Updated 1/29/2020

T1: Data models and query languages
L7: Alternative data models: NoSQL

Acknowledgments to Benny Kimelfeld for using many of his slides & examples in this lecture

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp20)

https://northeastern-datalab.github.io/cs7240/sp20/

Version 1/28/2019

Outline: Alternative data models

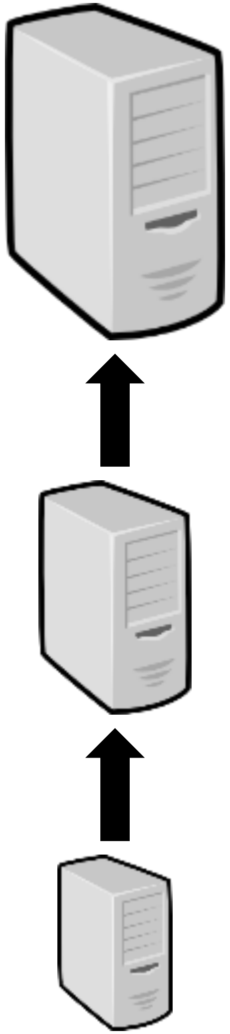
- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

SQL Means More than SQL

- SQL stands for the query language
- But commonly refers to the traditional RDBMS:
  - Relational storage of data
    - Each tuple is stored consecutively (per row, row-wise)
  - Joins as first-class citizens
    - In fact, normal forms prefer joins to maintenance
  - Strong guarantees on transaction management
    - No consistency worries when many transactions operate simultaneously (concurrently) on common data
- Focus on scaling up
  - That is, make a single machine do more, faster

Vertical vs. Horizontal Scaling

- **Vertical scaling ("scale up")**: you scale by adding more power (CPU, RAM)
- **Horizontal scaling ("scale out")**: you scale by adding more machines



"scaling up"

"scaling out"

Trends Drive Common Requirements

Social media + mobile computing

⇩

- Explosion in data, always available, constantly read and updated
- High load of simple requests of a common nature
- Some consistency can be compromised (e.g., 👍)

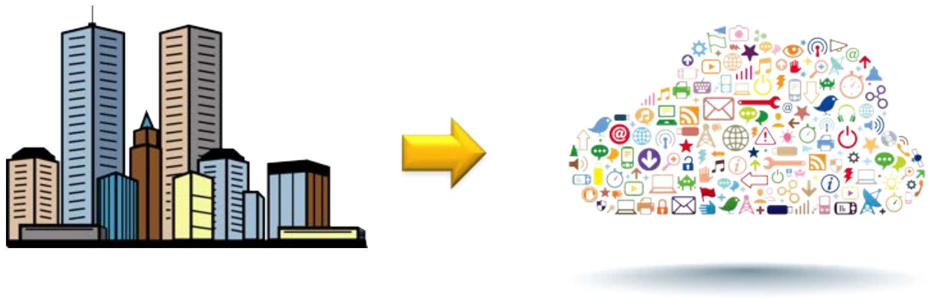
Cloud computing + open source

⇩

- Affordable resources for management / analysis of data
- People of various skills / budgets need software solutions for distributed analysis of massive data

Database solutions need to scale out
(utilize distribution, “scale horizontally”)

Compromises Required



What is needed for effective distributed, data- and user-intensive applications?

1. Use data models and storage that allow to avoid joins of big objects
2. Relax the guarantees on consistency

NoSQL

- Not Only SQL
  - May still support SQL-type languages
  - Term introduced by Carlo Strozzi in 1998 to describe an alternative database model
  - Became the name of a movement following Eric Evans’s reuse for a distributed-database event
- Seminal papers:
  - Google’s BigTable
    - Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, Gruber: Bigtable: A Distributed Storage System for Structured Data. OSDI 2006: 205-218
  - Amazon’s DynamoDB
    - DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007: 205-220

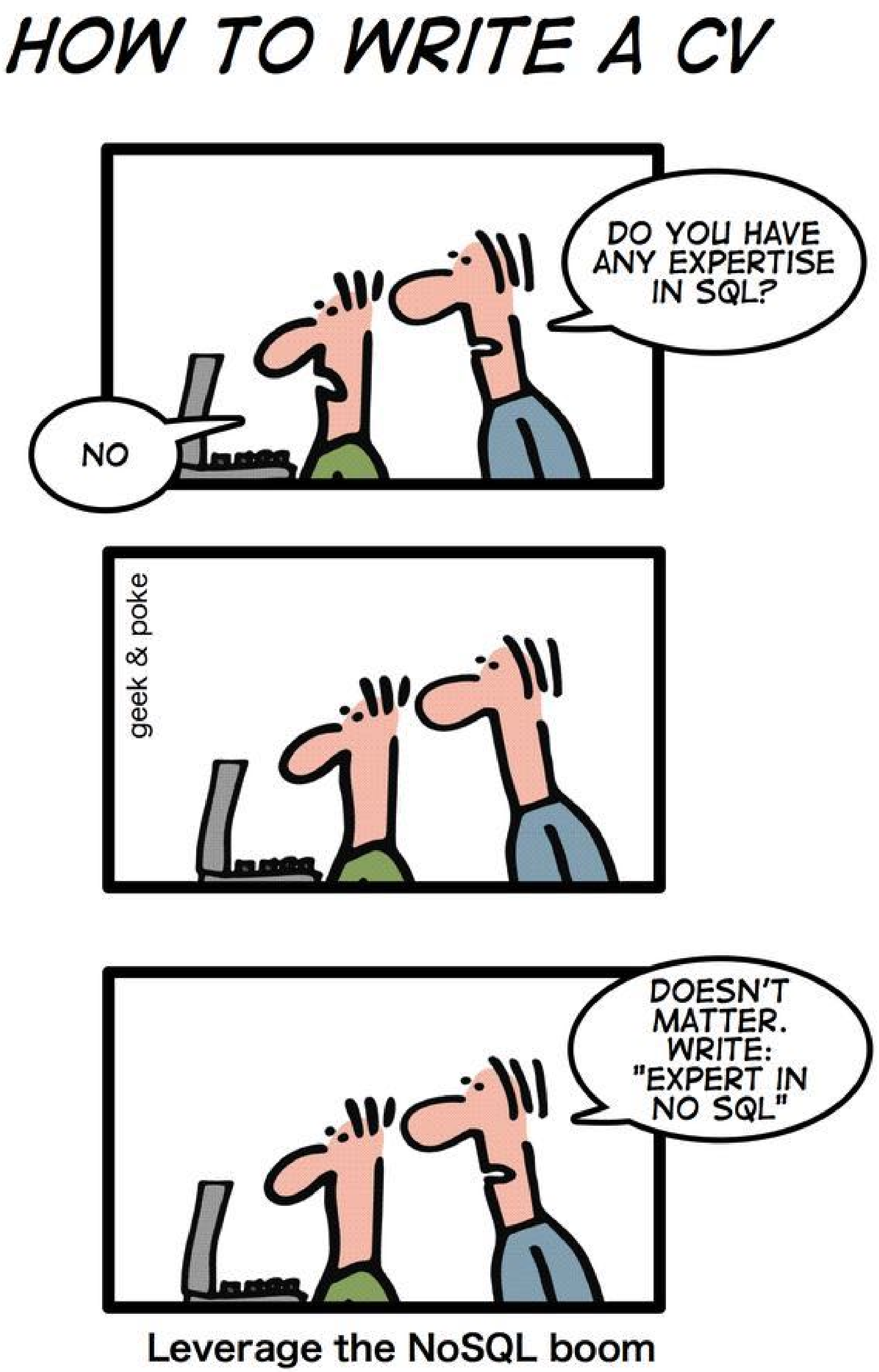
NoSQL from nosql-database.org

“

- Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontally scalable.
- The original intention has been modern web-scale databases. The movement began early 2009 and has been growing rapidly. Often more characteristics apply such as: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), a huge amount of data and more.
- So the misleading term “nosql” (the community now translates it mostly with “not only sql”) should be seen as an alias to something like the definition above.

”

What is NoSQL?



Source: Geek and Poke: http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html

Common NoSQL Features

- Non-relational data models
- Flexible structure
  - No need to fix a schema, attributes can be added and replaced on the fly
- Massive read/write performance; availability via horizontal scaling
  - Replication and sharding (data partitioning, we'll discuss that next)
  - Potentially thousands of machines worldwide
- Open source (very often)
- APIs to impose locality (opposite of joins)

When the database grows: Partitioning Tables

Key	Product Name	Short Description	Review	Picture
01	Americano @ Starbucks	Black, no sugar	I'd buy again	
02	BB @ Seattle's Best	Black, no sugar	The best	
03	TB @ Zoka Coffee	Black, no sugar	It's okay	
04	BC @ Coffee	Black, no sugar	Never again	

Source: http://cloudgirl.tech/data-partitioning-vertical-horizontal-hybrid-partitioning/

Vertical Partitioning

Key	Product Name	Short Description	Review	Picture
01	Americano @ Starbucks	Black, no sugar	I'd buy again	
02	BB @ Seattle's Best	Black, no sugar	The best	
03	TB @ Zoka Coffee	Black, no sugar	It's okay	
04	BC @ Coffee	Black, no sugar	Never again	

Key	Product Name	Review
01	Americano @ Starbucks	I'd buy again
02	BB @ Seattle's Best	The best
03	TB @ Zoka Coffee	It's okay
04	BC @ Coffee	Never again

Key	Short Description	Picture
01	Black, no sugar	
02	Black, no sugar	
03	Black, no sugar	
04	Black, no sugar	

Source: http://cloudgirl.tech/data-partitioning-vertical-horizontal-hybrid-partitioning/

Horizontal Partitioning ("sharding")

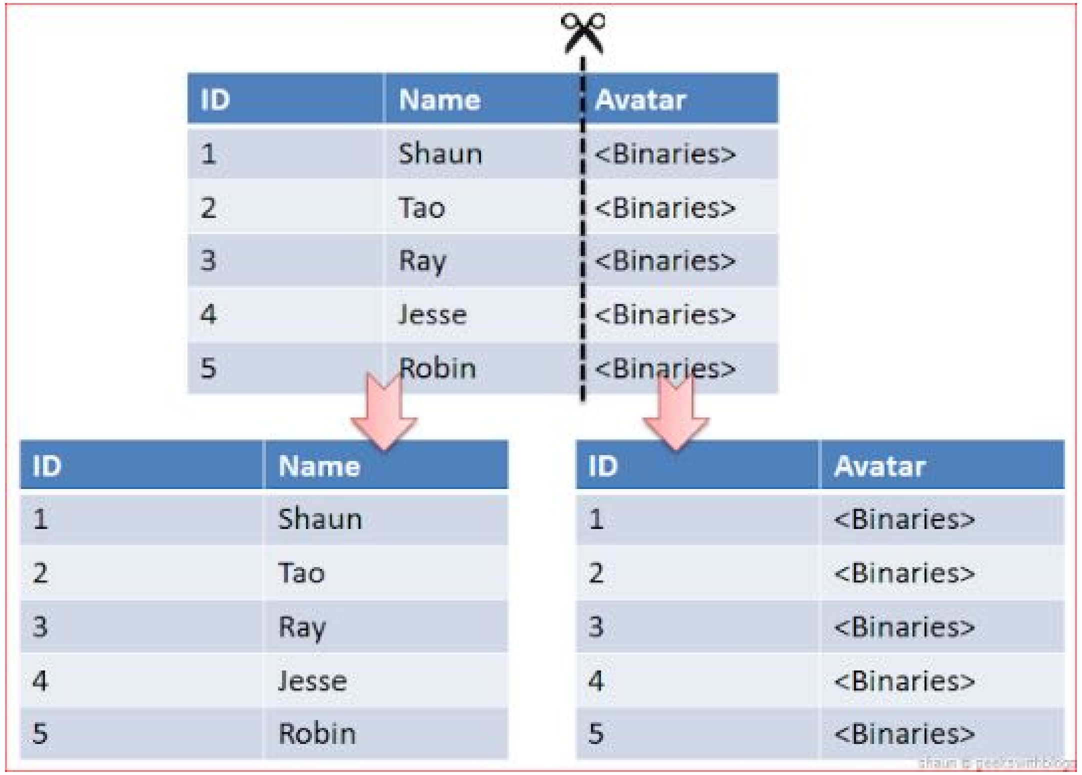
Key	Product Name	Short Description	Review	Picture
01	Americano @ Starbucks	Black, no sugar	I'd buy again	
02	BB @ Seattle's Best	Black, no sugar	The best	
03	TB @ Zoka Coffee	Black, no sugar	It's okay	
04	BC @ Coffee	Black, no sugar	Never again	

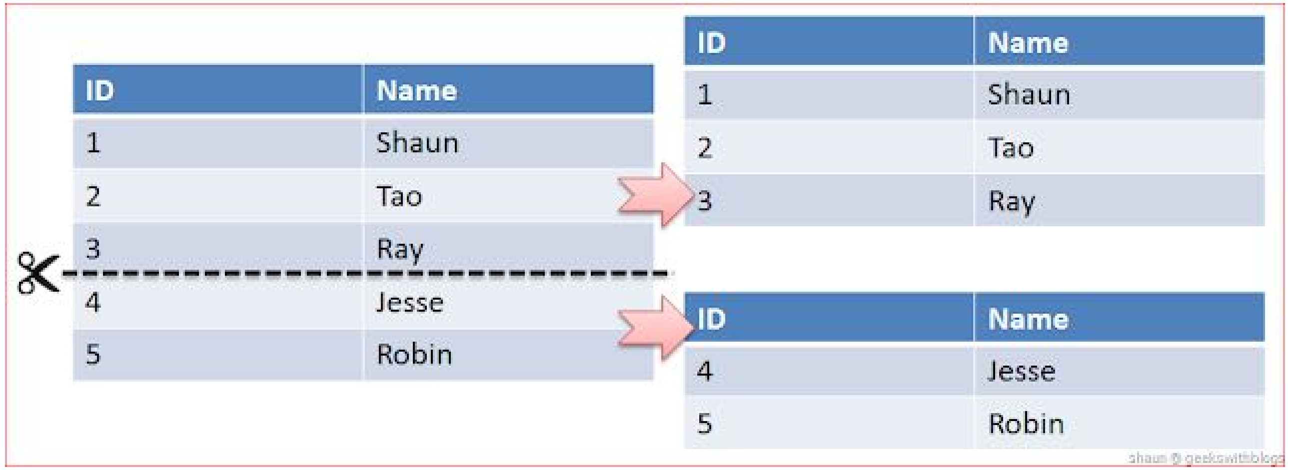
Key	Product Name	Short Description	Review	Picture
01	Americano @ Starbucks	Black, no sugar	I'd buy again	
02	BB @ Seattle's Best	Black, no sugar	The best	

