Advanced Databases

Lectures
December 2014.

NoSQL
2/3

Document databases

- Like Key-Value, with Value being document
- **Data model: (key, document)**
- Document: JSON, BSON, XML, YAML, some other semi-structured format, binary data
- Main operations:
  - Put(k, d)
  - Get(k)
  - Update(k, d)
  - Delete(k)
  - **Queries based od document content**! (not standardized, no query language)
- Some DBs support indexing
- Examples: *CouchDB, MongoDB, SimpleDB,…*

Example: MongoDB documents

Relational database: **relation**

fName	lName	DateBirth	BirthPlace
Ivan	Car	11.11.1971.	
Iva	Kralj		Šibenik

MongoDB: **collection**

```
{
    "_id": ObjectID("4efa8d2b7d284dad101e4bc9"),
    "fName" : "Ivan",
    "lName“ : "Car",
    "BirthDate" : "11.11.1971."
},
{
    "_id" : ObjectID("4efa8d2b7d284dad101e4bc7"),
    "fName" : "Iva",
    "lName" : "Kralj",
    "BirthPlace" : "Šibenik"
}
```

Primjer: MongoDB upiti

- Mongo queries are JSON (BSON) objects

SQL	MongoDB
CREATE TABLE student(id INT, lName CHAR(50))	Implicitly - by putting the first document in collection. Also explicitly: db.createCollection(„student")
ALTER TABLE student ADD…	Implicitly, every document can be changed, there is no schema
INSERT INTO student (100, 'Šostakovič');	db.student.insert({mbr:100, lName: 'Šostakovič'})
SELECT * FROM student;	db.student.find();
SELECT lName FROM student WHERE mbr = 200 ORDER BY lName;	db.student.find({mbr:100}, {lName:1}).sort({lName:1});
UPDATE student SET lName = 'Shostakovich' WHERE mbr = 100;	db.student.update({ mbr: 100 }, {$set : { lName : 'Shostakovich' } });
DELETE FROM student WHERE mbr = 100;	db.student.remove({ mbr: 100 });

Aggregate model - KV & Document databases

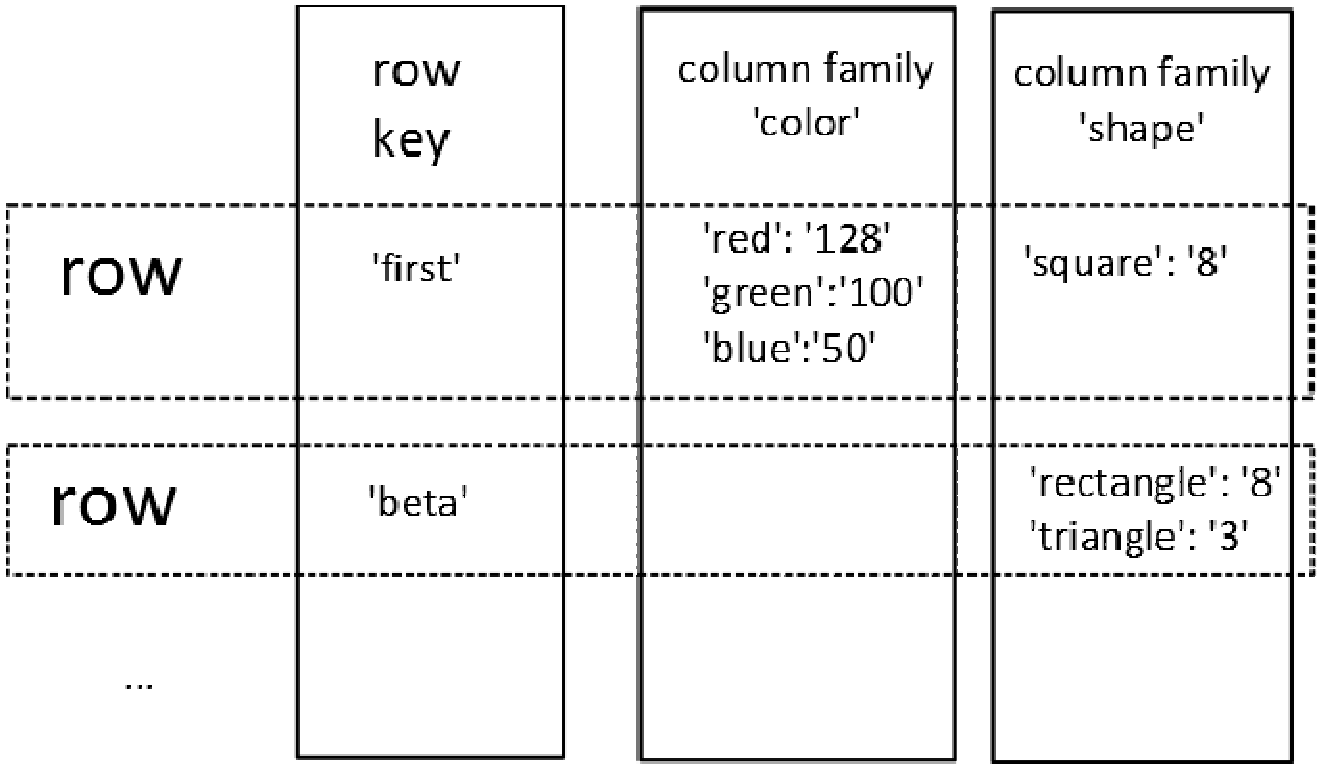
- KV and document DBs are based on the aggregate data type
- KV DBs
  - ✓✗ Retrieval by key
  - ✓✗ Value is BLOB
- Document DBs
  - ✓ Retrieval based on query
  - ✓ Part of the document can be retrieved
  - ✓ Indexing
  - ✗ Constraints on the value (not everything can be inserted)
- In practice, the distinction between KV & Document DB is blury

CF databases

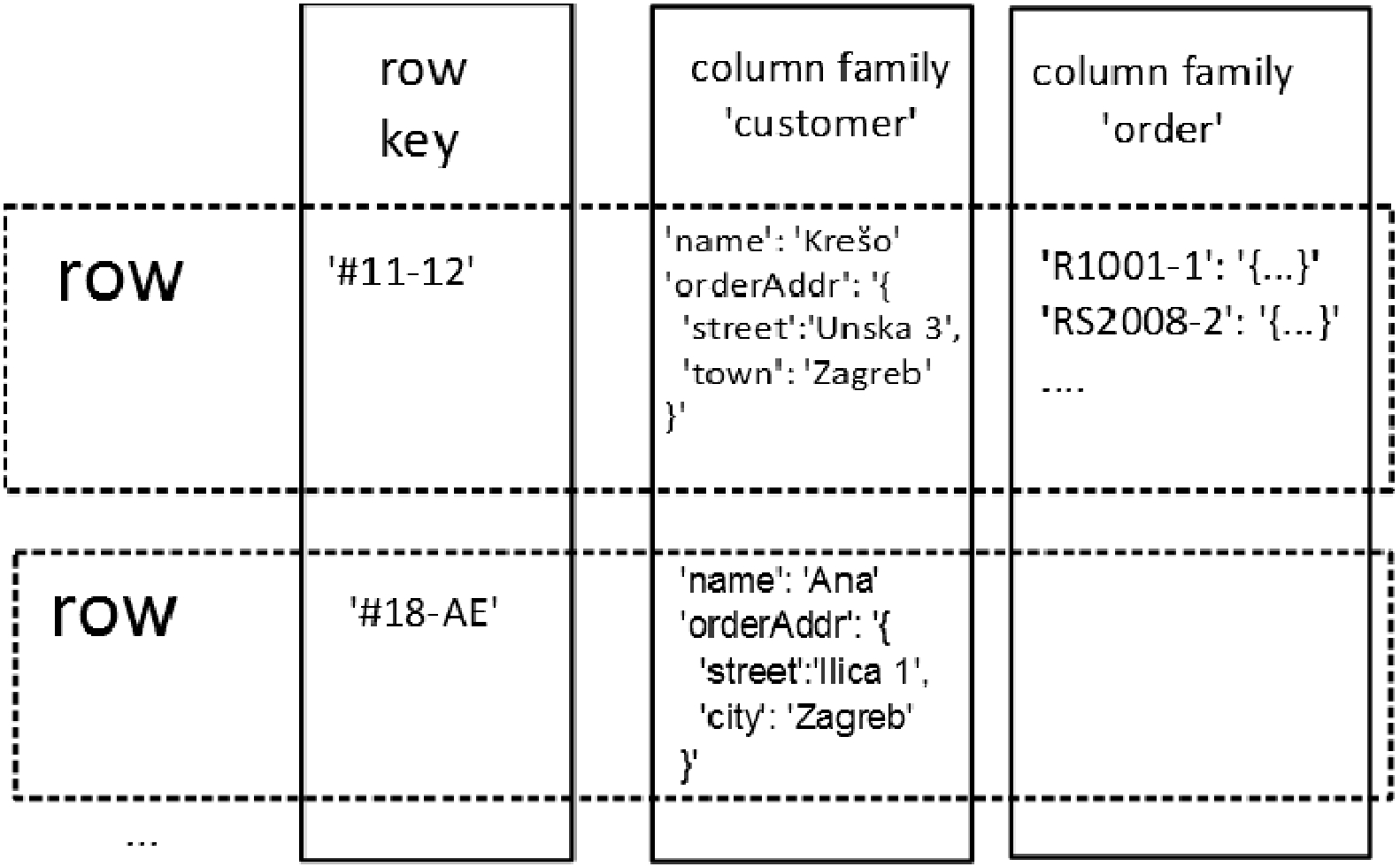
- *Chang et al. [2006], Bigtable: A Distributed Storage System for Structured Data*
- **Data model: *column family***
- **Not a table!**
- **Two-level hash map, two-level aggregate**
- First level key: row key
- Second level key: column key
- Each column is a member of single column family

CF example (1)

- `get('first', ' color:green')`

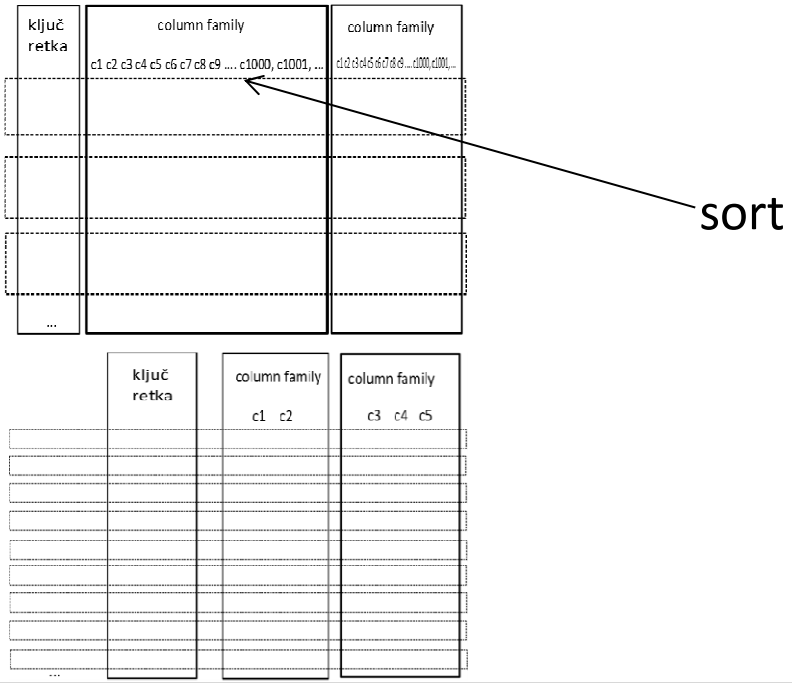


CF example (2)


row key
column family 'customer'
column family 'order'
row
'#11-12'
'name': 'Krešo'
'orderAddr': '{
'street':'Unska 3',
'town': 'Zagreb'
}'
'R1001-1': '{...}'
'RS2008-2': '{...}'
....
row
'#18-AE'
'name': 'Ana'
'orderAddr': '{
'street':'Ilica 1',
'city': 'Zagreb'
}'
...

