L23: NoSQL (continued)

CS3200 Database design (sp18 s2)

https://course.ccs.neu.edu/cs3200sp18s2/

4/9/2018

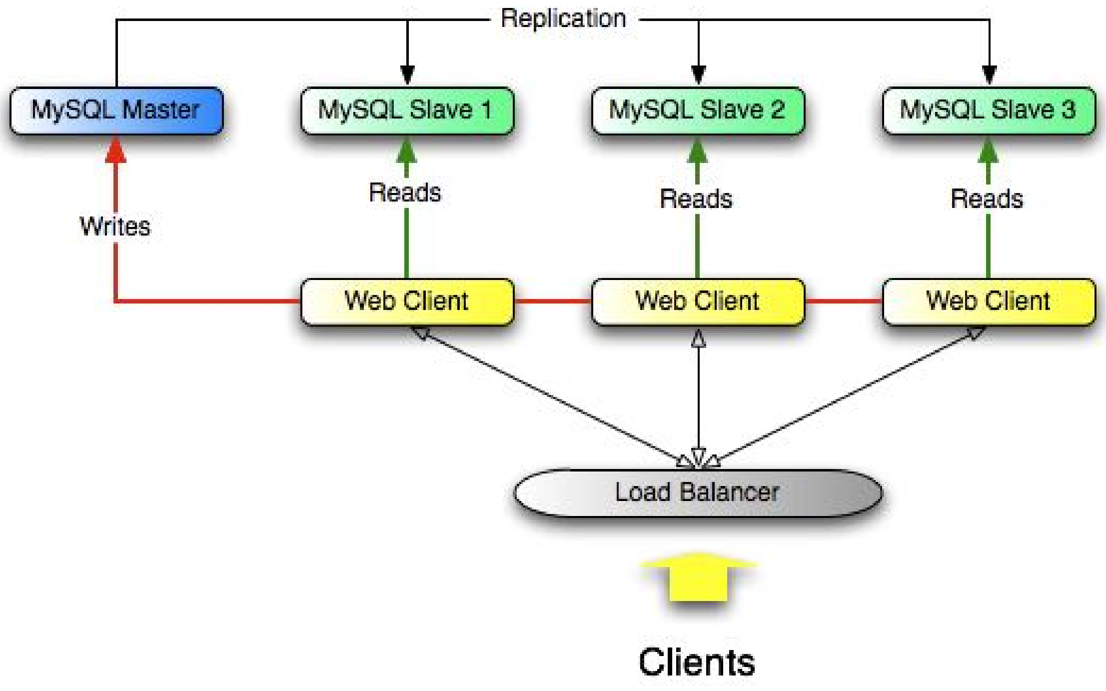
Announcements!

- Please pick up your exam if you have not yet
- HW6: start early
- NoSQL: focus on big picture
  - we see how things from earlier in class come together

NoSQL				
22	R Apr 5	Relational Algebra 2 & Query Optimization, NoSQL 1	GUW Ch 16.2	P2 (R 4/5), Q9 (FR 4/6)
23	M Apr 9	NoSQL 2		
24	R Apr 12	Class Review and Course Evaluation		Q10 (optional)
	M Apr 16	No class: Patriot's day		Optional PPTX (Wed 4/18)
	R Apr 19	No class: Reading day		HW6 (R 4/19)
	M Apr 23	**Exam 3** (1-3pm, location TBD)		

Database Replication

- Data replication: storing the same data on several machines ("nodes")
- Useful for:
  - Availability (parallel requests are made against replicas)
  - Reliability (data can survive hardware faults)
  - Fault tolerance (system stays alive when nodes/network fail)
- Typical architecture: master-slave



Replication example in MySQL
(dev.mysql.com)

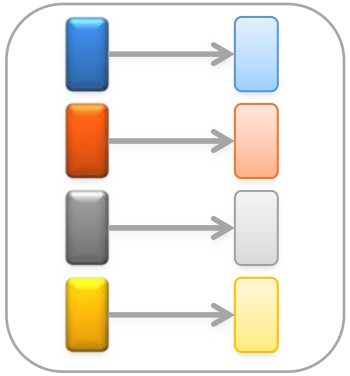
Open Source

- Free software, source provided
  - Users have the right to use, modify and distribute the software
  - But restrictions may still apply, e.g., adaptations need to be opensource
- Idea: community development
  - Developers fix bugs, add features, ...
- How can that work?
  - See [Bonaccorsi, Rossi, 2003. Why open source software can succeed. Research policy, 32(7), pp.1243-1258]
- A major driver of OpenSource is Apache

Apache Software Foundation

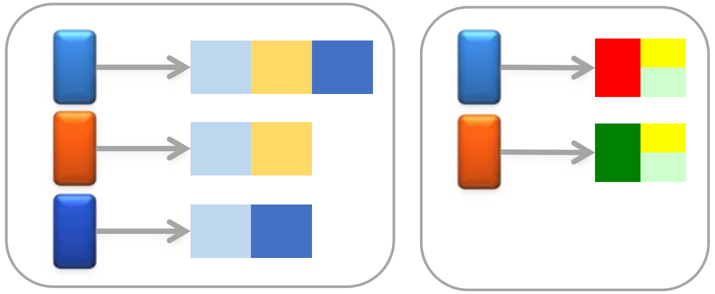
- Non-profit organization
- Hosts communities of developers
  - Individuals and small/large companies
- Produces open-source software
- Funding from grants and contributions
- Hosts very significant projects
  - Apache Web Server, Hadoop, Zookeeper, Cassandra, Lucene, OpenOffice, Struts, Tomcat, Subversion, Tcl, UIMA, ...