Key	Product Name	Short Description	Review	Picture
03	TB @ Zoka Coffee	Black, no sugar	It's okay	
04	BC @ Coffee	Black, no sugar	Never again	

Source: http://cloudgirl.tech/data-partitioning-vertical-horizontal-hybrid-partitioning/

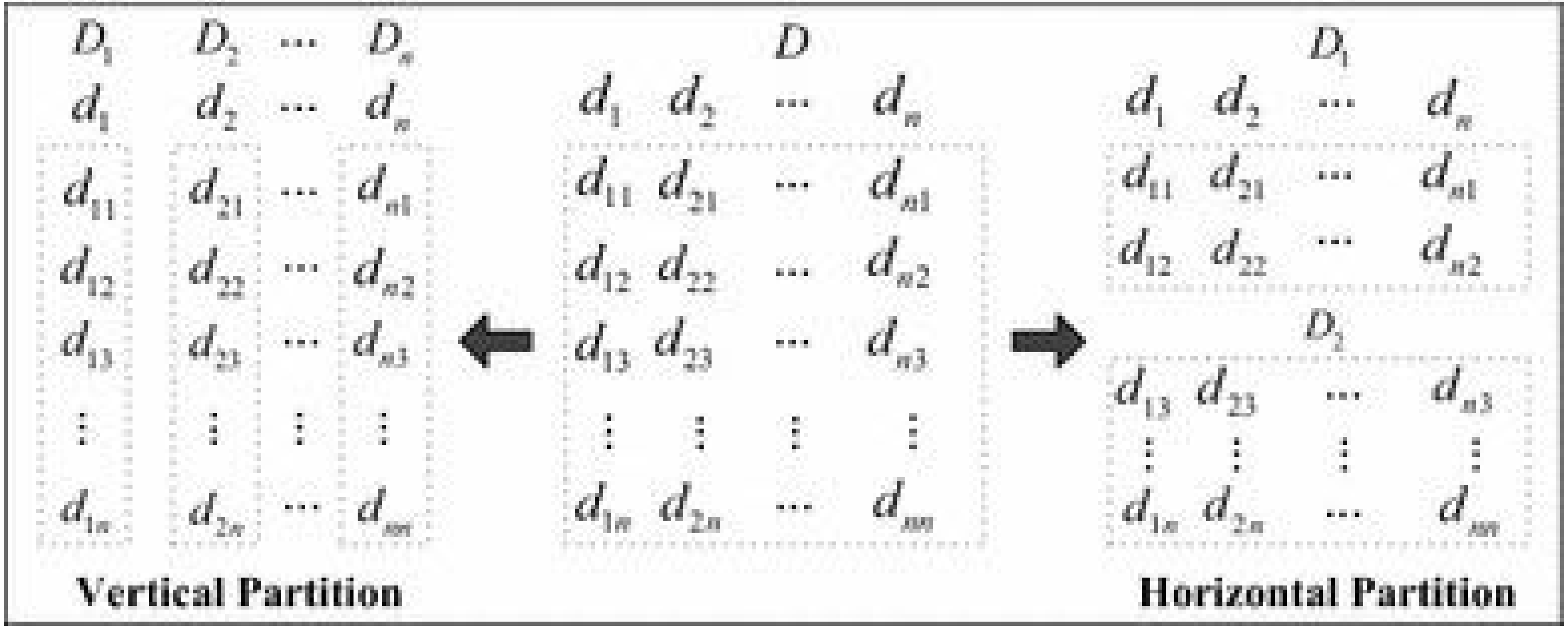
Vertical vs. Horizontal partitioning





Source: http://www.piyushgupta.co.uk/2016/04/database-scaling-jargons.html, http://slideplayer.com/slide/12131436/70/images/17/SQL+Azure+Azure+Custom+Sharding.jpg

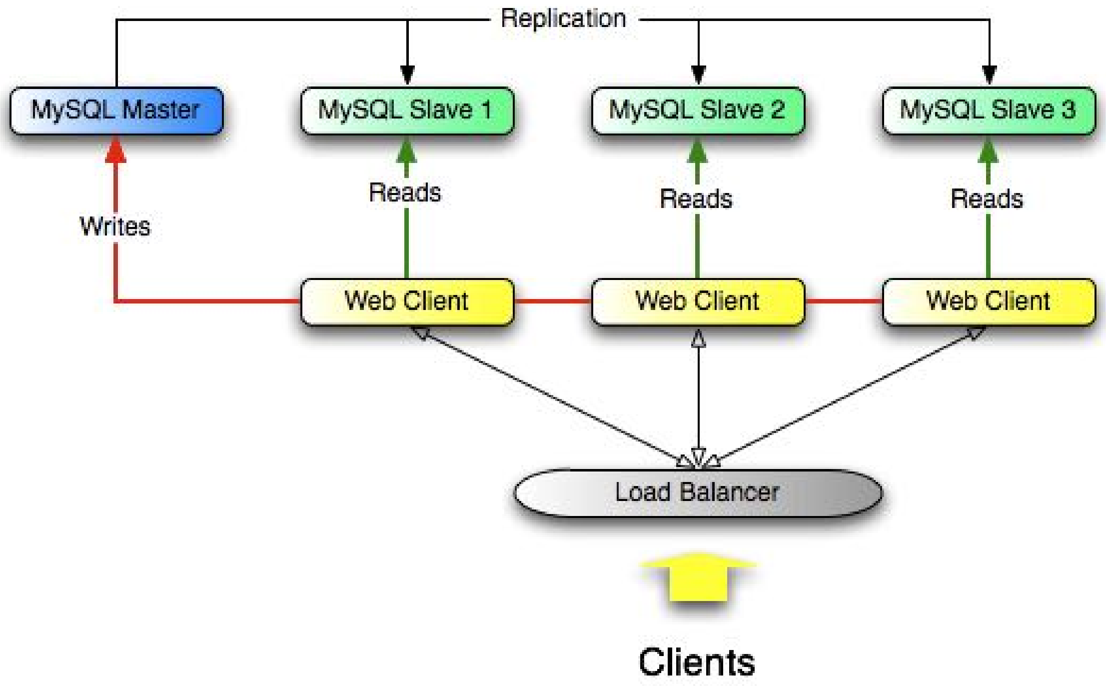
Cp. to concepts in Linear Algebra



Source: "An efficient scheme for probabilistic skyline queries over distributed uncertain data," Xiaoyong Li, Yijie Wang, Jie Yu, Telecommunication systems 2015.

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)

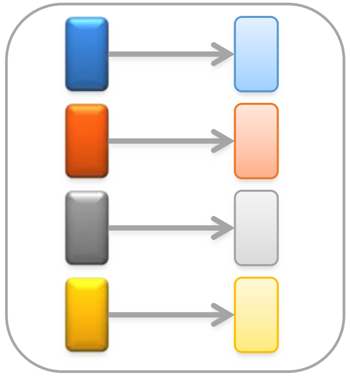
Open Source

- 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

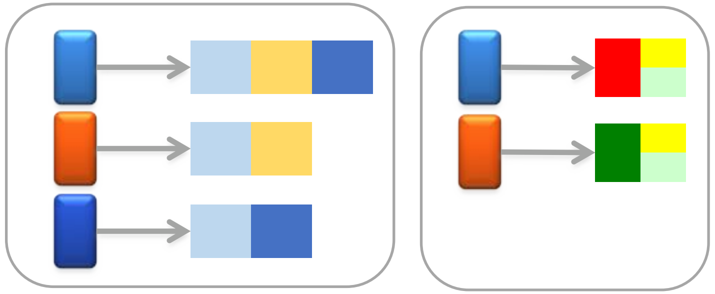
Apache Software Foundation

- 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, ...

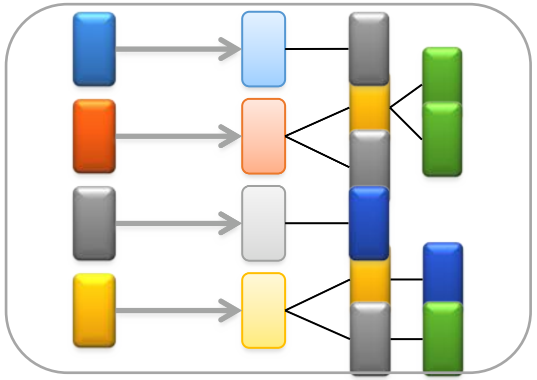
We Will Look at 4 Data Models



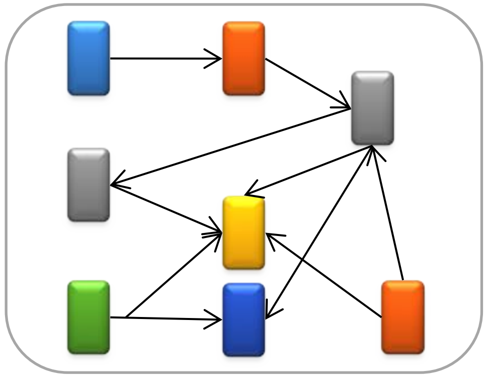
Key/Value Store
(e.g. REDIS)



Column-Family Store
(e.g. Cassandra)



Document Store
(e.g. MongoDB)



Graph Databases
(e.g. Neo4J)

Source: Benny Kimelfeld

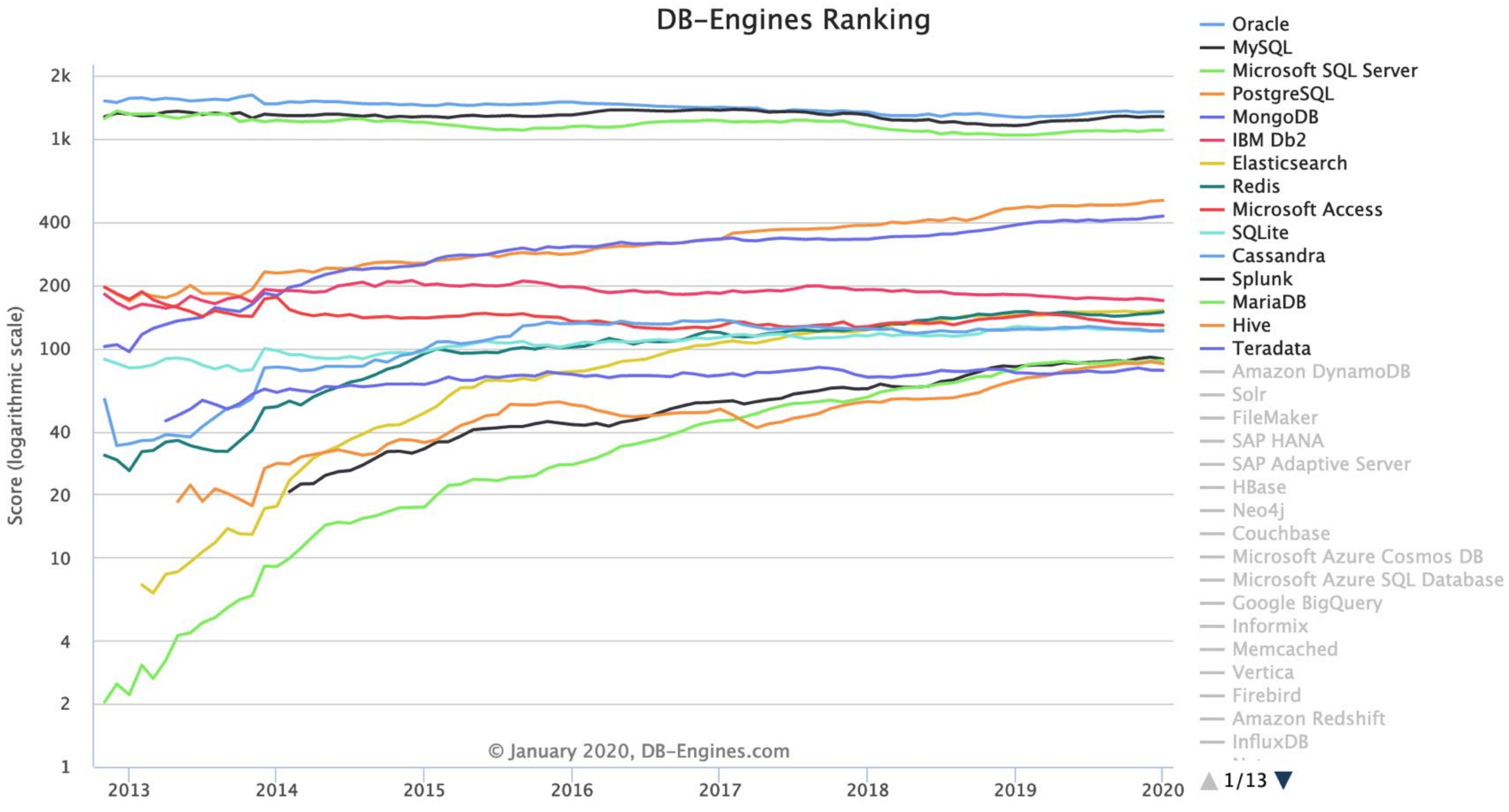
Database engines ranking by "popularity"