CF comment

- Dual view of the data:
  - By **rows**: each row can be considered an aggregate
  - By **columns**: each CF defines a record type (e.g. customer), with rows for each record
- Row = JOIN of records in all CFs
- Different row setups:
  - *Wide row*
  - *Skinny row*



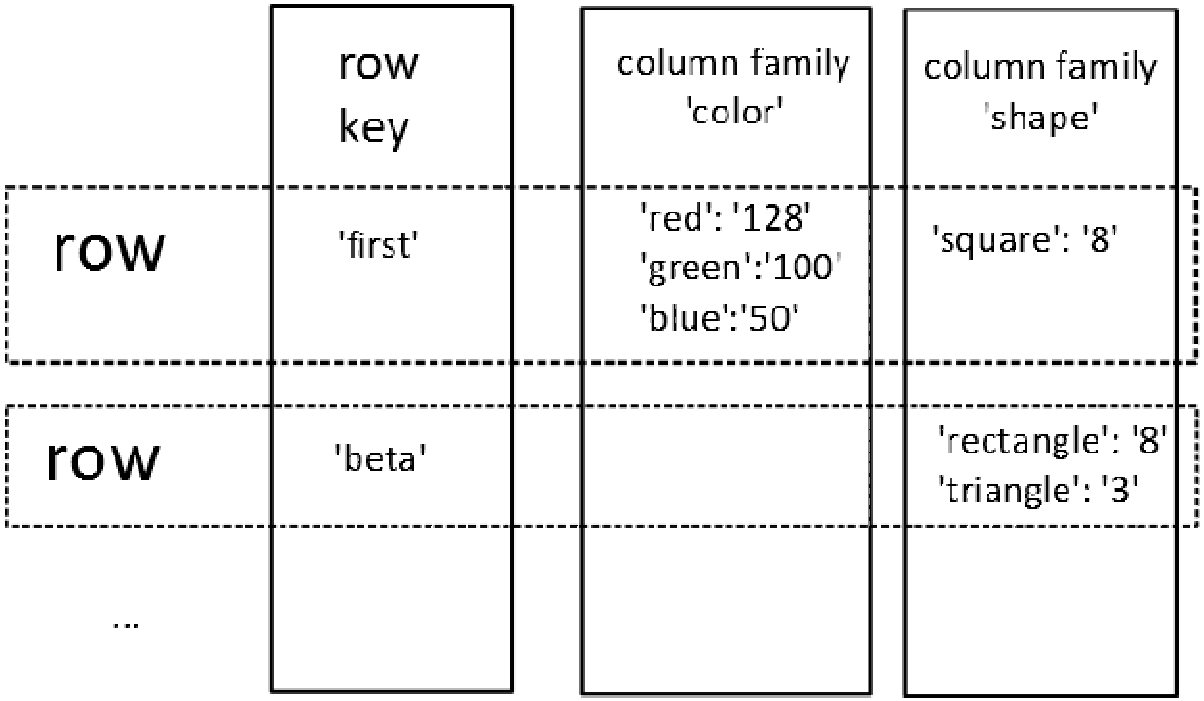
Example: HBase

- Column family DB, inspired with Google Big Table*
- *“Sparse, distributed, persistent multidimensional sorted map.”*
- Uses Hadoop
- Only for large data = GB+:
  "This project's goal is the hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware."
- Good properties:
  - Scalable
  - Versioning
  - Compression
  - Memory resident tables
  - Fault tolerant

**Chang et al. [2006], Bigtable: A Distributed Storage System for Structured Data*

Structure

- "table"* - key value - *map of maps*
- "row" (sorted)
- "column family"
- "column"
- "value", e.g.
  first/color:green
  is '100'

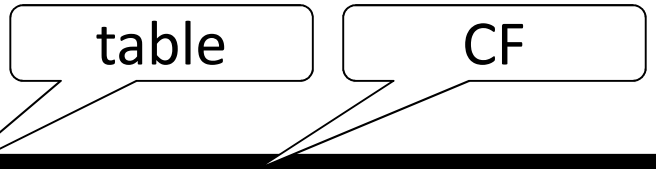


The terminology is reminiscent of the relational database, but the concepts are fundamentally very different.
To emphasize this, the names here are written in quotation marks.

Column families

- All Hbase operations are atomic on the row level – consistent rows
- Why not put all columns in the single CF?
  - CFs can be separately configured

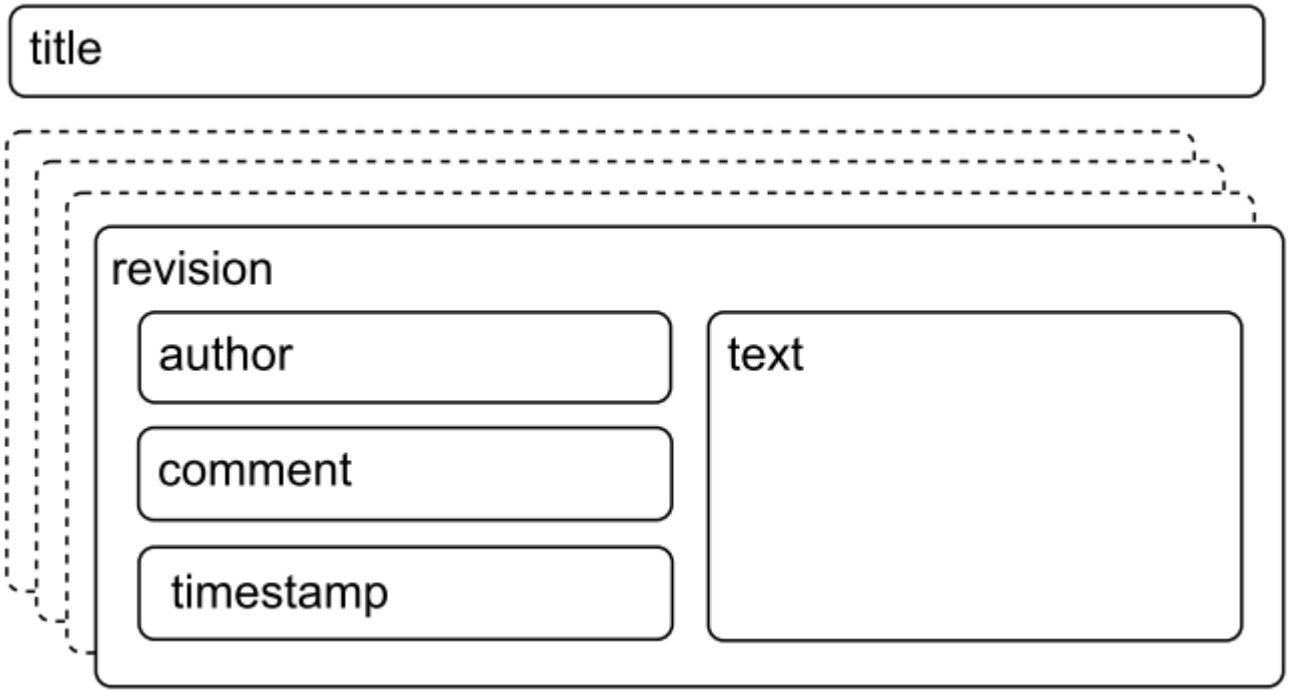
Versions




```
hbase(main):002:0> create 'test', 'semaphore'

hbase(main):004:0> put 'test', '1', 'semaphore:', 'red'

hbase(main):005:0> put 'test', '1', 'semaphore:', 'yellow'

hbase(main):006:0> put 'test', '1', 'semaphore:', 'green'
0 row(s) in 0.0040 seconds

hbase(main):007:0> get 'test', '1'
COLUMN                          CELL
 semaphore:                       timestamp=1372341842883, value=green

hbase(main):010:0> get 'test', '1', {COLUMN => 'semaphore:', VERSIONS => 4}
COLUMN                          CELL
 semaphore:                       timestamp=1372341842883, value=green
 semaphore:                       timestamp=1372341838768, value=yellow
 semaphore:                       timestamp=1372341829568, value=red
```

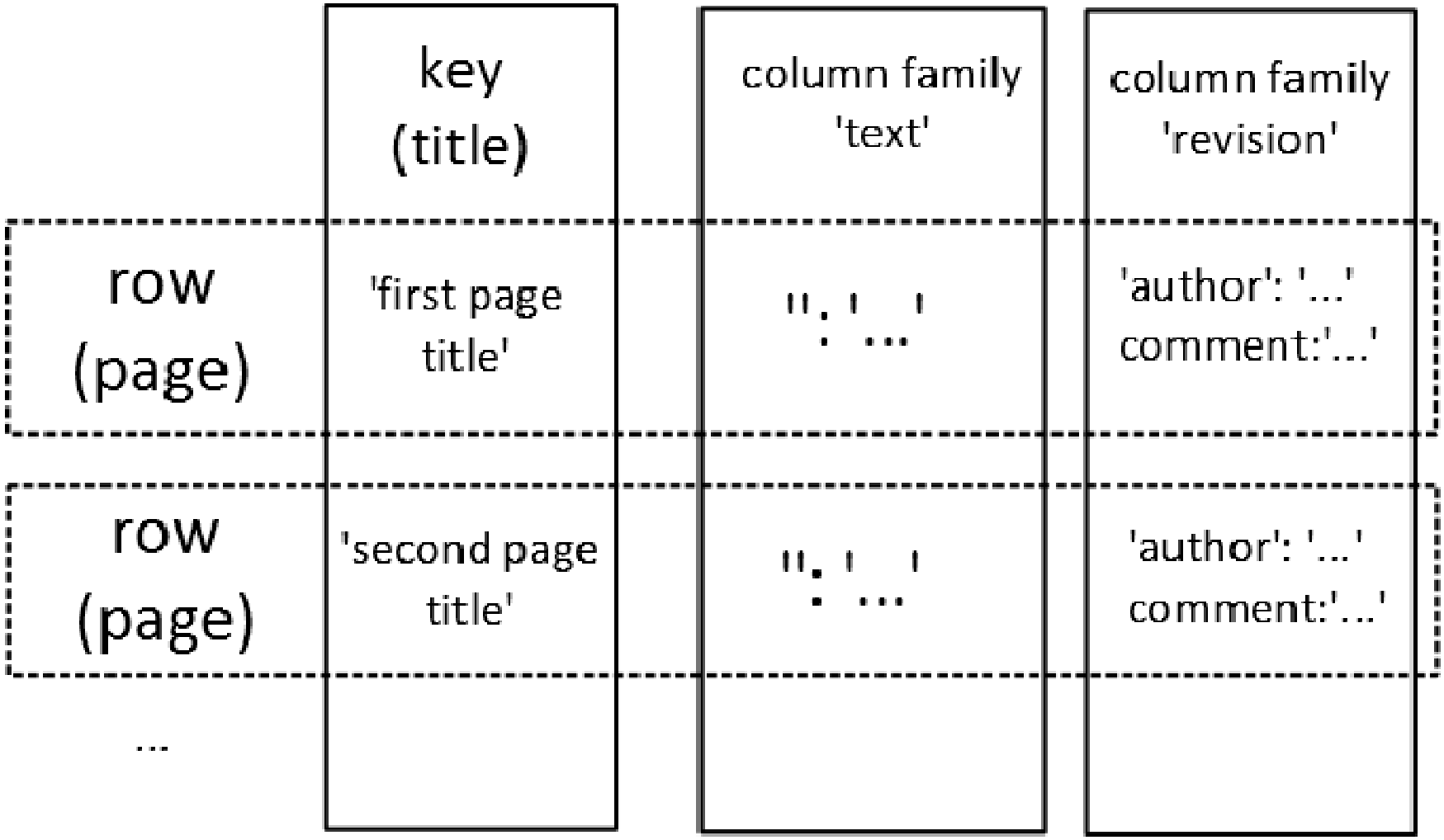
Example - wiki*

```
hbase(main):007:0> create 'wiki',
{NAME => 'text', VERSIONS => org.apache.hadoop.hbase.HConstants::ALL_VERSIONS
},
{NAME => 'revision', VERSIONS =>
org.apache.hadoop.hbase.HConstants::ALL_VERSIONS}
0 row(s) in 0.2040 seconds
```



