



# Scalable data processing, NoSQL

**Gergely Lukács**

Pázmány Péter Catholic University

Faculty of Information Technology

Budapest, Hungary

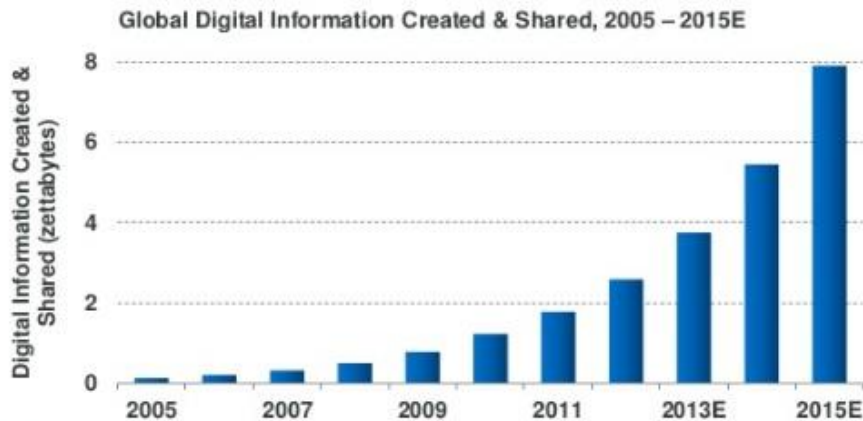
lukacs@itk.ppke.hu

NoSQL databases

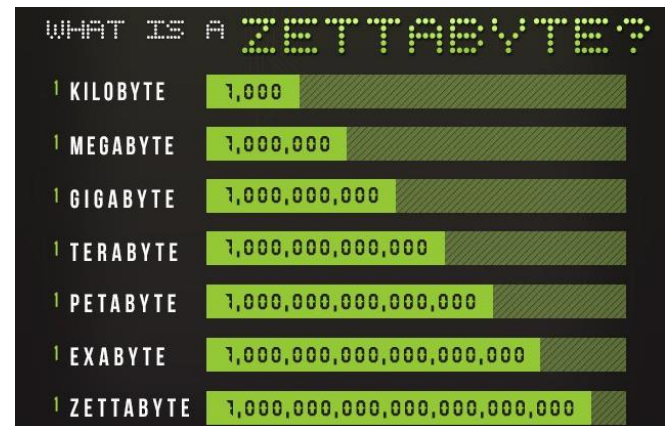
# How much data?

- Google processes 20 PB a day (2008)
- Wayback Machine has 3 PB + 100 TB/month (3/2009)
- Facebook has 2.5 PB of user data + 15 TB/day (4/2009)
- eBay has 6.5 PB of user data + 50 TB/day (5/2009)
- CERN's LHC will generate 15 PB a year (??)

Amount of data doubles  
every 20 months



Note: \* 1 zettabyte = 1 billion gigabytes. Source: IDC report "Extracting Value from Chaos" 6/11.



# NoSQL: The Name

- “SQL” = Traditional relational DBMS
  - efficient, reliable, convenient, and safe multi-user storage of and access to massive amounts of persistent data
- Recognition over past decade or so:  
Not every data management/analysis problem is best solved using a traditional relational DBMS
  - Web-based systems!
- “NoSQL” ( “No SQL” )
  - Not using traditional relational DBMS
  - **Not Only SQL**

# NoSQL – Definition ?

- Heterogeneous group of concepts, systems
  - Key-value stores
  - Wide column stores
  - Document stores
  - ...

# NoSQL – Advantages

- Depend on system/category (heterogeneous concept!!)
- Higher performance
- Easy distribution of data on nodes (sharding): scalability, fault tolerance
- Flexibility: schema free data model
- Simpler administration

# NoSQL – Methods

- No normalised relational data model
- Transaction management relaxed (ACID->BASE), fewer guarantees
  - **Basically available:** Nodes in the a distributed environment can go down, but the whole system shouldn't be affected.
  - **Soft State** (scalable): The state of the system and data changes over time.
  - **Eventual Consistency:** Given enough time, data will be consistent across the distributed system.
- Less powerful querying