350 systems in ranking, January 2020

Rank			DBMS	Database Model	Score		
Jan 2020	Dec 2019	Jan 2019			Jan 2020	Dec 2019	Jan 2019
1.	1.	1.	Oracle	Relational, Multi-model	1346.68	+0.29	+77.85
2.	2.	2.	MySQL	Relational, Multi-model	1274.65	-1.01	+120.39
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1098.55	+2.35	+58.29
4.	4.	4.	PostgreSQL	Relational, Multi-model	507.19	+3.82	+41.08
5.	5.	5.	MongoDB	Document, Multi-model	426.97	+5.85	+39.78
6.	6.	6.	IBM Db2	Relational, Multi-model	168.70	-2.65	-11.15
7.	7.	↑ 8.	Elasticsearch	Search engine, Multi-model	151.44	+1.19	+8.00
8.	8.	↓ 7.	Redis	Key-value, Multi-model	148.75	+2.51	-0.27
9.	9.	9.	Microsoft Access	Relational	128.58	-0.89	-13.04
10.	↑ 11.	10.	SQLite	Relational	122.14	+1.78	-4.66
11.	↓ 10.	11.	Cassandra	Wide column	120.66	-0.04	-2.32
12.	12.	12.	Splunk	Search engine	88.67	-1.85	+7.25
13.	13.	13.	MariaDB	Relational, Multi-model	87.45	+0.66	+8.63
14.	14.	↑ 15.	Hive	Relational	84.24	-1.81	+14.33
15.	15.	↓ 14.	Teradata	Relational, Multi-model	78.29	-0.21	+2.10
16.	16.	↑ 20.	Amazon DynamoDB	Multi-model	62.02	+0.39	+6.93

Source: https://db-engines.com/en/ranking, 1/2020

Database engines ranking by "popularity"



Source: https://db-engines.com/en/ranking_trend, 1/2020

Highlighted Database Features

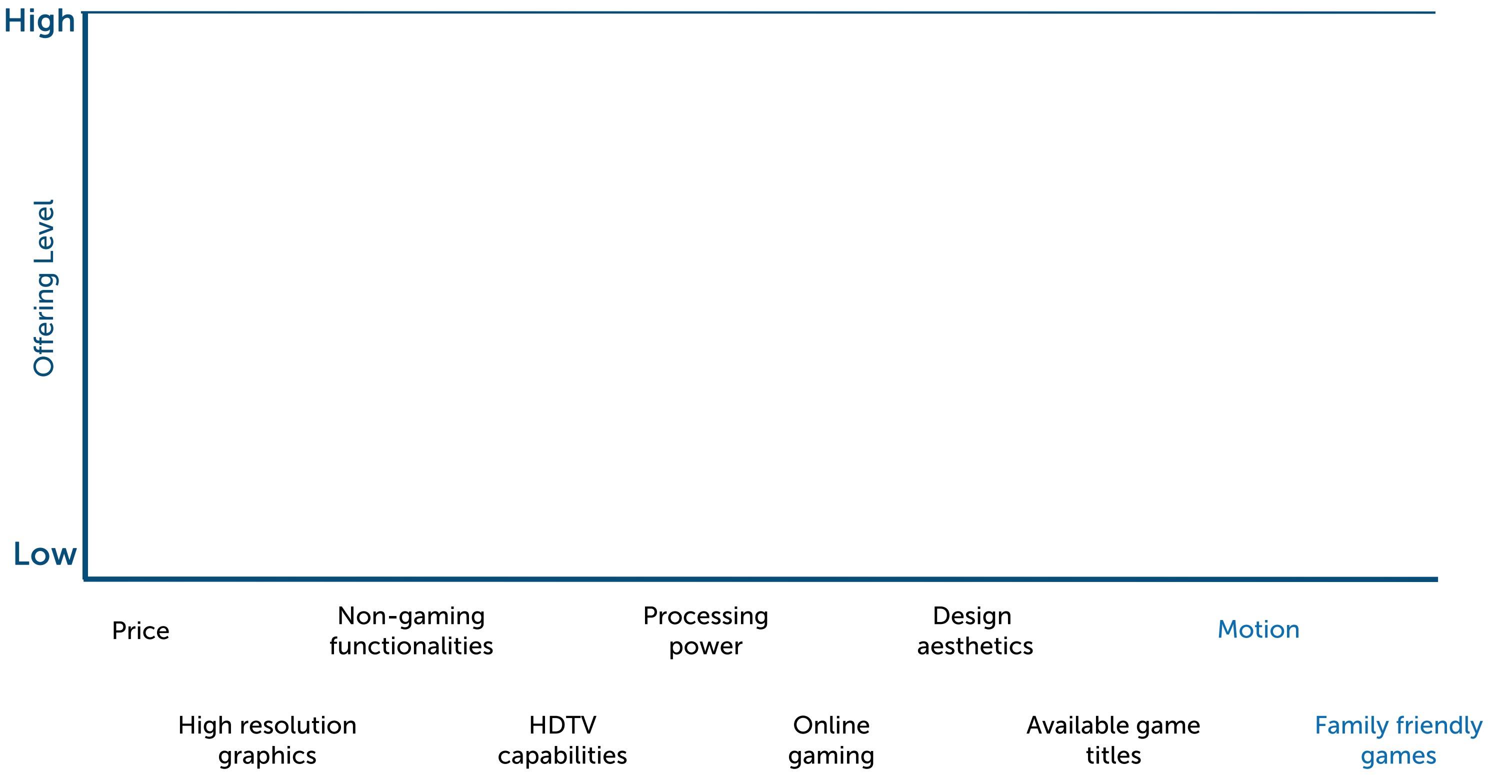
- Data model
  - What data is being stored?
- CRUD interface
  - API for Create, Read, Update, Delete
    - 4 basic functions of persistent storage (insert, select, update, delete)
  - Sometimes preceding S for Search
- Transaction consistency guarantees
- Replication and sharding model
  - What's automated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

Strategy Canvas: Example Nintendo Wii (1/3)

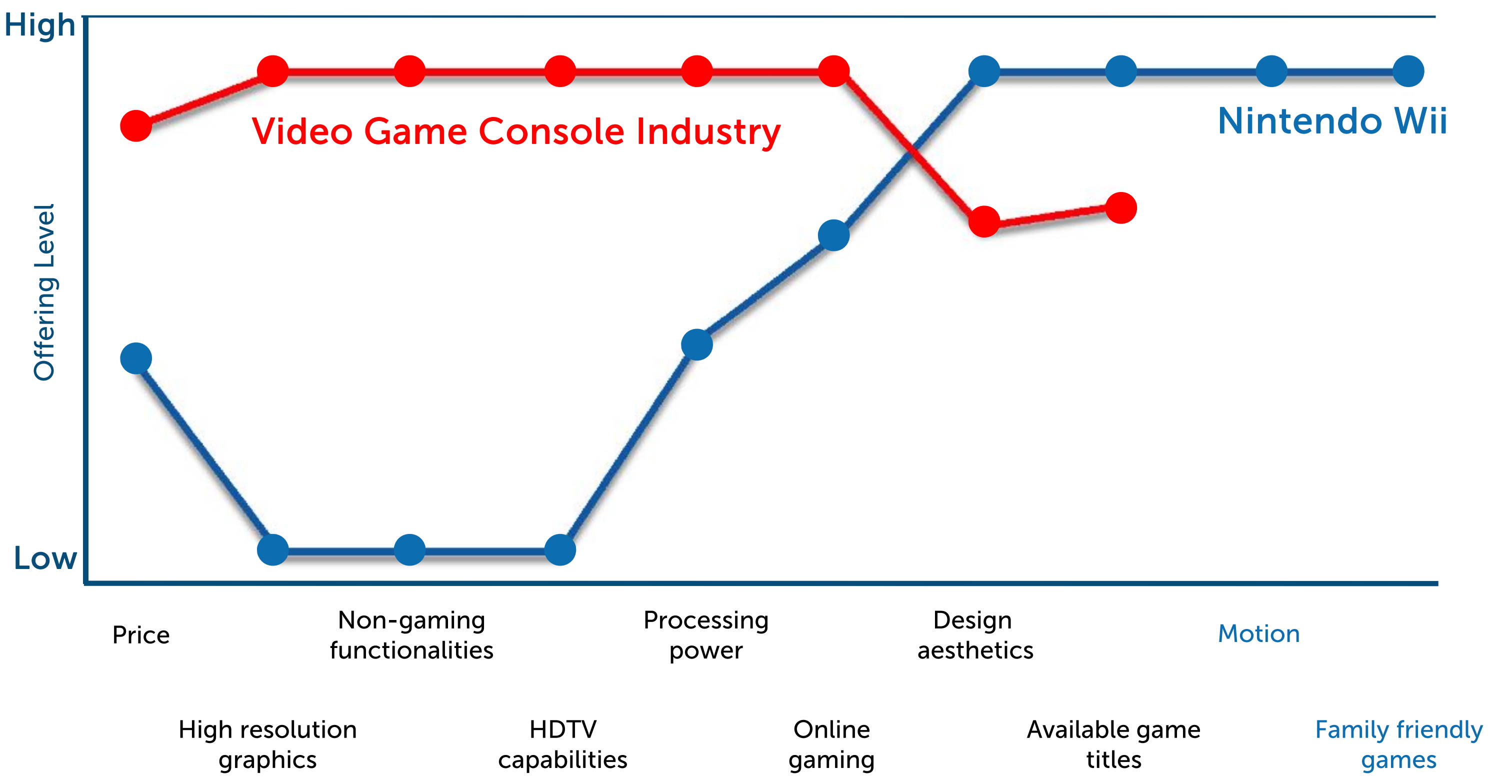
Nintendo Wii Strategy Canvas



Source: INSEAD, Blue Ocean Strategy Institute, 2013.

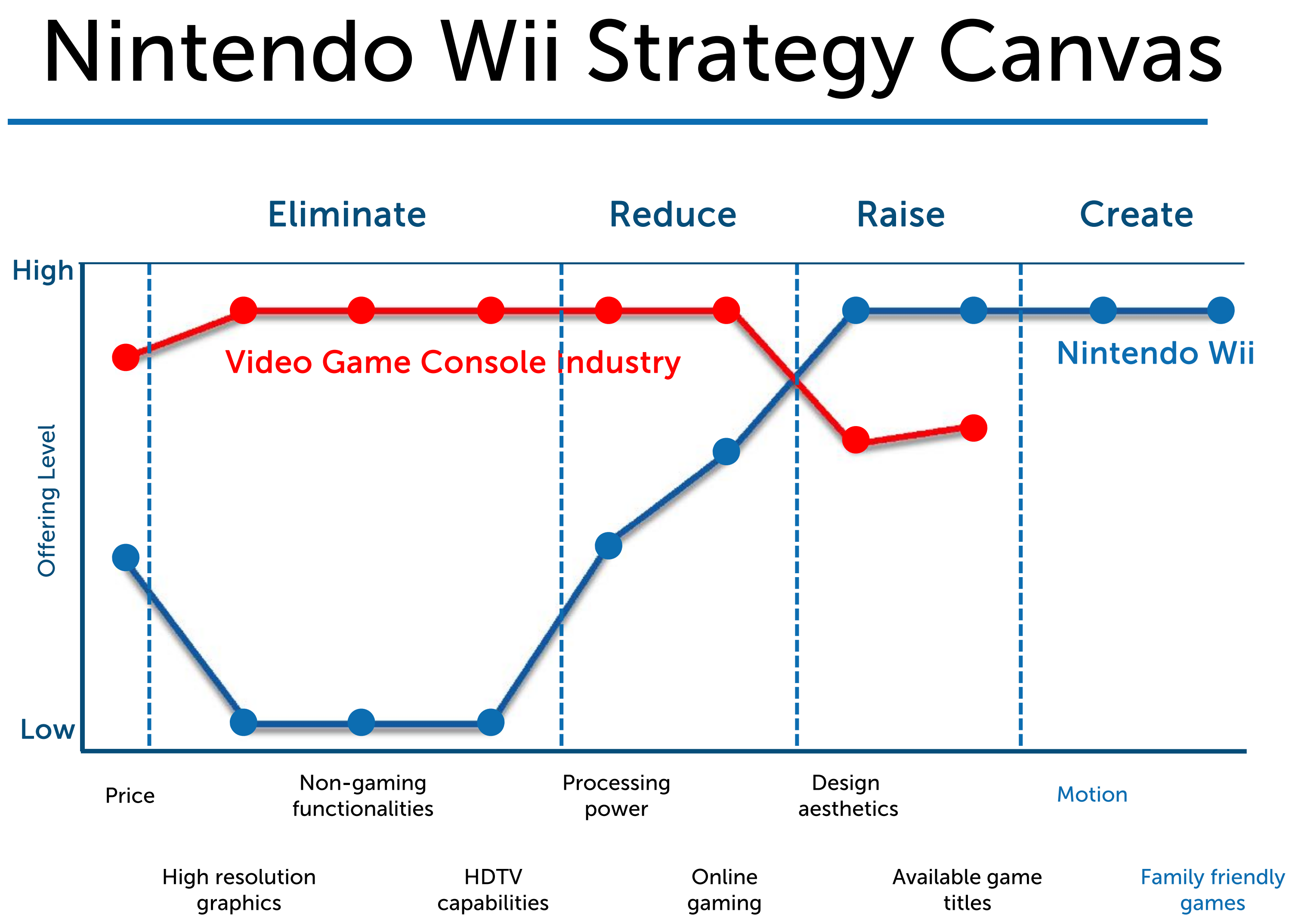
Strategy Canvas: Example Nintendo Wii (2/3)

Nintendo Wii Strategy Canvas



Source: INSEAD, Blue Ocean Strategy Institute, 2013.

Strategy Canvas: Example Nintendo Wii (3/3)



Source: INSEAD, Blue Ocean Strategy Institute, 2013.