**Taken from the book: Seven databases in seven weeks*

Example - wiki (2)



Example - wiki (3) – inserting a page

Insert our first page:

```
hbase(main):002:0> put 'wiki', 'HomePage', 'text:', 'Welcome'
0 row(s) in 0.6750 seconds

hbase(main):003:0> put 'wiki', 'HomePage', 'revision:author', 'igor'
0 row(s) in 0.0050 seconds

hbase(main):004:0> put 'wiki', 'HomePage', 'revision:comment', 'My first entry'
0 row(s) in 0.0130 seconds

hbase(main):005:0> get 'wiki', 'HomePage'
COLUMN                          CELL
 revision:author                timestamp=1372332177394, value=igor
 revision:comment               timestamp=1372332182860, value=My first entry
 text:                          timestamp=1372332177312, value=Welcome
3 row(s) in 0.0340 seconds
```

Not ideal - timestamp is different. Shell does not allow multiple commands at once, must use API (e.g. Java) for that.

Example - wiki (4) - wikipedia import

- Download the wikipedia dump from the internet and pipe it to hbase table (using ruby script that parses xml)*

```
hbase(main):004:0> curl
http://dumps.wikimedia.org/enwikibooks/20130626/enwikibooks-20130626-pages-
articles-multistream.xml.bz2 | bzcat | ./bin/hbase shell
./ruby/import_from_wikipedia.rb
```

```
…
 95  116M   95  110M    0     0   765k      0  0:02:35  0:02:28  0:00:07  800k 100500 records
inserted (Ba Zi/Date Selection in 1927)
101000 records inserted (Development Cooperation Handbook/The video resources linked to this
handbook/The Documentary Story/Searching for the questions to ask)
 97  116M   97  113M    0     0   756k      0  0:02:37  0:02:33  0:00:04  489k 101500 records
inserted (Template:DPL)
 97  116M   97  113M    0     0   753k      0  0:02:38  0:02:34  0:00:04  380k 102000 records
inserted (Ba Zi/HP H6)
 98  116M   98  114M    0     0   748k      0  0:02:39  0:02:36  0:00:03  380k 102500 records
inserted (Managing your business with anyPiece/Inventory system for recycle business/Storage
Packaging/Storage packaging - History)
103000 records inserted (How to Ace FYLSE/October 2012 Exam)
 99  116M   99  115M    0     0   750k      0  0:02:38  0:02:37  0:00:01  480k 103500 records
inserted (ElementarySpanish/Meeting people/Lesson 1)
100  116M  100  116M    0     0   752k      0  0:02:38  0:02:38 --:--:--  712k
```

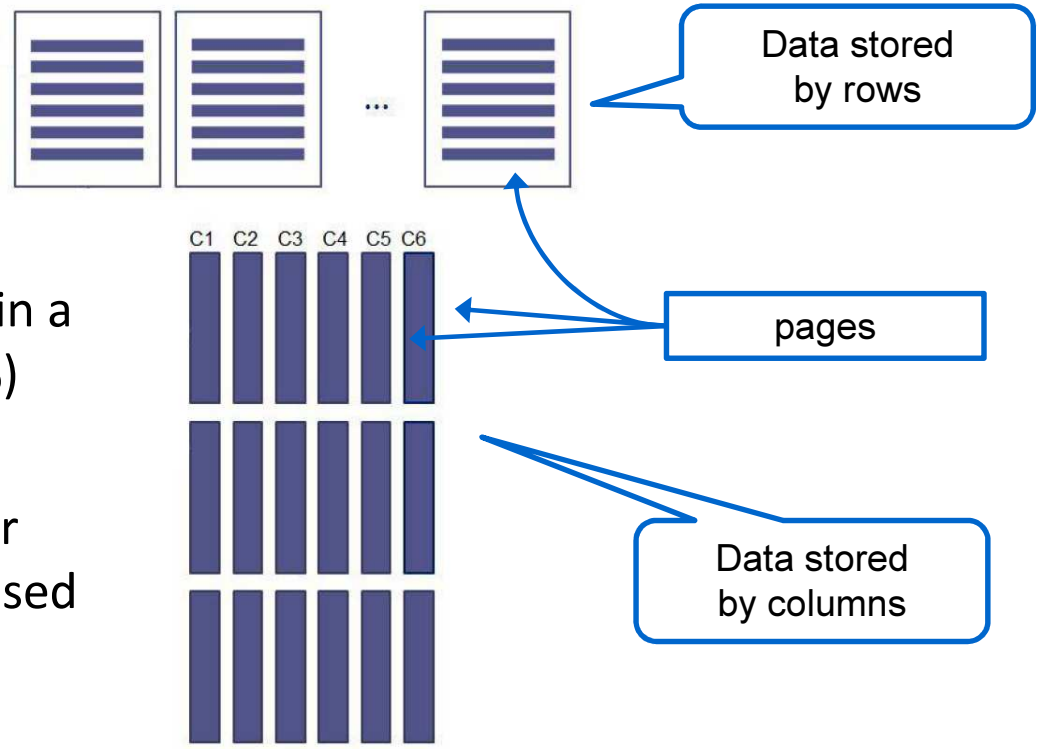
*For details, see: *Seven databases in seven weeks*

Example - wiki (5) – wikipedia import

```
hbase(main):004:0> count 'wiki', INTERVAL => 10000
Current count: 10000, row: Blender 3D: Noob to Pro/Basic Animation/Rendering
Current count: 20000, row: Chemical engineering
Current count: 30000, row: Development Cooperation Handbook/The video resources linked to
this handbook/The Documentary
Story/The KFI story
Current count: 40000, row: File:NotesCornell.png
Current count: 50000, row: Hebrew Roots/The Law and the Covenants/Covenants:The Covenant
of Israeli Sovereignty
Current count: 60000, row: Mandarin Chinese/Sentences/He is a student
Current count: 70000, row: Programming:Game Maker/Intro
Current count: 80000, row: Structural Biochemistry/Protein function/Heme
group/Hemoglobin/Affinity Constant
Current count: 90000, row: Template:User language/sah
Current count: 100000, row: Wikijunior Languages/Finnish
103927 row(s) in 8.5280 seconds

hbase(main):005:0> get 'wiki', 'Chemical engineering'
COLUMN                          CELL
 revision:author                timestamp=1277299531, value=Adrignola
 revision:comment               timestamp=1277299531, value=Redirected page to
[[Subject:Chemical engineering]]
 text:                          timestamp=1277299531, value=#REDIRECT [[Subject:Chemical
engineering]]
3 row(s) in 0.0470 seconds
```

Digression: CF are not columnar DBs (1)

- Data **stored** by columns, e.g. *C-Store*
- *Column oriented* DBMS http://en.wikipedia.org/wiki/Columnar_database
- Some vendors (*Oracle, Informix, Microsoft, …*) introduce columnar storage model (as indexes) into RDBMSs.



- ✓ Retrieves only the columns required to resolve queries (in a typical fact table, below 15%)
- ✓ Better compression
- ✓ Increased utilization of buffer (better compression, often used columns)

Digression: CF are not columnar DBs (2)

- ✓ Order of magnitude (sometimes several) faster query times
- Useful when: often read, seldom write
- E.g. Microsoft SQL Server 2012 Vertipaq*:
  - Test: 1 TB star join (1,44 billion rows), 32processors, 256 GB RAM:

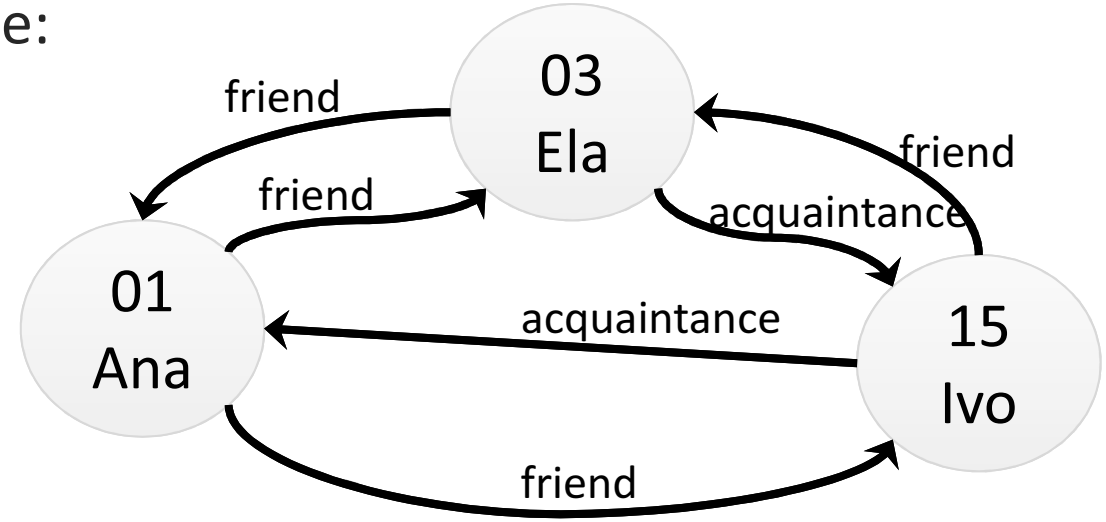
	Total CPU time (seconds)	Elapsed time (seconds)
Columnstore	31.0	1.10
No columnstore	502	501
Speedup	16X	455X

  - ✓ Can provide acceleration from hundreds to thousands of times, at least tenfold
  - ✓ Compression factor of 4-20 on real data
  - ✗ You can not do INSERT
  - ✗ 2-3 times slower index creation in comparison to the B-tree

*
http://download.microsoft.com/download/8/C/1/8C1CE06B-DE2F-40D1-9C5C-3EE521C25CE9/Columnstore%20Indexes%20for%20Fast%20DW%20QP%20SQL%20Server%2011.pdf

