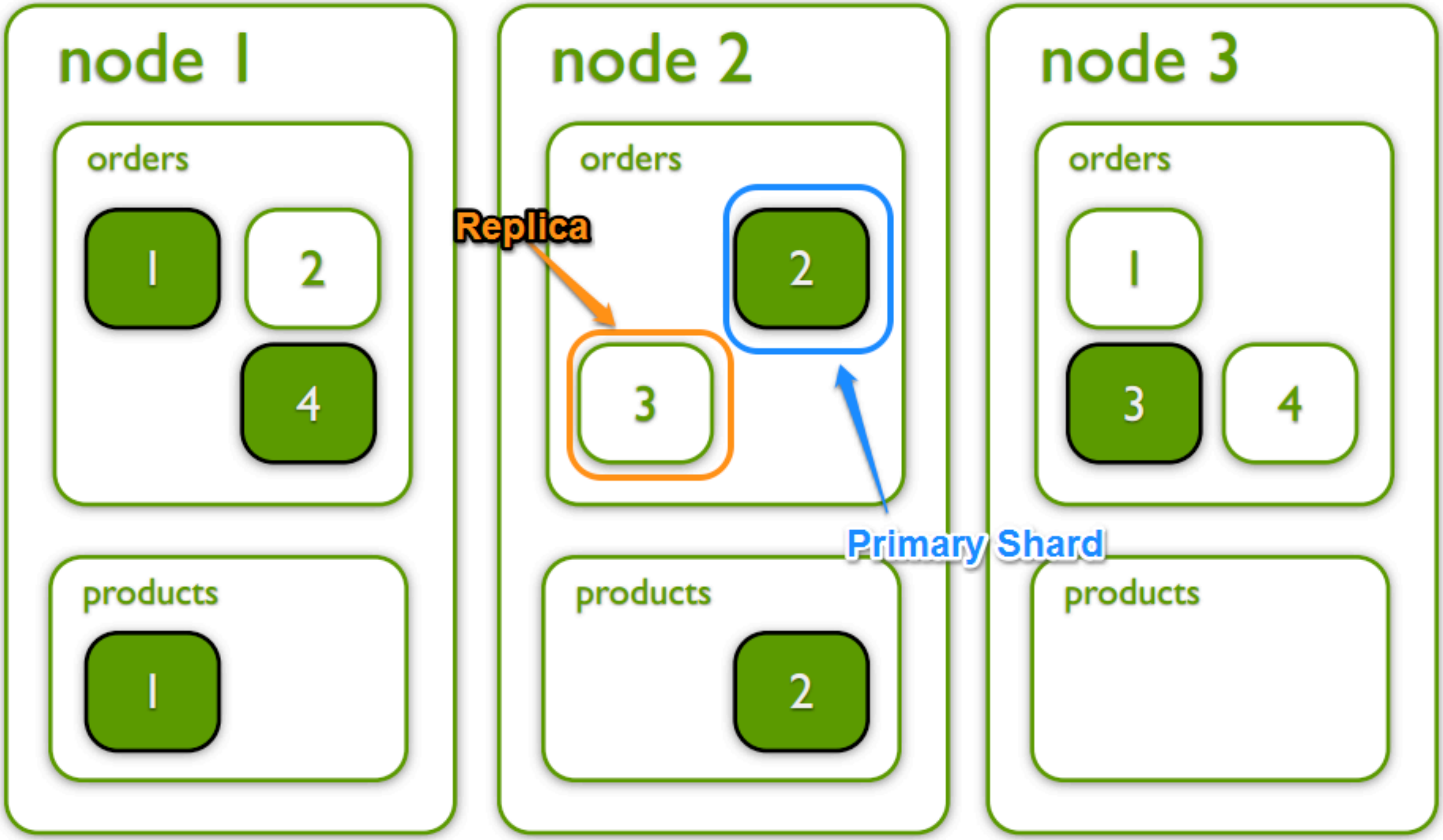
Sharding (Partitioning)



Sharding (Partitioning)

- Meaning
  - Store huge data sets across multiple machines
  - Can handle huge amounts of data that could not be kept on a single machine
- Consequences
  - Some parallelization possible (e.g. sequential search)
  - Smaller indices
  - Partitioning function has to decide in which shard data is stored
  - A single query might have to use multiple shards
  - Application has to be partition aware
  - If a shard is not available, queries might stall
  - Referential integrity is hard to achieve

Sharding and Replicas in Elastic Search



Common Properties of Horizontal Scaling

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

- Multiple nodes are involved
- Nodes need synchronization and locking
- The more nodes involved the more the likelihood of node failures increases (e.g. over internet connections)
- ACID properties and low latencies are very hard to achieve

EFMD EQUIS ACCREDITED

Tradeoff between Consistency (ACID) and Availablility (BASE)

- BASE:
  - Basically Available
  - Soft-State
  - Eventually consistency (after some time consistent)
- BASE properties:
  - Weak consistency (stale reads are ok)
  - Approximate answers are ok
  - Availability first (best effort, optimistic policies, faster, simpler, easier evolution)

Consistency Models

- ***Strong Consistency*:** After the update completes any subsequent access is *guaranteed* to return the updated value.
- ***Weak Consistency*:** The system *does not guarantee* that subsequent accesses will return the updated value. Typically after some time (the *inconsistency window*) the correct value will be returned.
- ***Eventual Consistency*:** Special form of the weak consistency: The storage system *guarantees* that if no new updates are made to the object, eventually all accesses will return the last updated value.

  Examples:
  - DNS (Domain Name System): Updates to a name are distributed according to a configured pattern and in combination with time-controlled caches; eventually, all clients will see the update.
  - Asynchronous master/slave replication on an RDBMS (also on MongoDB)
  - Caching (e.g. in front of a database) with memchached

Variations of Eventual Consistency

- **Causal consistency:** If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.

- **Read-your-writes consistency:** This is an important model where process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.

- **Session consistency:** This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.

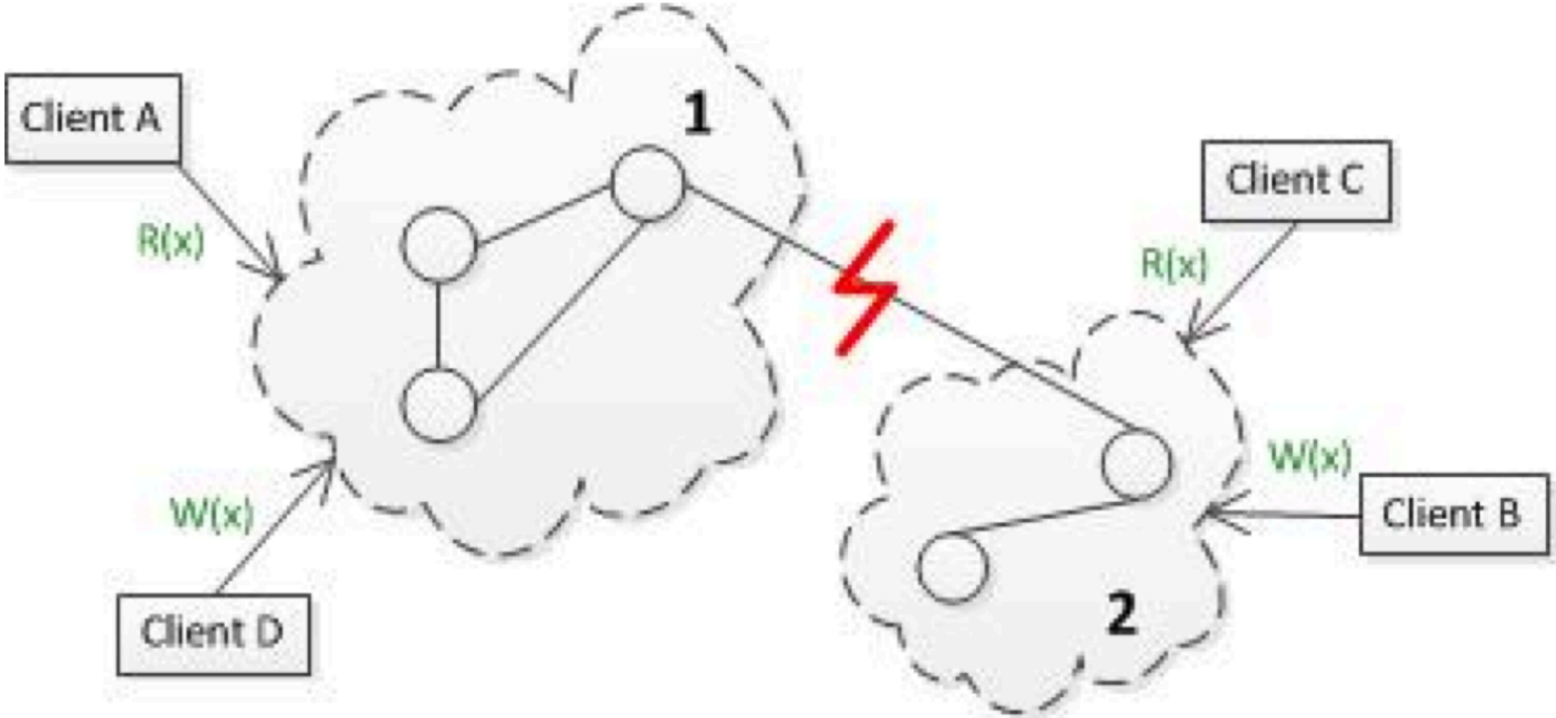
- **Monotonic read consistency:** If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

- **Monotonic write consistency:** In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

CAP Theorem

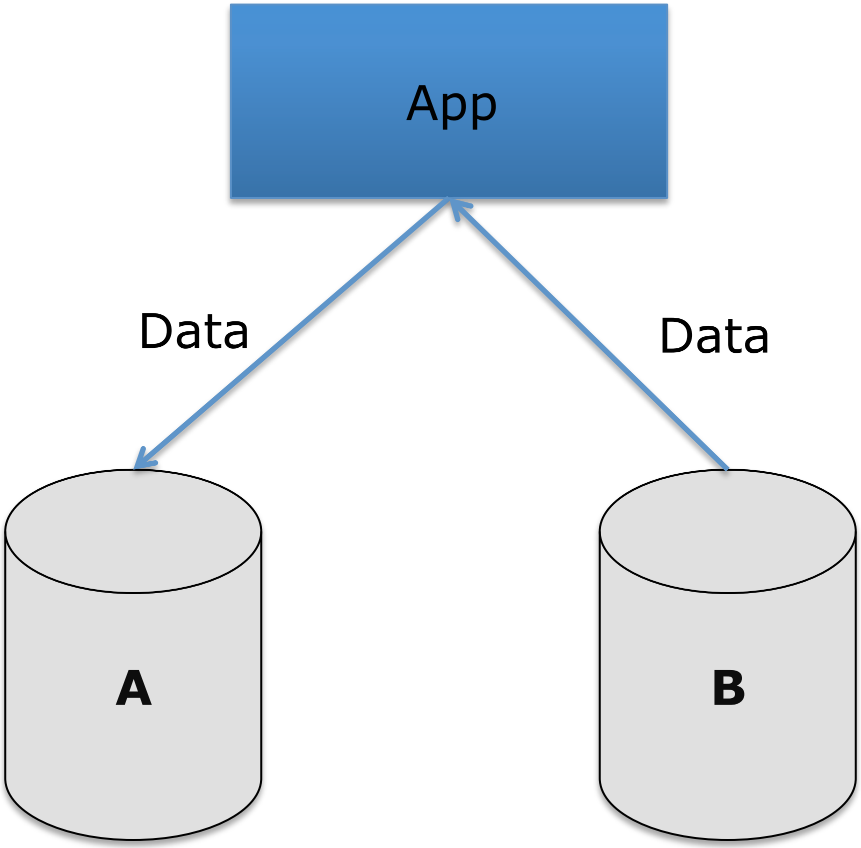
- CAP:
  - Consistency
  - Availability
  - Tolerance to network partitions
    (nodes become unreachable, e.g. network failures)
- Theorem (Brewer 2000):
  - One can have **at most two** of these properties for any shared-data system
  - PODC Keynote 2000 (Principles of Distributed Computing)

Network Partitioning



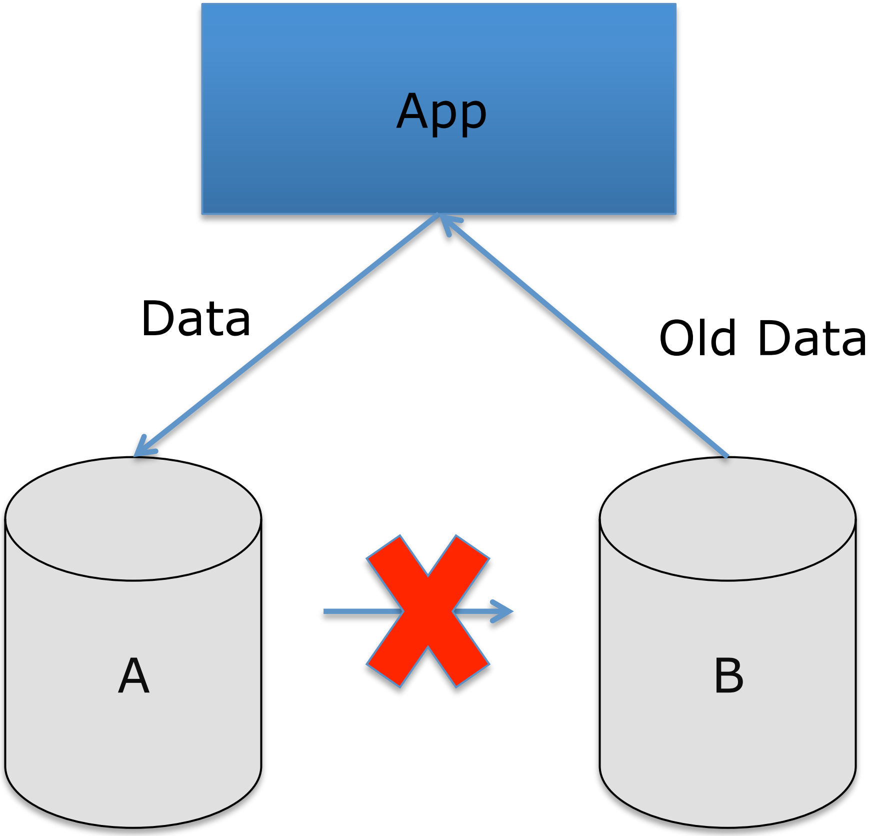
CAP Scenarios: C + A + P

Consistent and available
No partition.