# CAP Theorem

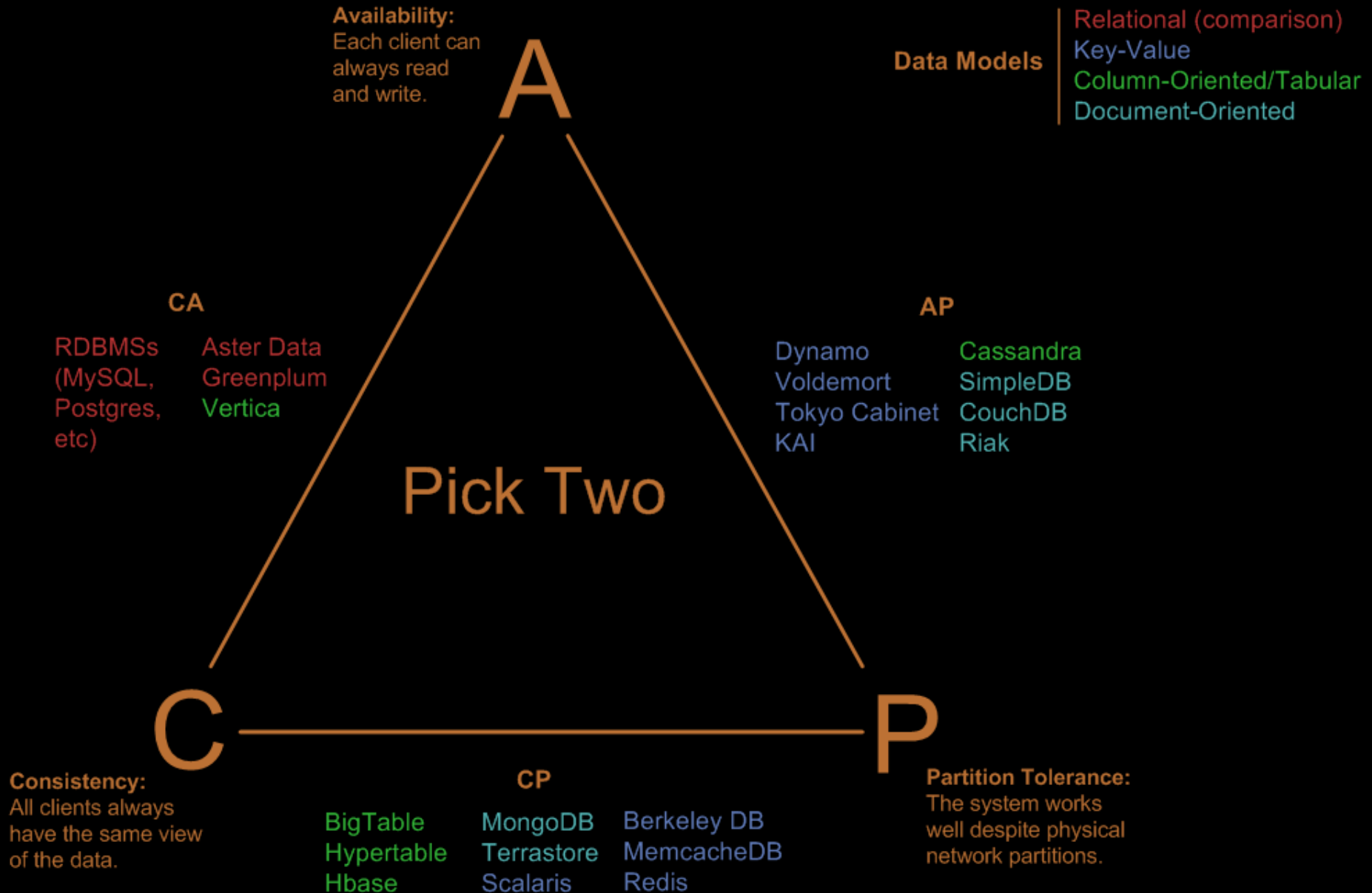
- Also known as Brewer's Theorem by Prof. Eric Brewer, published in 2000 at University of Berkeley.
- “Of three properties of a shared data system: data consistency, system availability and tolerance to network partitions, only two can be achieved **at any given moment**.”
- Proven by Nancy Lynch et al. MIT labs.
- <http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf>