We Will Look at 4 Data Models

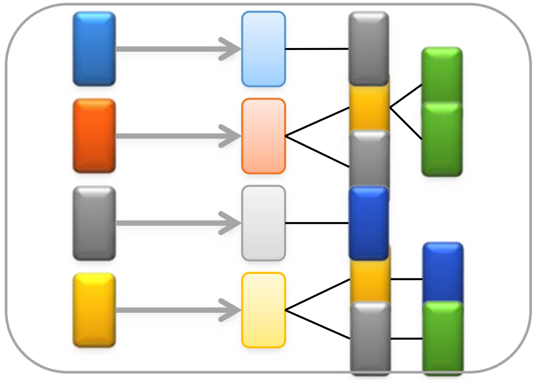


Key/Value Store

Redis

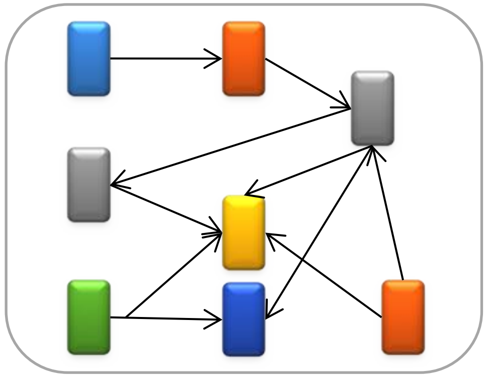


Column-Family Store



Document Store

MongoDB



Graph Databases

Source: Benny Kimelfeld

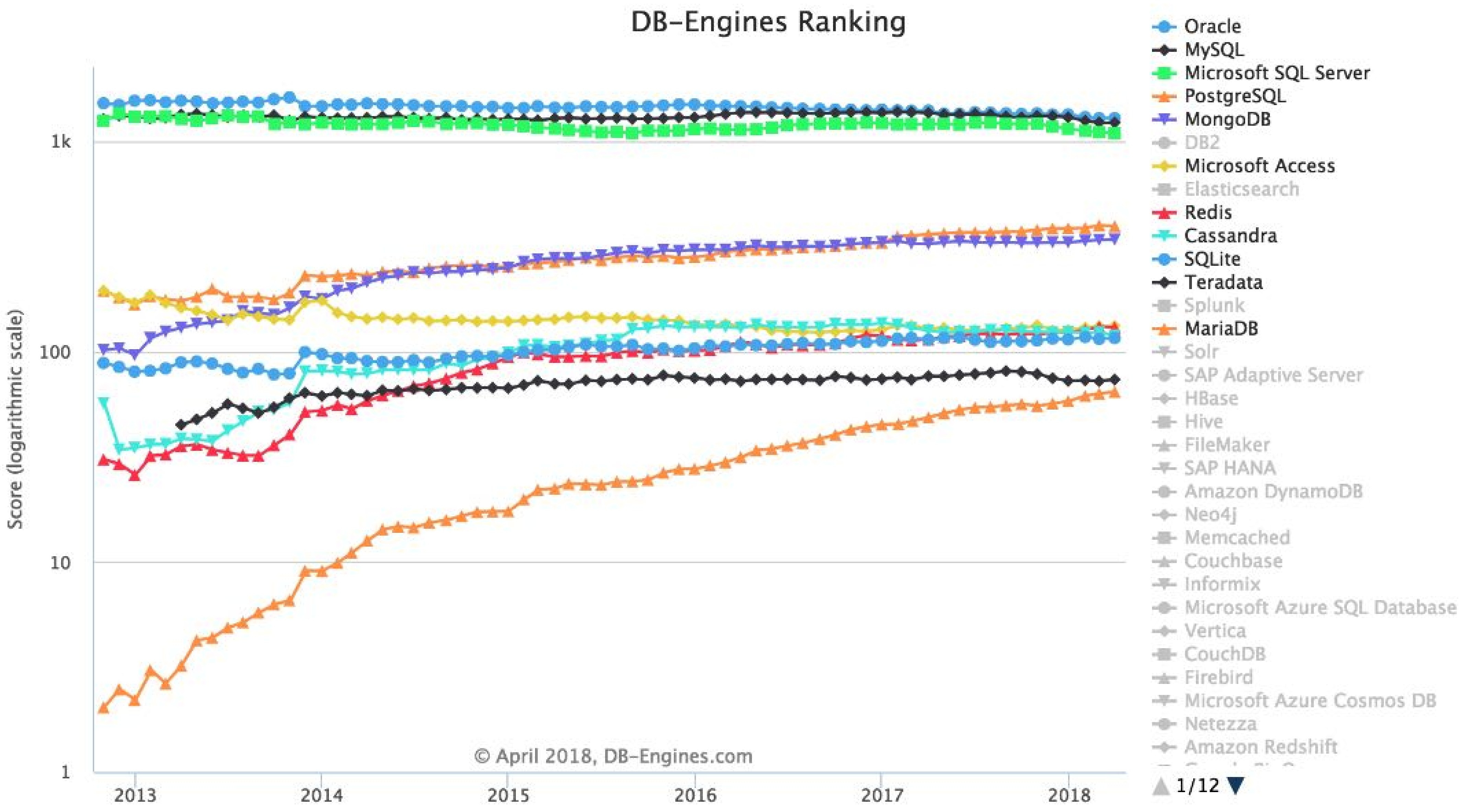
Database engines ranking by "popularity"

342 systems in ranking, April 2018

Rank			DBMS	Database Model	Score		
Apr 2018	Mar 2018	Apr 2017			Apr 2018	Mar 2018	Apr 2017
1.	1.	1.	Oracle	Relational DBMS	1289.79	+0.18	-112.21
2.	2.	2.	MySQL	Relational DBMS	1226.40	-2.46	-138.22
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1095.51	-9.28	-109.26
4.	4.	4.	PostgreSQL	Relational DBMS	395.47	-3.88	+33.69
5.	5.	5.	MongoDB	Document store	341.41	+0.89	+15.98
6.	6.	6.	DB2	Relational DBMS	188.95	+2.28	+2.29
7.	7.	7.	Microsoft Access	Relational DBMS	132.22	+0.27	+4.04
8.	↑ 9.	↑ 11.	Elasticsearch	Search engine	131.36	+2.81	+25.69
9.	↓ 8.	9.	Redis	Key-value store	130.11	-1.12	+15.75
10.	10.	↓ 8.	Cassandra	Wide column store	119.09	-4.40	-7.10
11.	11.	↓ 10.	SQLite	Relational DBMS	115.99	+1.17	+2.19

Source: https://db-engines.com/en/ranking , 4/2018

Database engines ranking by "popularity"



Source: https://db-engines.com/en/ranking_trend , 4/2018

Highlighted Database Features

- Data model
  - What data is being stored?
- CRUD interface
  - API for Create, Read, Update, Delete
    - 4 basic functions of persistent storage (insert, select, update, delete)
  - Sometimes preceding S for Search
- Transaction consistency guarantees
- Replication and sharding model
  - What's automated and what's manual?

True and False Conceptions

- True:
  - SQL does not effectively handle common Web needs of massive (datacenter) data
  - SQL has guarantees that can sometimes be compromised for the sake of scaling
  - Joins are not for free, sometimes undoable
- False:
  - NoSQL says NO to SQL
  - Nowadays NoSQL is the only way to go
  - Joins can always be avoided by structure redesign

Outline

- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

Transaction

- A sequence of operations (over data) viewed as a single higher-level operation
  - Transfer money from account 1 to account 2
- DBMSs execute transactions in parallel
  - No problem applying two “disjoint” transactions
  - But what if there are dependencies?
- Transactions can either commit (succeed) or abort (fail)
  - Failure due to violation of program logic, network failures, credit-card rejection, etc.
- DBMS should not expect transactions to succeed

Examples of Transactions

- Airline ticketing
  - Verify that the seat is vacant, with the price quoted, then charge credit card, then reserve
- Online purchasing
  - Similar
- “Transactional file systems” (MS NTFS)
  - Moving a file from one directory to another: verify file exists, copy, delete
- Textbook example: bank money transfer
  - Read from acct#1, verify funds, update acct#1, update acct#2

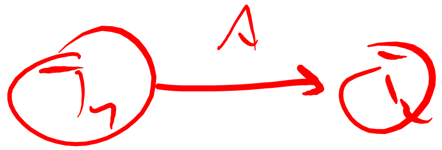
Transfer Example

```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```

txn_1	txn_2
Begin	Begin
Read(A,v)	Read(A,x)
v = v-100	x = x-100
Write(A,v)	Write(A,x)
Read(B,w)	Read(C,y)
w=w+100	y=y+100
Write(B,w)	Write(C,y)
Commit	Commit

- *Scheduling* is the operation of interleaving transactions
  - *Why is it good?*
- A *serial schedule* executes transactions one at a time, from beginning to end
- A *good* ("serializable") scheduling is one that *behaves like some serial scheduling* (typically by locking protocols)

Scheduling Example 1



txn$_1$

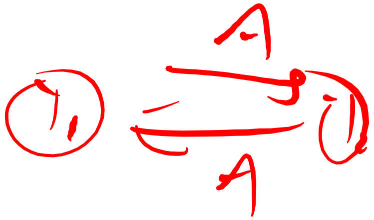
```
Begin
Read(A,v)
v = v-100
Write(A,v)
Read(B,w)
w=w+100
Write(B,w)
Commit
```

txn$_1$	txn$_2$
Read(A,v)	
v = v-100	
Write(A,v)	
	Read(A,x)
	x = x-100
	Write(A,x)
Read(B,w)	
	Read(C,y)
w=w+100	
	y=y+100
Write(B,w)	
	Write(C,y)

txn$_2$