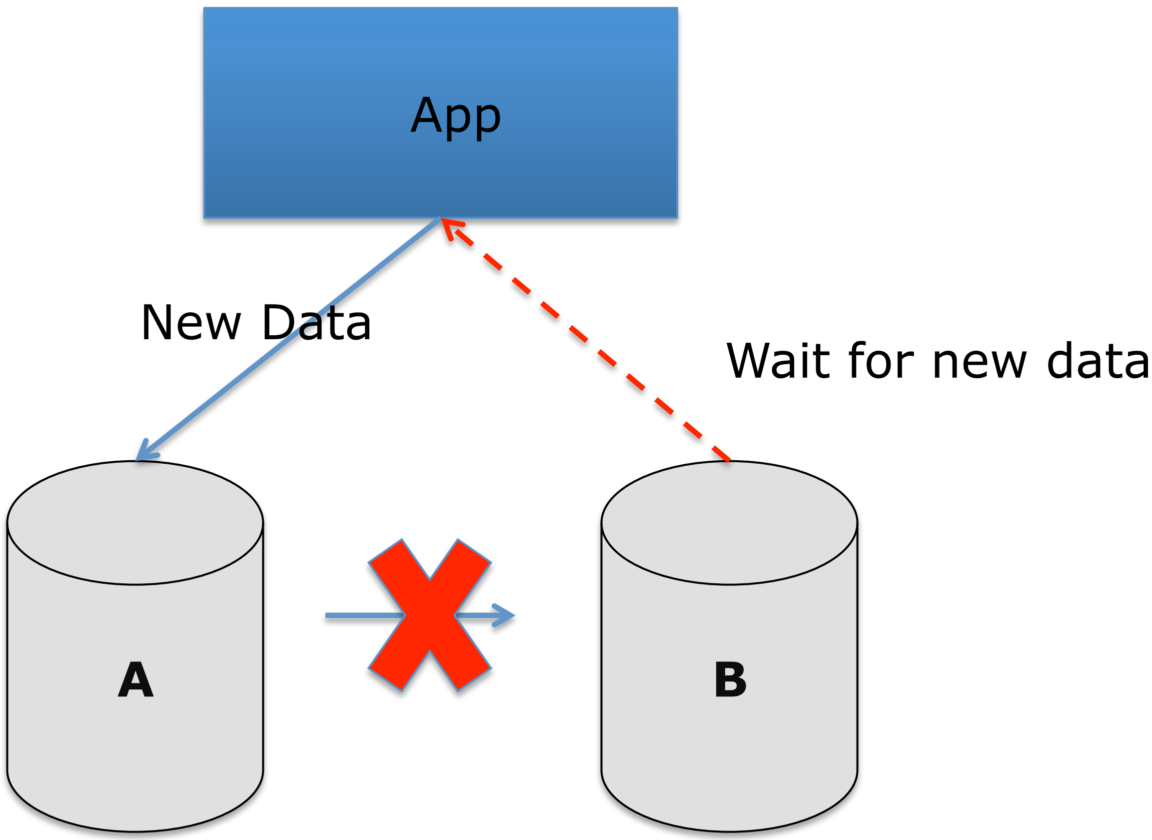
CAP Scenarios: A + P

Available and partitioned
Not consistent, we get back old data.

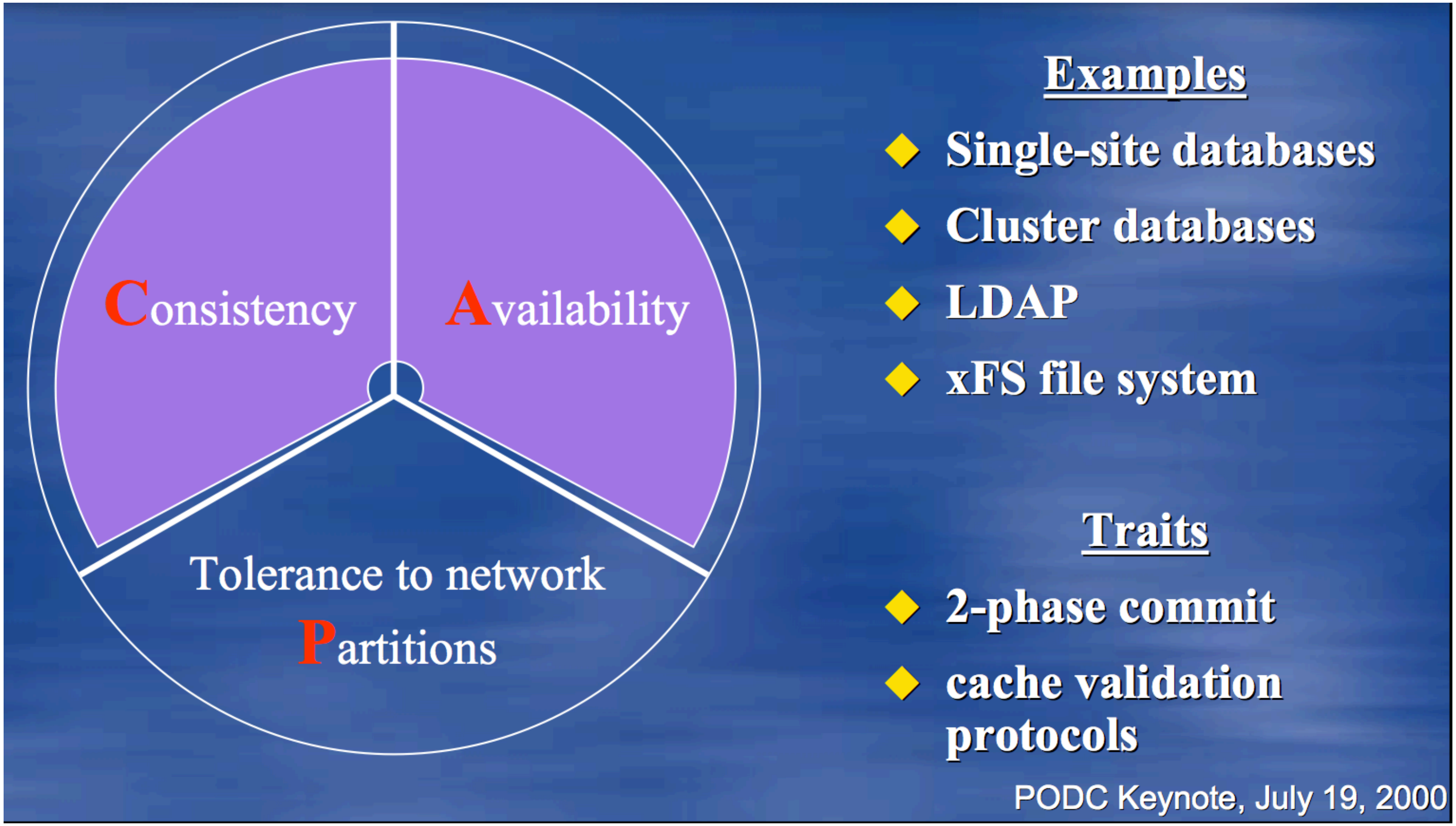


CAP Scenarios: C + P

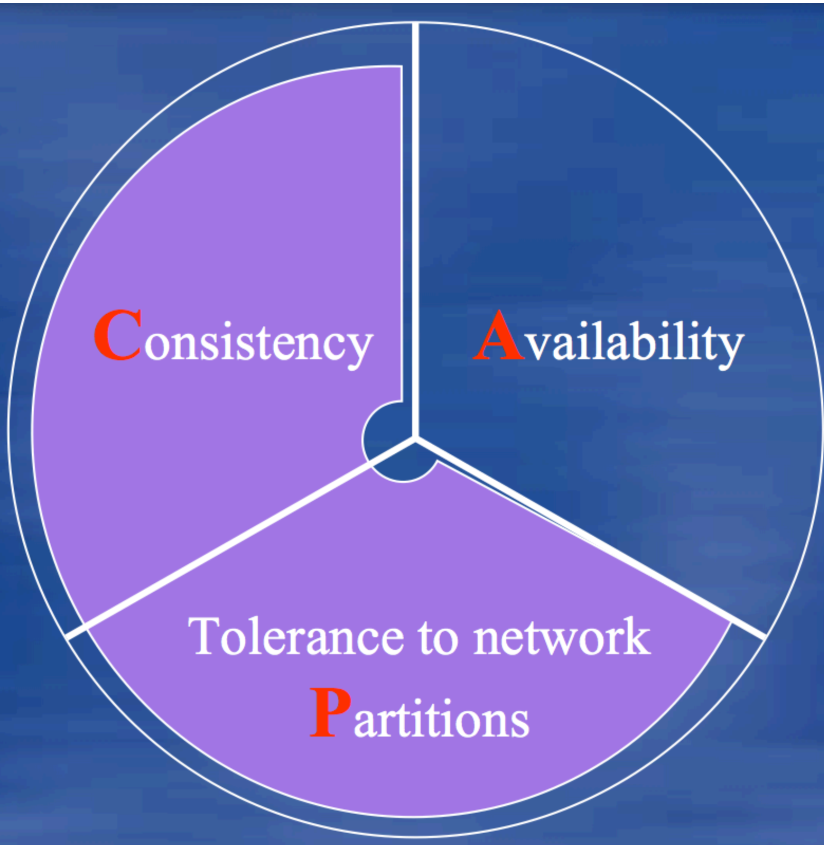
Consistent and partitioned
Not available, waiting…



Systems Achieving only C + A



Systems Achieving only C + P



Examples

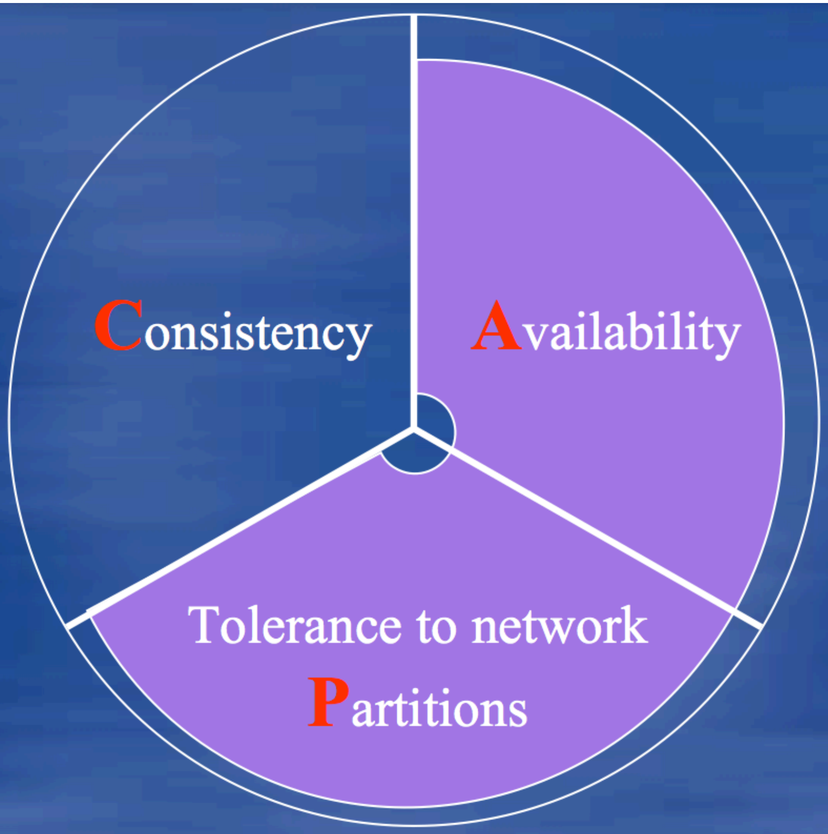
- Distributed databases
- Distributed locking
- Majority protocols

Traits

- Pessimistic locking
- Make minority partitions unavailable

PODC Keynote, July 19, 2000

Systems Achieving only A + P



Examples

- Coda
- Web caching
- DNS

Traits

- expirations/leases
- conflict resolution
- optimistic

PODC Keynote, July 19, 2000

NoSQL Databases

- Since relational databases theory (and SQL) requires ACID properties, the CAP theorem caused and advent (or revival) of NoSQL databases
- NoSQL Databases sacrifice strong consistency for availability when multiple replicas are involved

Post on your blog!