Redefine the Market

Outline: Alternative data models

- 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 (conflicts)?
- 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
- Textbook example: bank money transfer
  - Read from acct#1, verify funds, update acct#1, update acct#2
- Online purchasing
  - Similar
- "Transactional file systems" (MS NTFS)
  - Moving a file from one directory to another: verify file exists, copy, delete

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
```

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$_1$

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

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

txn$_2$

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

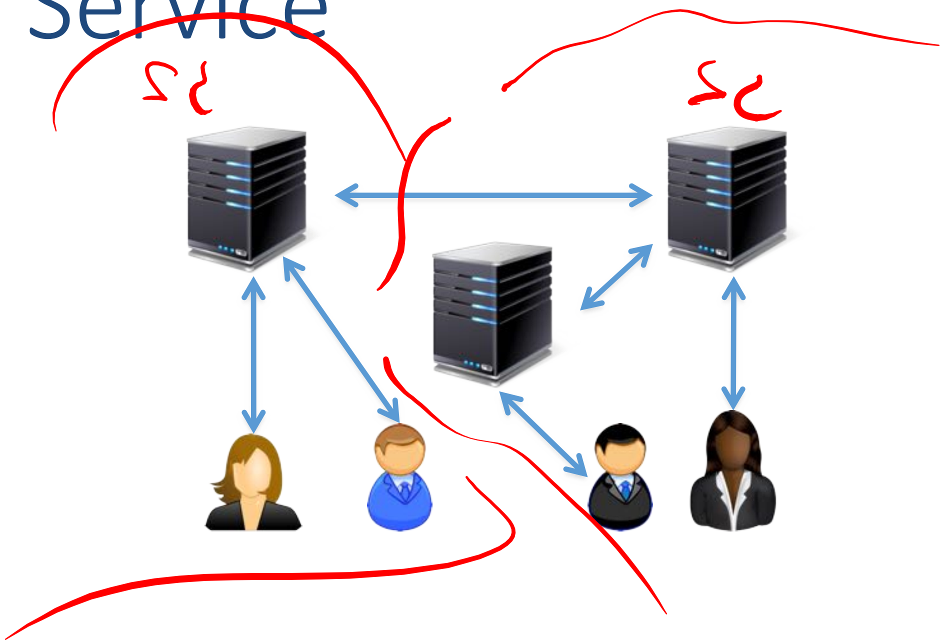
ACID

- **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



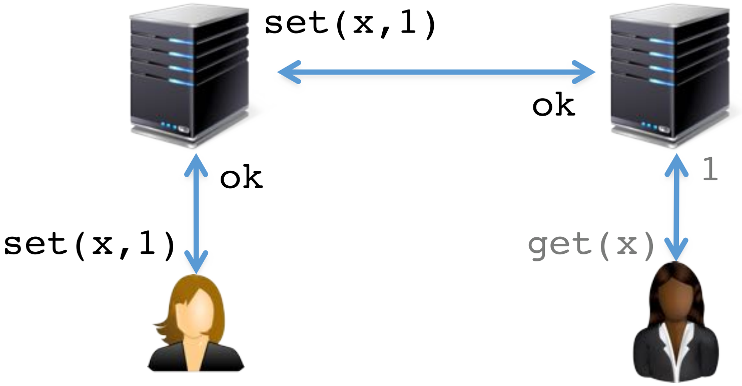
- Context: distributed service
  - e.g., social network
- Clients make get / set requests
  - e.g., setLike(user,post), getLikes(post)
  - Each client can talk to any server
- Servers return responses
  - e.g., ack, {$user_1$,....,$user_k$}
- Failure: the network may occasionally disconnect due to failures (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 returns 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

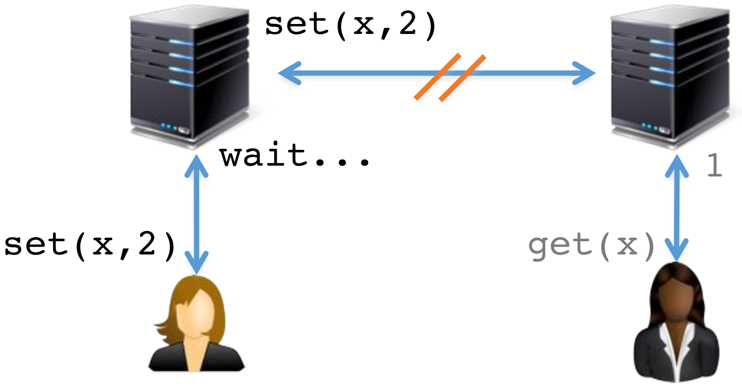
Simple Illustration

Our Relational Database world so far …



CA

Consistency, Availability

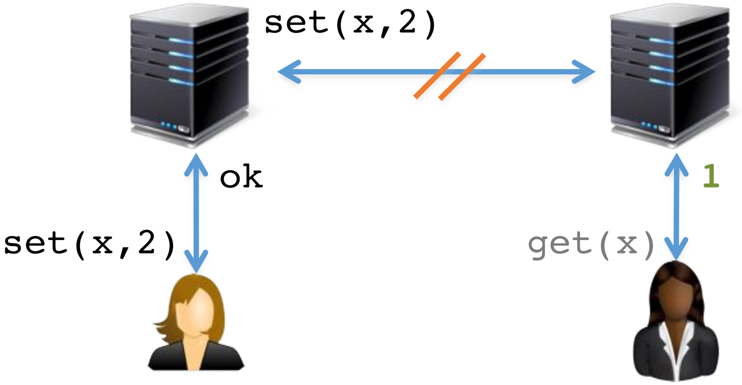


In a system that may suffer partitions, you have to trade off consistency vs. availability

CP

Consistency, Partition tolerance

~~Availability~~



AP

Availability, Partition tolerance

~~Consistency~~

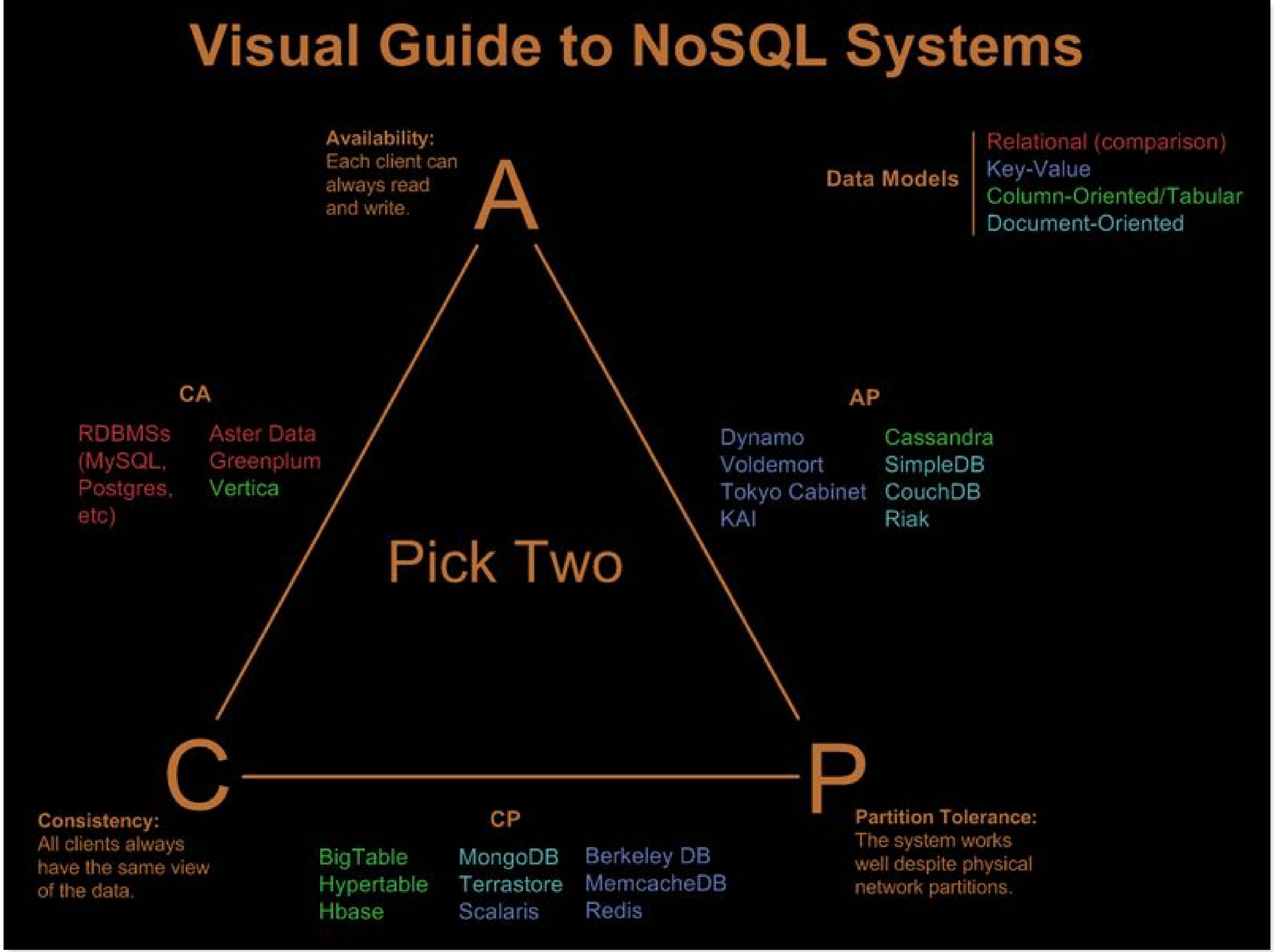
The CAP Theorem

Eric Brewer's CAP Theorem:

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

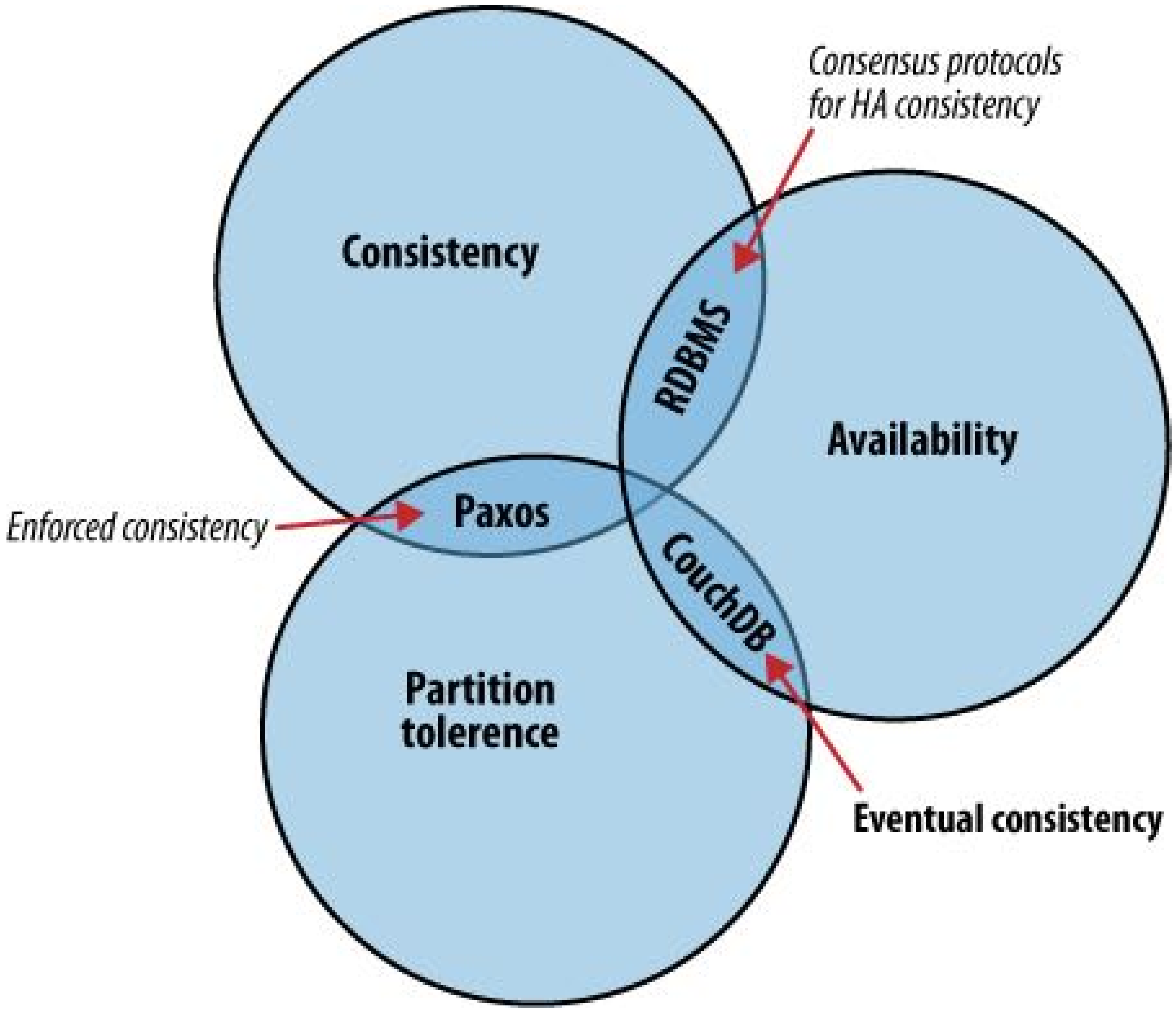
Historical Note

- 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: Alternative data models

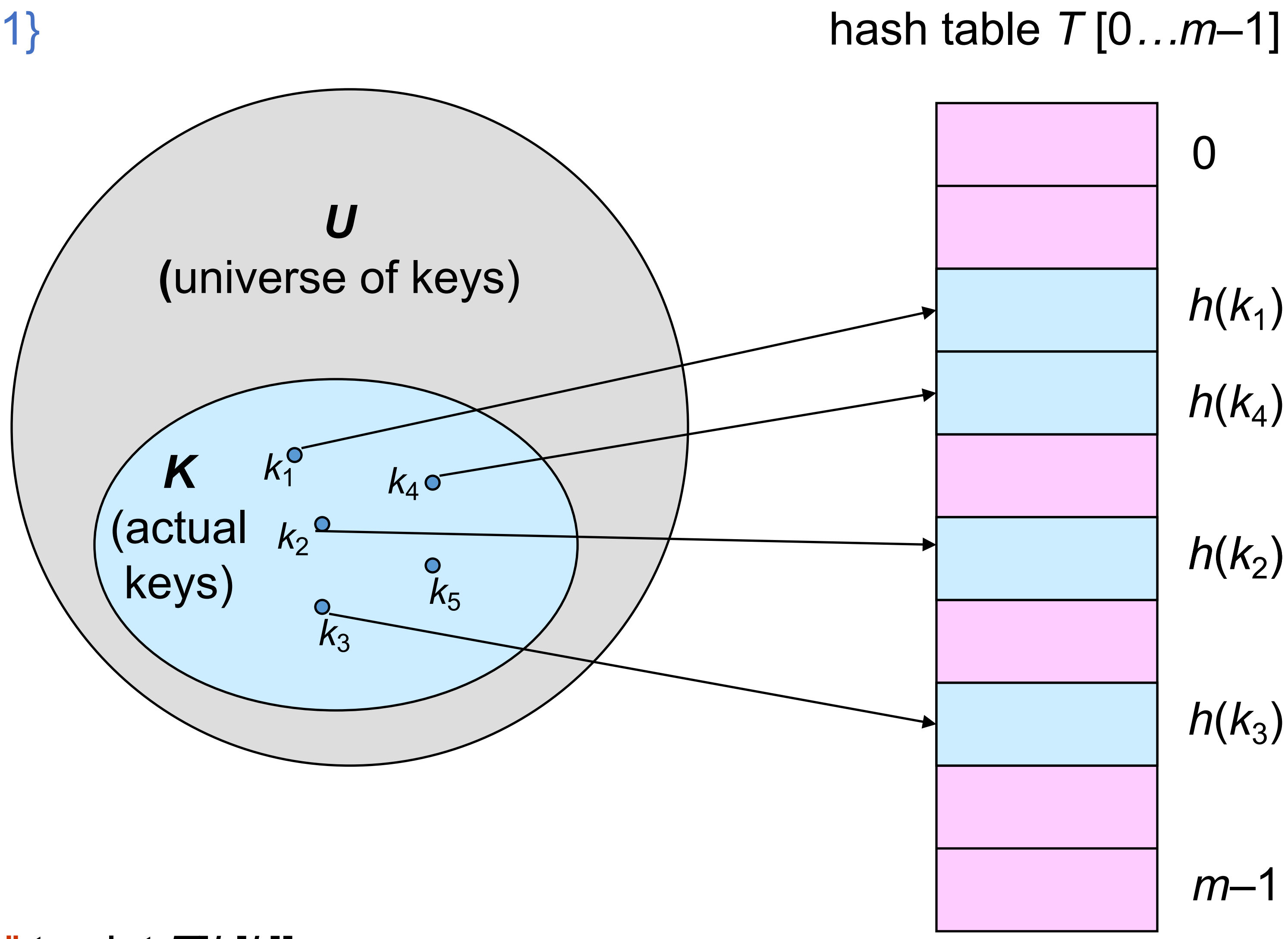
- Introduction
- Transaction Consistency
- 4 main data models
  - Key-Value Stores (e.g., Redis)
  - Column-Family Stores (e.g., Cassandra)
  - Document Stores (e.g., MongoDB)
  - Graph Databases (e.g., Neo4j)
- Concluding Remarks

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)

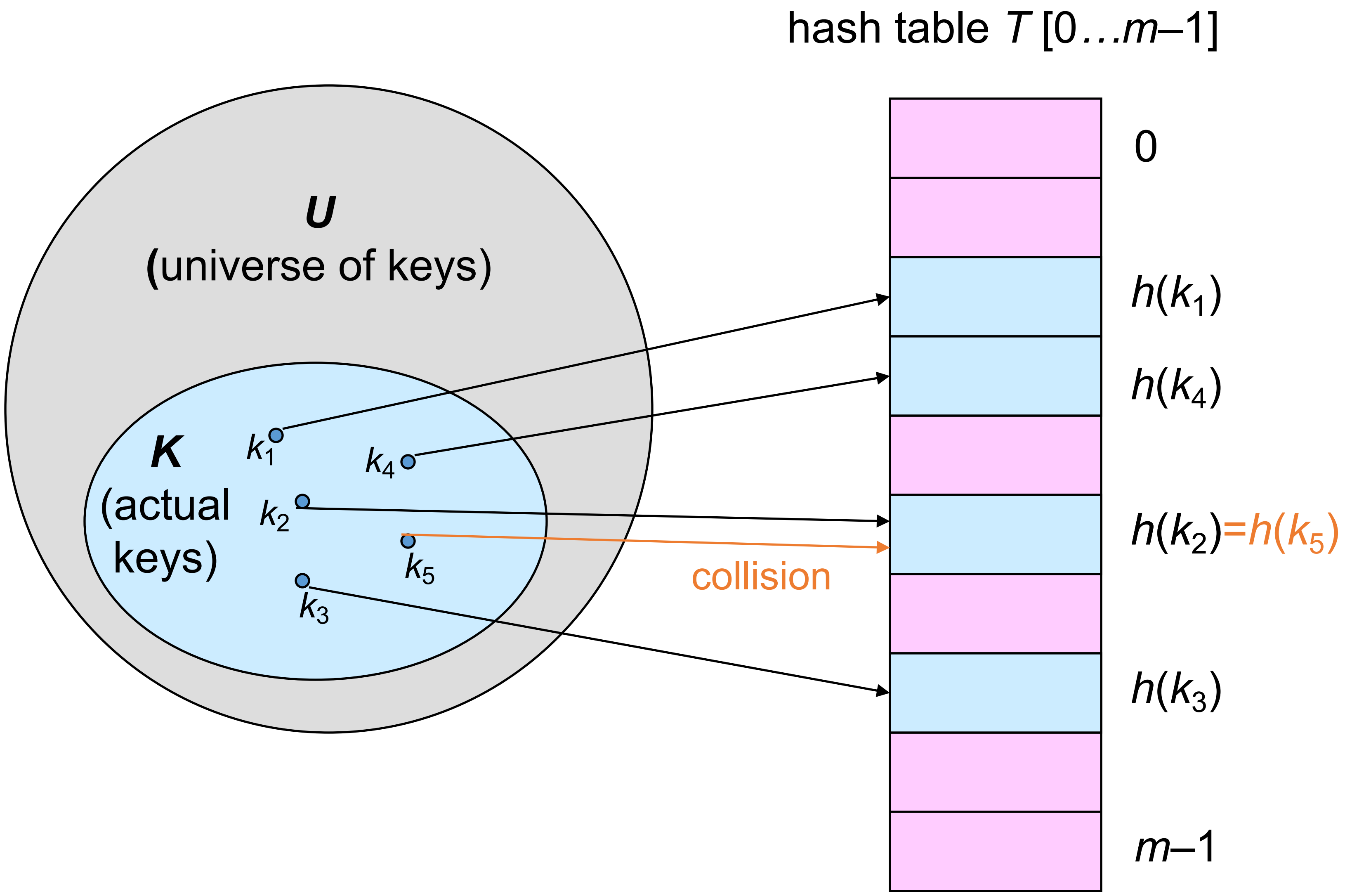
$h : U \rightarrow \{0,1,\ldots, m-1\}$



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

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 optional Jupyter notebooks to play with
- 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

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
- In-memory 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
```