```
Begin
Read(A,x)
x = x-100
Write(A,x)
Read(C,y)
y=y+100
Write(C,y)
Commit
```

Scheduling Example 2



txn$_1$			txn$_2$
Begin	Read(A,v)		Begin
Read(A,v)	v = v-100		Read(A,x)
v = v-100		Read(A,x)	x = x-100
Write(A,v)		x = x-100	Write(A,x)
Read(B,w)		Write(A,x)	Read(C,y)
w=w+100	Write(A,v)		y=y+100
Write(B,w)		Read(C,y)	Write(C,y)
Commit	Read(B,w)		Commit
		y=y+100	
	w=w+100		
		Write(C,y)	
	Write(B,w)		

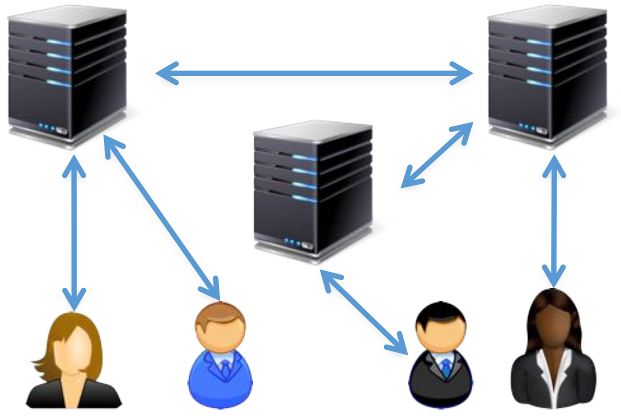
ACID

- **A**tomicity
  - Either all operations applied or none are (hence, we need not worry about the effect of incomplete / failed transactions)
- **C**onsistency
  - Each transaction can start with a consistent database and is required to leave the database consistent (bring the DB from one to another consistent state)
- **I**solation
  - The effect of a transaction should be as if it is the only transaction in execution (in particular, changes made by other transactions are not visible until committed)
- **D**urability
  - Once the system informs a transaction success, the effect should hold without regret, even if the database crashes (before making all changes to disk)

ACID May Be Overly Expensive

- In quite a few modern applications:
  - ACID contrasts with key desiderata: high volume, high availability
  - We can live with some errors, to some extent
  - Or more accurately, we prefer to suffer errors than to be significantly less functional
- *Can this point be made more "formal"?*

Simple Model of a Distributed Service



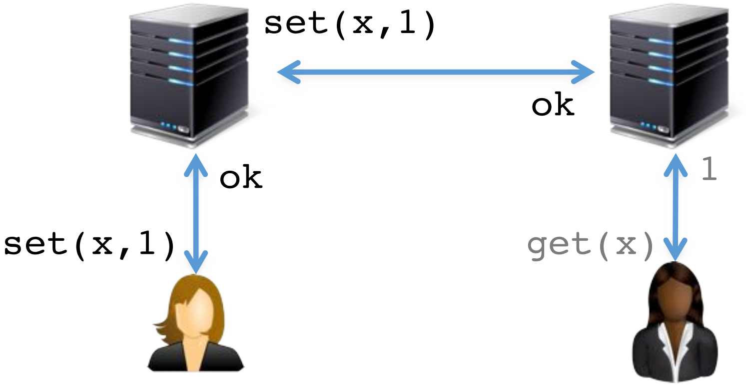
- Context: distributed service
  - e.g., social network
- Clients make get / set requests
  - e.g., setLike(user,post), getLikes(post)
  - Each client can talk to any server
- Servers return responses
  - e.g., ack, {$user_1$,....,$user_k$}
- Failure: the network may occasionally disconnect due to failures (e.g., switch down)
- Desiderata: **C**onsistency, **A**vailability, **P**artition tolerance

CAP Service Properties

- **C**onsistency:
  - every read (to any node) gets a response that reflects the most recent version of the data