Top Job Trends

1. HTML5
2. **MongoDB**
3. iOS
4. Android
5. Mobile app
6. Puppet
7. Hadoop
8. jQuery
9. PaaS
10. Social Media

NoSQL Databases

- 3 main groups:
  - Key-value Databases

    Examples: Memcached, **Redis**, Tokyo Cabinet, Dynamo

  - Column-oriented Databases

    Examples: BigTable, **Cassandra**, HBase

  - Document Databases

    Examples: **MongoDB**, CouchDB

DB Engines Ranking

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

210 systems in ranking, January 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	Oracle	Relational DBMS	1467.79	-0.26
2.	2.	MySQL	Relational DBMS	1296.91	-12.38
3.	3.	Microsoft SQL Server	Relational DBMS	1226.02	+20.14
4.	4.	PostgreSQL	Relational DBMS	228.25	-2.71
5.	5.	DB2	Relational DBMS	188.31	-2.30
6.	6.	MongoDB	Document store	178.23	-4.84
7.	7.	Microsoft Access	Relational DBMS	174.99	+3.32
8.	8.	SQLite	Relational DBMS	97.30	-2.20
9.	9.	Sybase	Relational DBMS	94.51	-0.77
10.	10.	Cassandra	Wide column store	81.18	+0.67
11.	11.	Teradata	Relational DBMS	61.45	-2.27
12.	12.	Solr	Search engine	60.33	-2.12
13.	13.	Redis	Key-value store	52.49	+0.72

Ranking based on web sites, Google Trends, technical discussions (stack overflow), job offers, job profiles)

DB Engines Ranking

339 systems in ranking, December 2017

Rank			DBMS	Database Model	Score		
Dec 2017	Nov 2017	Dec 2016			Dec 2017	Nov 2017	Dec 2016
1.	1.	1.	Oracle	Relational DBMS	1341.54	-18.51	-62.86
2.	2.	2.	MySQL	Relational DBMS	1318.07	-3.96	-56.34
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1172.48	-42.59	-54.17
4.	4.	4.	PostgreSQL	Relational DBMS	385.43	+5.51	+55.41
5.	5.	5.	MongoDB	Document store	330.77	+0.29	+2.09
6.	6.	6.	DB2	Relational DBMS	189.58	-4.48	+5.24
7.	7.	↑ 8.	Microsoft Access	Relational DBMS	125.88	-7.43	+1.18
8.	↑ 9.	↑ 9.	Redis	Key-value store	123.24	+2.05	+3.34
9.	↓ 8.	↓ 7.	Cassandra	Wide column store	123.21	-1.00	-11.07
10.	10.	↑ 11.	Elasticsearch	Search engine	119.78	+0.37	+16.51
11.	11.	↓ 10.	SQLite	Relational DBMS	115.19	+2.44	+4.36
12.	12.	12.	Teradata	Relational DBMS	74.74	-3.49	+1.37
13.	13.	↑ 14.	Solr	Search engine	66.30	-2.86	-2.70
14.	14.	↓ 13.	SAP Adaptive Server	Relational DBMS	65.68	-1.35	-4.74
15.	15.	↑ 16.	Splunk	Search engine	63.79	-1.08	+8.87
16.	16.	↓ 15.	HBase	Wide column store	63.41	-0.15	+4.79

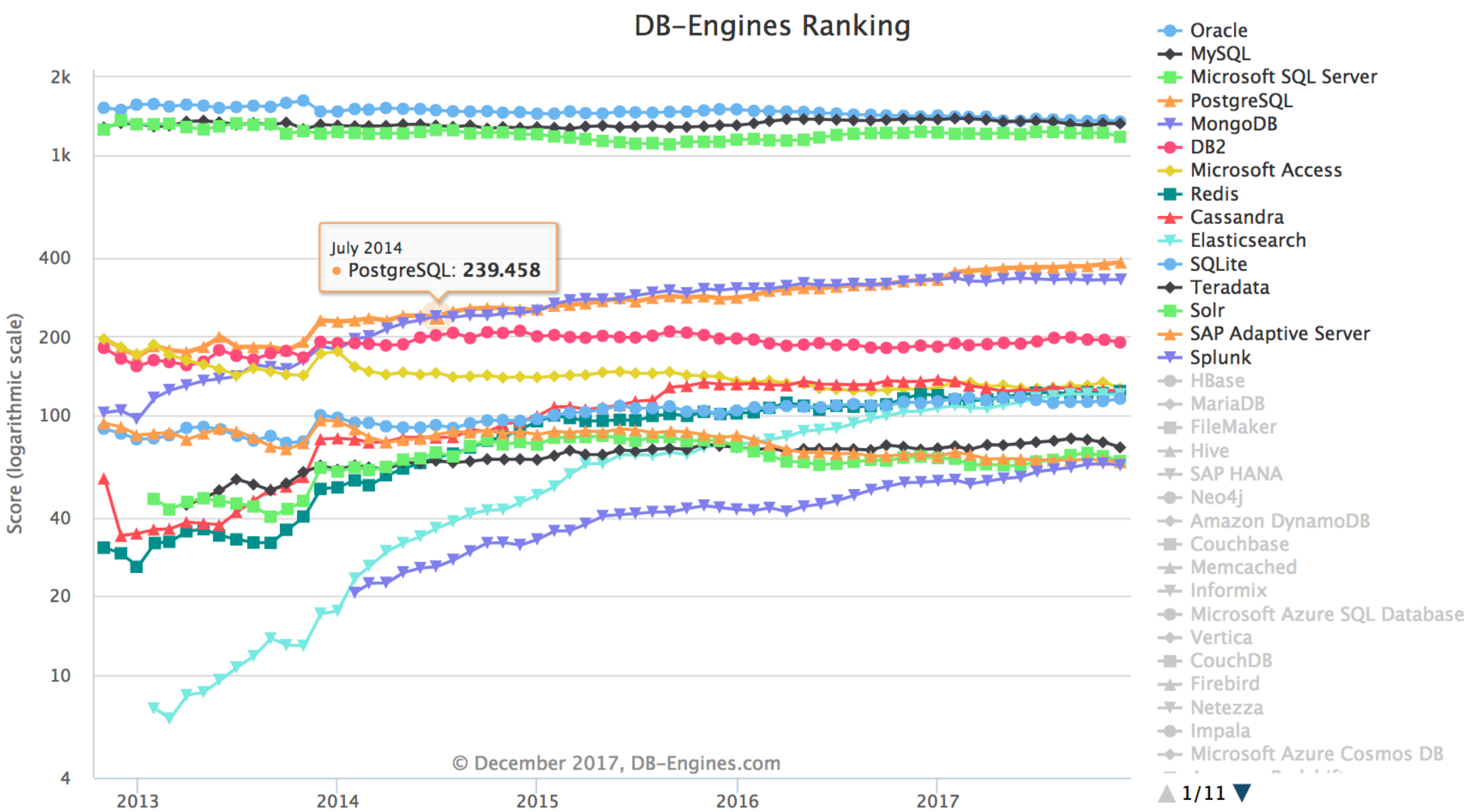
Ranking based on web sites, Google Trends, technical discussions (stack overflow), job offers, job profiles)

DB Engines Ranking (Trend)

Read more about the method of calculating the scores.

3	◆	SQL Server	1278	- 40
4		PostgreSQL	174	- 3
5		MS Access	161	- 8
6		DB2	155	- 4

ranking table
December 2017



Redis

- Redis is an key-value data store
- Redis stores in its values not only strings, but as well data structures such as lists, hashes, sets, and sorted sets
- Redis works with an in-memory dataset
- It is possible to persist dataset either by
  - dumping the dataset to disk every once in a while (e.g. after some seconds, after some changes)
  - or by appending each command to a log
- Popular e.g. for caching values

Redis Keys and Values

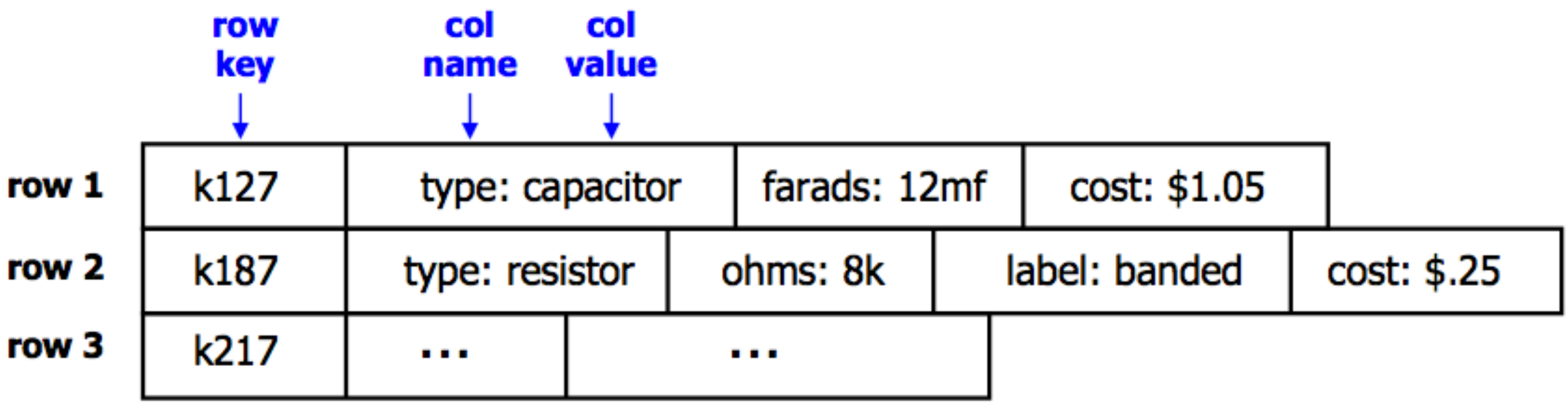
- Keys
  - Keys are binary safe - it is possible to use any binary sequence as a key
  - The empty string is also a valid key
  - A nice idea is to use some kind of schema, like: "`object-type:id:field`"
- Values
  - Built-in operations for data structures such as: Lists: „`LPUSH list:xxx a`“
  - Hashes: „`HMSET user:123 username gn password foo`” (similar to dicts in Tcl)
  - Caching: `GET` returns "" if value does not exist, `SETNX` (set value, if it does not exist)

Cassandra

- Originally developed at Facebook in 2007
- Combines Google's BigTable (2004) data model with Amazon's Dynamo (2006)
- Column-oriented
- Multi-Master replication
- Became Apache Incubator project in 2009
- Written in Java
- Uses Apache Thrift as API

Cassandra Data Model

- Similar to relational model, but:
- No schema necessary, columns can be added to rows, columns can vary per row

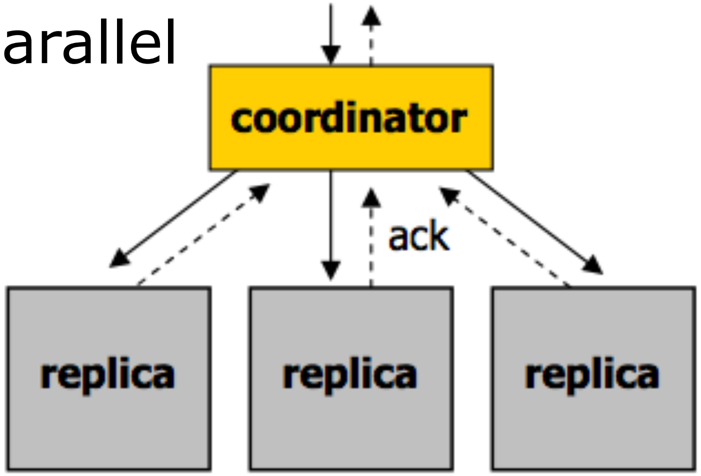


- Columns are grouped into column families
  - Similar to tables, allow for separate storage, vertical partitioning
- Columns can be "super columns" (somewhat deprecated in 2012)
- Keyspaces group column families together, base for replication (similar to "database")

Cassandra and Consistency

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

- Cassandra has programmable read/writable consistency
  - Writes are sent to all replicas in parallel
  - Developer can choose to read from R replicas and wait for W acks for writes
  - R and W can be tuned for latency/consistency requirements



Cassandra Basic AP and Query Language

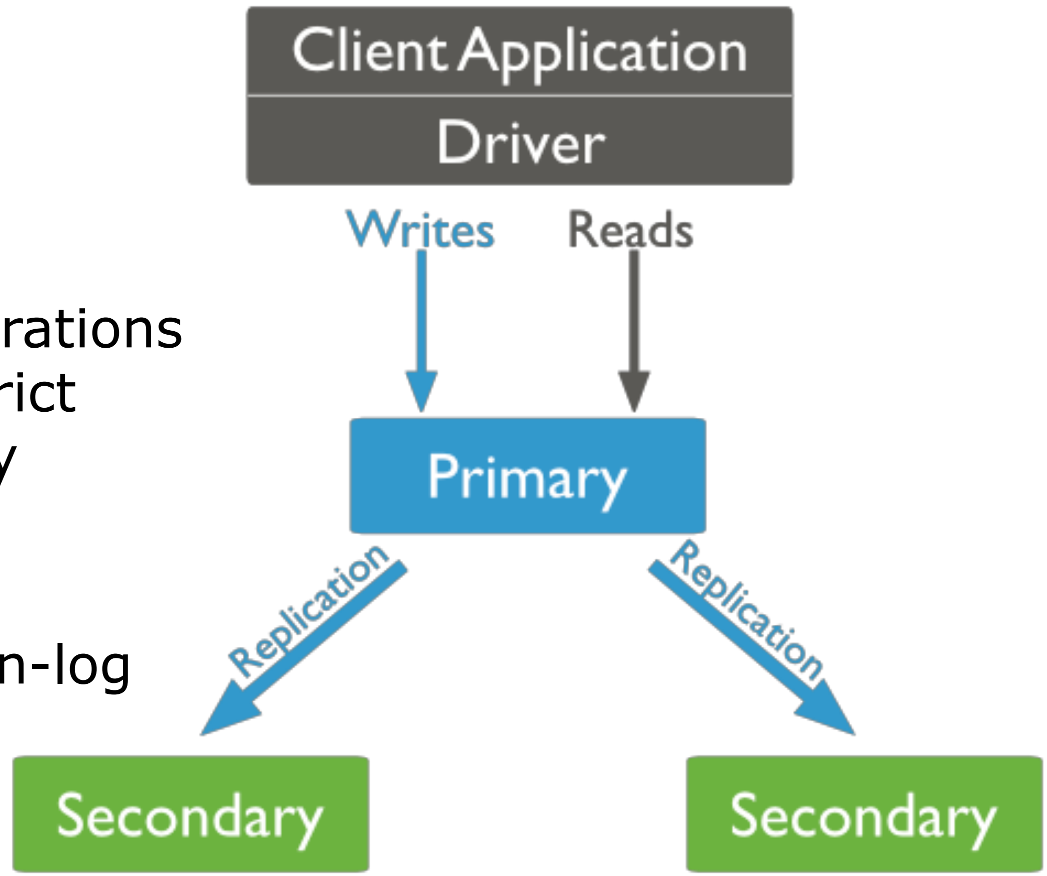
- Basic API (examples, … actually many APIs)
  - `get_column() / now GetColumn()`
  - `get_slice(keyspace, key, startColumn, acs/desc, count)`
  - Efficient row/column index support
  - Using RPC (Thrift, Avro)
- CQL – Cassandra query language (0.8, 2011)
  - High-level, similar to SQL, (e.g. "select", but "update" instead of "insert")
  - Subset of "classical" SQL + extensions for Cassandra features (column families, keyspaces, TTL, ….)
  - Since Cassandra 1.2, CQL is preferred interface.