An excursion into indexing

“If you don’t find it in the index,
look very carefully through the entire catalog”

- Sears, Roebuck and Co., Consumers Guide, 1897

High-level overview: indexes

id	age	salary	other
006	19	50k	...
005	20	55k	...
004	25	50k	...
007	30	80k	...
002	35	75k	...
003	35	70k	...
001	40	65k	...

data file = index file
clustered (primary) index

id	age	salary	other
006	19	50k	...
004	25	50k	...
005	20	55k	...
001	40	65k	...
003	35	70k	...
002	35	75k	...
007	30	80k	...

index file
unclustered (secondary) index

Indexes: High-level

- An index on a file speeds up selections on the search key fields for the index.
  - Search key properties
    - Any subset of fields
    - is not the same as key of a relation

- Example:

`Product(name, maker, price)`

On which attributes would you build indexes?

More precisely

- An **index** is a data structure mapping search keys to sets of rows in a database table
    - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (primary index) or pointers to those rows (secondary index)

Operations on an Index

- Search: Quickly find all records which meet some condition on the search key attributes
  - More sophisticated variants as well. Why?

- Insert / Remove entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is <u>covering</u> *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

High-level Categories of Index Types

- B-Trees
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - We will look at a variant called B+ Trees

- Hash Tables
  - There are variants of this basic structure to deal with IO
  - Called linear or extendible hashing- IO aware!

The data structures we present here are "IO aware"

Real difference between structures: costs of ops *determines which index you pick and why*

Activity-51.ipynb

Further Motivation for Indexes: NoSQL!

- NoSQL engines are (basically) just indexes!
  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!
  - Sometimes use B+ Trees, sometimes hash indexes