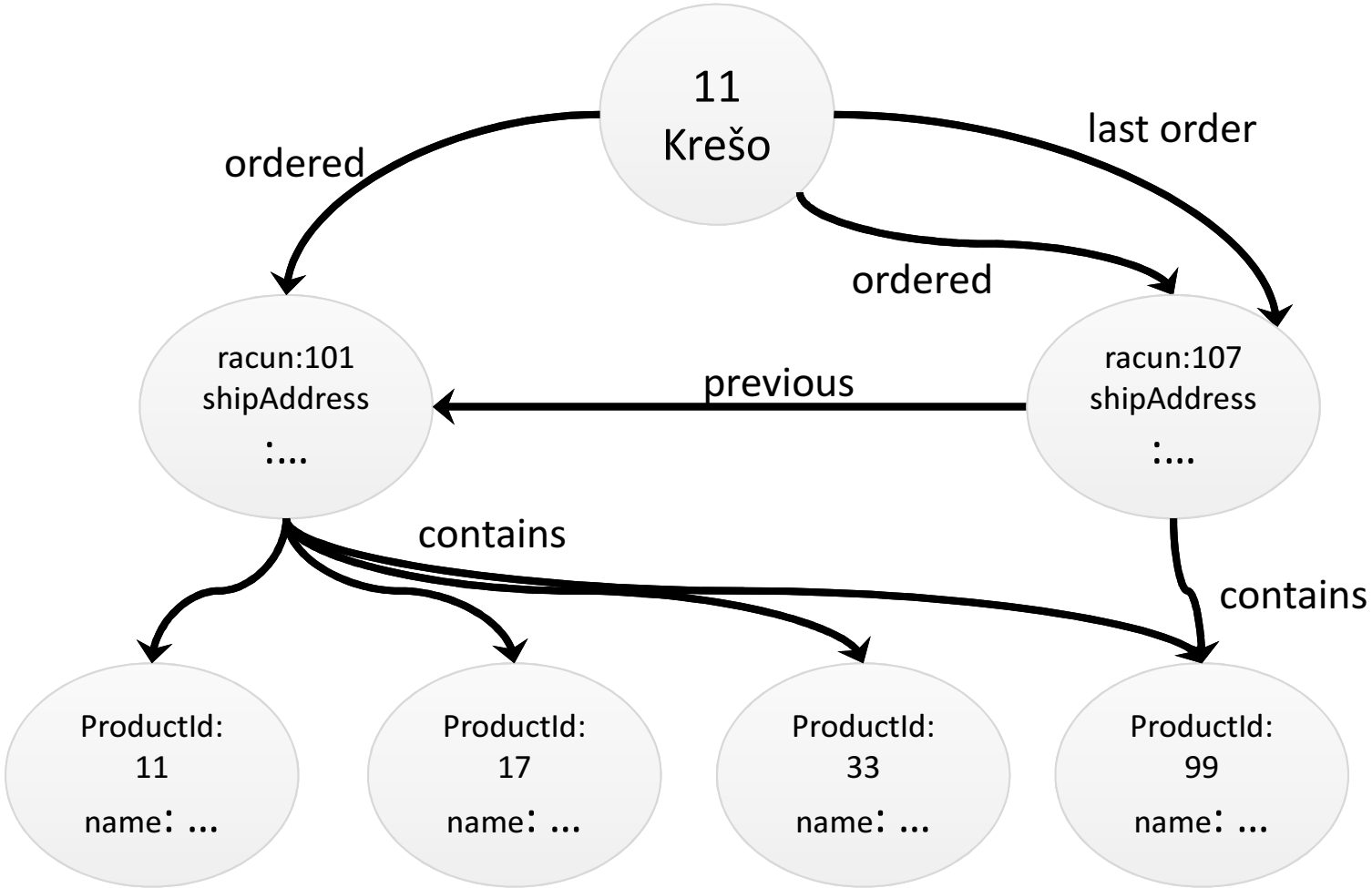
Graph databases

- **Data model: nodes, edges (arcs), properties:**
  - Nodes can have properties (KV pairs)
  - Edges have tags, directions and start and end node
  - Edges also have properties
- Interfaces and query languages are not standardized (*Cypher, SPARQL, Gremlin*)
- Example:



- Some DBs: *Neo4j, GraphDB, DEX, FlockDB, InfoGrid, OrientDB, Pregel, …*

Graph databases - example



Example: Neo4j

- "whiteboard friendly,"
- All about relationships
- In different „sizes":
  - Embedded
  - Cluster support, MS replication
- Can store tens of billions of nodes and edges

- Query languages:
  - Gremlin
  - Cypher
- Language binding
- Node/edge properties can be indexed (for query start objects)
  (uses *Lucene* to index)

Cypher

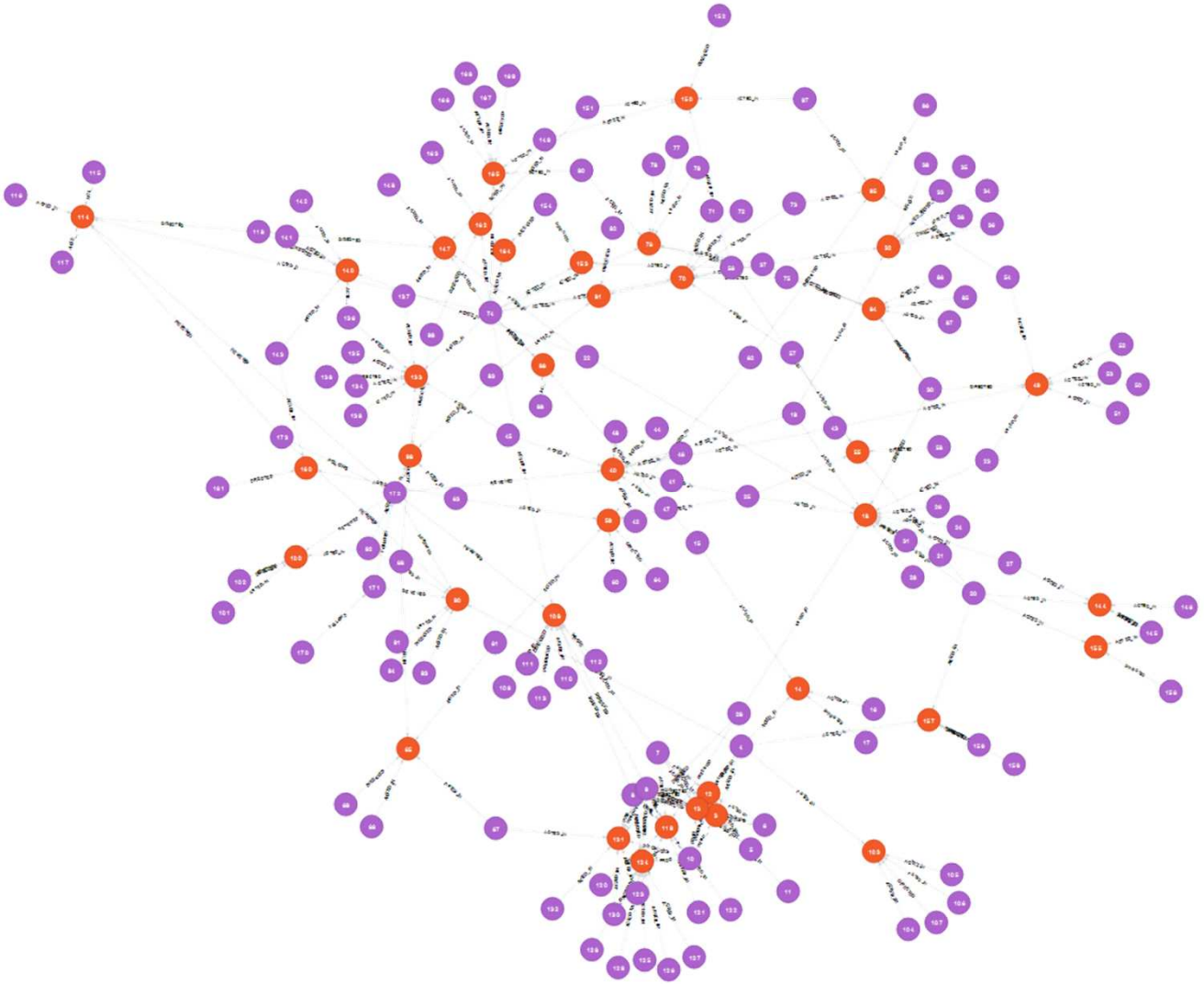
- See: http://docs.neo4j.org/refcard/1.9/
- On exam, you can be asked to give a simple Cypher query. Reference card will be available to you during the exam.

```
START
[MATCH]
[WHERE]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Example: insert

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the
Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (AndyW:Person {name:'Andy Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
CREATE
  (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
  (AndyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),
  (JoelS)-[:PRODUCED]->(TheMatrix)

CREATE (Emil:Person {name:"Emil Eifrem", born:1978})
CREATE (Emil)-[:ACTED_IN {roles:["Emil"]}]->(TheMatrix)

CREATE (TheMatrixReloaded:Movie {title:'The Matrix Reloaded', released:2003,
tagline:'Free your mind'})
...
```

Example: movies and actors



Cypher - examples

- Get a node having name attribute value equal to „Tom Hanks” :

```
MATCH (tom {name: "Tom Hanks"})
RETURN tom
```

- Get nodes with title = „Cloud Atlas”:

```
MATCH (cloudAtlas {title: "Cloud Atlas"})
RETURN cloudAtlas
```

- Get names of ten persons (node type is Person):

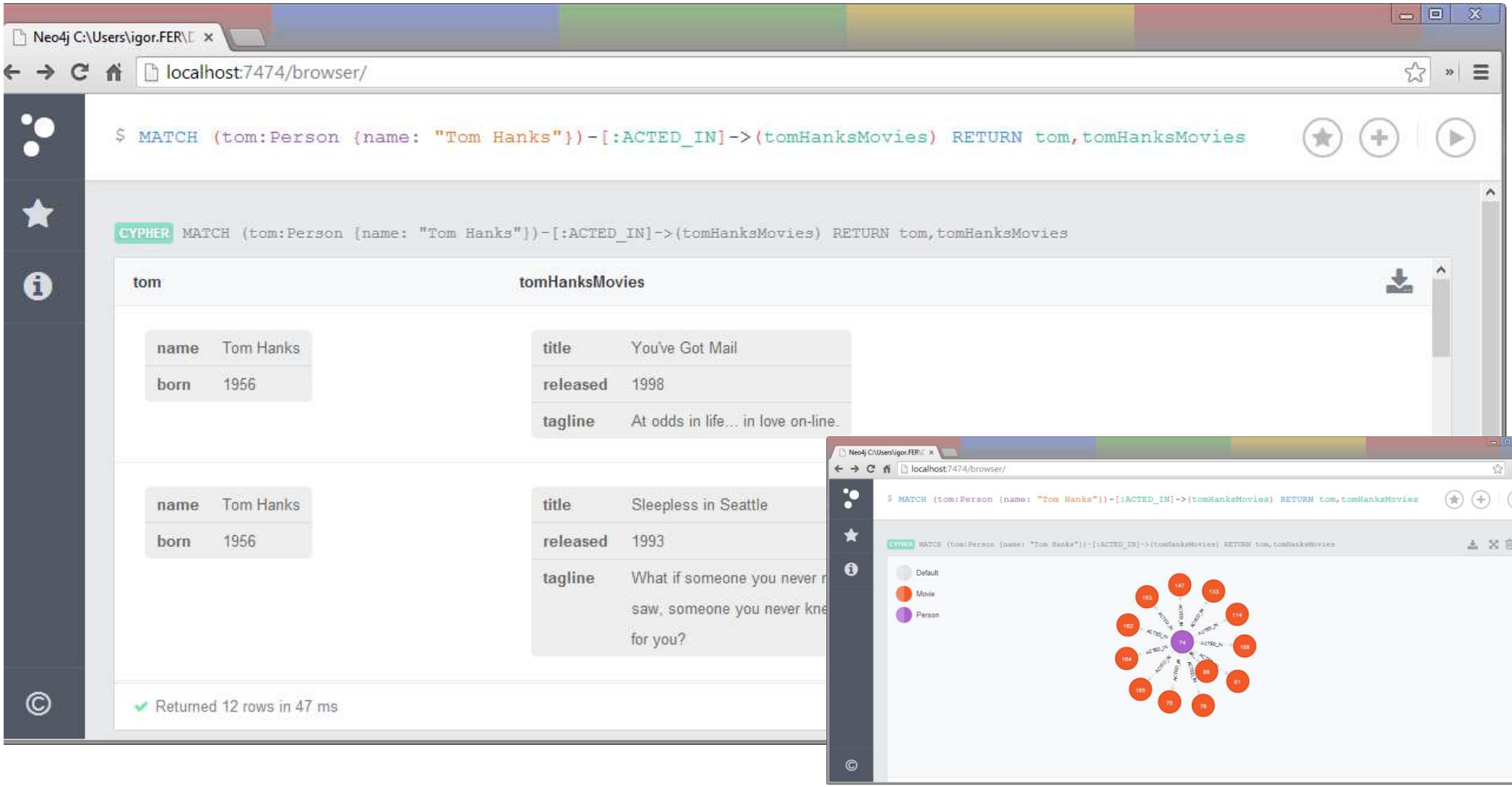
```
MATCH (people:Person)
RETURN people.name LIMIT 10
```