# CAP Semantics

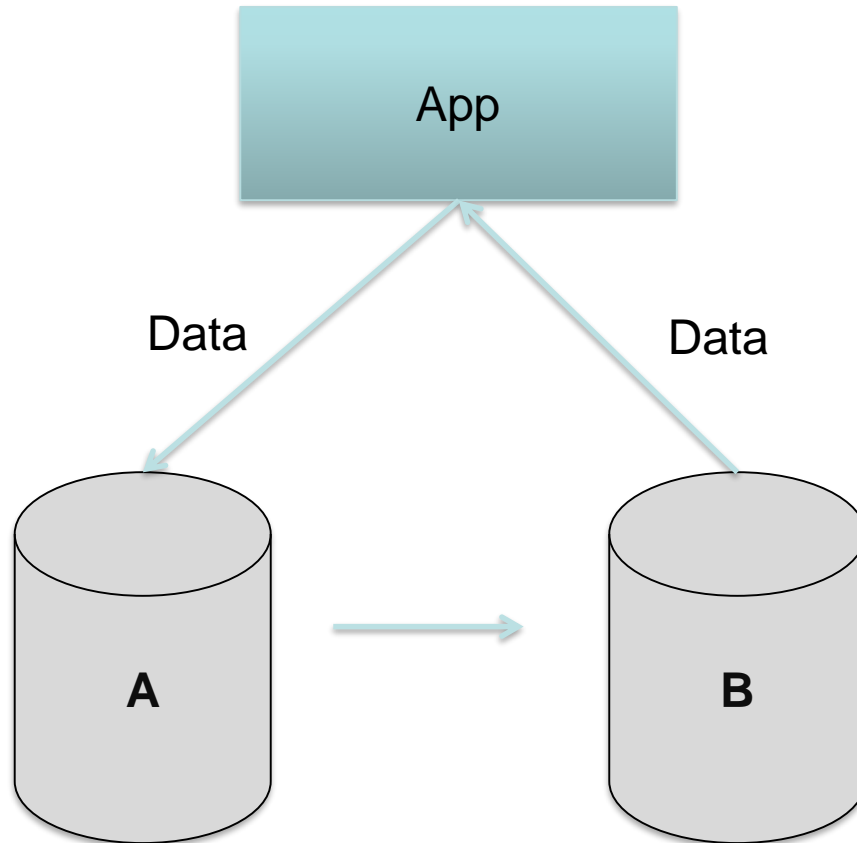
- **Consistency:** Clients should read the same data. There are many levels of consistency.
  - Strict Consistency – RDBMS.
  - Tunable Consistency – Cassandra.
  - Eventual Consistency – Amazon Dynamo.
- **Availability:** Data to be available.
- **Partition Tolerance:** Data to be partitioned across network segments due to network failures.