Indexes are critical across all DBMS types

Back to NoSQL!
Concretely, Key-values

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
```

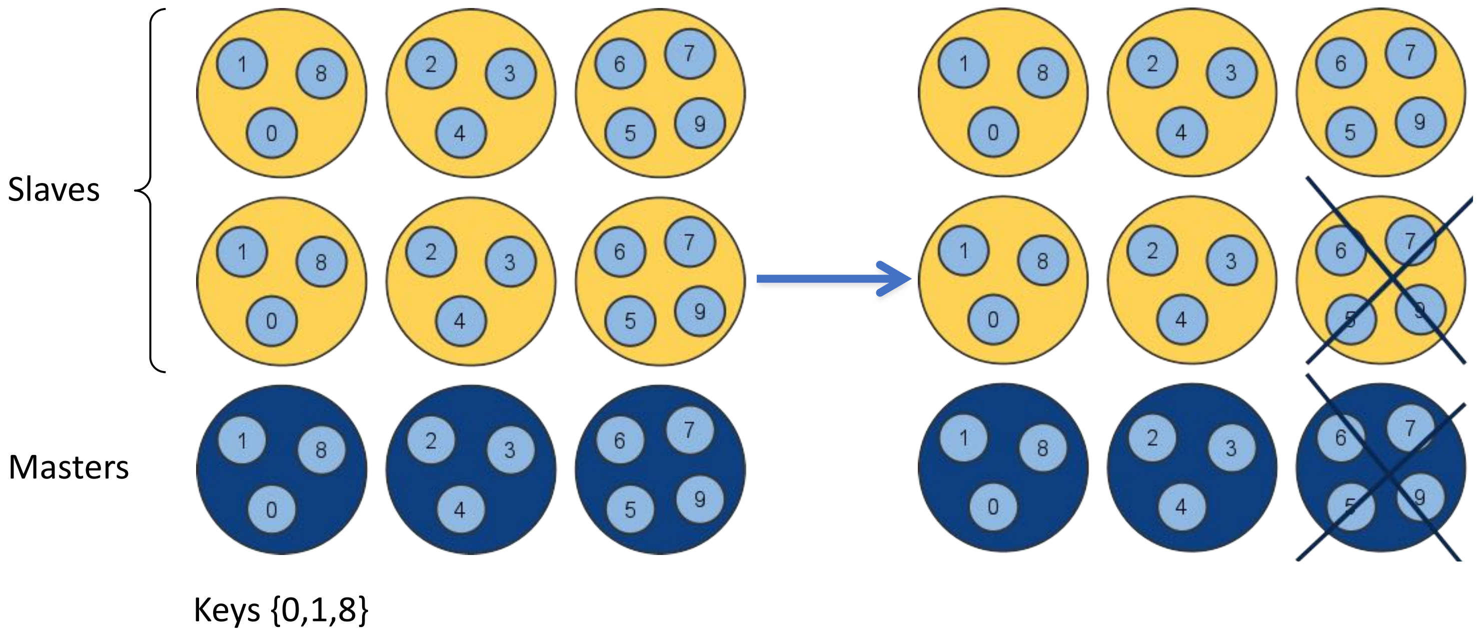
```
rpop l
>> b
```

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 redundancy

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

Do-it-yourself secondary indexing

How to find users by email?

Base Table USERS

Rowkey	UserName	Email	Country
100001	dbman	dbman@gmail	USA
100002	georgy	georgy@gmail	USA
100003	jane	jane@outlook	USA
100004	kate	kate@yahoo	Australia
100005	Mike	Mike@gmail	USA
100006	crilly1	crilly1@aol	USA
100007	bamflux	bamflux@gmail	Australia

Source: 2016 - Harrison - Next Generation Databases -- NoSQL, NewSQL, and Big Data: Fig 10-13

Do-it-yourself secondary indexing

Index table on EMAIL

Rowkey	BaseTableKey
bamflux@gmail	100007
crilly1@aol	100006
dbman@gmail	100001
georgy@gmail	100002
jane@outlook	100003
kate@yahoo	100004
Mike@gmail	100005

How to find users by email?

Base Table USERS

Rowkey	UserName	Email	Country
100001	dbman	dbman@gmail	USA
100002	georgy	georgy@gmail	USA
100003	jane	jane@outlook	USA
100004	kate	kate@yahoo	Australia
100005	Mike	Mike@gmail	USA
100006	crilly1	crilly1@aol	USA
100007	bamflux	bamflux@gmail	Australia

How to find users by country?

Source: 2016 - Harrison - Next Generation Databases -- NoSQL, NewSQL, and Big Data: Fig 10-13

Do-it-yourself secondary indexing

How to find users by email?

Index table on EMAIL

Rowkey	BaseTableKey
bamflux@gmail	100007
crilly1@aol	100006
dbman@gmail	100001
georgy@gmail	100002
jane@outlook	100003
kate@yahoo	100004
Mike@gmail	100005

Base Table USERS

Rowkey	UserName	Email	Country
100001	dbman	dbman@gmail	USA
100002	georgy	georgy@gmail	USA
100003	jane	jane@outlook	USA
100004	kate	kate@yahoo	Australia
100005	Mike	Mike@gmail	USA
100006	crilly1	crilly1@aol	USA
100007	bamflux	bamflux@gmail	Australia

Index table on COUNTRY

RowKey	100001	100002	100003	100005	100006
USA					

RowKey	100004	100007
Australia		

How to find users by country?

Source: 2016 - Harrison - Next Generation Databases -- NoSQL, NewSQL, and Big Data: Fig 10-13

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: Alternative data models

- 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	Vienna	2	NULL
753	Amir	London	NULL	CS
955	Armin	NULL	2	IE

Column store *(still SQL)*

id	sid
1	861
2	753
3	955

id	name
1	Alice
2	Amir
3	Armin

id	address
1	Vienna
2	London

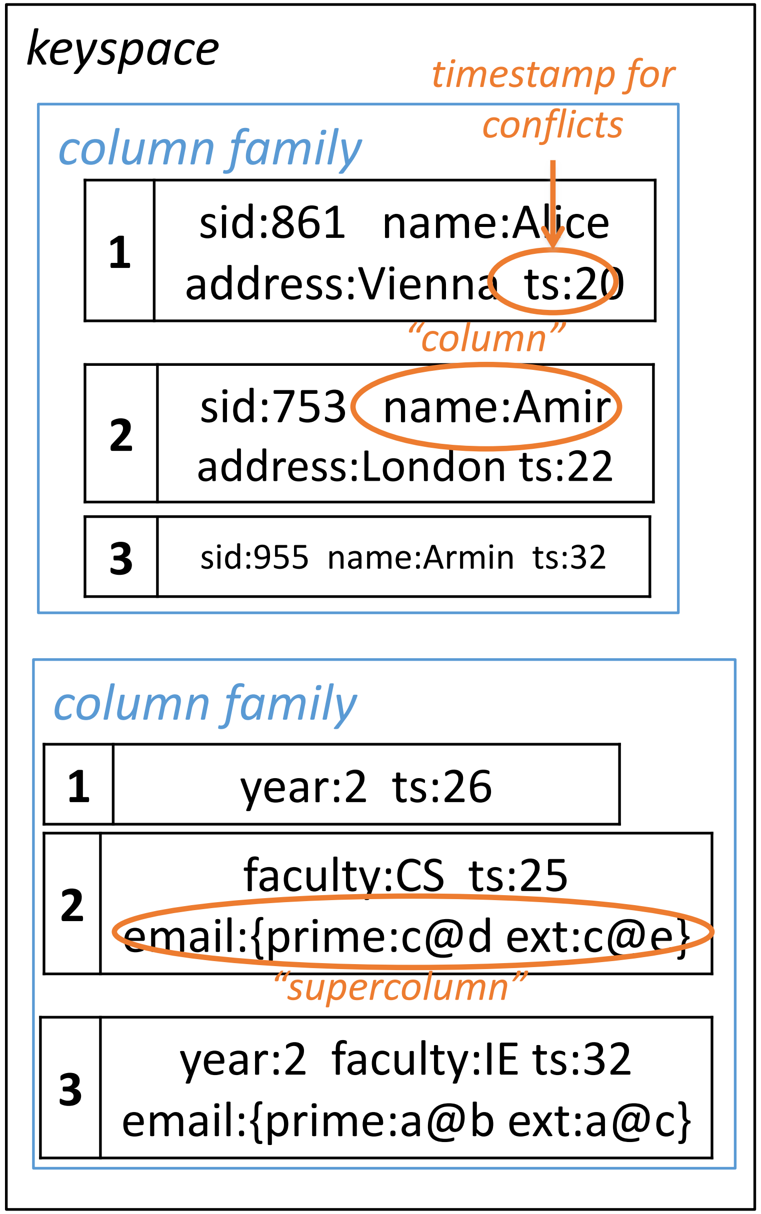
id	year
1	2
3	2

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

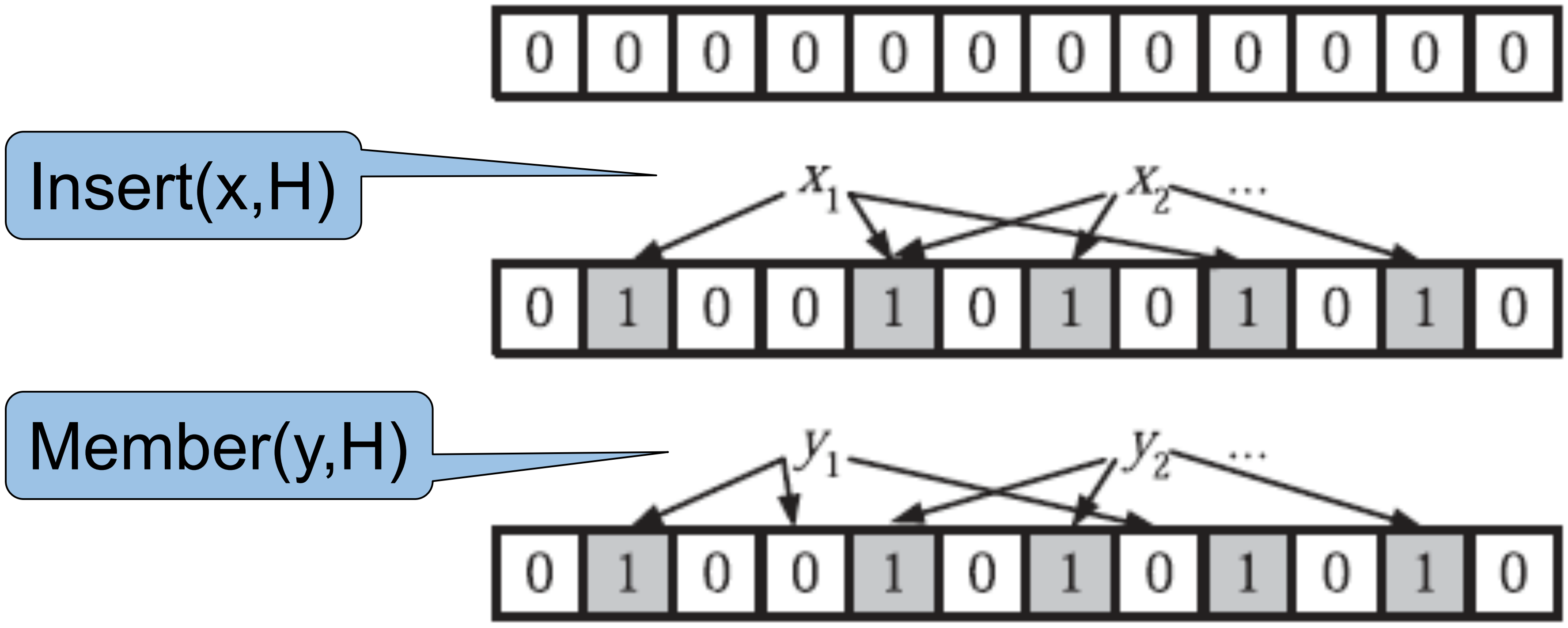
Examples Systems

- 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

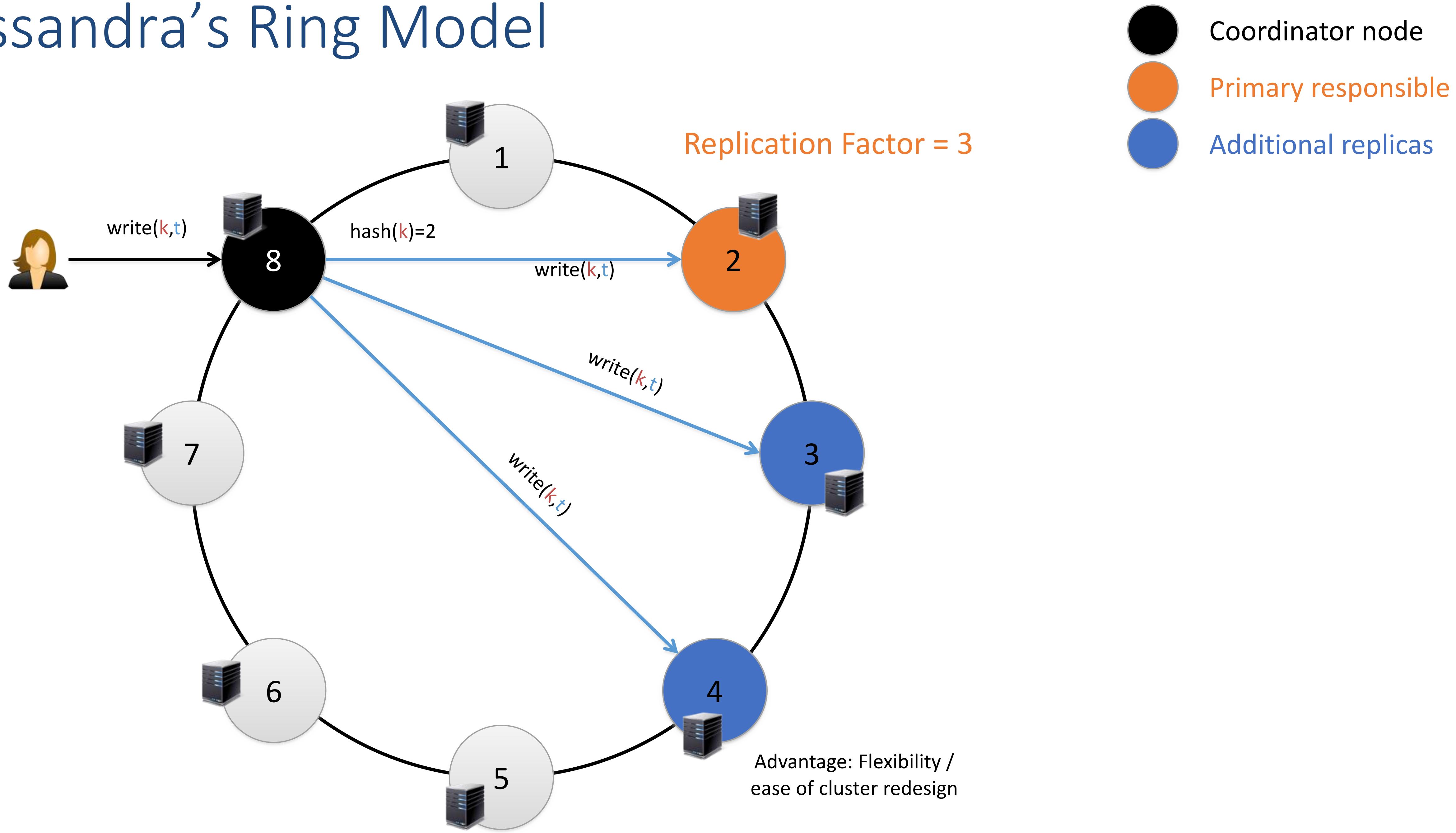
Example Bloom Filter k=3



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 change query patterns often (in RDMS schema changes are costly, in Cassandra query changes too: require changing the column family design)

Outline: Alternative data models

- 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"]
}
```

~ record / row / tuple

Collection

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

~ table

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 → `{ status: "A" }`

projection → `{ cust_id: 1, price: 1, _id: 0 }`

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"

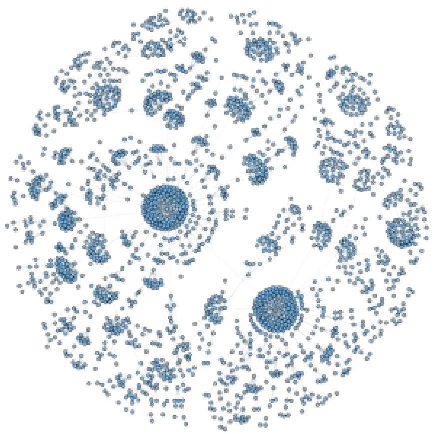
When to use it

- 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: Alternative data models

- 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

Graph Databases



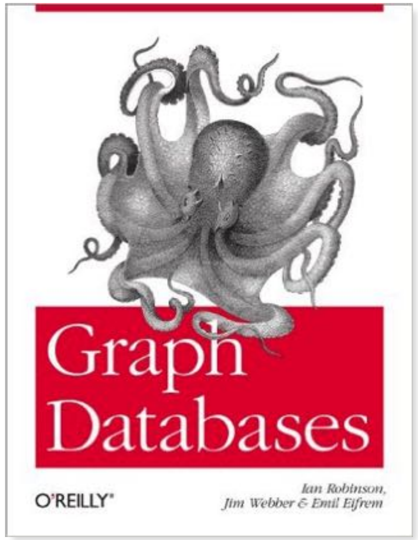
- Restricted case of a relational schema:
  - Nodes (+labels/properties)
  - Edges (+labels/properties)
- Motivated by the popularity of network/communication oriented applications
- Efficient support for graph-oriented queries
  - *Reachability*, *graph patterns*, *path patterns*
  - Ordinary RDBs either not support or inefficient for such queries
    - Path of length k is a k-wise self join; yet a very special one...
- Specialized languages for graph queries
  - For example, pattern language for paths
- Plus distributed, 2-of-CAP, etc.
  - Depending on the design choices of the vendor

Example Databases

- Graph with nodes/edges marked with labels and properties (labeled property graph)
  - Sparksee (DEX) (Java, 1st release 2008)
  - neo4j (Java, 1st release 2010)
  - InfiniteGraph (Java/C++, 1st release 2010)
  - OrientDB (Java, 1st release 2010)
- Triple stores: Support W3C RDF and SPARQL, also viewed as graph databases
  - MarkLogic, AllegroGraph, Blazegraph, IBM SystemG, Oracle Spatial & Graph, OpenLink Virtuoso, ontotext

neo4j

- Open source, written in Java
  - First version released 2010
- Supports the Cypher query language (declarative graph QL)
- Clustering support
  - Replication and sharding through master-slave architectures
- Used by ebay, Walmart, Cisco, National Geographic, TomTom, Lufthansa, ...

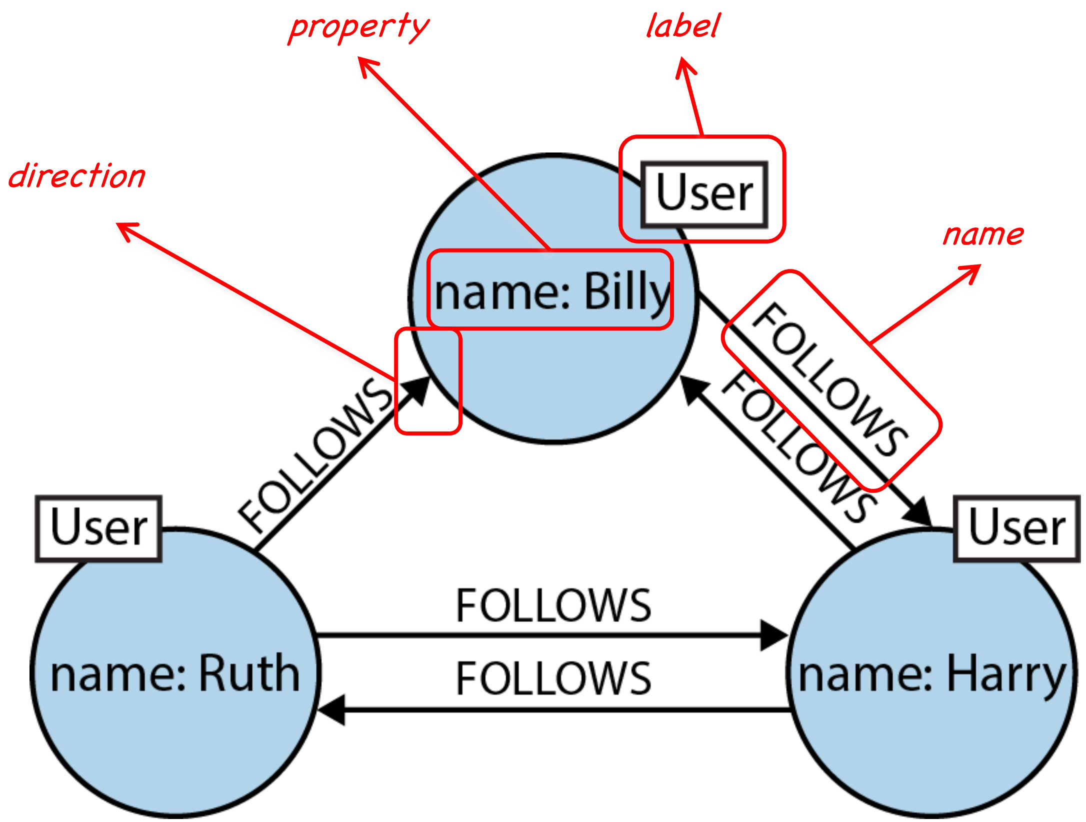


Examples taken from *Graph Databases* by Robinson, Webber, and Eifrem (O'Reilly) – free eBook

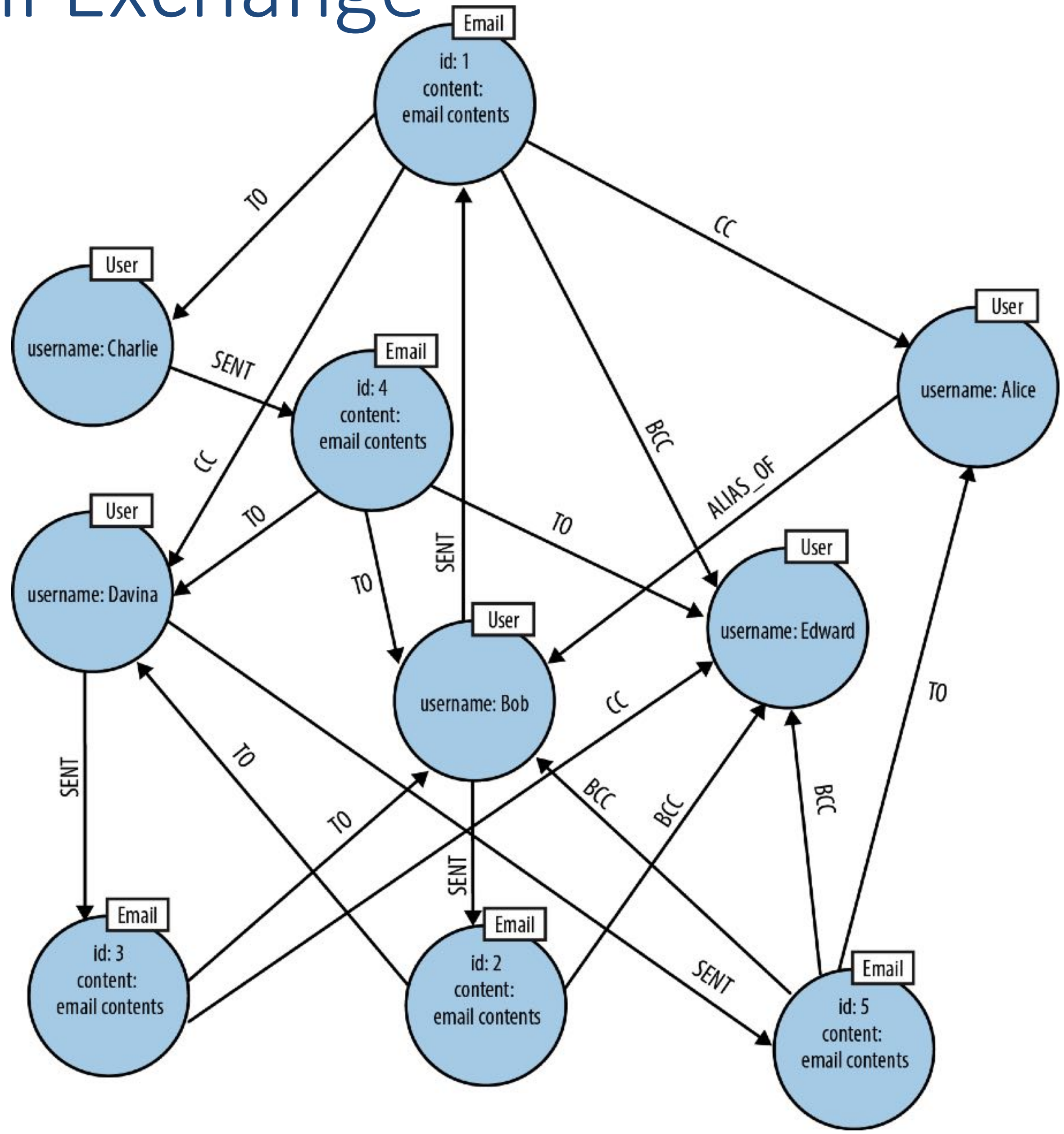
The Graph Data Model in Cypher

- Labeled property graph model
- Node
  - Has a set of *labels* (typically one label)
  - Has a set of *properties* key:value (where value is of a primitive type or an array of primitives)
- Edge (relationship)
  - Directed: node→node
  - Has a *name*
  - Has a set of *properties* (like nodes)

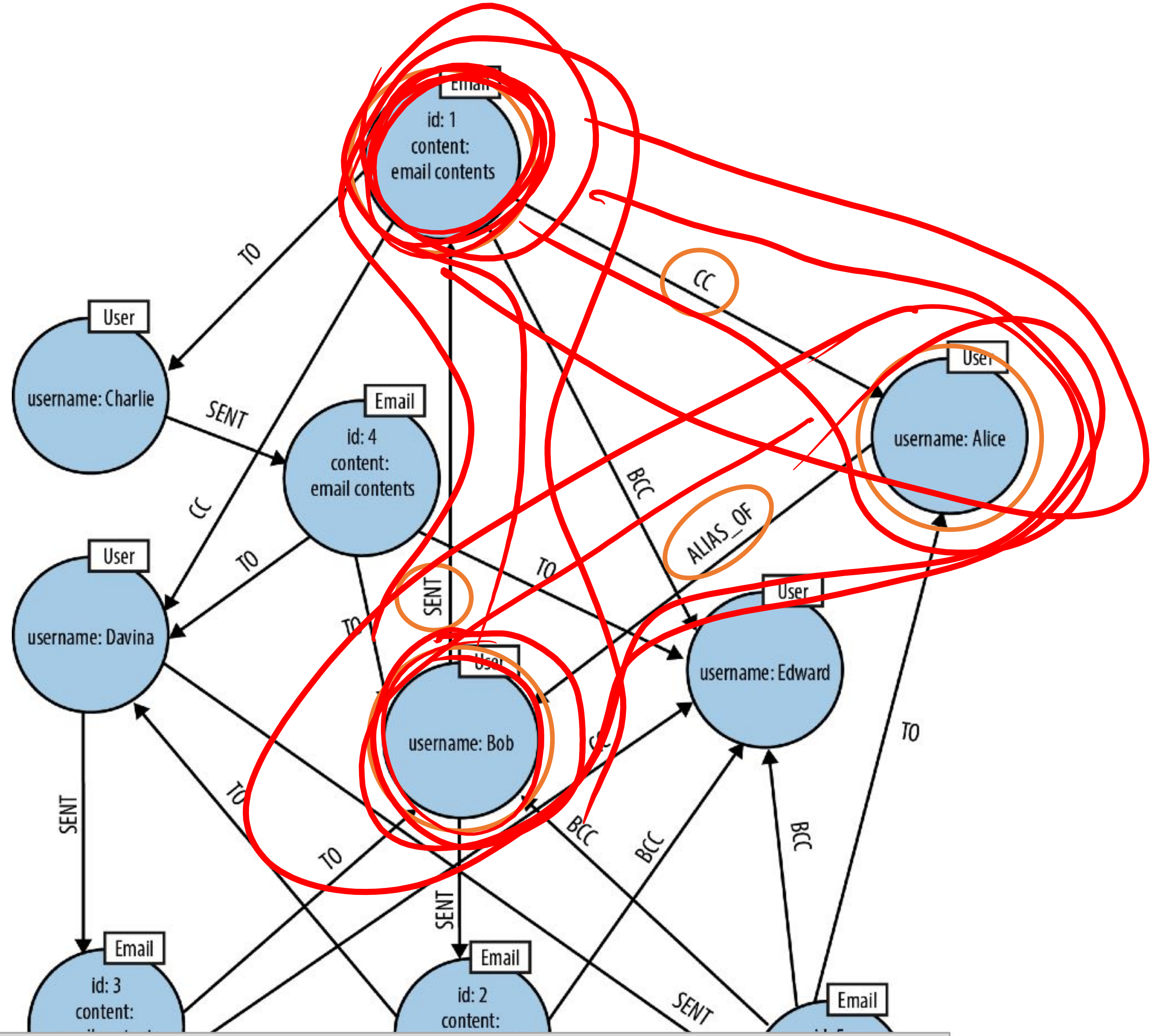
Example: Cypher Graph for Social Networks



Another Example: Email Exchange



Query Example

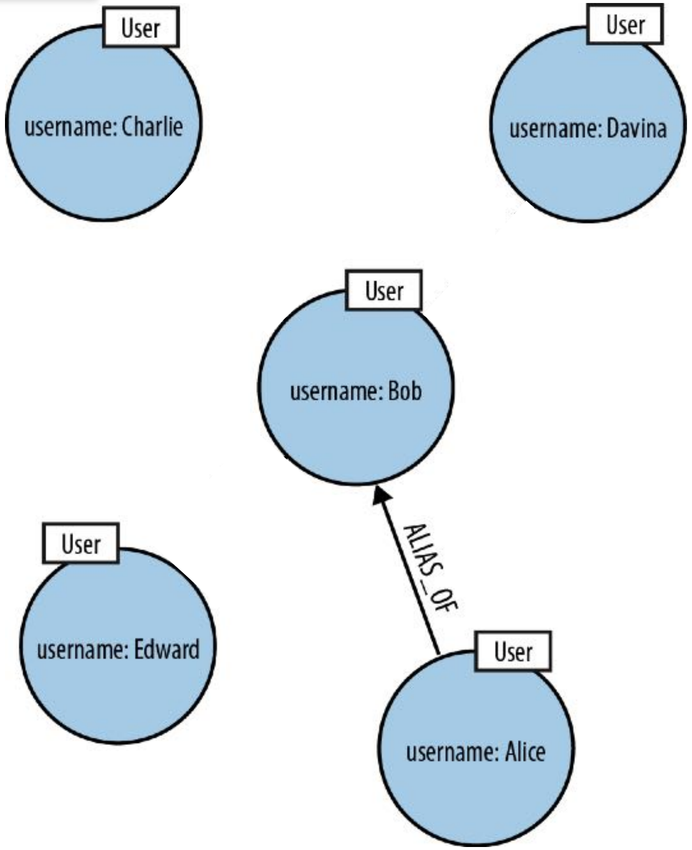


email
Node{id:"1",content:"..."}

```
MATCH (bob:User{username:'Bob'})-[:SENT]->(email)-[:CC]->(alias),
      (alias)-[:ALIAS_OF]->(bob)
RETURN email
```

Creating Graph Data

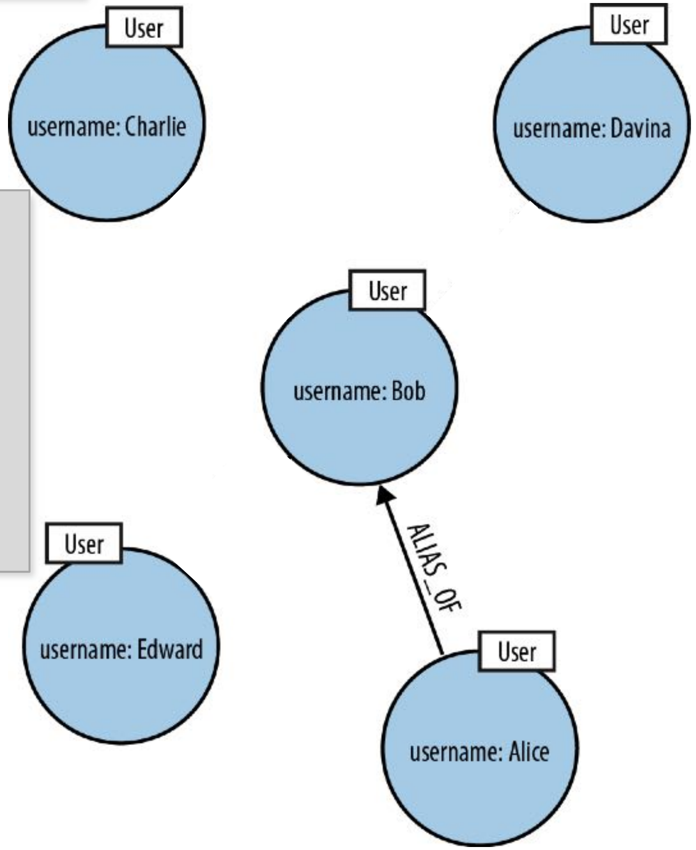
```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'}),
       (alice)-[:ALIAS_OF]->(bob)
```



Creating Graph Data

```
CREATE (alice:User {username:'Alice'}),
       (bob:User {username:'Bob'}),
       (charlie:User {username:'Charlie'}),
       (davina:User {username:'Davina'}),
       (edward:User {username:'Edward'}),
       (alice)-[:ALIAS_OF]->(bob)
```

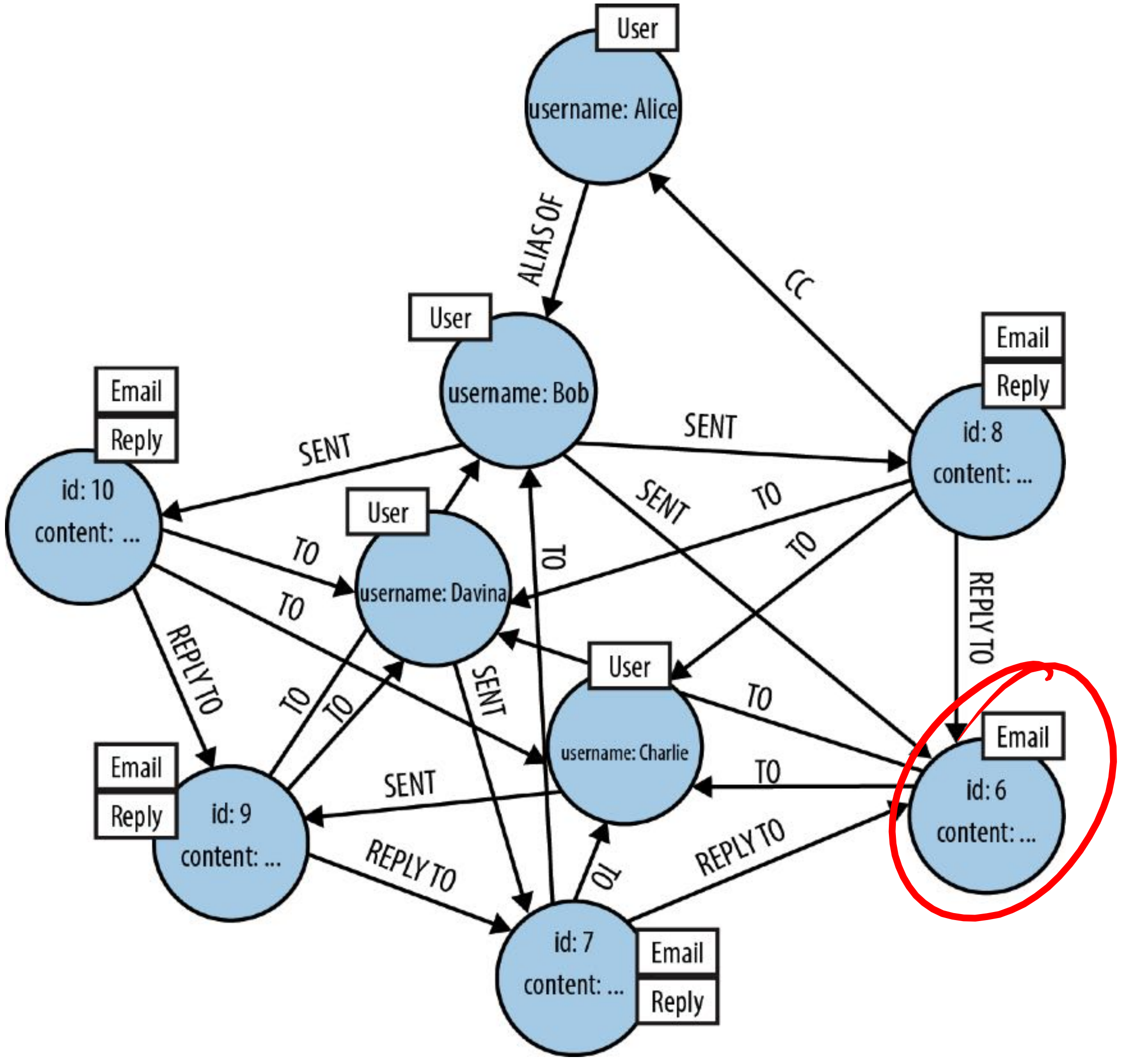
```
MATCH (bob:User {username:'Bob'}),
      (charlie:User {username:'Charlie'}),
      (davina:User {username:'Davina'}),
      (edward:User {username:'Edward'})
CREATE (bob)-[:EMAILED]->(charlie),
       (bob)-[:CC]->(davina),
       (bob)-[:BCC]->(edward)
```



Another Example

Path assignment

replier	depth
Davina	1
Bob	1
Charlie	2
Bob	3

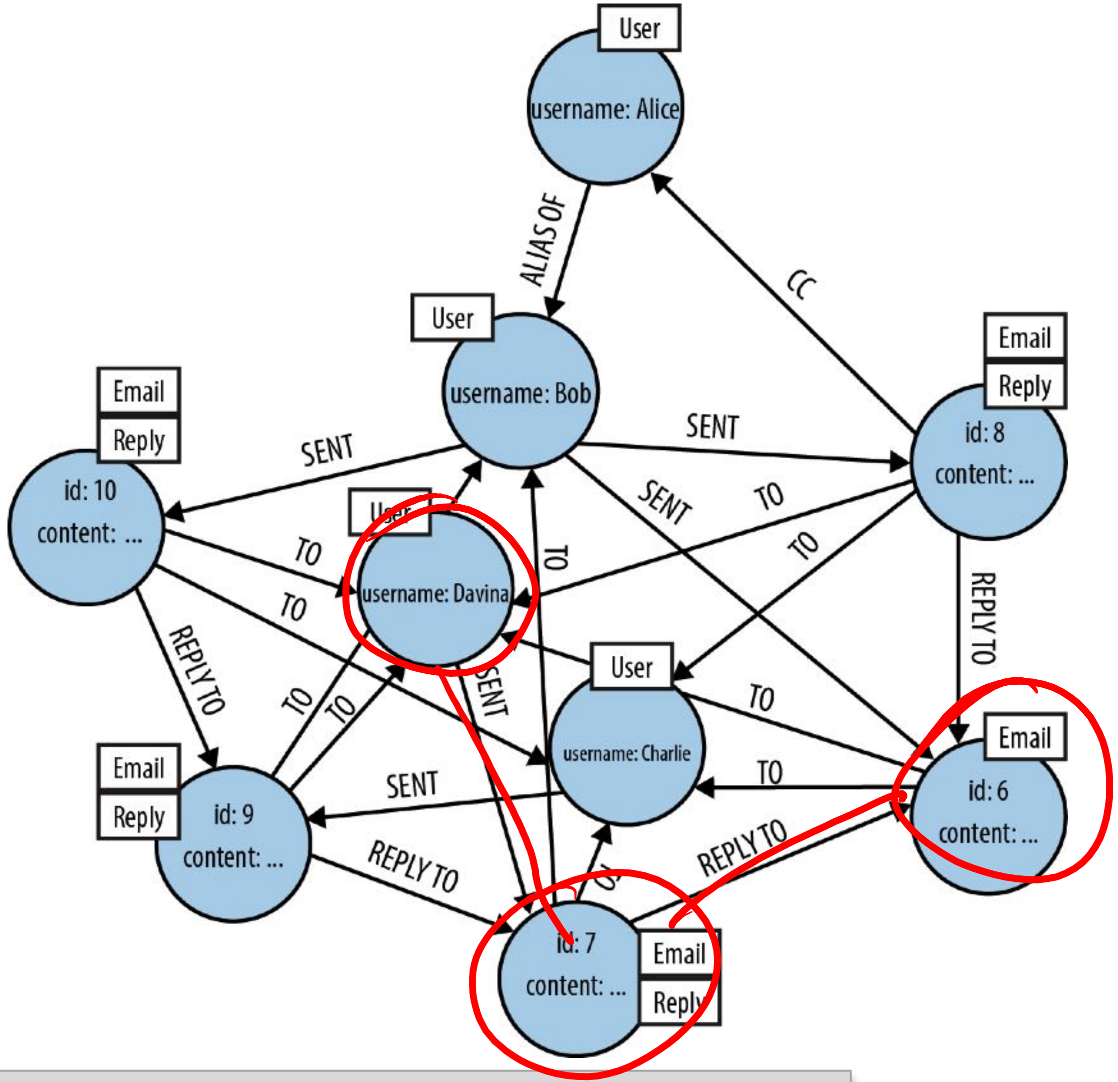



```
MATCH p = (email:Email {id:'6'})
      <-[:REPLY_TO*1..4]-(:Reply)<-[:SENT]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
```

Another Example

Path assignment

replier	depth
Davina	1
Bob	1
Charlie	2
Bob	3

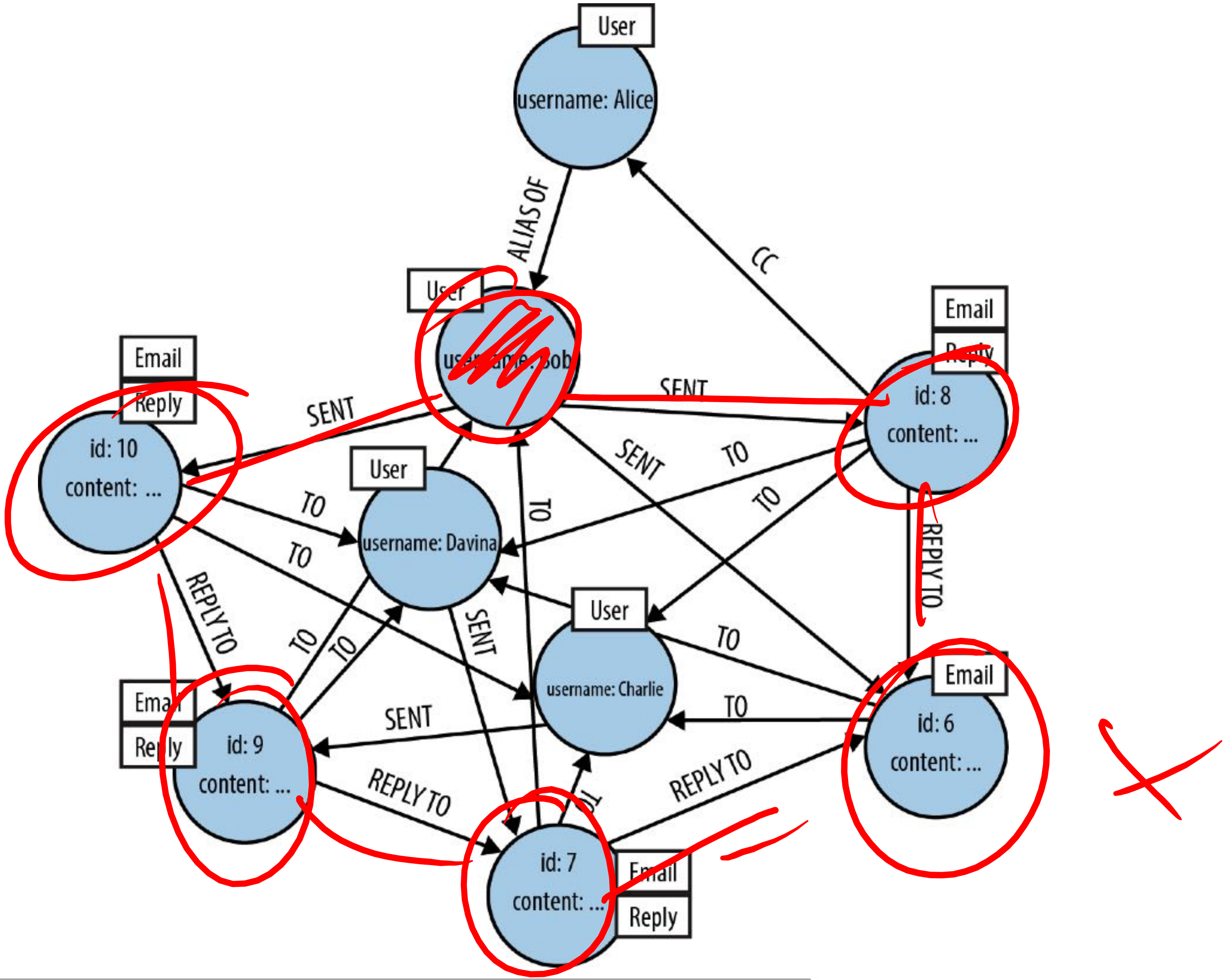



```
MATCH p = (email:Email {id:'6'})
      <-[:REPLY_TO*1..4]-(:Reply)<-[:SENT]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
```

Another Example

Path assignment

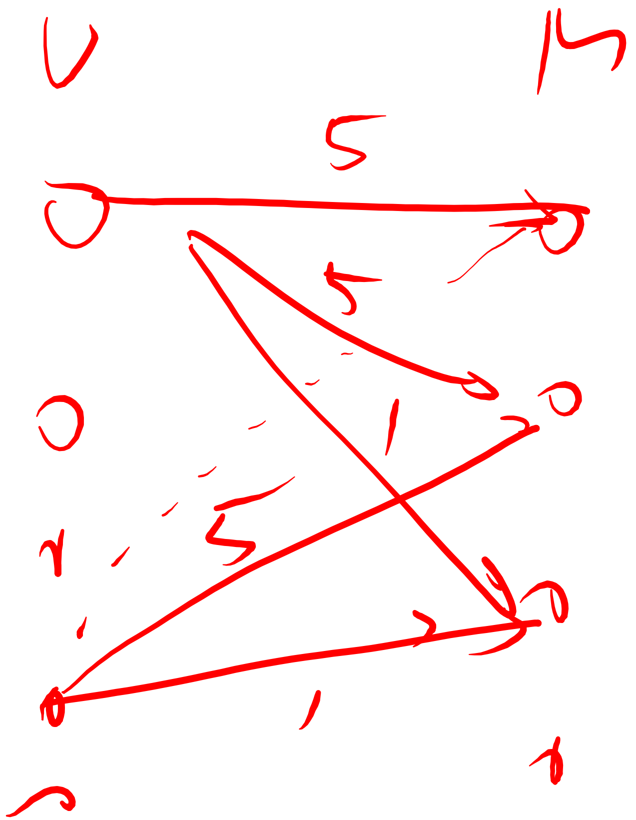
replier	depth
Davina	1
Bob	1
Charlie	2
Bob	3




```
MATCH p = (email:Email {id:'6'})
      <-[:REPLY_TO*1..4]-(:Reply)<-[:SENT]-(replier)
RETURN replier.username AS replier, length(p) - 1 AS depth
ORDER BY depth
```

When to use it

- Use it:
  - Connected data, e.g. social graphs, employees where they worked
  - ~~Location-based services~~
  - Recommendation engines
- Don't use it:
  - Change properties on many entities



Outline: Alternative data models

- 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

Concluding Remarks on Common NoSQL

- Aim to avoid join & ACID overhead
  - Joined within, correctness compromised for quick answers; believe in best effort
- Avoid the idea of a schema
- Query languages are more imperative
  - And less declarative
  - Developer better knows what's going on; less reliance on smart optimization plans
  - More responsibility on developers
- No standard well studied languages (yet)