- Get movies from the 1990’s:

```
MATCH (nineties:Movie)
WHERE nineties.released > 1990 AND nineties.released < 2000
RETURN nineties.title
```

Example: movies starring TH

```
MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies)
RETURN tom,tomHanksMovie
```

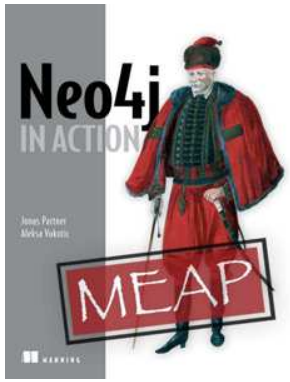


Relational databases and relationships

- *"relational databases deal poorly with relationships"* ☺
- Friends of friends of my friends? (reminder: FOAF, advanced SQL)

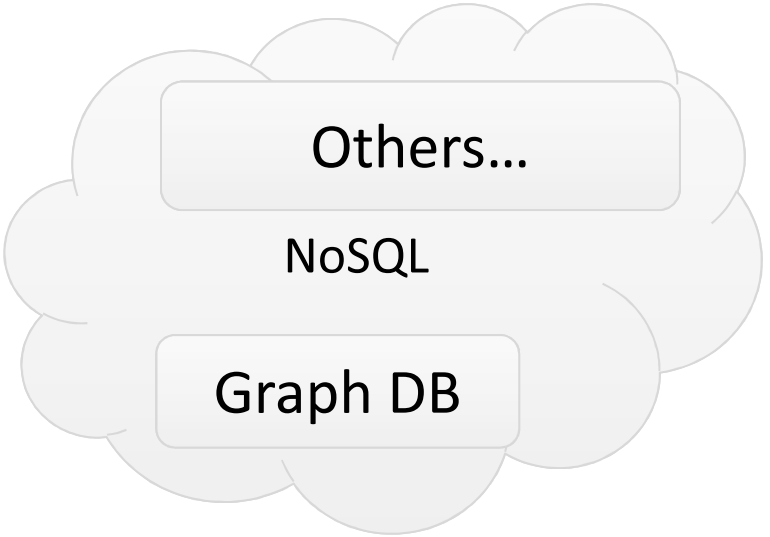
Depth	Execution Time – MySQL	Execution Time –Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	Not Finished in 1 Hour	2.132

http://www.neotechnology.com/how-much-faster-is-a-graph-database-really/

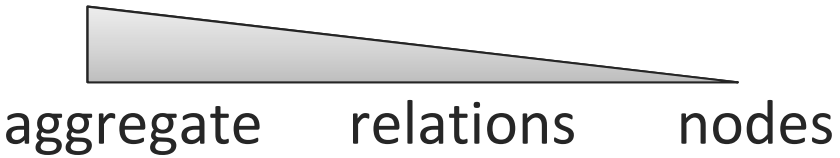


Graph database

- *"Strange fish in SQL pond"*



- Breaks the data into even smaller units than RDB



- Not suitable for distribution
- Query language
- ACID
- In common with others: non-relational model, popularity
- Suitable for complex, semi-structured, highly connected data

Schemaless databases

- *Schemaless*
- With RDBs the (fixed) schema has to be defined **beforehand**
- NoSQL databases are schemaless:
  - ✓ Can store "anything"
  - ✓ Data „format” can be easily changed, new atrributes added
  - ✓ Suitable for heterogeneous data
    - **Implicit schema** exists never the less – in the application ☹
  - ✗ DB has no knowledge of implicit schema – cannot use it for more efficient storage and retrieval
  - ✗ DB cannot enforce integrity
- What happens when multiple apps access the DB?
- No-schema flexibility valid only within aggregate boundaries

Schema migration

- Eg. Attribute name change, new attribute, new table ...
- RDB
  - SQL delta scripts, DBDiff
  - Tools for automatization: *DBDeploy, Liquibase, DBMantain*
  - Problem, with multiple apps, especially legacy apps
  - *Transition phase*
- NoSQL
  - Small changes are trivial (eg. adding an attribute), otherwise no easier than the RDB
  - Incremental migration:
    - *schema version*
    - Gradually changing on save ☹
  - In GraphDB, what happens when the edge tag changes?
    - We can maintain parallel edges (old + new), add metadata (version, timestamp, etc.)

Materialized views

- Inspired with RDB's views
- NoSQL don't have views
- Materialized views – queries evaluted beforehand and (their results) stored on the *disk*
- E.g. Particular product group sales in the current month
- Suitable for multiple read, few writes data
- Two approaches:
  - *Eager* – update on insert
  - *Stale* – update asynchronously after the insert. Can also be done outside of the DB and stored in the DB.

Data model - conclusion

- In data modelling, a general rule is that the data be denormalized on insert (write), in order to be suitable for the targeted reads
- Aggregate DBs (KV,D,CF) have difficulties dealing with relatinships crossing the aggregate boundaries
- Graph DBs decompose data to " small" parts. They are suitable for data that have complex interdependencies
- Schemaless DBs are more flexible on writes, make it easy to change the attributes to be entered, but the implicit scheme still exists - "somewhere" in the application code.

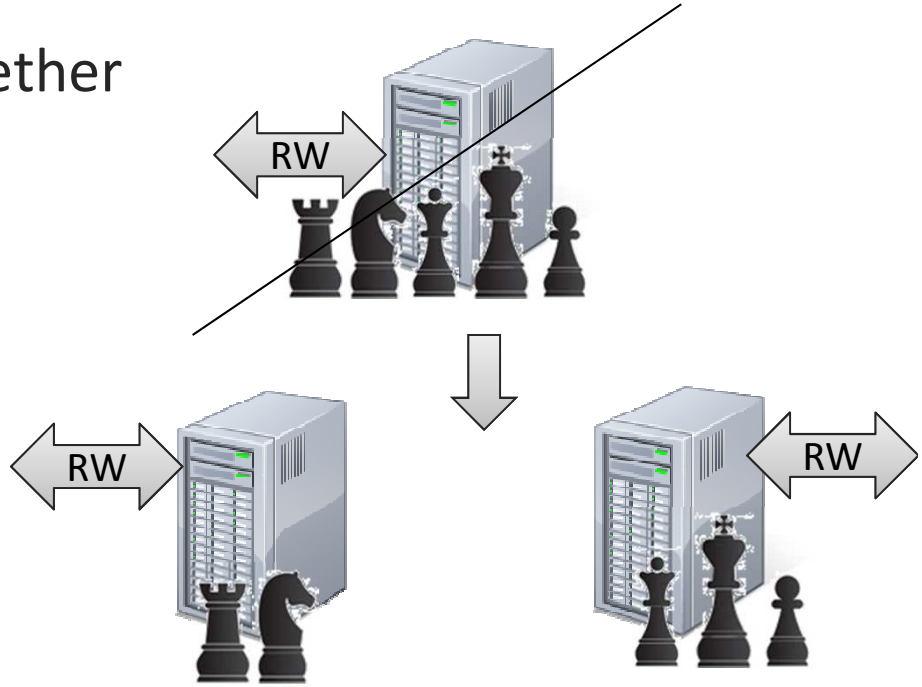
Distribution models

Data distribution

- Popularity of NoSQL systems ~ executing on clusters
- Distribution:
  - ✓ Can handle larger data volumes
  - ✓ Bigger R/W throughput
  - ✓ Better availability
  - ✗ More complex, new issues
- Two modes of distribution:
  - Fragmentation (sharding)
  - Replication
- The best: non-distributed ☺

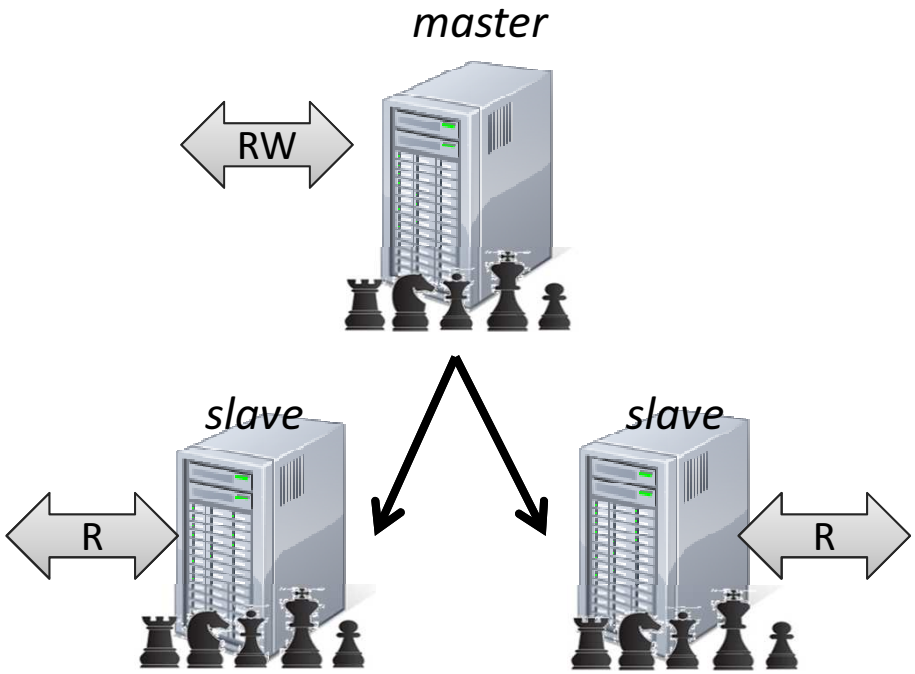
Sharding

- Join data that often accessed together - Aggregates
- How to distribute across servers?
  - Geographically
  - Uniformly
  - Domain rules (eg. Domain names in *BigTable*)
- Auto-sharding - DB does the sharding
- Improves reads and writes
- Does not improve the error resilience, even contrary (more servers to administer!)
- When to shard (in the start, or later)?