    - More accurately, a transaction should behave as if it changes the entire state correctly in an instant, Idea similar to serializability
- **A**vailability:
  - every request (to a living node) gets an answer: set succeeds, get retunes a value (if you can talk to a node in the cluster, it can read and write data)
- **P**artition tolerance:
  - service continues to function on network failures (cluster can survive
    - As long as clients can reach servers

Simple Illustration

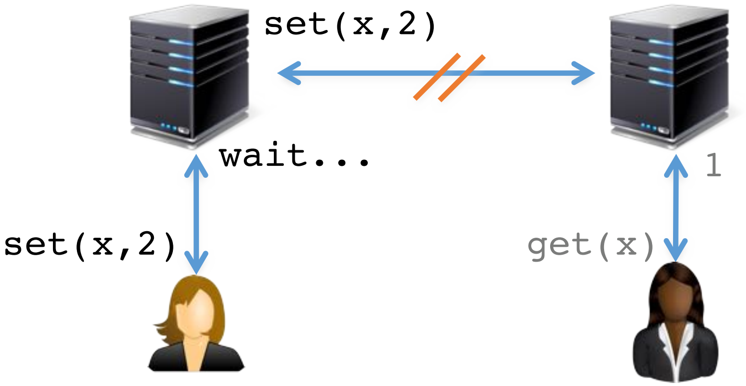
Our Relational Database world so far …



CA

Consistency, Availability

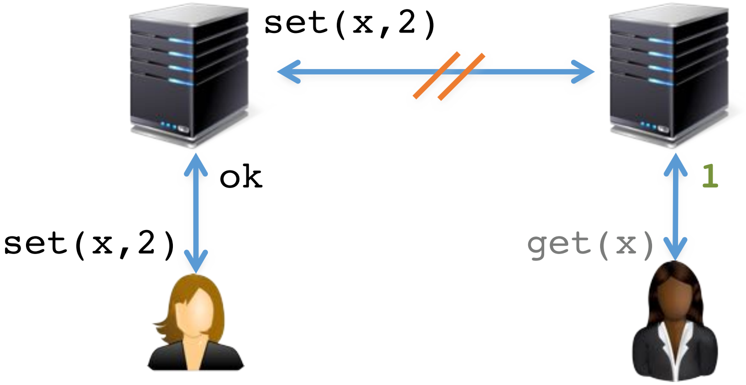
In a system that may suffer partitions, you have to trade off consistency vs. availability



CP

Consistency, Partition tolerance

~~Availability~~



AP

Availability, Partition tolerance

~~Consistency~~

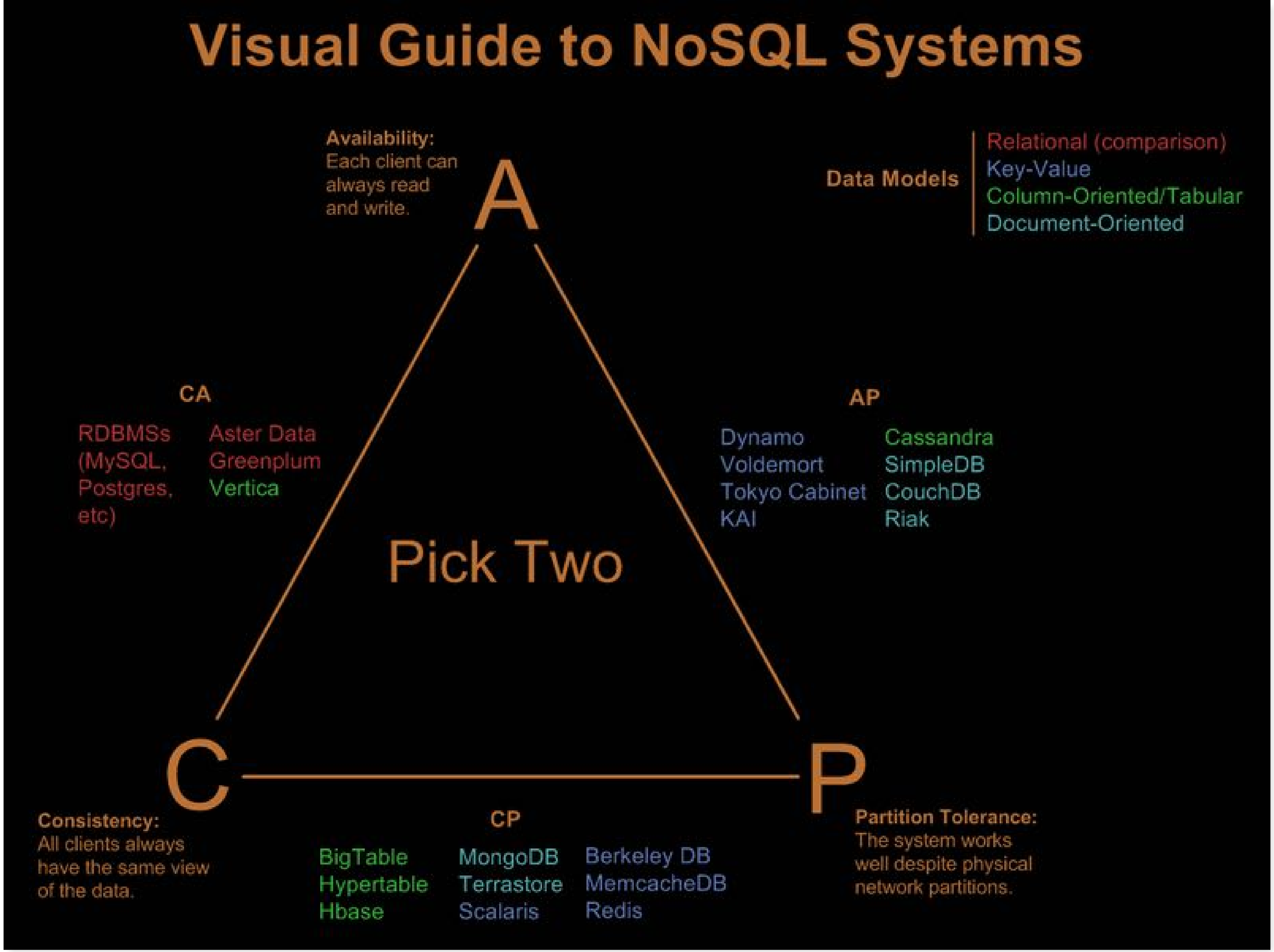
The CAP Theorem

Eric Brewer's CAP Theorem:

A distributed service
can support at most two
out of **C**, **A** *and* **P**

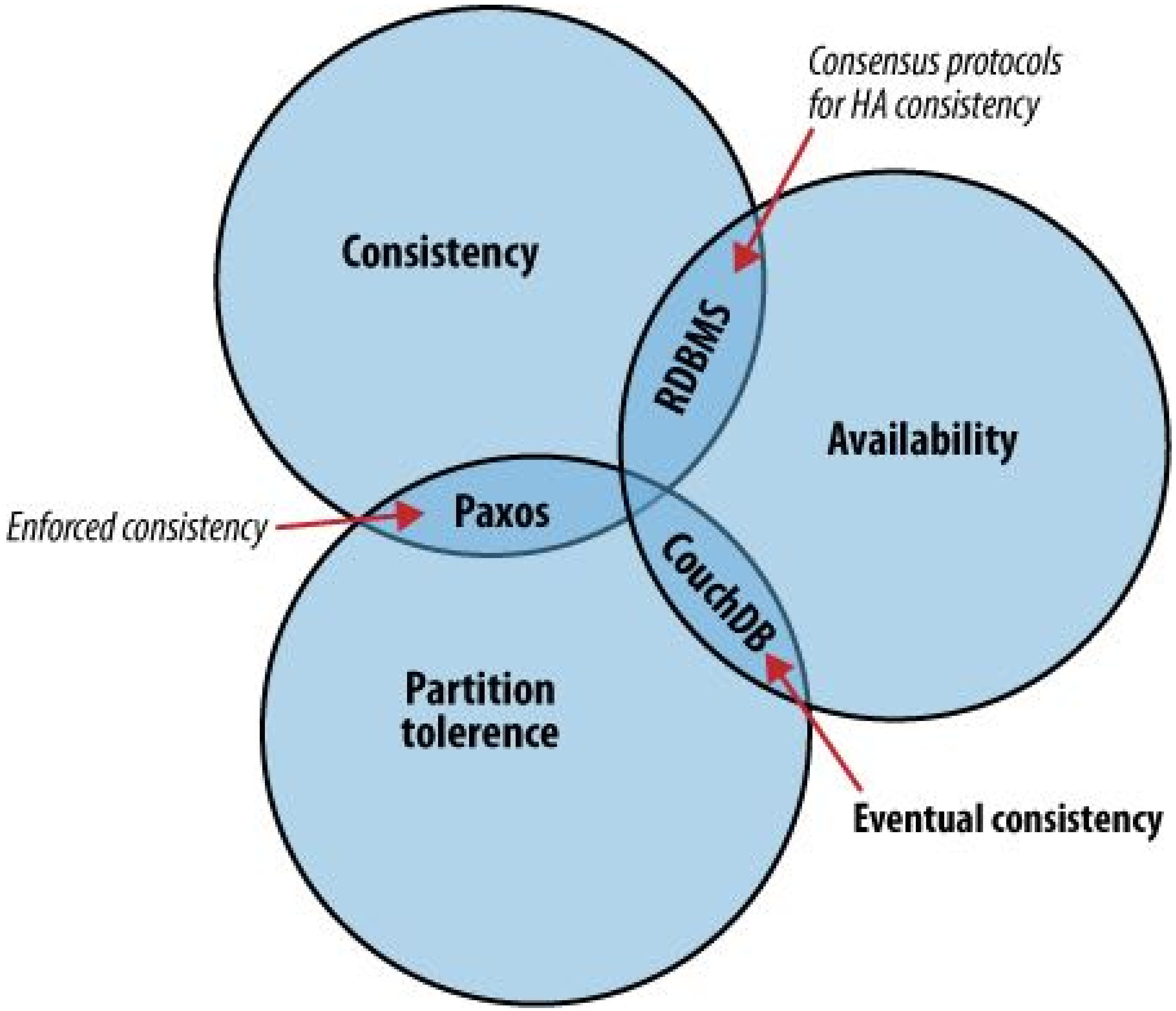
Historical Note

- Brewer presented it as the CAP principle in a 1999 article
  - Then as an informal conjecture in his keynote at the PODC 2000 conference
- In 2002 a formal proof was given by Gilbert and Lynch, making CAP a theorem
  - [Seth Gilbert, Nancy A. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2): 51-59 (2002)]
  - It is mainly about making the statement formal; the proof is straightforward



Source: http://blog.nahurst.com/visual-guide-to-nosql-systems , 2010

CAP theorem



Source: http://guide.couchdb.org

The BASE Model

- Applies to distributed systems of type AP
- **B**asic **A**vailability
  - Provide high availability through distribution: There will be a response to any request. Response could be a ‘failure’ to obtain the requested data, or the data may be in an inconsistent or changing state.
- **S**oft state
  - Inconsistency (stale answers) allowed: State of the system can change over time, so even during times without input, changes can happen due to ‘eventual consistency’
- **E**ventual consistency
  - If updates stop, then after some time consistency will be achieved