mongoDB

- Developed by company 10gen, Founded in 2007, Open Source
- Document-oriented, schema-less:
  - Tree structure
  - Supports nesting
- High Performance
- Consistency might be strict or eventually consistent depending on options in asynchronous master/slave replication
- Infrastructure support (to reduce maintenance support) for
  - Replication & High Availability (automatic failover)
  - Sharding (auto-sharding)
- Written in C++

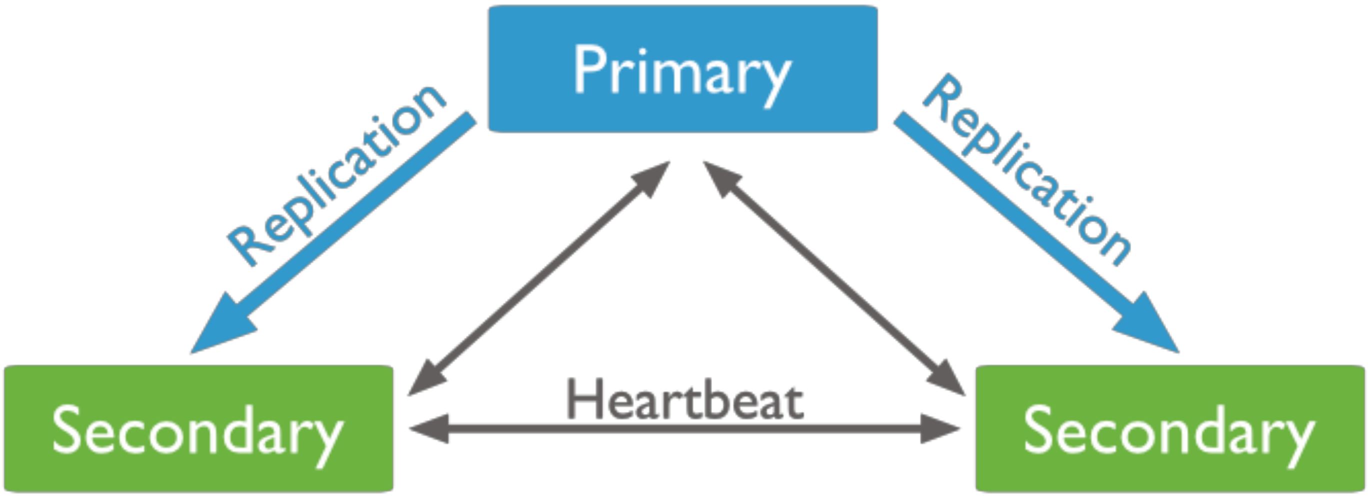
Replication

- Replica set:
  Group of instances
- Primary:
  Receives all **write** operations (like master-slave), strict consistency on primary
- Secondaries:
  receive/apply operation-log of primary, identical data



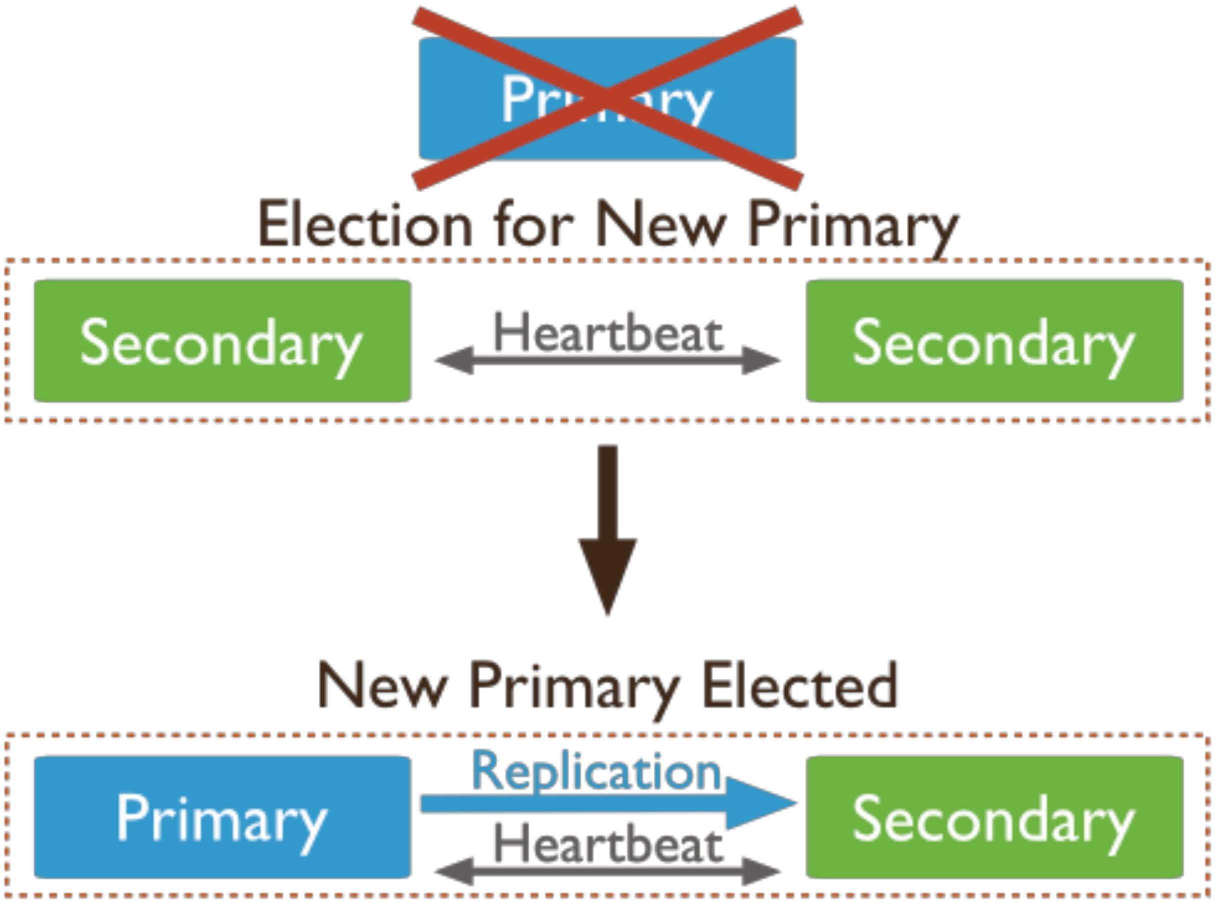
Replication

- Secondaries:
  - Default: clients read from primary, can be altered via read preferences



  - When primary becomes unavailable, they **elect** a new one

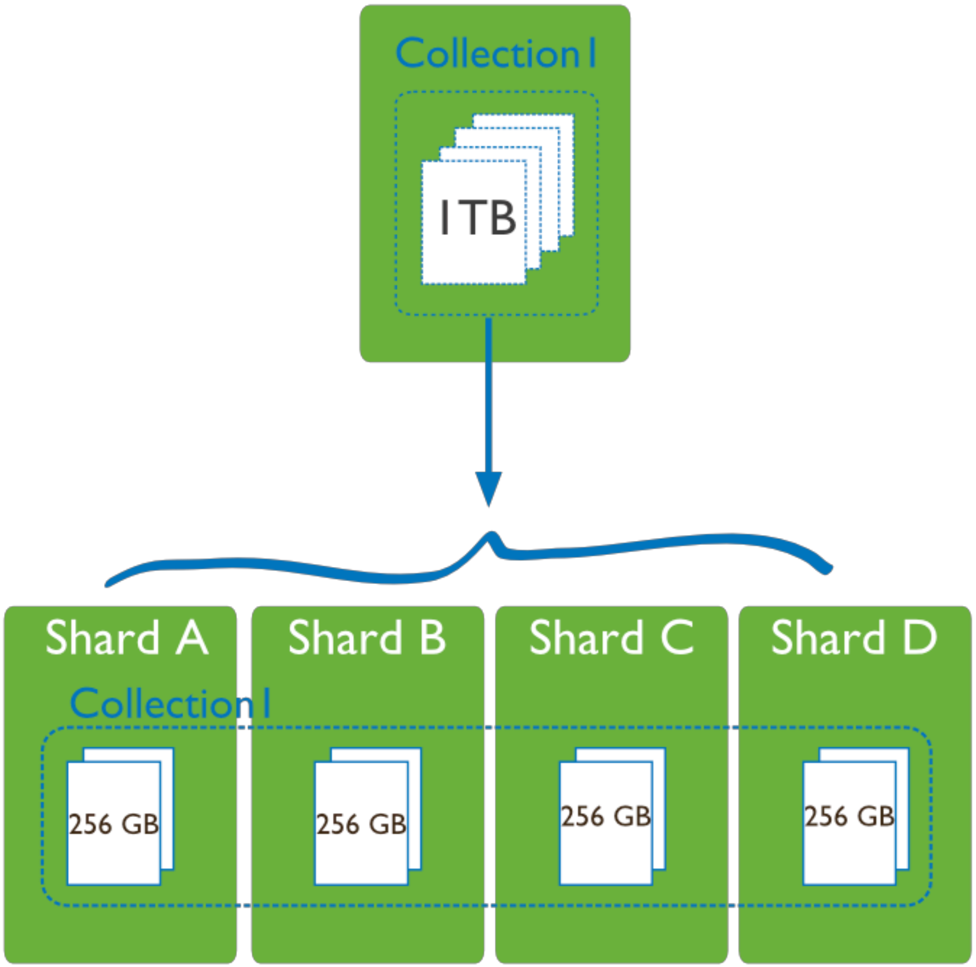
Automatic Failover



Sharding

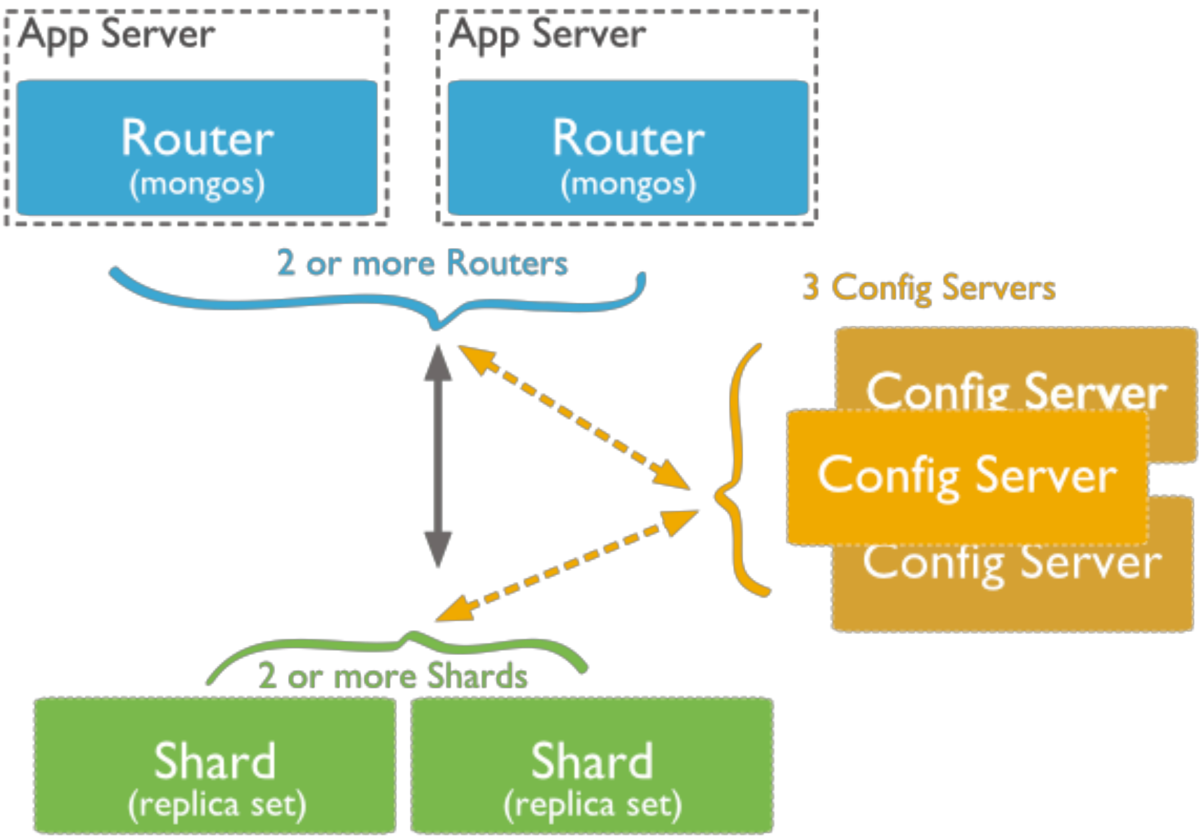
- Sharding:
  - store data across multiple machines.
  - divides the data set and distributes the data over multiple servers, or **shards**.
  - Each shard is an independent database, and collectively, the shards make up a single logical database.
- MongoDB uses sharding to support deployments with very large data sets and high throughput operations.