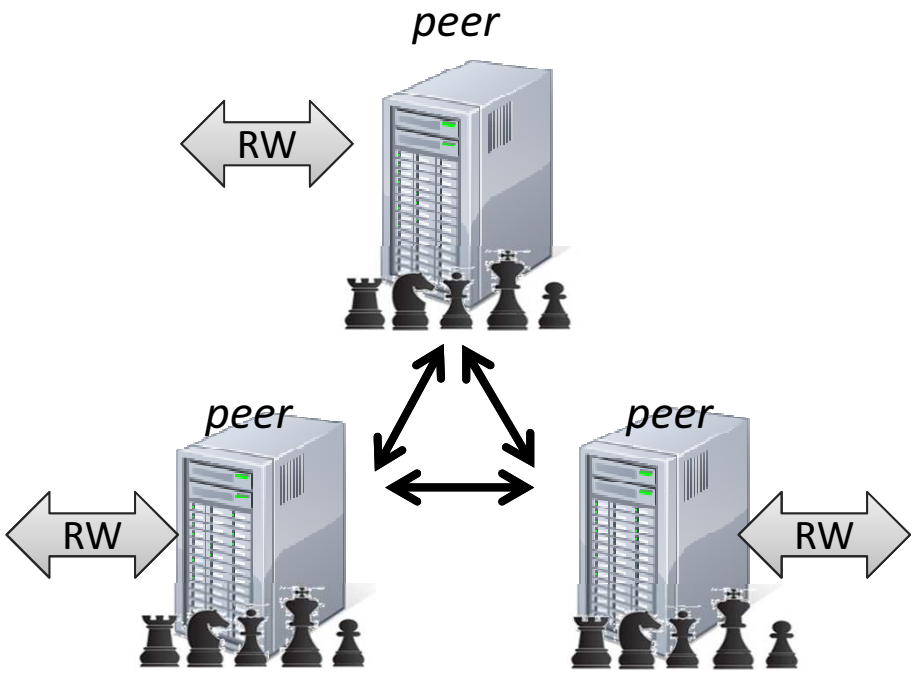
Replication: *Master-Slave*

- ✓ Useful for scaling when there are **many reads**
- Read resilience – should the master fail, you can still read
- ✓ Fast recovery – new master election (~ *hot backup*):
  - Manually (configured)
  - Automatically("elections")
- Read resilience – different connections for R & W (how?)
- ✗ Inconsistency (what happens if the master fails?)

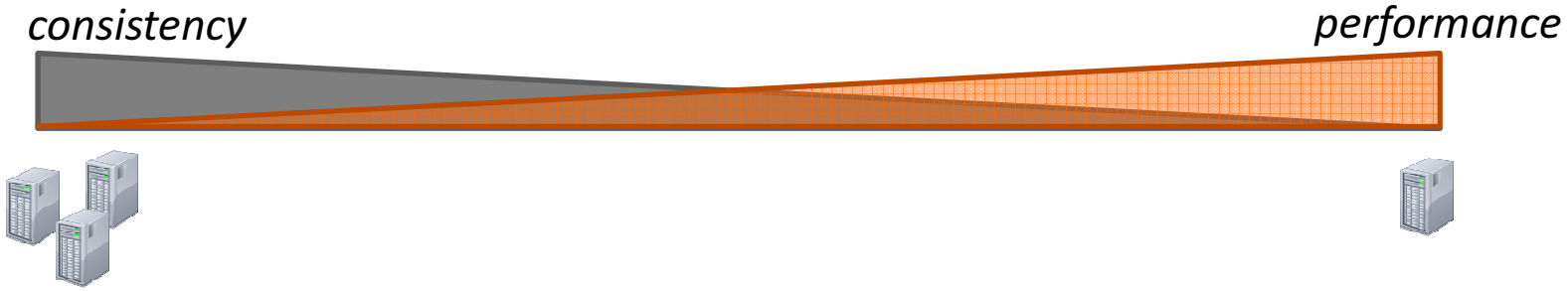


Replication: Peer-to-Peer

- ✓ Useful for scaling on both **reads and writes**
- Equal nodes = peers
- ✓ It is trivial to boost performance:
  - add a node!

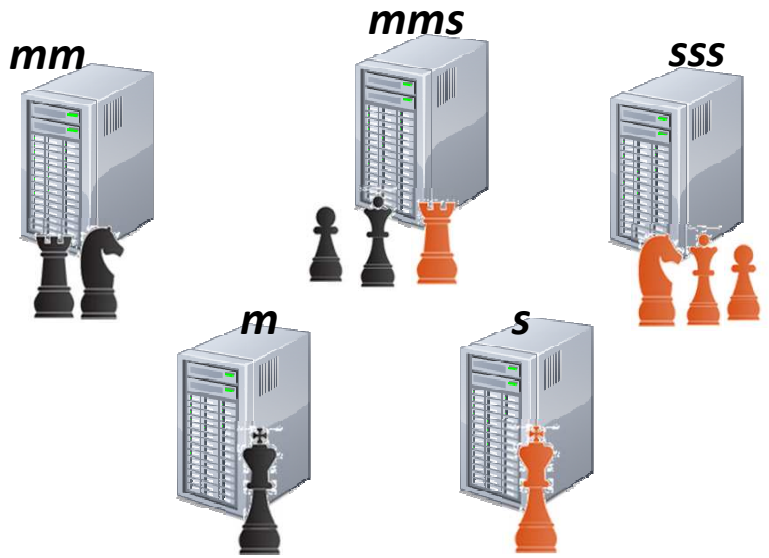


- ✗ **Inconsistency:**
  - ✗ R (same as MS), transient
  - ✗ WW conflict - *inconsistent writes are forever* ☹

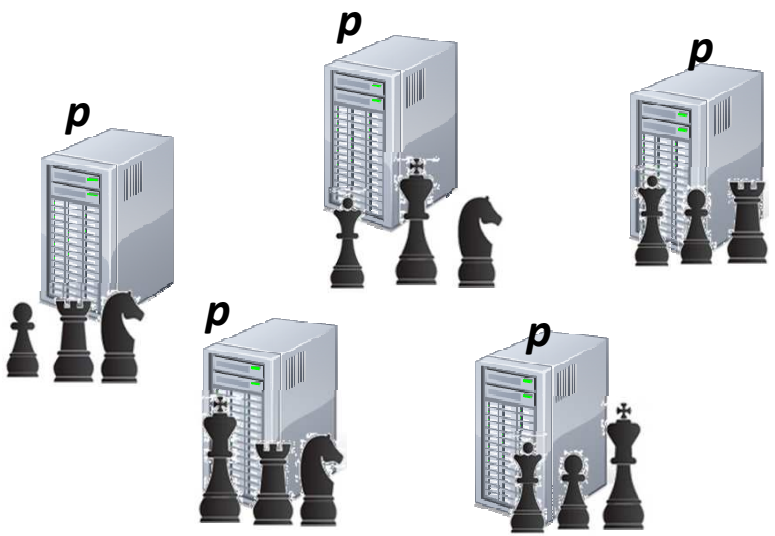


Sharding + replication

- MS+R=1



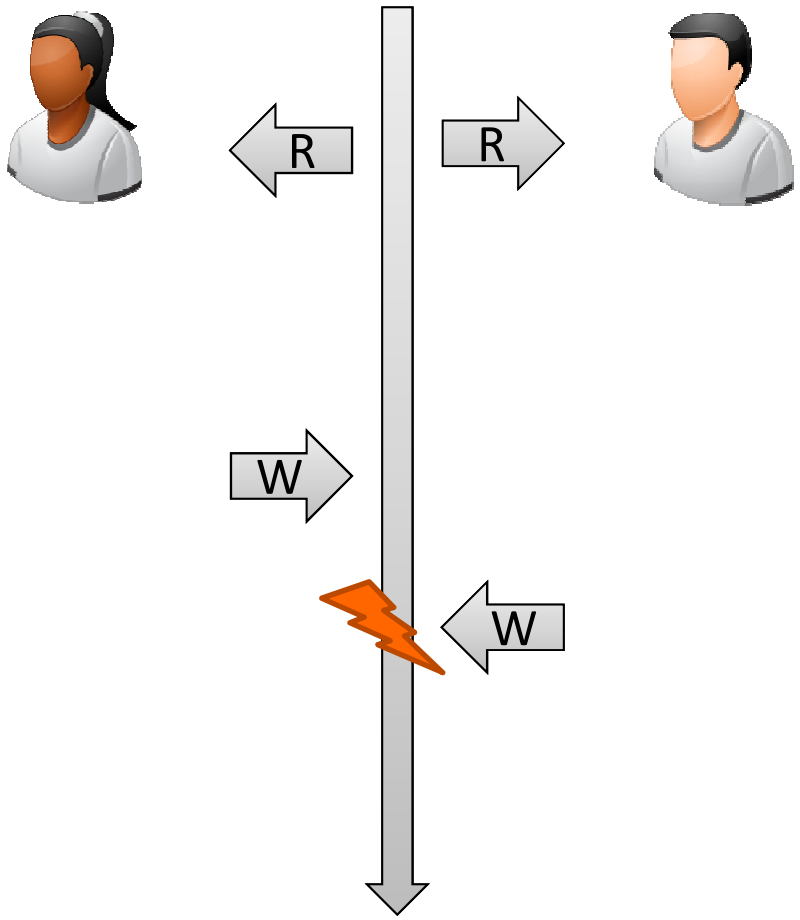
- P2P+R=3



Consistency

Consistency on writes - example (1)

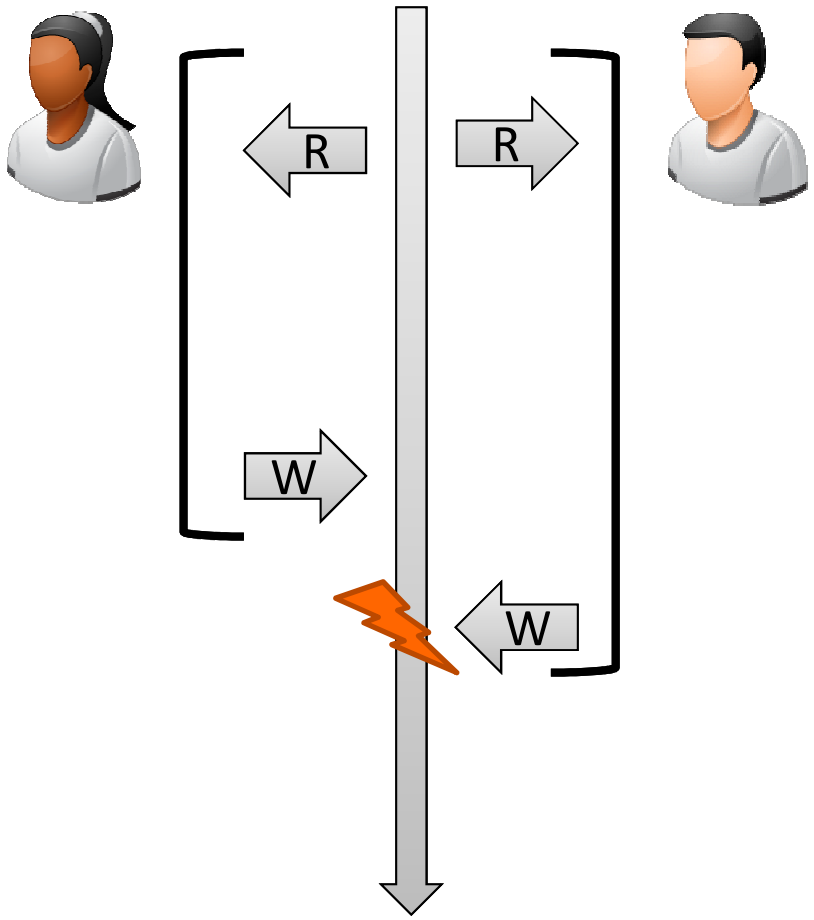
- RDB - *strong consistency*
- Nevertheless, potential problems,
- Example:



- Solutions?