# Visual Guide to NoSQL Systems



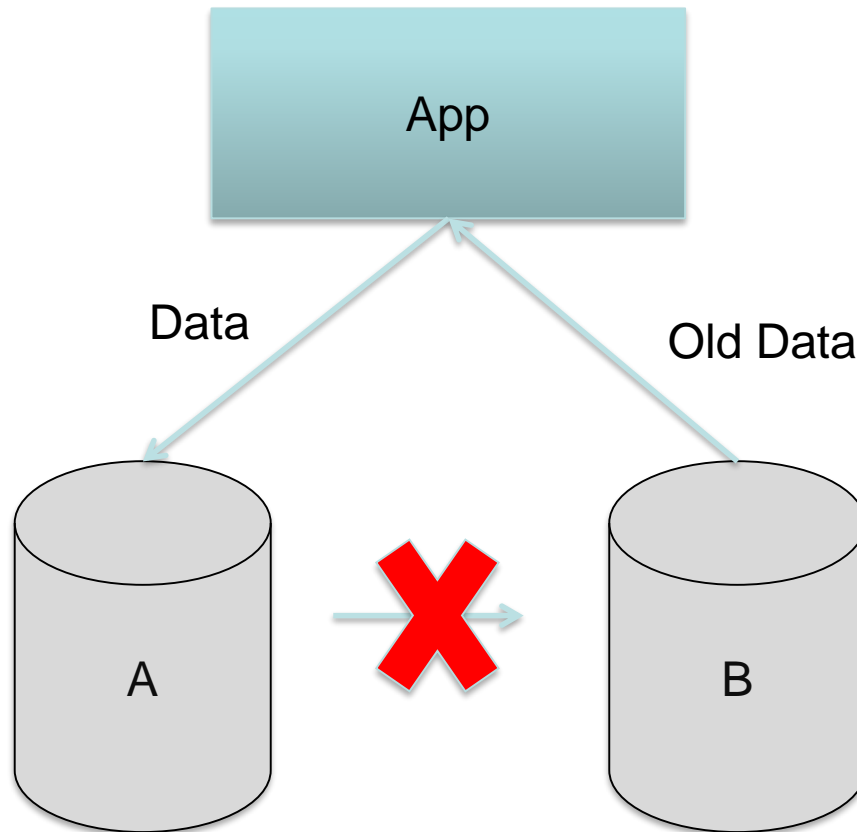
# A Simple Proof

Consistent and available  
No partition.



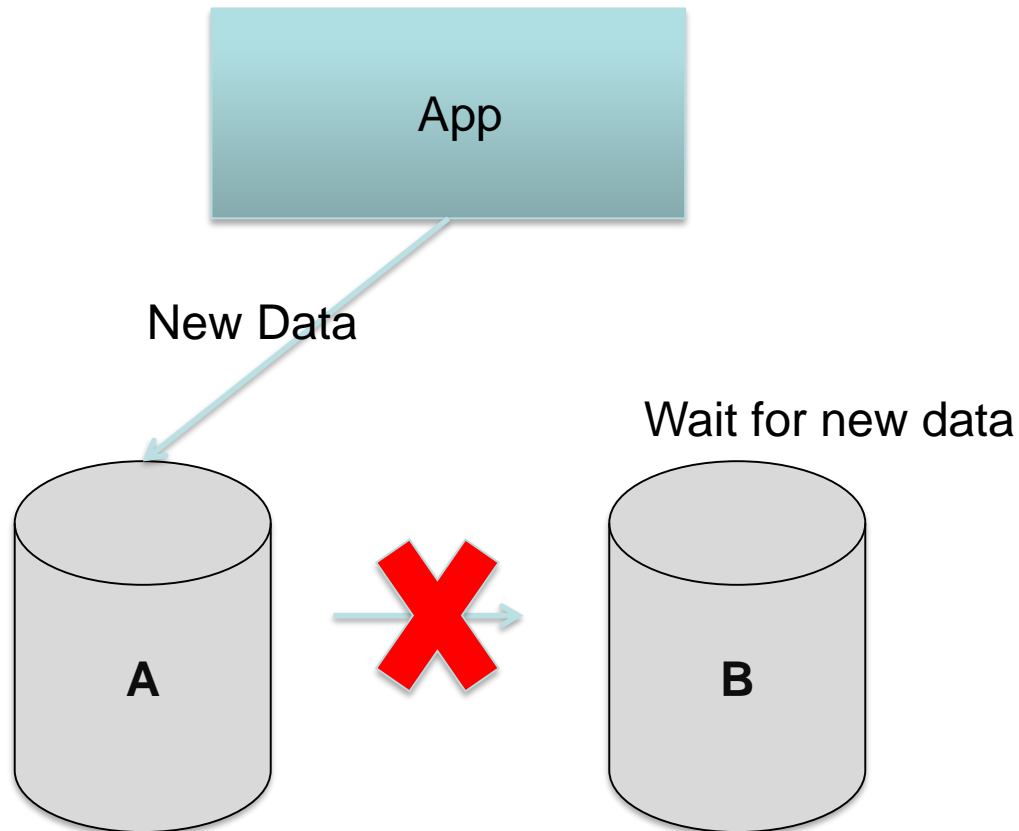
# A Simple Proof

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



# A Simple Proof

Consistent and partitioned  
Not available, waiting...



# Google Cloud Spanner

Spanner is Google's highly available global SQL database [CDE+12]. It manages replicated data at great scale, both in terms of size of data and volume of transactions. It assigns globally consistent real-time timestamps to every datum written to it, and clients can do globally consistent reads across the entire database without locking.

The CAP theorem [Bre12] says that you can only have two of the three desirable properties of:

- C: Consistency, which we can think of as serializability for this discussion;
- A: 100% availability, for both reads and updates;
- P: tolerance to network partitions.