Sharding



Collection: similar to TABLE or VIEW in relational database systems

Sharding and Replica Sets



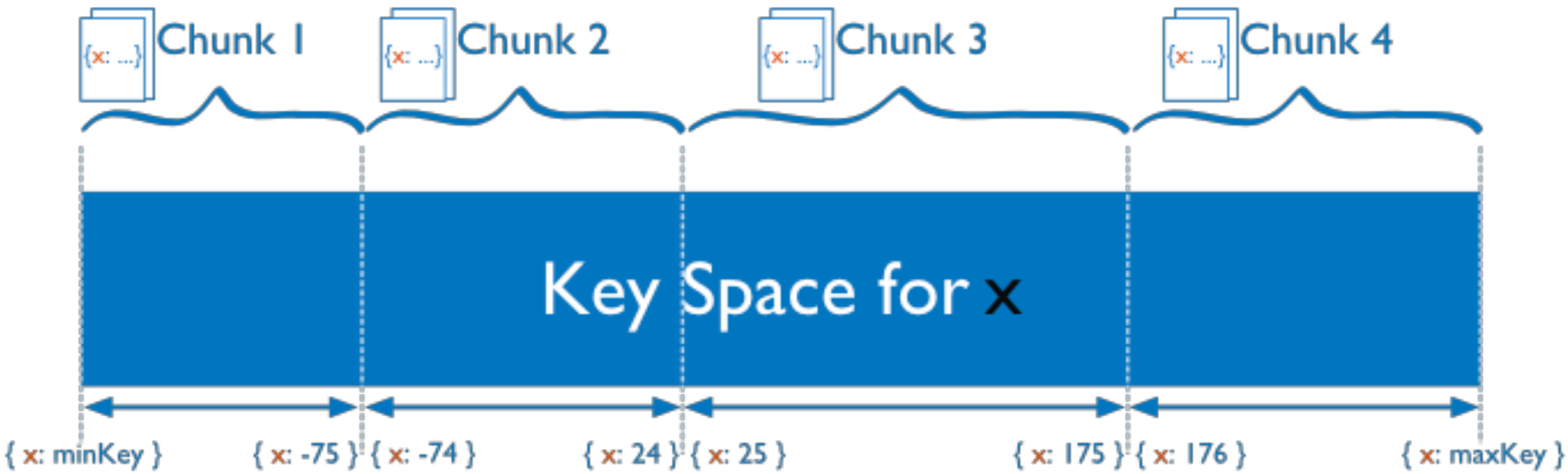
Query Router:
Direct query to shard

Shards: for high availability, shards should be replica sets

For development:
a shard can be a single `mongod`.

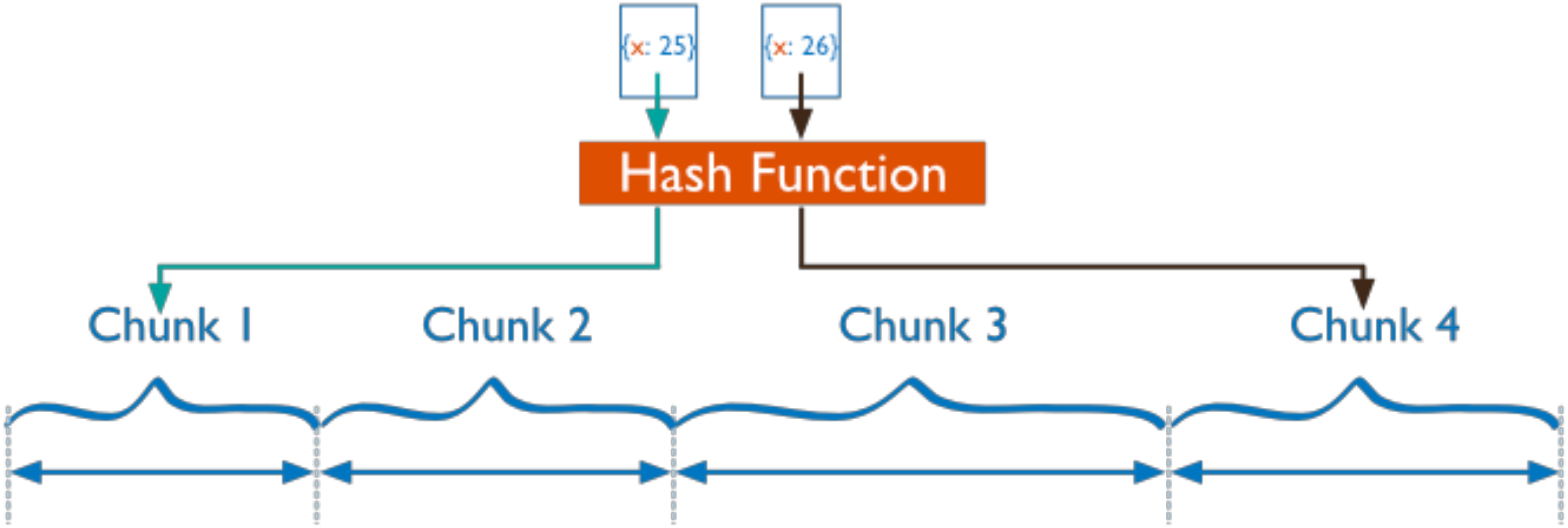
For production:
clusters with 3 config servers + replica sets

Range-based Partitioning



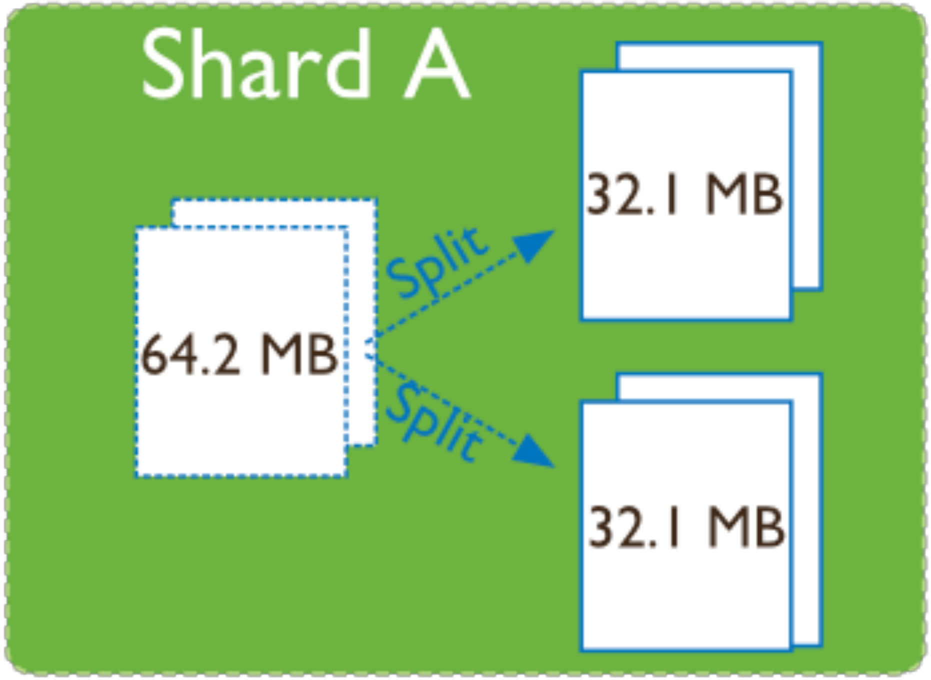
- Define application specific, non overlapping ranges to distribute data to different chunks (which might exist in different shards)
- Similar values are kept in the same shard (good for range-queries)

Hash-based Partitioning



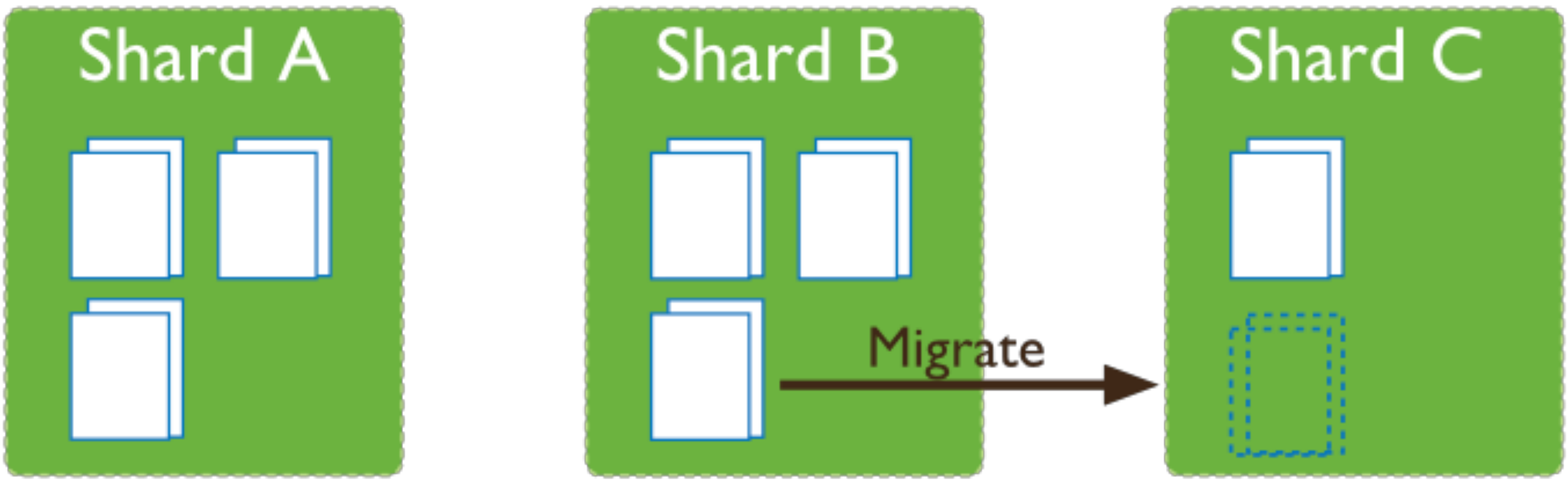
- Hash value is used for partitioning -> even distribution of values
- Range-based partitioning is better for range-queries, but may lead to uneven shard populations

Shard Splitting



- When data is added, shard populations can become unbalanced.
- *Splitter:* background process, splits chunks when size is exceeded
- Splitter does not migrate between shards

Shard Migration



- *Balancer:* background process to manager chunk migration, runs in query routers
- Chunks are migrated from the shard that has the largest number of chunks to the shard with the least number of chunks until the collection balances

Configurable Write Concerns

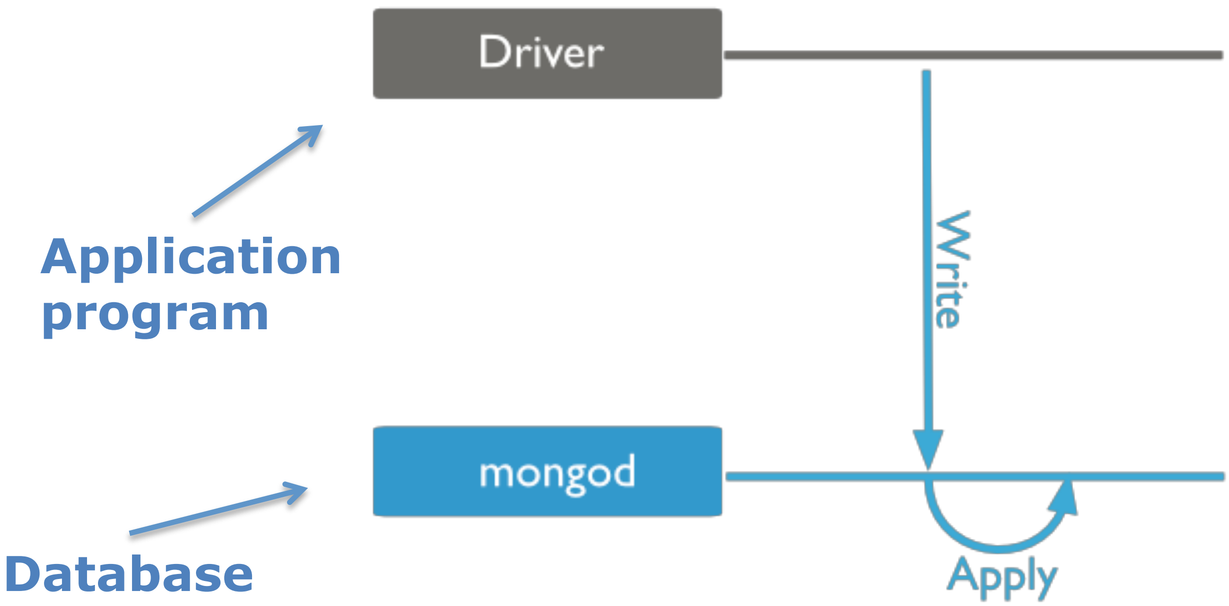
- **Write Concern:** defines what MongoDB guarantees when reporting the success of a write operation (insert, update, delete)

- Determine the level of guarantee:
  - *Weak write concern* leads to better performance,
  - *strong write concern* leads to higher reliability
- MongoDB provides different levels of write concerns to better address the specific needs of applications, level of guarantee can be specified up to *single database operations*

- Configurable write concerns
  - Unacknowledged (written to the socket)
  - Acknowledged (use default write concern)
  - Journaled (commit to disk)
  - Majority
  - W1, W2, W3