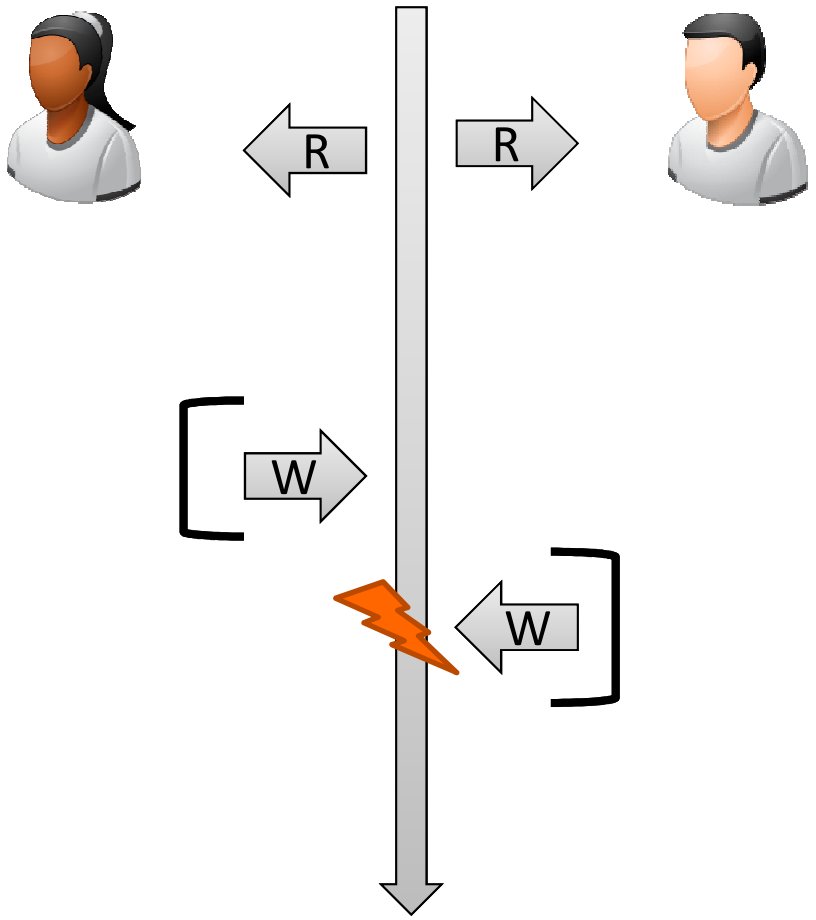
Consistency on writes - example (2)

- Solution:
  transactions (performance?)
- Suitable for a small number of users



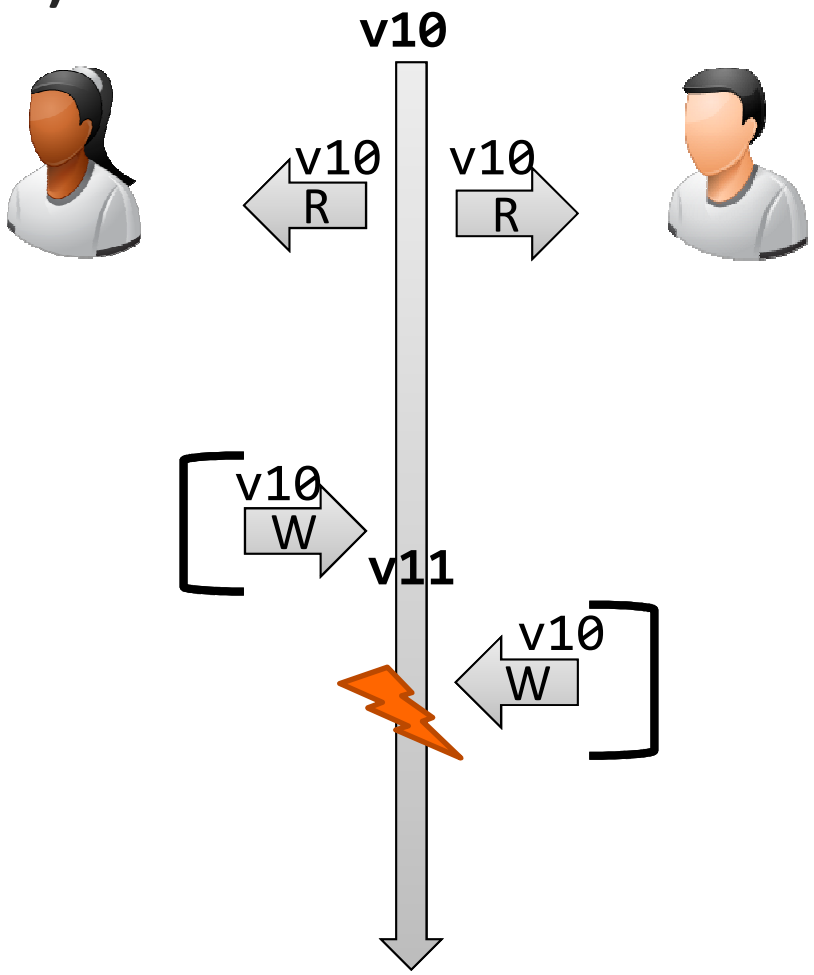
Consistency on writes - example (3)

- ✓✗ Solution:
  transactions(performance?)
- ✗ Still, WW conflict



Consistency on writes - example (4)

- **Better solution: *Offline locks***
  (i.e. versions, conditional update)

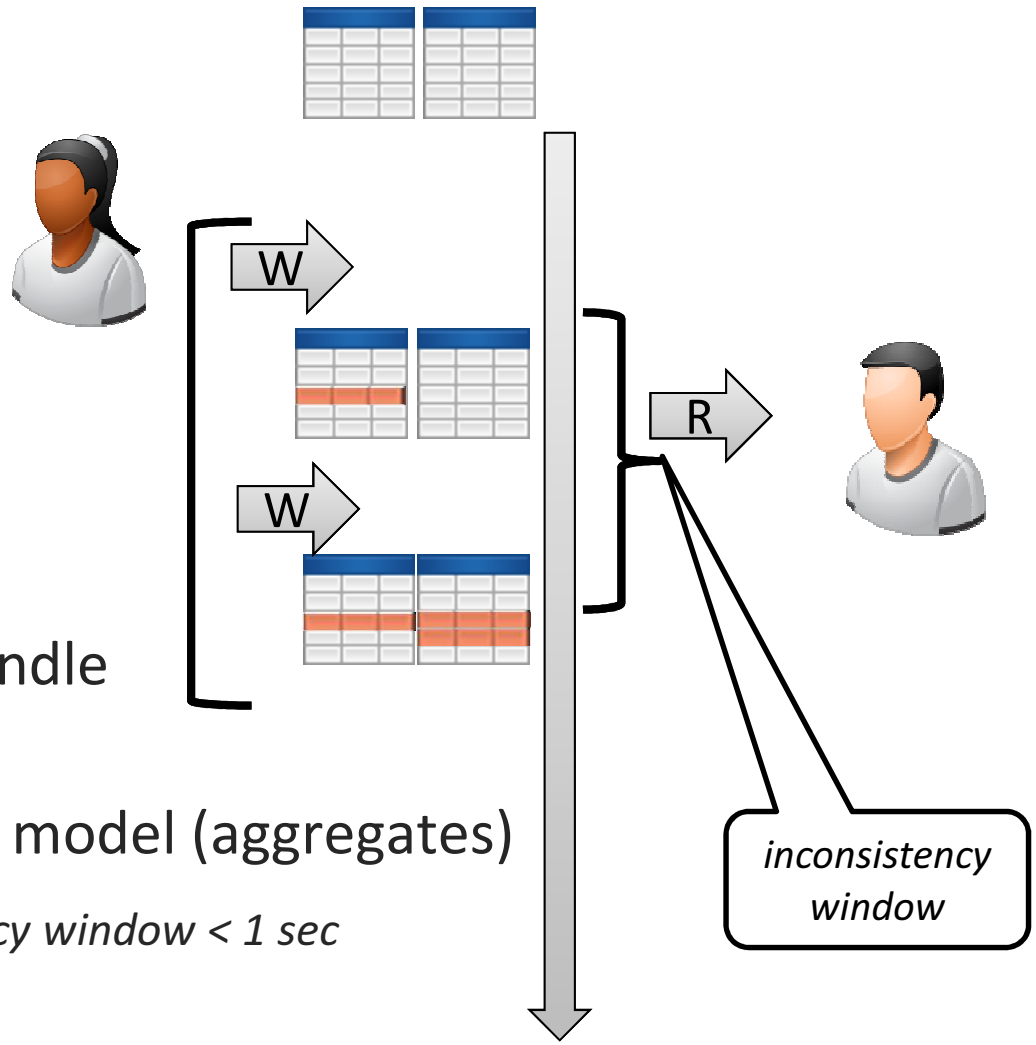


Consistency on writes

- E.g. Two users changing mutual info on their webpage – *lost update*
- Two approaches:
  - **Pessimistic** – prevent WW conflict (*write locks*)
  - **Optimistic** - allow, detect, resolve (e.g. *conditional update* (previous example), *automatic merge ~ CVS*)
- Both approaches rely on a consistent order of actions
  - In a single server mode – trivial – pick one or the other update
  - P2P? – different values
- How to do that in distributed environment?
- One solution – W via a single node
- Conflicts must be resolved

Consistency on reads (1)

- **Logical (in)consistency**
  = make sure different objects make sense together
- Example:
  inconsistent read
  or *RW conflict*