    - Achieved by protocols to propagate updates and verify correctness of propagation (gossip protocols)
- Philosophy: best effort, optimistic, staleness and approximation allowed

Outline

- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

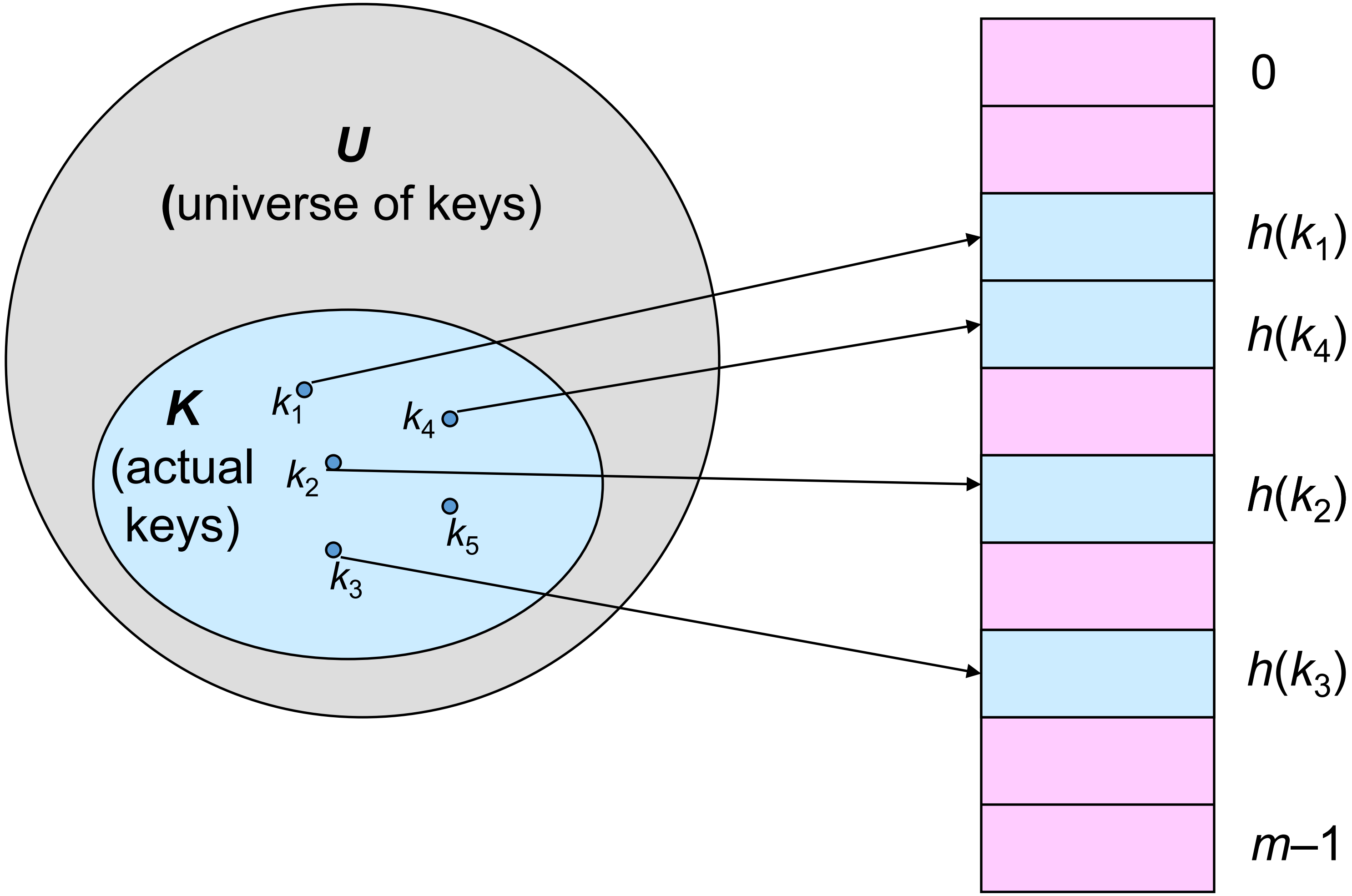
Key-Value Stores

- Essentially, big distributed hash maps
- Origin attributed to Dynamo – Amazon’s DB for world-scale catalog/cart collections
  - But Berkeley DB has been here for >20 years
- Store pairs ⟨key, opaque-value⟩
  - Opaque means that DB does not associate any structure/semantics with the value; oblivious to values
  - This may mean more work for the user: retrieving a large value and parsing to extract an item of interest
- Sharding via partitioning of the key space
  - Hashing, gossip and remapping protocols for load balancing and fault tolerance

Hashing (Hash tables, dictionaries)

$h : U \rightarrow \{0,1,\ldots, m-1\}$

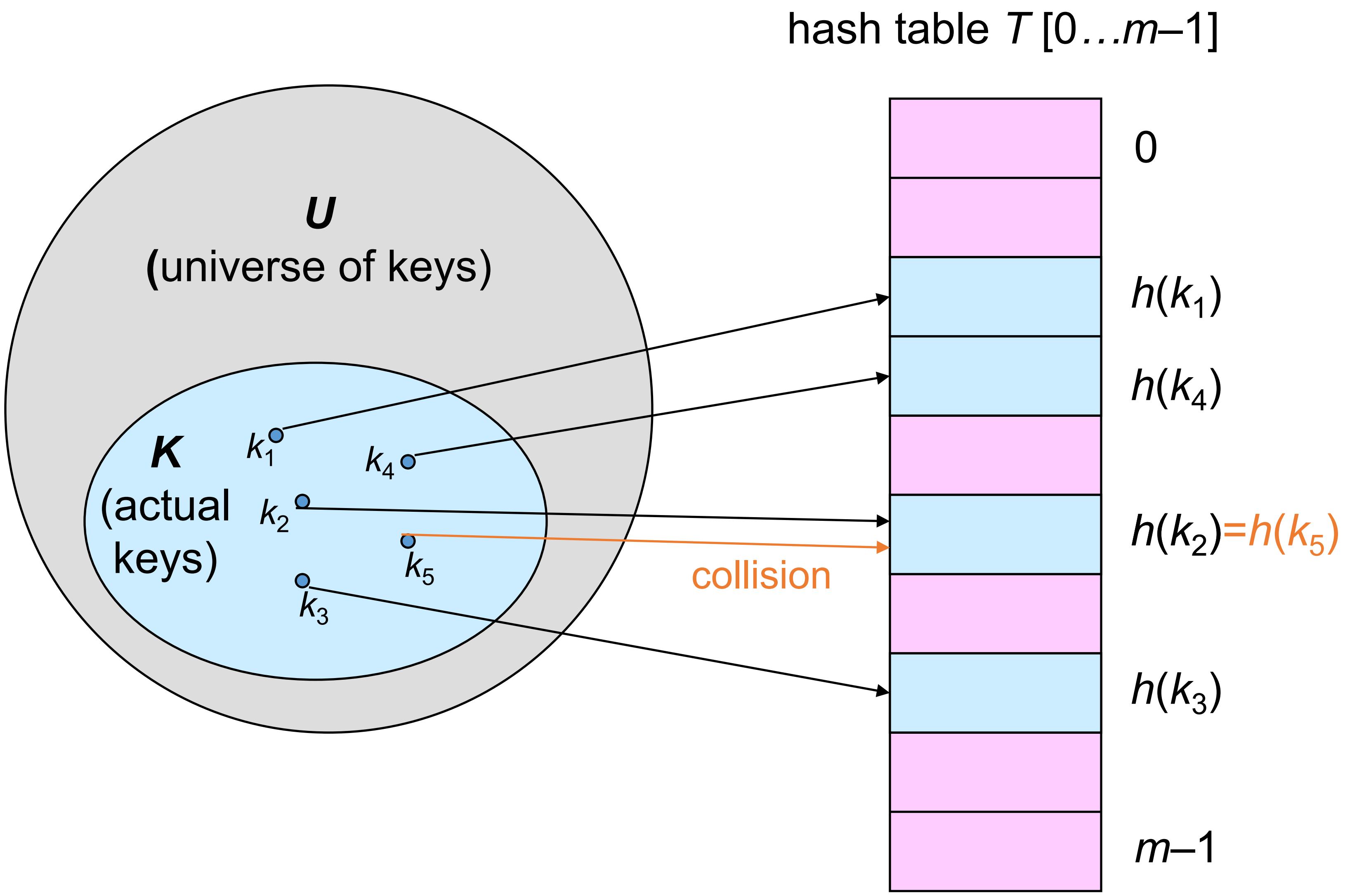
hash table $T[0\ldots m-1]$



$n = |K| << |U|$.

key k "hashes" to slot $T[h[k]]$

Hashing (Hash tables, dictionaries)



Example Databases

- Amazon’s DynamoDB
  - Originally designed for Amazon’s workload at peaks
  - Offered as part of Amazon’s Web services
- Redis
  - Next slides and in our Jupyter notebooks
- Riak
  - Focuses on high availability, BASE
  - “As long as your Riak client can reach one Riak server, it should be able to write data.”
- FoundationDB
  - Focus on transactions, ACID
- Berkeley DB (and Oracle NoSQL Database)
  - First release 1994, by Berkeley, acquired by Oracle
  - ACID, replication

Redis

- Basically a data structure for strings, numbers, hashes, lists, sets
- Simplistic “transaction” management
  - Queuing of commands as blocks, really
  - Among ACID, only Isolation guaranteed
    - A block of commands that is executed sequentially; no transaction interleaving; no roll back on errors
- <u>In-memory</u> store
  - Persistence by periodical saves to disk
- Comes with
  - A command-line API
  - Clients for different programming languages
    - Perl, PHP, Rubi, Tcl, C, C++, C#, Java, R, …

Example of Redis Commands

key maps to:

key	value