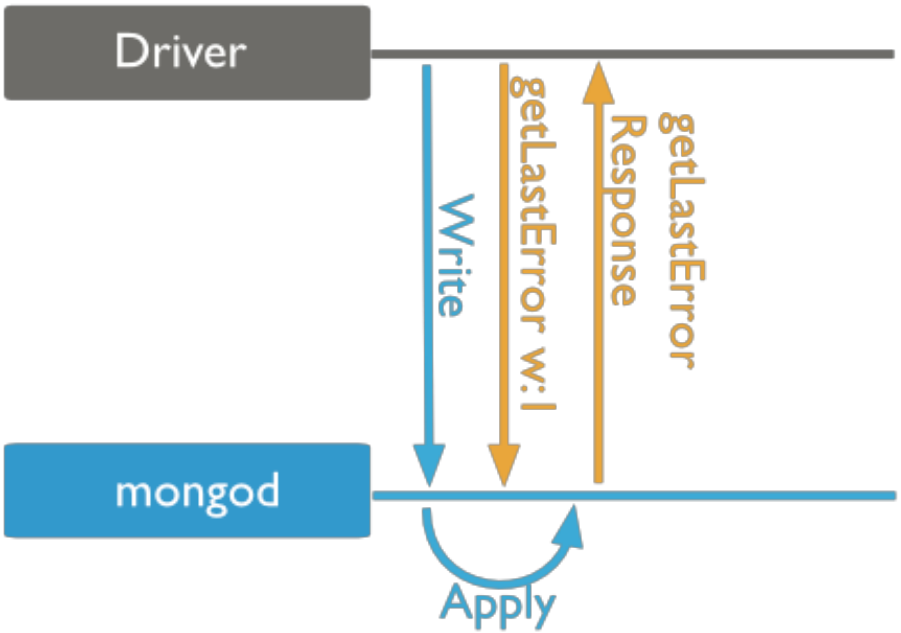
- Also: Configurable read preferences

Write Concern: Unacknowledged



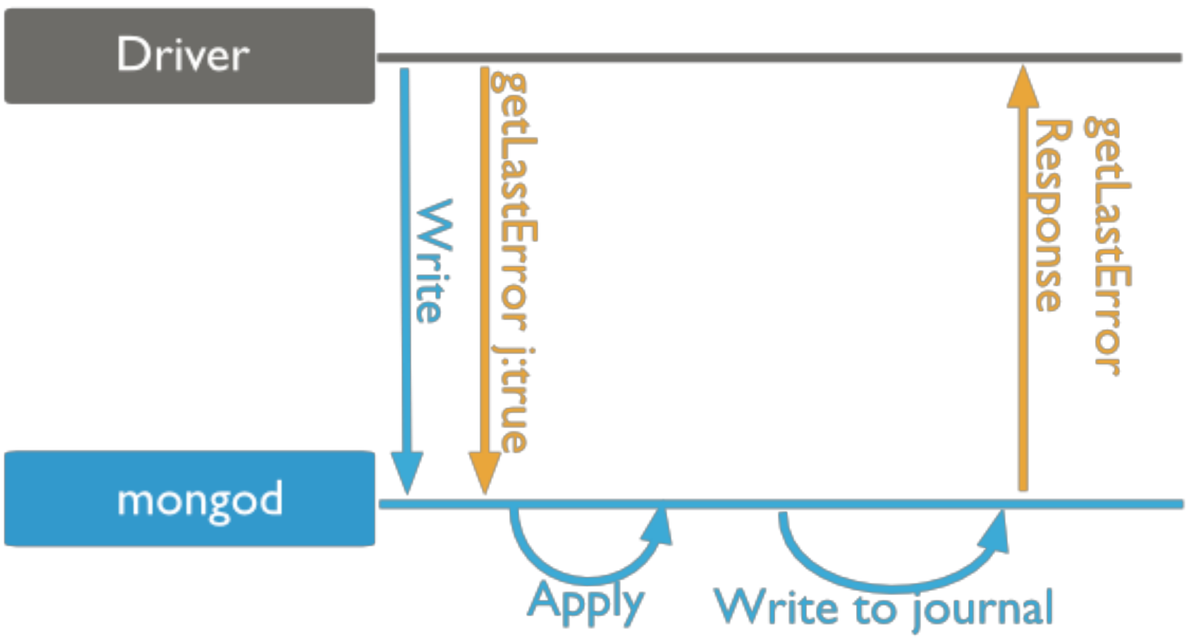
- **Driver:** Interface of application program to MongoDB
- **Write Concern Unacknowledged:**
  Submit write operation and continue in application program while database stores and distributes values.

Write Concern: Acknowledged



- **Write Concern Acknowledged:**
  Application program submits "`getLastError`" with "`w`" set to "`1`"
- MongoDB confirms the receipt of the write operation.
- Allows clients to catch network, duplicate key, and other errors

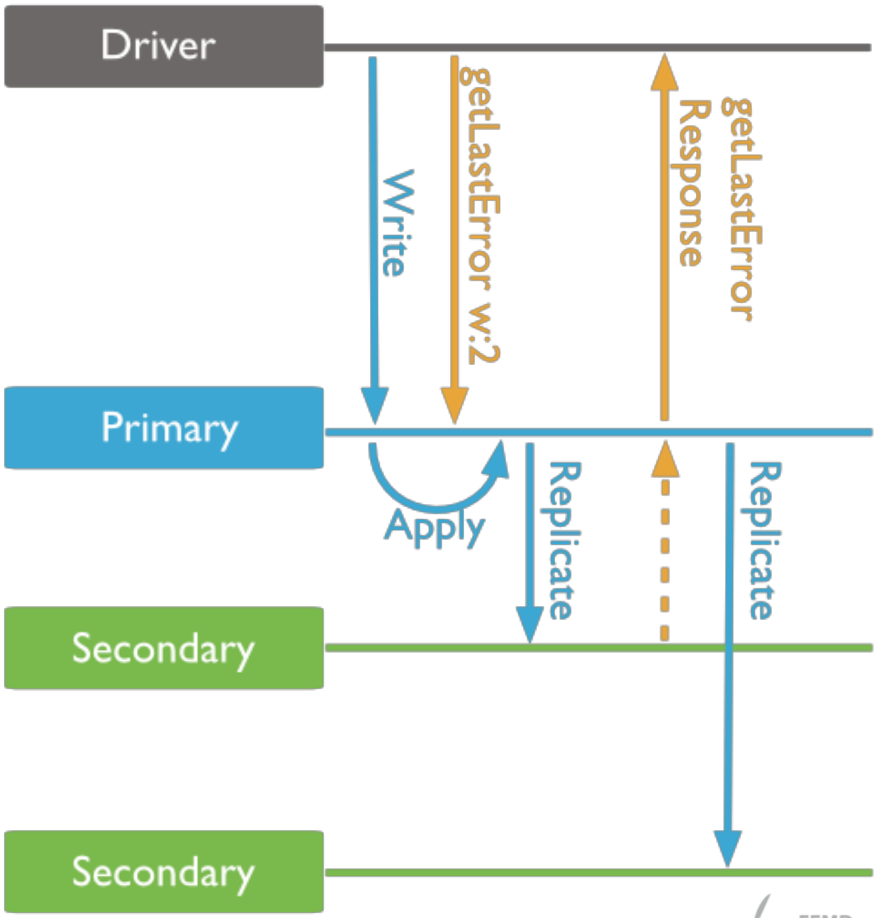
Write Concern: Journaled



- **Write Concern Journaled:**
  Application program submits “`getLastError`“ with “`j`” set to “`1`”
- “Write to journal” -> data written to the disk
- MongoDB can recover from a power interruption without loosing this data

Write Concern: Replica Acknowledged

- **Write Concern Replica Acknowledged:** Application program submits "`getLastError`" with "`w`" set to "`2`"
- Wait for the acknowledge of at least one secondary
- Other values for "`w`":
  - `3, 4` …
  - `majority`
- `getLastError` can also turn on
  - `fsync`
  - `wtimeout`



Connection String URI Format (Setting Default Preferences)

- **Write concerns** and **read preferences** can be defined programmatically, and/or via special URI format used when connectiong from an application to the mongodb server.

  `mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]`

- **Replica Set with Members on localhost**
  Connect to a replica set with three members running on localhost on ports 27017, 27018, and 27019:
  `mongodb://localhost,localhost:27018,localhost:27019`

- **Replica Set with Read Distribution**
  Connect to a replica set with three members and distributes reads to secondary servers:
  `mongodb://example1.com,example2.com,example3.com/?readPreference=secondary`

- **Replica Set with a High Level of Write Concern**
  Connects to a replica set with write concern configured to wait for replication to succeed on at least two members, with a two-second timeout.
  `mongodb://example1.com,example2.com,example3.com/?w=2&wtimeoutMS=2000`

MongoDB Data Structures and Queries

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

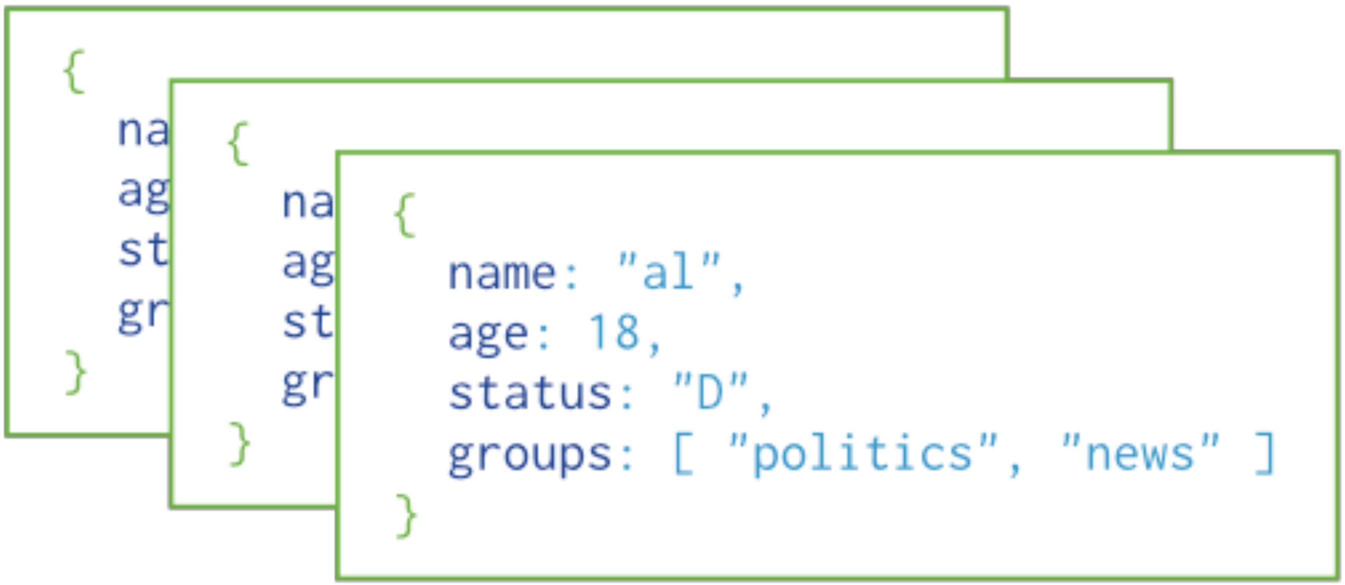
EFMD EQUIS ACCREDITED

Document


```
{
  name: "sue",                       ← field: value
  age: 26,                           ← field: value
  status: "A",                       ← field: value
  groups: [ "news", "sports" ]       ← field: value
}
```

- Data is stored in **documents** (similar to a tuple in an RDBMS)
- Document are JSON-like data structure with field and value pairs (tree structure)
- JSON: JavaScript Object Notation
- MongoDB uses BSON (binary „JSON") with various data types
- MongoDB supports multivalued attributes (here: "`groups`")

Collection



Collection

- **Collection:** Multiple documents of same kind
- Documents use same indices
- Comparable to a TABLE in relational database systems
- Creation: `db.createCollection("users")`

CRUD API
(here in JavaScript syntax)

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

- Create (data)
  - `db.collection.insert( <document> )`
  - `db.collection.save( <document> )`
  - `db.collection.update( <query>, <update>, { upsert:true } )`
- Read
  - `db.collection.find( <query>, <projection> )`
  - `db.collection.findOne( <query>, <projection> )`
- Update
  - `db.collection.update( <query>, <update>, <options> )`
- Delete
  - `db.collection.remove( <query>, <justOne> )`

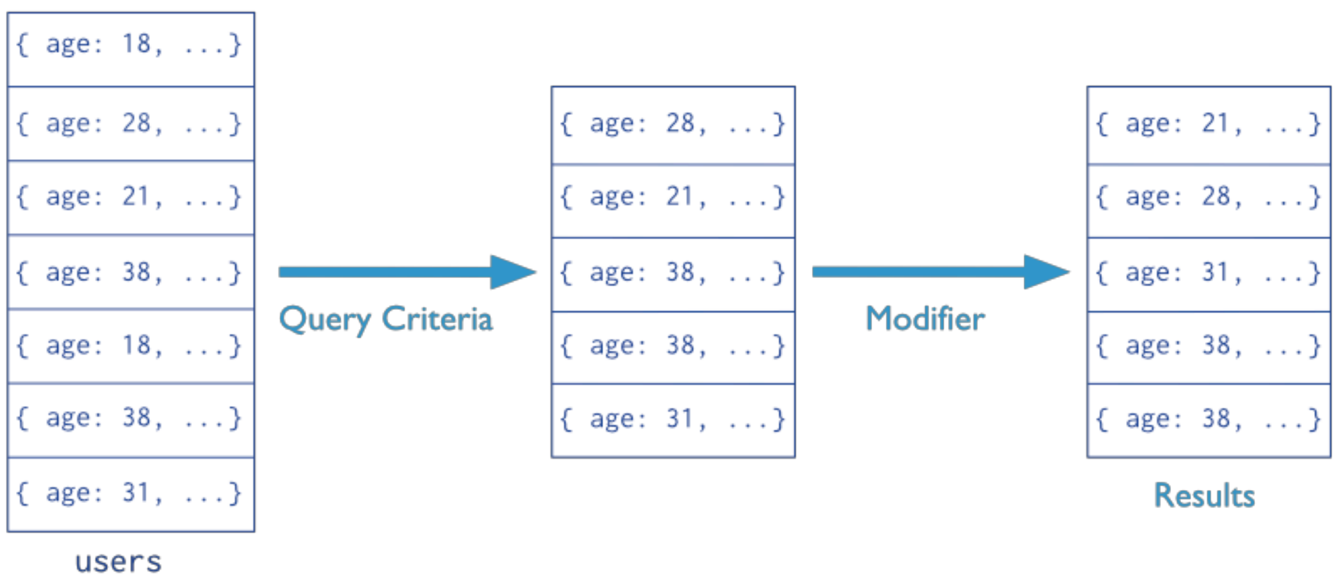
- „Upsert": insert or update in **one** operation