This leads to three kinds of systems: CA, CP and AP, based on what letter you leave out. Note that you are not entitled to 2 of 3, and many systems have zero or one of the properties. For **distributed systems** over a “wide area”, it is generally viewed that **partitions are inevitable**, although not necessarily common [BK14]. Once you believe that partitions are inevitable, any distributed system must be prepared to forfeit either consistency (AP) or availability (CP), which is not a choice anyone wants to make. In fact, the original point of the CAP theorem was to get designers to take this tradeoff seriously. But there are two important caveats: first, you **only** need forfeit something **during an actual partition**, and even then there are many mitigations (see the “12 years” paper [Bre12]). Second, the actual theorem is about 100% availability, while the interesting discussion here is about the **tradeoffs involved for realistic high availability**.

# Key-value stores

# Examples for Data

## Extremely simple interface

- Data model: (key, value) pairs

Color	Red
Age	18
Size	Large
Name	Smith
Title	The Brown Dog

user1923_color	Red
user1923_age	18
user3371_color	Blue
user4344_color	Brackish
user1923_height	6' 0"
user3371_age	34

- Operations
  - Insert(key,value)
  - Fetch(key)
  - Update(key, value)
  - Delete(key)
















# Key-Value Stores

Implementation: efficiency, scalability, fault-tolerance

- Records distributed to nodes based on key
- Replication
- Single-record transactions, “eventual consistency”

52 systems in ranking, April 2016

Rank			DBMS	Database Model	Score		
Apr 2016	Mar 2016	Apr 2015			Apr 2016	Mar 2016	Apr 2015
1.	1.	1.	Redis 	Key-value store	111.24	+5.02	+16.69
2.	2.	2.	Memcached	Key-value store	28.01	-1.23	-6.04
3.	3.	3.	Amazon DynamoDB 	Multi-model 	23.12	+0.89	+8.54
4.	4.	4.	Riak KV 	Key-value store	11.49	-0.60	-1.60
5.	5.	 6.	Hazelcast	Key-value store	6.68	-0.13	+1.00
6.	6.	 5.	Ehcache	Key-value store	6.46	-0.25	-1.25
7.	7.	 8.	OrientDB	Multi-model 	6.31	-0.35	+2.93
8.	8.	 11.	Aerospike	Key-value store	4.20	+0.10	+1.71
9.	 10.	9.	Oracle Coherence	Key-value store	3.13	-0.24	-0.19
10.	 9.	 7.	Berkeley DB	Key-value store	3.06	-0.38	-0.86
11.	11.	 10.	Amazon SimpleDB	Key-value store	2.96	+0.05	-0.24
12.	12.	12.	Oracle NoSQL	Key-value store	2.51	-0.02	+0.39