```
get x
>> 10
```

```
hget h y
>> 5
```

```
hkeys p:22
>> name , age
```

```
smembers s
>> 20 , Alice
```

```
scard s
>> 2
```

```
llen l
>> 3
```

```
lrange l 1 2
>> a , b
```

```
lindex l 2
>> b
```

```
lpop l
>> c
```

```
rpop l
>> b
```

Example of Redis Commands

key maps to:

		key	value
(simple value)	`set x 10`	x	10
(hash table)	`hset h y 5`	h	y→5
	`hset h1 name two` `hset h1 value 2`	h1	name→two value→2
	`hmset p:22 name Alice age 25`	p:22	name→Alice age→25
(set)	`sadd s 20` `sadd s Alice` `sadd s Alice`	s	{20,Alice}
(list)	`rpush l a` `rpush l b` `lpush l c`	l	(c,a,b)

```
get x
>> 10
```

```
hget h y
>> 5
```

```
hkeys p:22
>> name , age
```

```
smembers s
>> 20 , Alice
```

```
scard s
>> 2
```

```
llen l
>> 3
```

```
lrange l 1 2
>> a , b
```

```
lindex l 2
>> b
```

```
lpop l
>> c
```

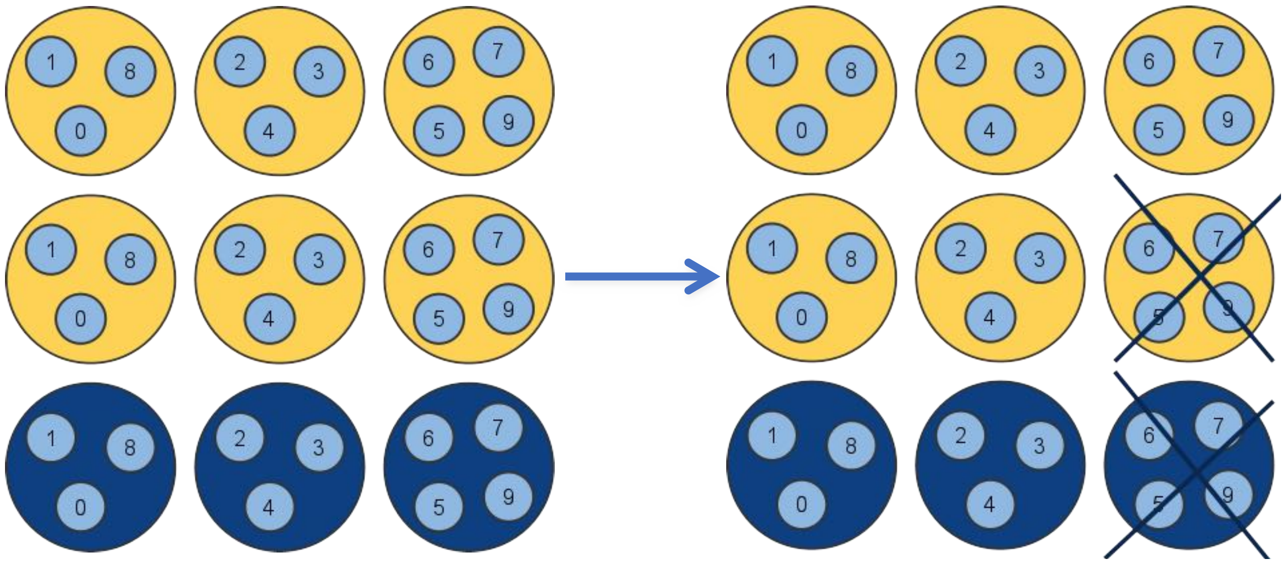
```
rpop l
>> b
```

Additional Notes

- A key can be any <256MB binary string
  - For example, JPEG image
- Some key operations:
  - List all keys: `keys *`
  - Remove all keys: `flushall`
  - Check if a key exists: `exists k`
- You can configure the persistency model
  - `save m k` means save every `m` seconds if at least `k` keys have changed

Redis Cluster

- Add-on module for managing multi-node applications over Redis
- Master-slave architecture for sharding + replication
  - Multiple masters holding pairwise disjoint sets of keys, every master has a set of slaves for replication and sharding



Blue … master,
Yellow … replicas
Up to 2 random nodes can go down without issues because of redudancy

Source: http://redis.io/presentation/Redis_Cluster.pdf

When to use it

- Use it:
  - All access to the databases is via primary key
  - Storing session information (web session)
  - user or product profiles (single GET operation)
  - shopping card information (based on userid)
- Don't use it:
  - relationships between different sets of data
  - query by data (based on values)
  - operations on multiple keys at a time

Outline

- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

2 Types of Column Stores

Standard RDB

sid	name	address	year	faculty
861	Alice	Haifa	2	NULL
753	Amir	London	NULL	CS
955	Ahuva	NULL	2	IE

Column store *(still SQL)*

id	sid
1	861
2	753
3	955

id	name
1	Alice
2	Amir
3	Ahuva

id	address
1	Haifa
2	London

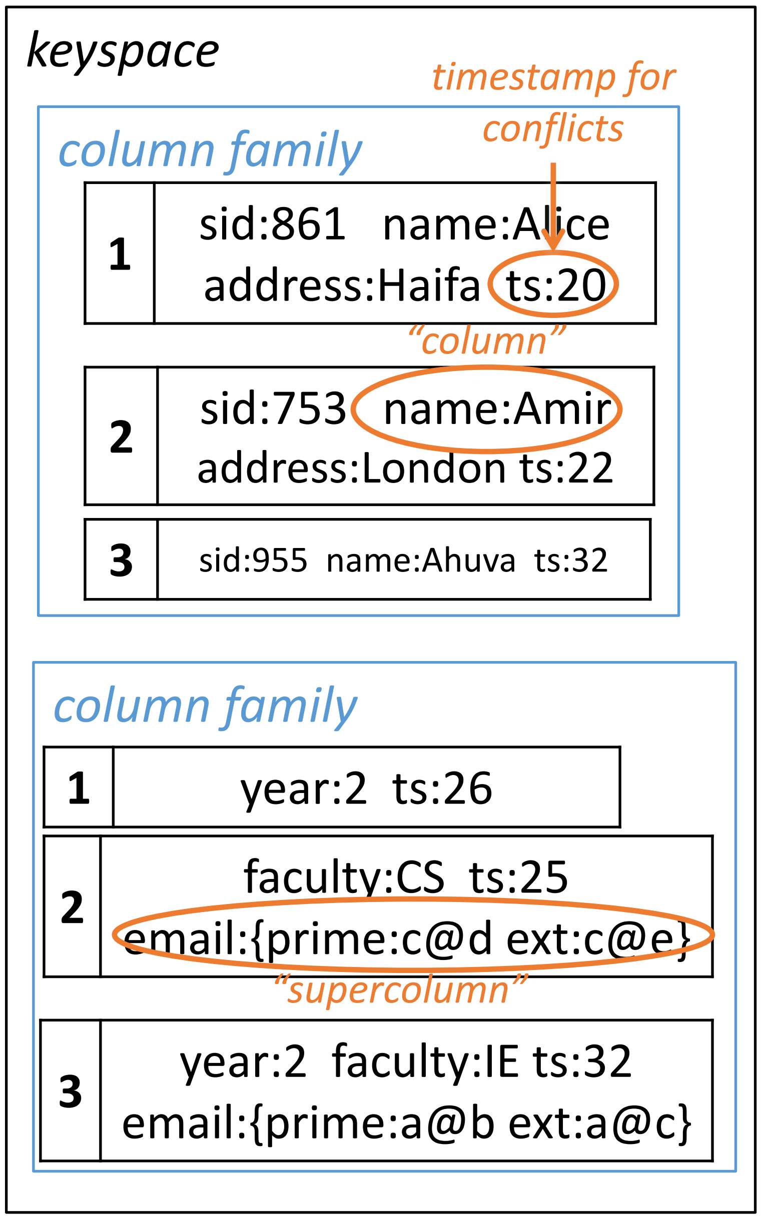
id	year
1	2
3	2

id	faculty
2	CS
3	IE

Each column stored separately. Why?
Efficiency (fetch only required columns),
compression, sparse data for free

Column-Family Store (*NoSQL*)

Cassandra data model



Column Stores

- The two often mixed as "column store" → confusion
  - See Daniel Abadi's blog: http://dbmsmusings.blogspot.com/2010/03/distinguishing-two-major-types-of_29.html
- Common idea: don't keep a row in a consecutive block, split via projection
  - Column store: each column is independent
  - Column-family store: each column family is independent
- Both provide some major efficiency benefits in common read-mainly workloads
  - Given a query, load to memory only the relevant columns
  - Columns can often be highly compressed due to value similarity
  - Effective form for sparse information (no NULLs, no space)