EFMD EQUIS ACCREDITED

mongoDB Query

Collection — Query Criteria — Modifier

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



- **Query, Data, and Result** in form of a BSON document, similar to JSON (JavaScript Object Notation)
- MongoDB Operators: reserved words, starting with "$"

BSON

Collection — Query Criteria — Modifier

```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```

- BSON: Abbreviation of Binary JSON
- Data Structure is a tree structured document with potentially multivalued-attributes (arrays)
- Allows compact representation and efficient operations (e.g. sort by date, using internal ordinal representation rather than strings)
- Data types: UTF-8-Strings, 32- und 64-bit-integer, float, date, Boolean, NULL value, ObjectId, UUID, MD5, …

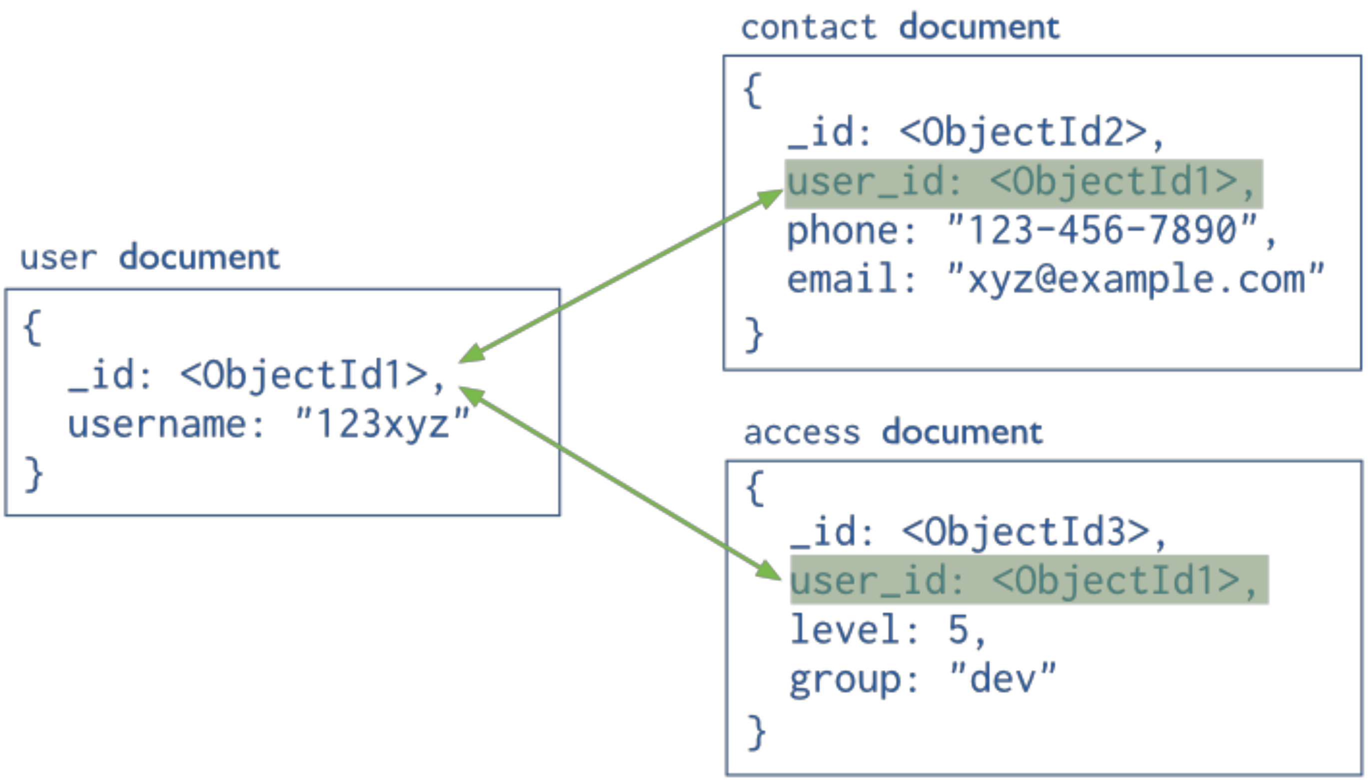
- Here: Query in JavaScript syntax (`mongo` shell)
- Binding for various programming languages

Multi-valued Attributes

- First class data-type, can be used e.g. for full-text search
- Index support for efficient access to multi valued attributes
- Atomic operations for maintaining multi-valued attributes in update operations: $push, $addToSet, $each, $slice, $sort, …

```
db.users.update( { name: "sue" }, { $push: { groups: "wu" }})
```

Document References



- MongoDB maintains per document an object ID "`_id`"
- ID can be used as field value (no guarantee of referential integrity)

Embedding Documents

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
             level: 5,
             group: "dev"
           }
}
```

Embedded sub-document (contact)

Embedded sub-document (access)

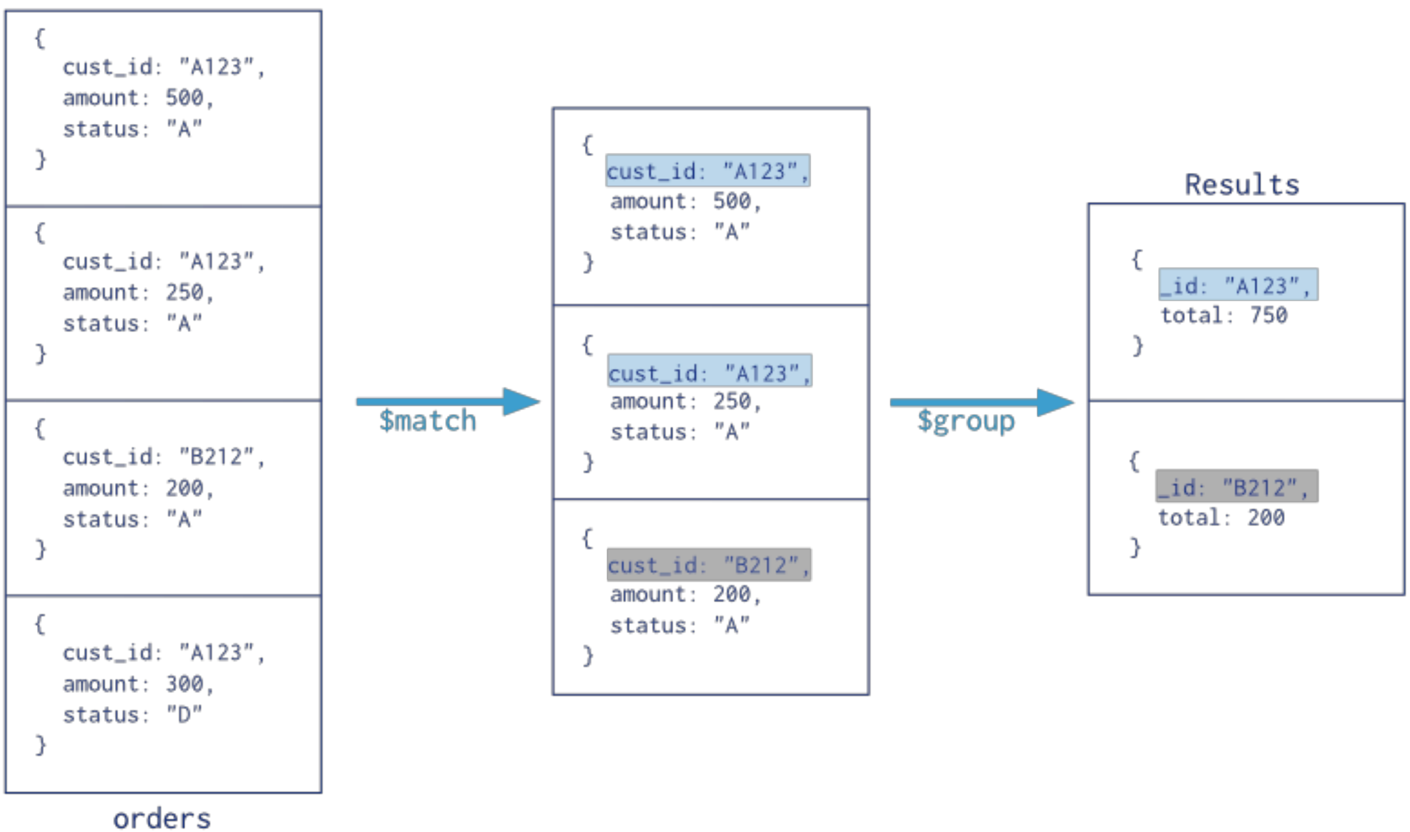
- **Embedding** in "`user`" document: "`contact`" and "`access`" documents

Aggregation

WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

Collection

```
db.orders.aggregate(
    $match phase ──▶ { $match: { status: "A" } },
    $group phase ──▶ { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
)
```



Operators:

- $match, $group, $sum

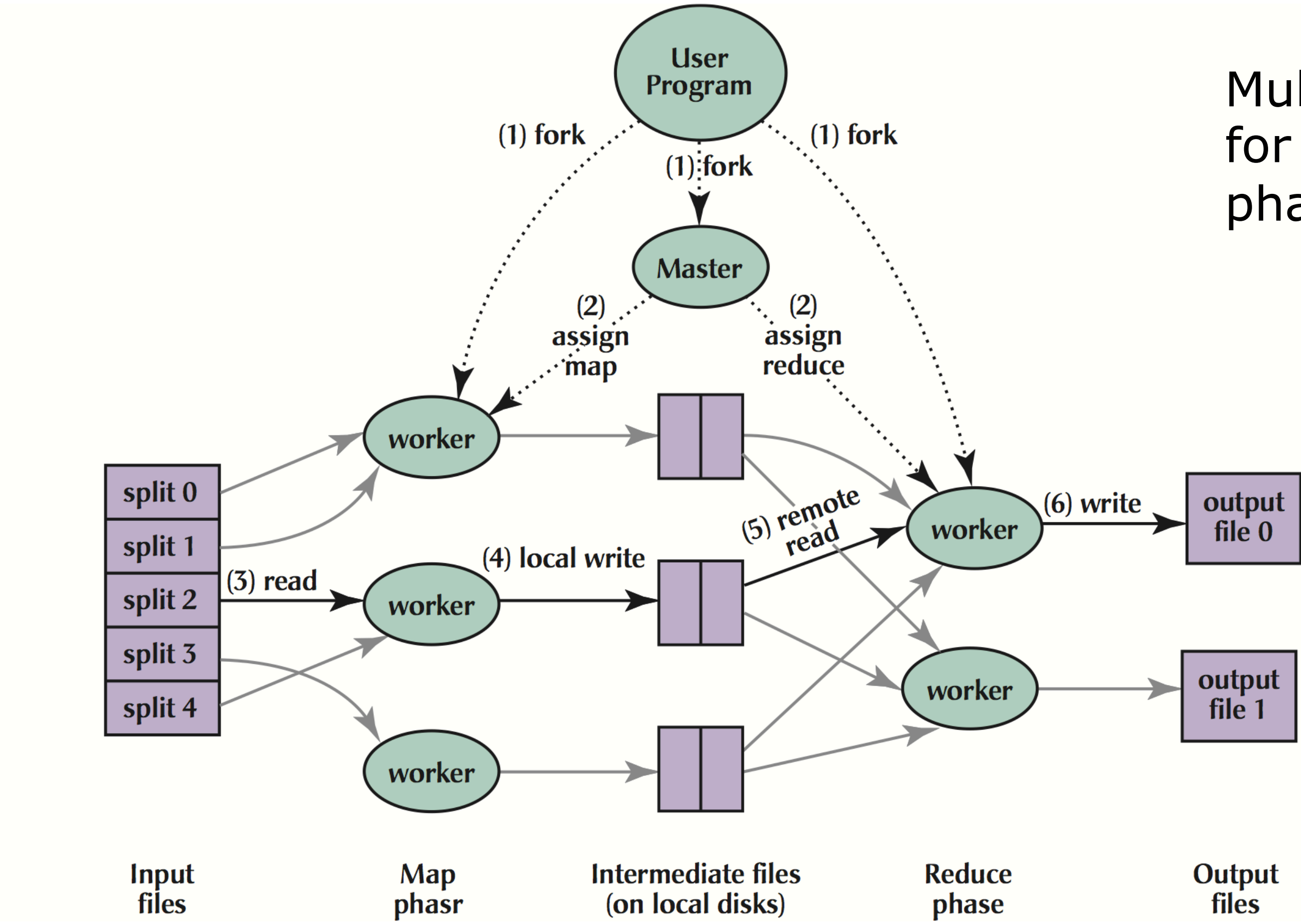
Placeholders:

- $cust_id, $amount

Map-Reduce

- Programming paradigm for aggregation/summarizing
- Allows for massive scalability across hundreds or thousands of servers through parallelization
- Name inspired from functional programming
- Introduced by Google 2004 for large-scale indexing (Google patent in 2010; novelty challenged)
- Open-Source implementations (after Google article): Hadoop, Phoenix, …
- 2 Phases:
  - Map: filtering, sorting, projecting of data
  - Reduce: summarizer

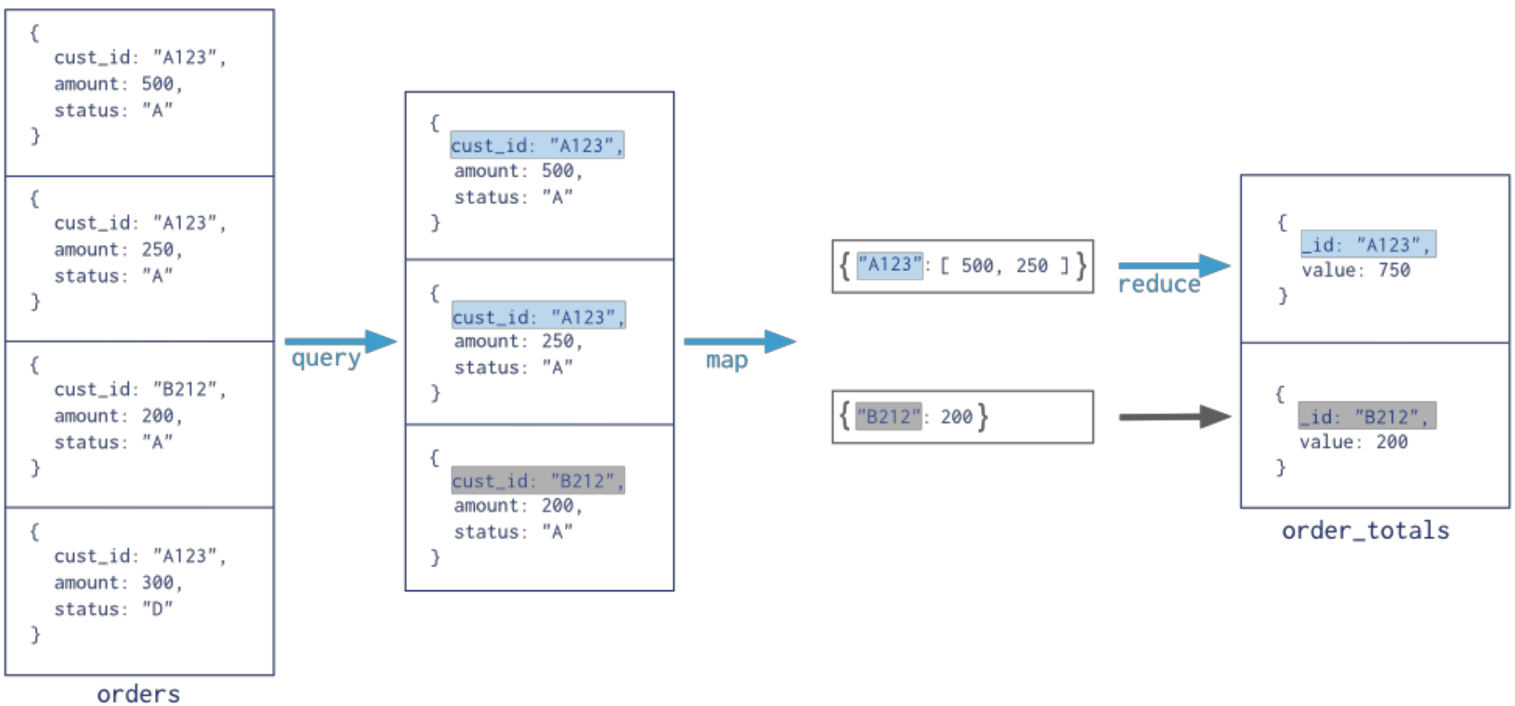
Map-Reduce



Multiple workers for `map` and `reduce` phases

Map-Reduce

```
            Collection
                 ↓
db.orders.mapReduce(
        map     →  function() { emit( this.cust_id, this.amount ); },
        reduce  →  function(key, values) { return Array.sum( values ) },
                   {
        query   →    query: { status: "A" },
        output  →    out: "order_totals"
                   }
)
```



WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

EFMD EQUIS ACCREDITED

Implementing a Web-App with MongoDB and NX

Example: Business Insider

- One of the largest growing professional news sites (claimed by businessinsider.com)

- Blog-like data model:

```
{ title: ‘Too Big to Fail’,
  author: ‘John S’,
  ts: Date(“05-Nov-09 10:33”),
  comments: [ { author: 'Ian White',
                comment: 'Great article!' },
              { author: 'Joe Smith',
                comment: 'But how fast is it?',
                replies: [ {author: 'Jane Smith',
                            comment: 'scalable?' } ]
              } ],
  tags: [‘finance’, ‘economy’]
}
```

- From Dwight Merriman, founder of 10gen: http://www.slideshare.net/mongodb/nosql-the-shift-to-a-nonrelational-world

Business Insider as (Simplified) Relational Model

Entries	entry_id
	223117
	112456
	112457
	112458

Replies	entry_id
	112458

Postings	ID	Title	Author	ts
	223117	Too Big To Fail	John S	05-Nov-09 10:33

Comments	ID	Comment	Author	entry_id
	112456	Great article!	Jan White	223117
	112457	But how fast is it?	Joe Smith	223117
	112458	scalable?	Jane Smith	112457

Tags	posting_id	Tag
	223117	finance
	223117	econonmy

- 5 tables
- Complex referential structure
- Query requires several join operations over potentially huge tables (which are maybe stored on different nodes)
- Update of one logical entry requires locks

Business Insider Data Model for MongoDB in NX

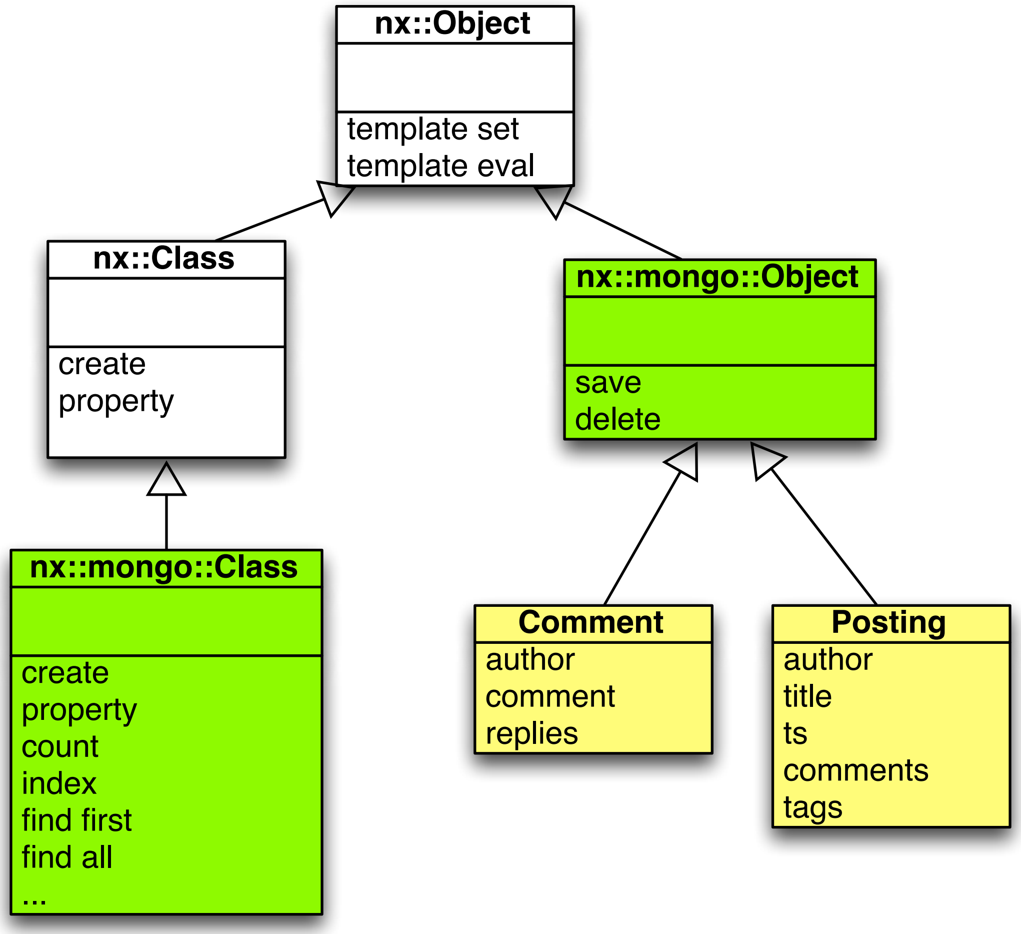
WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

```
################## Data Model ################

nx::mongo::Class create ::bi::Comment {
   :property author:required
   :property comment:required
   :property -incremental replies:embedded,type=::bi::Comment,0..n
}

nx::mongo::Class create ::bi::Posting {
   :index tags
   :property title:required
   :property author:required
   :property ts:required
   :property -incremental comments:embedded,type=::bi::Comment,0..n
   :property -incremental {tags:0..n ""}
}
```

Type spec for property in nx::Class can be “embedded” or “reference”, multiplicity: “0..1”, “0..n”, “1..1”, “1..n”

EFMD EQUIS ACCREDITED

Simplified Relational Model



- Mapping Layer
  - `nx::mongo::Object`
  - `nx::mongo::Class`
- Application classes for Data to be stored in MongoDB
  - Classes created via `nx::mongo::Class`
  - MongoDB classes create MongoDB objects
  - Objects inherit from `nx::mongo::Object`

Object oriented CRUD API in NX (simplified)

- Create
  - *nx::mongo::Object* `save`
- Read
  - *nx::mongo::Class* `find first ...`
  - *nx::mongo::Class* `find all ...`
- Update
  - *nx::mongo::Object* `save`
- Delete
  - *nx::mongo::Object* `delete`

Sample MongoDB Interactions 1: Insert

```
# ... Source data-model if outside of NaviServer

# Connect to the database
::nx::mongo::db connect -db "tutorial"

# Create a nested object, like every other nx object
set p [Posting new \
       -title "Too Big to Fail" -author "John S." -ts "05-Nov-09 10:33" \
       -tags {finance economy} \
       -comments [list \
           [Comment new -author "Walter White" -comment "Great Article!"] \
           [Comment new -author "Joe Smith" -comment "But how fast is it?" \
              -replies [list [Comment new -author "Jane Smith" —comment "scalable?"]]] \
           ]]
# Save in MongoDB
$p save
```

Sample MongoDB Interactions 2: Query and Update

```
# ... Source data-model if outside of NaviServer ...
# Connect to MongoDB
nx::mongo::db connect –db tutorial
# Count entries having "finance" as a tag, result is 1 with sample DB
Posting count -cond {tags = finance}

# OO update: Fetch posting from MongoDB as object, update it, and save it
set p [Posting find first -cond {tags = finance}]
$p tags add wi
$p save

# Optional: Lower level interface using triples to map BSON structures to Tcl lists
# {tags: "finance", { $addToSet: { tags: "wu" }}}
nx::mongo::db update "tutorial.postings" \
              {tags string finance} \
              {$addToSet object {tags string wu}}

# show updated value
Posting count -cond {tags = wu}
```

Desired View

- Display all `Postings` with `tags`, `comments` and `replies` using nested HTML lists:

Postings:

- 05-Nov-09 10:33: **John S.** posts: *Too Big to Fail*
  - **Walter White** comments: '*Great Article!*'
  - **Joe Smith** comments: '*But how fast is it?*'
    - reply: **Jane Smith** comments: '*scalable?*'

  tags: finance economy

Counting entries, obtaining all values, rendering output

```
::nx::mongo::db connect -db "tutorial"
. . .
if {[Posting count] > 0} {

   # Build result object containing the instance variable :postings,
   # which is a list of objects:
   set result [nx::Object new {
     set :postings [Posting find all -orderby ts]
   }]

   # Set template for result, iterating over the postings with FOREACH
   $result template set {
     Postings: <ul><FOREACH var='p' in=':postings' type='list'><li>@p;obj@<p></li>
     </FOREACH></ul>
   }

   # Obtain the rendered HTML output
   set html [$result template eval]
}
```

Templates for “Posting” and “Comment”

```
#
# Default templates
#
Posting template set {
  @:ts@: <b>@:author@</b> posts: <em>@:title@</em> <br>
  <ul><FOREACH var='c' in=':comments' type='list'><li>@c;obj@</li>
  </FOREACH></ul>
  tags: @:tags@<br>
}

Comment template set {
  <b>@:author@</b> comments: <em>'@:comment@'</em>
  <ul><FOREACH var='r' in=':replies' type='list'>
    <li>reply: @r;obj@</li>
  </FOREACH></ul>
}}
```

Program Examples

- Copy

  `mongo-*.tcl`

  into

  `/usr/local/ns/pages/`

- and

  `oo-templating.tcl`
  `bi.tcl`

  into

  `/usr/local/ns/modules/tcl/`

- Make sure "`mongod`" is running; restart NaviServer

- Try out in from a browser

  `http://localhost:8080/mongo-setup.tcl`

  and look into the page source

Review of mongo-*.tcl

- Good:
  - Preserved all "good" properties from last examples
  - High-speed database access and persistence
  - MongoDB is fully compliant with the dynamic object model of XOTcl, NX
  - Uses MongoDB specific connection pooling
  - Timings quite good, study where time is spent on page `http://localhost:8080/mongo-edit.tcl`

- Limitations:
  - No language support for configurable write concerns (just via connection URI)
  - Many details can be improved (e.g. nicer editing, etc.)

Project Assignment

- Turn Business Informer example implementation in NX into a “query and answer” application (specialized Forum)
  - Start from business informer data model and files
  - Use problem-specific names (adp-files, data model)
  - Use features such as tags, ratings, up-voting to implement social feedback (let you inspire by stackoverflow.com)
  - Bootstrap Interface
  - Add User-management with cookies (using ns_cookie, http://naviserver.sourceforge.net/n/naviserver/files/ns_cookie.html)

Literature

- Eric A. Brewer. Towards Robust Distributed Systems, Keynote at the *Symposium on Principles of Distributed Computing PODC,* (*2000*)
- Werner Vogels. 2008. Eventually Consistent. *Queue* 6, 6 (October 2008), 14-19
- Decandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, G.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. (2007). "Dynamo: *Amazon's Highly Available Key-value Store*". Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles - SOSP '07. p. 205-220
- Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *CACM* 51, 1 (January 2008)