✓ RDBs use transactions to handle that

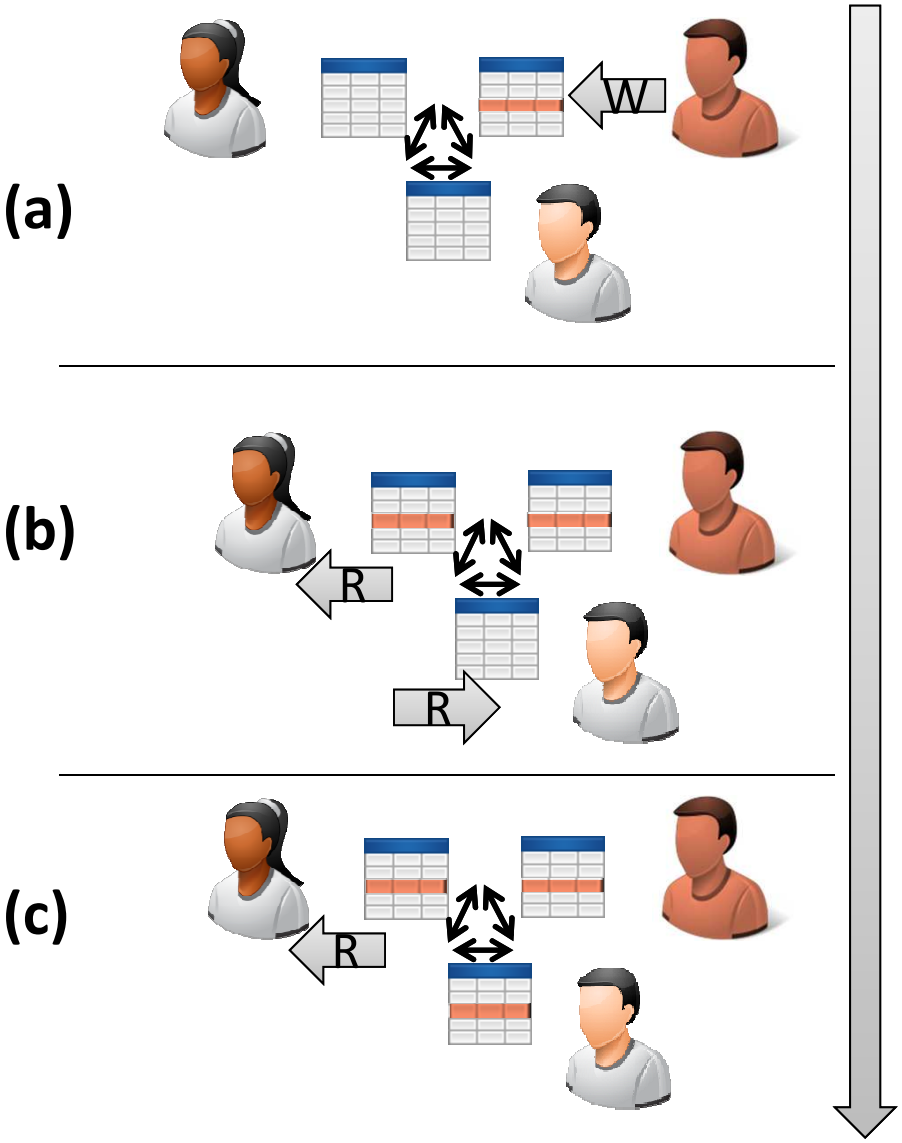
✓✗ NoSQL somewhat with data model (aggregates)

- E.g. *Amanzon SimpleDB inconsistency window < 1 sec*

Consistency on reads (2)

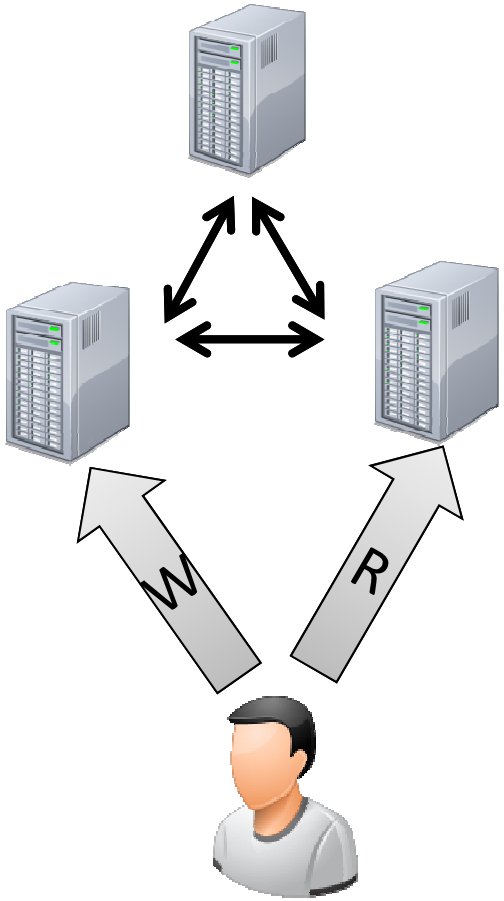
- **Replication (in)consistency = m**ake sure **all replicas of the same data** have the **same value**
- Inconsistency in (b)
- (c) -*"Eventually consistent"*
- Independent of logical inc., but:
- Replication can elongate the *inconsistency window* of logical consistency

- Consistency is not a global application property, usually can be set *per request*: sometimes weak, sometimes strong



Consistency on reads (3)

- **Replication (in)consistency**
- Example: *blog post*
- Read-your-writes consistency
- Solution:
  - *Sticky session, session affinity*
    (can the slave, in MS architecture, temporarily take over W duty?
    -> out of the ordinary behaviour)
  - *Version stamps*



Relaxing consistency

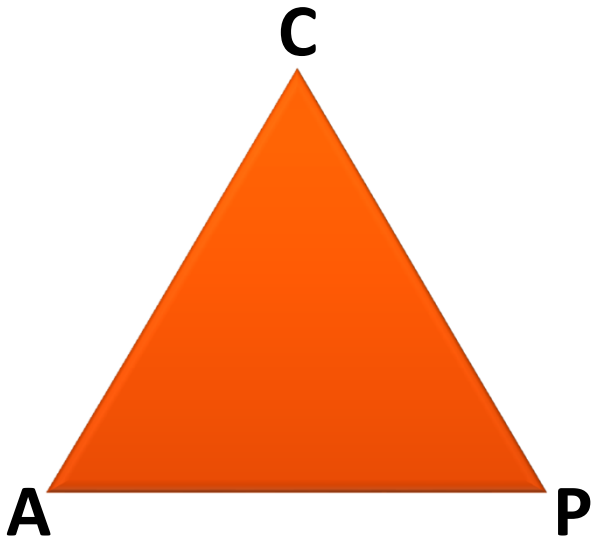
- Compromise between performance and consistency
- Even RDBs can do it - isolation levels
- For example, MySQL was popular before it supported transactions
- *eBay*
- *Amazon*

CAP theorem

- E. Brewer, 2000.: *Towards Robust Distributed Systems*
- Dokazan 2002. - *Lynch & Gilbert*
- ***C**onsistency, **A**vailability, **P**artition tolerance*

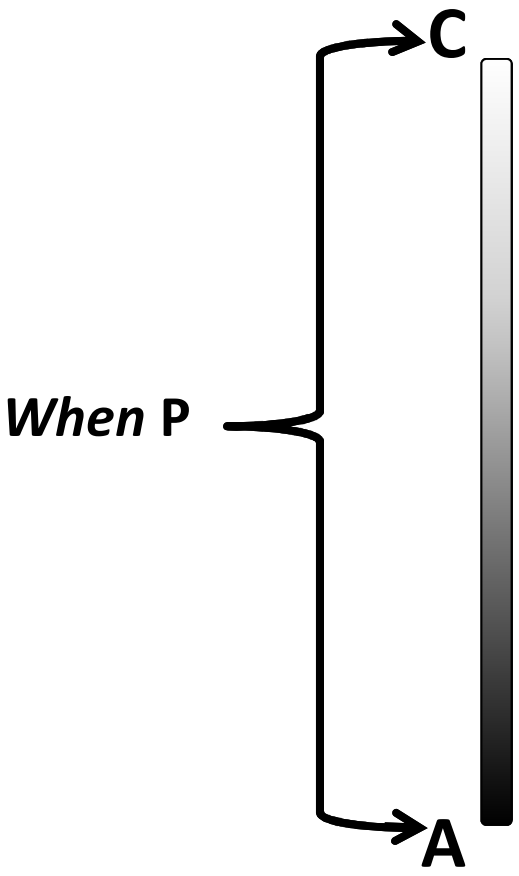
In distributed systems it is possible to achieve only **two out of the three** properties.

- ***Consistency*** (**Cap** <> a**C**id):
  - Every response sent to client is accurate
- ***Availability***
  - Each request, received by a functioning server, must result in response (both R & W)
- ***Partition tolerance***
  - The system is working even when „islands" (isolated sets) of computes occur

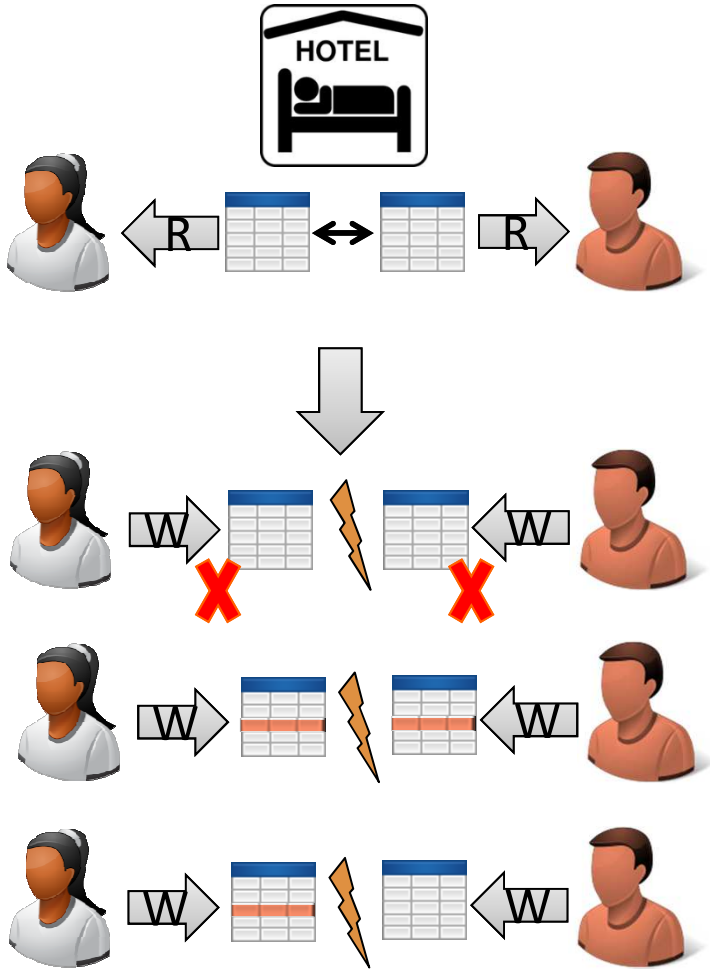


Za "one koji žele znati više": http://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

CAP theorem - reformulated



- Example:

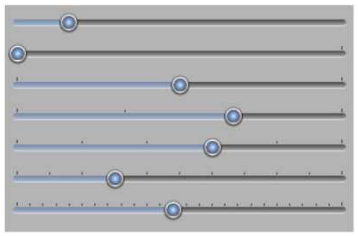


or: **Response time (latency), example: *Amazon shopping cart***

BASE

- For many applications, availability and partition tolerance are more important than strict consistency (e.g. (large) web applications, search engines, ...)
- BASE:
  - ***B**asically **A**vailable* – app is practically always available (regardless occasional errors)
  - ***S**oft-state* – not always consistent („soft state"), system is ever-changing, fluid
  - ***E**ventual consistency* – will be, eventually, in some known state (changes will be eventually propagated for all to see)

ACID
Strong consistency
Isolation
Availability?
Pessimistic (conservative)
…



not exc. or

BASE
Weak consistency(stale data)
Availability first
Approximate answers are acceptable
Aggressive (optimistic)
…