- Column-family store is handled differently from RDBs, often requiring a designated query language

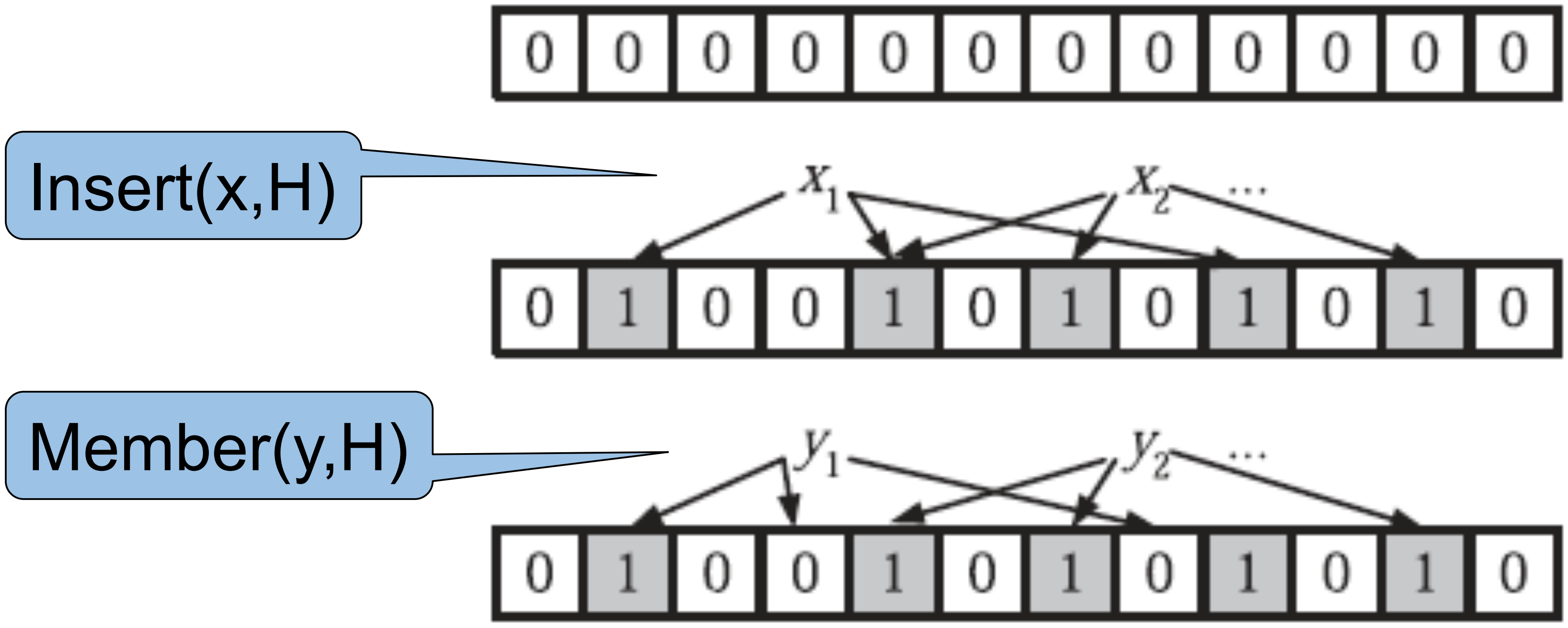
Examples Systems

- Column store (*SQL*):
  - MonetDB (started 2002, Univ. Amsterdam)
  - VectorWise (spawned from MonetDB)
  - Vertica (M. Stonebraker)
  - SAP Sybase IQ
  - Infobright
- Column-family store (*NoSQL*):
  - Google's BigTable (main inspiration to column families)
  - Apache HBase (used by Facebook, LinkedIn, Netflix..., CP in CAP)
  - Hypertable
  - Apache Cassandra (AP in CAP)

Example: Apache Cassandra

- Initially developed by Facebook
  - Open-sourced in 2008
- Used by 1500+ businesses, e.g., Comcast, eBay, GitHub, Hulu, Instagram, Netflix, Best Buy, ...
- Column-family store
  - Supports key-value interface
  - Provides a SQL-like CRUD interface: CQL
- Uses Bloom filters
  - An interesting membership test that can have false positives but never false negatives, behaves well statistically
- BASE consistency model (AP in CAP)
  - Gossip protocol (constant communication) to establish consistency
  - Ring-based replication model

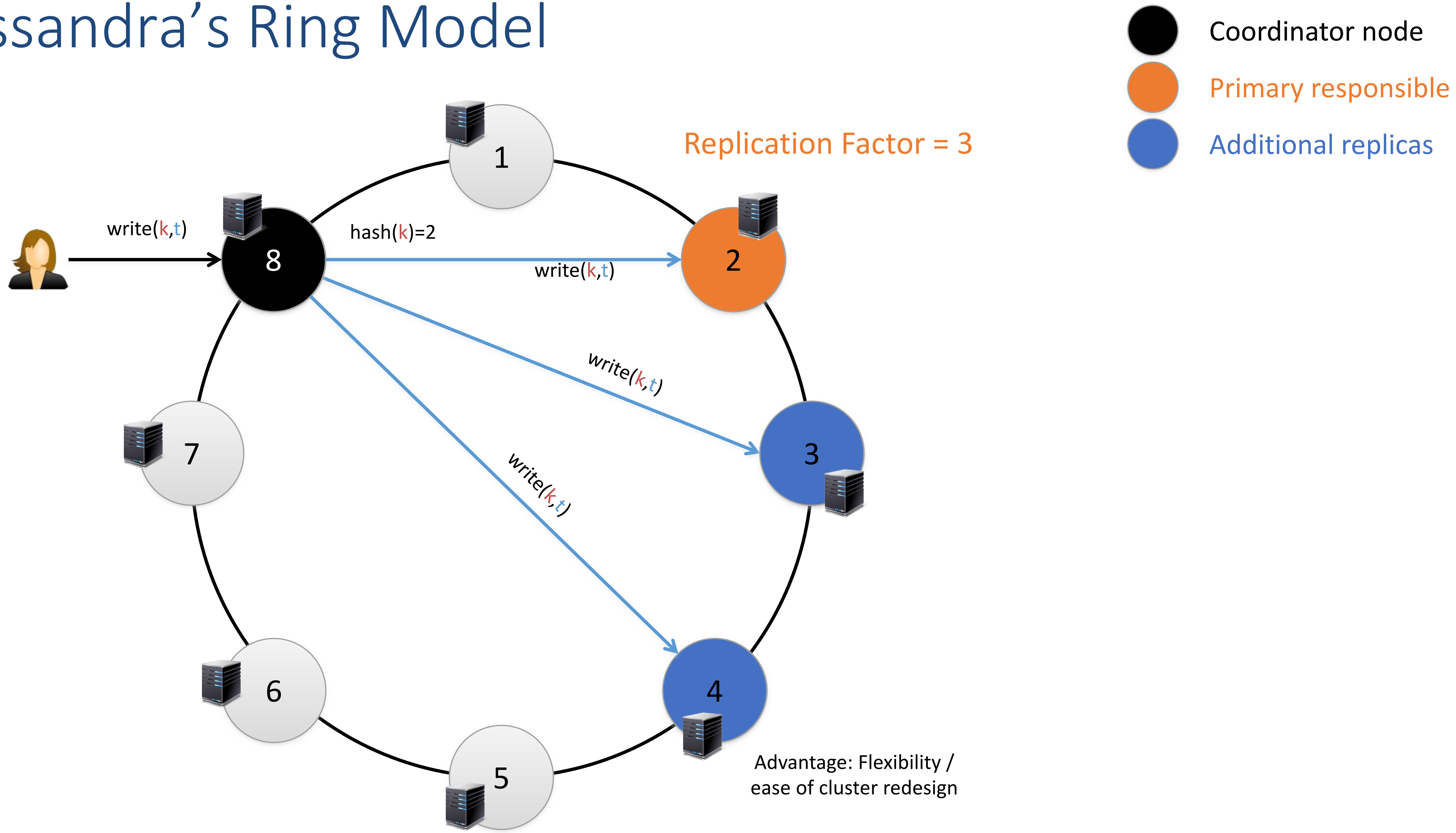
Example Bloom Filter k=3



y_1 = is not in H (why ?)

y_2 may be in H (why ?)

Cassandra's Ring Model



Advantage: Flexibility / ease of cluster redesign

See more: https://www.hakkalabs.co/articles/how-cassandra-stores-data

When to use it (e.g. Cassandra)

- Use it:
  - Event logging (multiple applications can write in different columns and row-key: appname:timestamp)
  - CMS: Store blog entries with tags, categories, links in different columns
  - Counters: e.g. visitors of a page
- Don't use it:
  - if you require ACID, consistency
  - if you have to aggregates across all the rows
  - if you change query patterns often (in RDMS schema changes are costly, in Cassandrda query changes are)

Outline

- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

Document Stores

- Similar in nature to key-value store, but value is tree structured as a document
- Motivation: avoid joins; ideally, all relevant joins already encapsulated in the document structure
- A document is an atomic object that cannot be split across servers
  - But a document collection will be split
- Moreover, transaction atomicity is typically guaranteed within a single document
- Model generalizes column-family and key-value stores

Example Databases

- MongoDB
  - Next slides
- Apache CouchDB
  - Emphasizes Web access
- RethinkDB
  - Optimized for highly dynamic application data
- RavenDB
  - Deigned for .NET, ACID
- Clusterpoint Server
  - XML and JSON, a combined SQL/JavaScript QL

MongoDB

- Open source, 1st release 2009, document store
  - Actually, an extended format called BSON (Binary JSON = JavaScript Object Notation) for typing and better compression
- Supports replication (master/slave), sharding (horizontal partitioning)
  - Developer provides the "shard key" – collection is partitioned by ranges of values of this key
- Consistency guarantees, CP of CAP
- Used by Adobe (experience tracking), Craigslist, eBay, FIFA (video game), LinkedIn, McAfee
- Provides connector to Hadoop
  - Cloudera provides the MongoDB connector in distributions

Data Example: High-level

Document