# Document stores

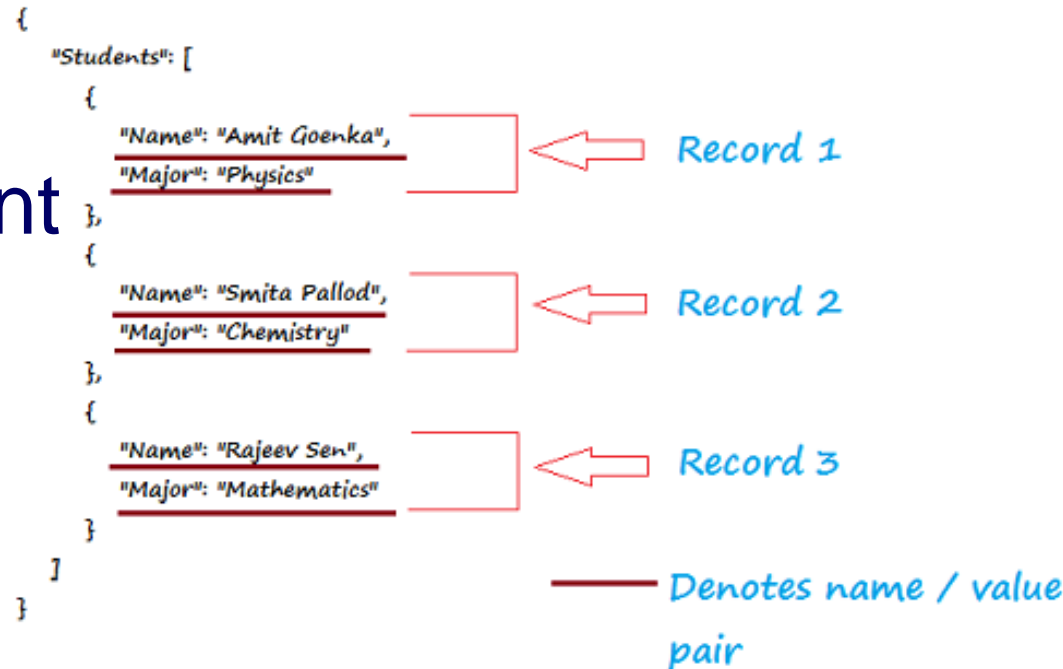
# Document Stores

- Like Key-Value Stores except value is document

- XML
- YAML
- JSON
- BSON

- binary forms











- PDF
- Microsoft Office documents (MS Word, Excel, and so on).



# Document Stores

- Data model: (key, document) pairs
  - Basic operations:
    - Insert(key,document), Fetch(key), Update(key), Delete(key)
  - Also Fetch based on document contents

39 systems in ranking, April 2016

Rank			DBMS	Database Model	Score		
Apr 2016	Mar 2016	Apr 2015			Apr 2016	Mar 2016	Apr 2015
1.	1.	1.	MongoDB 	Document store	312.44	+7.11	+33.85
2.	2.	 3.	Couchbase 	Document store	25.02	-0.78	-0.55
3.	 4.	 4.	Amazon DynamoDB 	Multi-model 	23.12	+0.89	+8.54
4.	 3.	 2.	CouchDB	Document store	22.36	-1.02	-4.58
5.	5.	5.	MarkLogic	Multi-model 	9.12	-0.25	-1.09

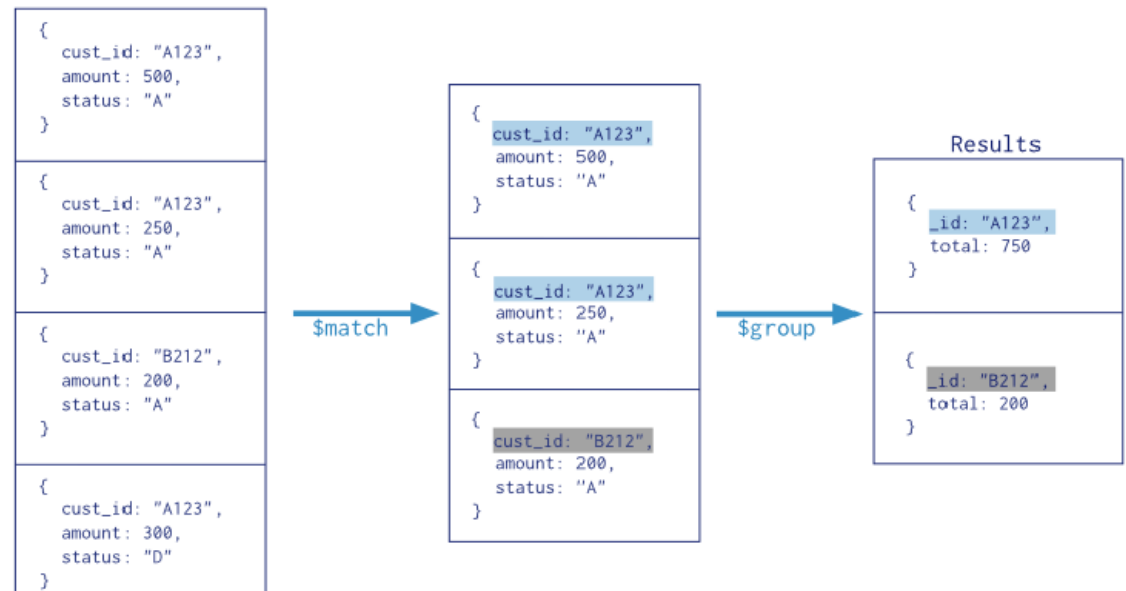
# MongoDB

- High performance
- High availability
- Horizontal scalability
  - Sharding: distributing data across a cluster of machines
- Rich query language
  - Data aggregation
  - Text search
  - Geospatial queries

# MongoDB - aggregation

- Aggregation pipeline

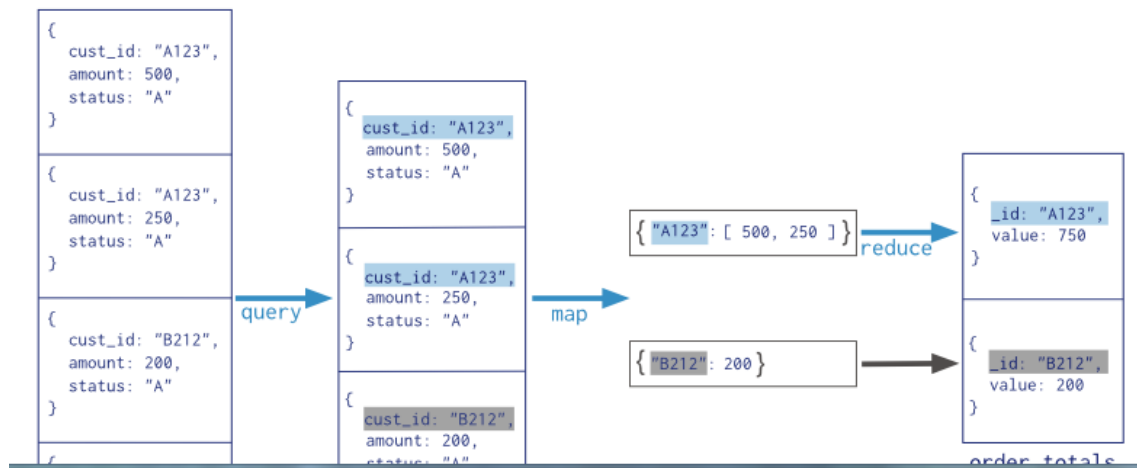
Collection  
↓  
`db.orders.aggregate( [`  
    \$match stage → `{ $match: { status: "A" } },`  
    \$group stage → `{ $group: { _id: "$cust_id", total: { $sum: "$amount" } } }`  
`]` )



# MongoDB - aggregation

- Aggregation MapReduce

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 → "order\_totals"  
})



# MongoDB - aggregation

- Single purpose aggregation operations
  - `Db.collection.count()`
  - `Db.collection.group()`
  - `Db.collection.distinct()`



# MongoDB – geospatial queries

The screenshot shows a web browser window with the URL <https://docs.mongodb.org/manual/tutorial/geospatial-tutorial/>. The page is titled "Find Restaurants with Geospatial Queries" and is part of the "Indexes > 2dsphere Indexes" section. The left sidebar contains a navigation menu with links to Introduction, Installation, The mongo Shell, MongoDB CRUD Operations, Aggregation, Text Search, Data Models, Administration, and Indexes. The main content area has a heading "Find Restaurants with Geospatial Queries" and a section "On this page" with links to Overview, Differences Between Flat and Spherical Geometry, Distortion, and Searching for Restaurants. The "Overview" section explains that MongoDB's geospatial indexing allows for efficient execution of spatial queries on collections containing geospatial shapes and points. It mentions that the tutorial will introduce the concepts of geospatial indexes and demonstrate their use with `$geoWithin`, `$geoIntersects`, and `geoNear`. A sidebar on the right promotes "GET THE MONGODB QUICK REFERENCE CARDS".

Find Restaurants with Geospatial Queries

On this page

- [Overview](#)
- [Differences Between Flat and Spherical Geometry](#)
- [Distortion](#)
- [Searching for Restaurants](#)

Overview

MongoDB's [geospatial](#) indexing allows you to efficiently execute spatial queries on a collection that contains geospatial shapes and points. This tutorial will briefly introduce the concepts of geospatial indexes, and then demonstrate their use with `$geoWithin`, `$geoIntersects`, and `geoNear`.

To showcase the capabilities of geospatial features and compare different approaches, this tutorial will guide you through the process of writing queries for a simple geospatial application.

# Wide column stores

- Key-value database
- Columns: name, format can vary from row to row
- Apache Cassandra