```
{
	name: "Alice",
	age: 21,
	status: "A",
	groups: ["algorithms", "theory"]
}
```

Collection

```
{
	name: "Alice",
}
{
	name: "Bob",
}
{
	name: "Charly",
}
{
	name: "Dorothee",
	age: 16,
	status: "A",
	groups: ["cars", "sports"]
}
```

Source: Modified from https://docs.mongodb.com/v3.0/core/crud-introduction/

MongoDB Terminology

RDBMS

- Database
- Table
- Record/Row/Tuple
- Column
- Primary key
- Foreign key

MongoDB

- Database
- Collection
- Document
- Field
- _id

MongoDB Data Model

- JavaScript Object Notation (JSON) model
- *Database* = set of named *collections* ← *generalizes relation*
- *Collection* = sequence of *documents* ← *generalizes tuple*
- *Document* = {attribute_1:value_1,...,attribute_k:value_k}
- *Attribute* = string ($\text{attribute}_i \neq \text{attribute}_j$)
- *Value* = primitive value (string, number, date, ...), or a document, or an array
  - *Array* = [value_1,...,value_n]

- Key properties: hierarchical (like XML), no schema
  - Collection docs may have different attributes

Data Example

Collection inventory

```
{
	item: "ABC2",
	details: { model: "14Q3", manufacturer: "M1 Corporation" },
	stock: [ { size: "M", qty: 50 } ],
	category: "clothing”
}

{
	item: "MNO2",
	details: { model: "14Q3", manufacturer: "ABC Company" },
	stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
	category: "clothing”
}
```

```
db.inventory.insert(
  {
   item: "ABC1",
   details: {model: "14Q3",manufacturer: "XYZ Company"},
   stock: [ { size: "S", qty: 25 }, { size: "M", qty: 50 } ],
   category: "clothing"
  }
)
```

Document insertion

Source: Modified from https://docs.mongodb.com/v3.0/core/crud-introduction/

Example of a Simple Query

Collection orders

```
{
    _id: "a",
    cust_id: "abc123",
    status: "A",
    price: 25,
    items: [ { sku: "mmm", qty: 5, price: 3 },
             { sku: "nnn", qty: 5, price: 2 } ]
}
{
    _id: "b",
    cust_id: "abc124",
    status: "B",
    price: 12,
    items: [ { sku: "nnn", qty: 2, price: 2 },
             { sku: "ppp", qty: 2, price: 4 } ]
}
```

```
db.orders.find(
   { status: "A" },
   { cust_id: 1, price: 1, _id: 0 }
)
```

selection

projection

In SQL it would look like this:

```
SELECT cust_id, price
  FROM orders
WHERE status="A"
```

```
{
    cust_id: "abc123",
    price: 25
}
```

Find all orders and price with with status "A"

When to use it

- Use it:
  - Event logging: different types of events across an enterprise
  - CMS: user comments, registration, profiles, web-facing documents
  - E-commerce: flexible schema for products, evolve data models
- Don't use it:
  - if you require atomic cross-document operations
  - queries against varying aggregate structures

Outline

- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks