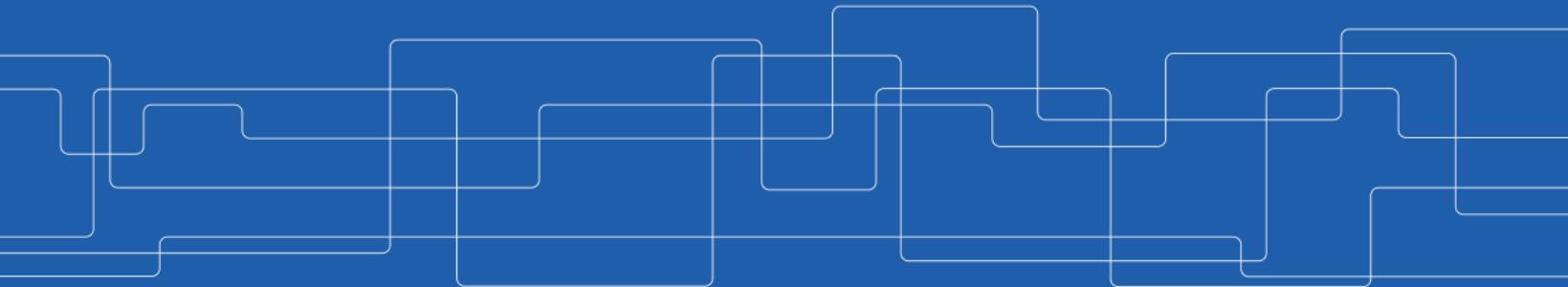




NoSQL Databases

Amir H. Payberah
payberah@kth.se
2020-09-01





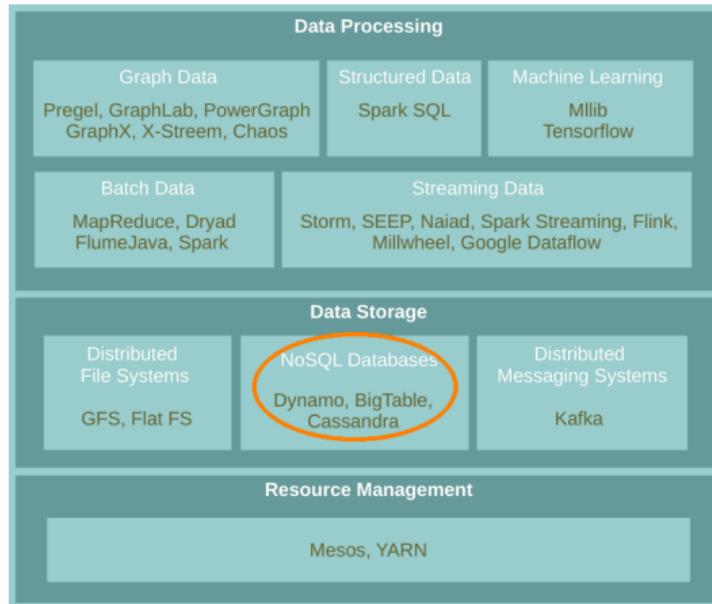
The Course Web Page

<https://id2221kth.github.io>

<https://tinyurl.com/y4qph82u>



Where Are We?



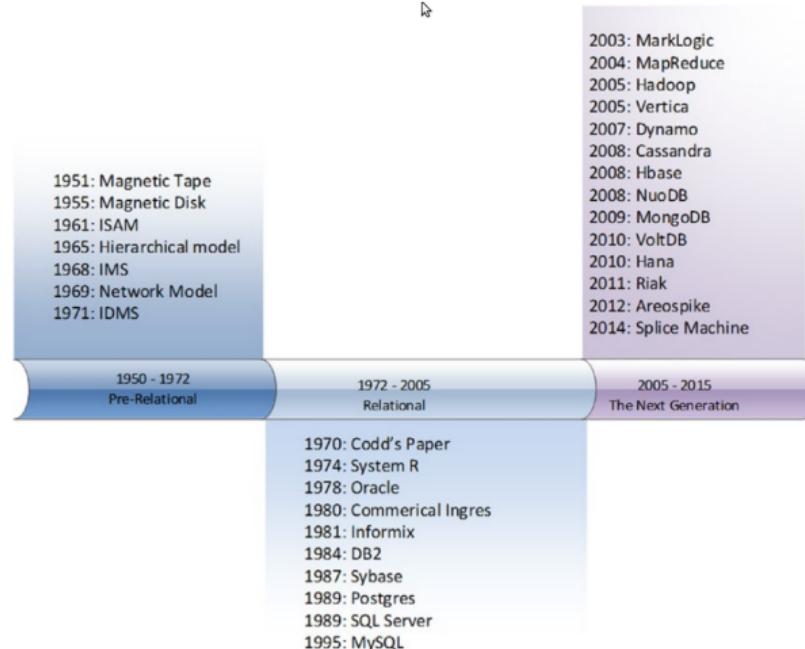
Database and Database Management System

- ▶ **Database:** an **organized** collection of **data**.
- ▶ **Database Management System (DBMS):** a **software** to **capture** and **analyze** data.





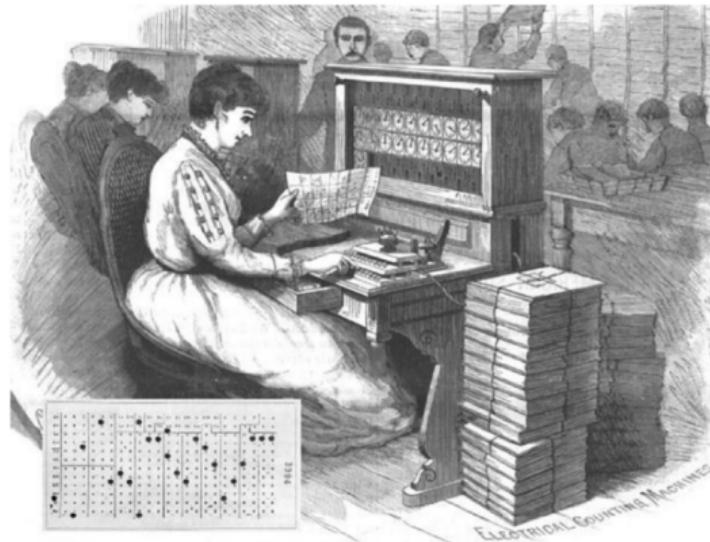
Three Database Revolutions



[Guy Harrison, Next Generation Databases: NoSQLLand Big Data, 2015]

Early Database Systems

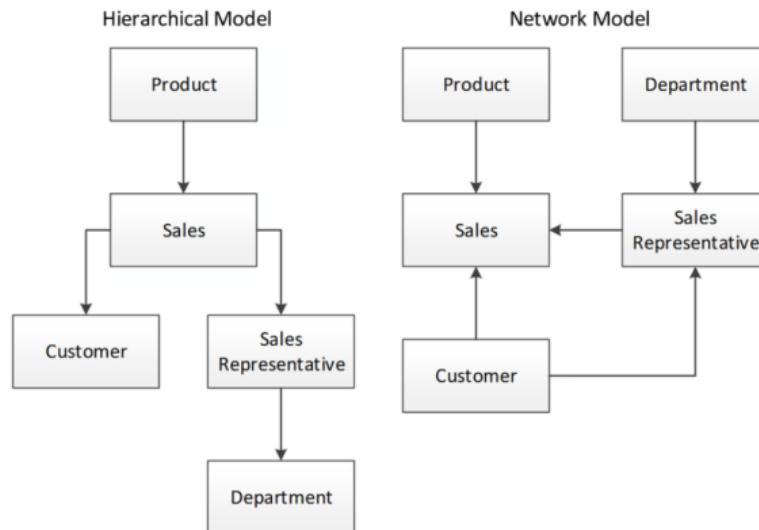
- ▶ There were databases but **no Database Management Systems (DBMS)**.



[Guy Harrison, Next Generation Databases: NoSQLand Big Data, 2015]

The First Database Revolution

- ▶ Navigational data model: hierarchical model (IMS) and network model (CODASYL).
- ▶ Disk-aware

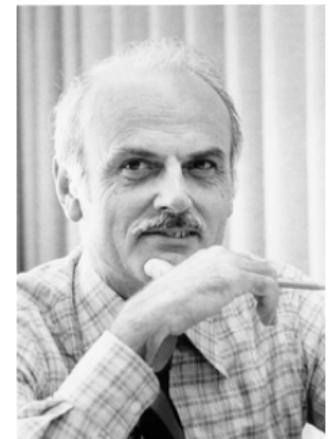


[Guy Harrison, Next Generation Databases: NoSQL and Big Data, 2015]



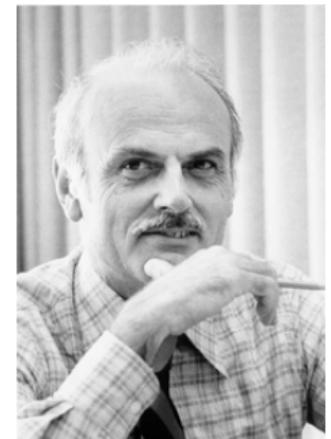
The Second Database Revolution

- ▶ Relational data model: Edgar F. Codd paper
 - Logical data is disconnected from physical information storage



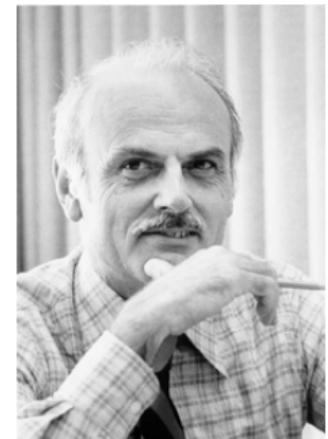
The Second Database Revolution

- ▶ Relational data model: Edgar F. Codd paper
 - Logical data is disconnected from physical information storage
- ▶ ACID transactions
 - Atomic, Consistent, Isolated, Durable



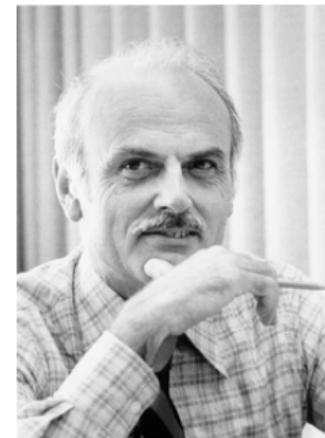
The Second Database Revolution

- ▶ Relational data model: Edgar F. Codd paper
 - Logical data is disconnected from physical information storage
- ▶ ACID transactions
 - Atomic, Consistent, Isolated, Durable
- ▶ SQL language



The Second Database Revolution

- ▶ Relational data model: Edgar F. Codd paper
 - Logical data is disconnected from physical information storage
- ▶ ACID transactions
 - Atomic, Consistent, Isolated, Durable
- ▶ SQL language
- ▶ Object databases
 - Information is represented in the form of objects





ACID Properties

► Atomicity

- All included statements in a transaction are either **executed** or the **whole** transaction is **aborted** without affecting the database.



ACID Properties

► Atomicity

- All included statements in a transaction are either **executed** or the **whole** transaction is **aborted** without affecting the database.

► Consistency

- A database is in a **consistent** state before and after a transaction.



ACID Properties

► Atomicity

- All included statements in a transaction are either **executed** or the **whole** transaction is **aborted** without affecting the database.

► Consistency

- A database is in a **consistent** state before and after a transaction.

► Isolation

- Transactions can not see **uncommitted changes** in the database.



ACID Properties

► Atomicity

- All included statements in a transaction are either **executed** or the **whole** transaction is **aborted** without affecting the database.

► Consistency

- A database is in a **consistent** state before and after a transaction.

► Isolation

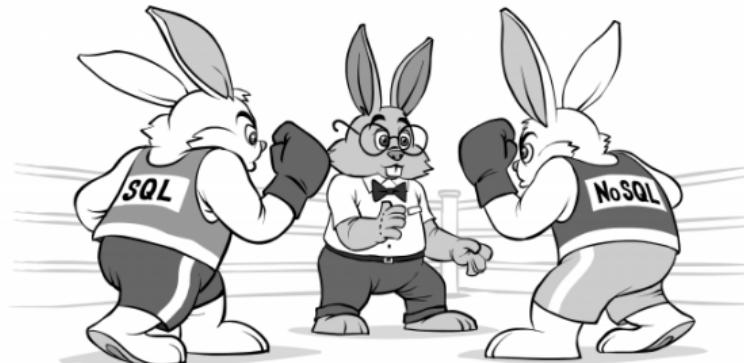
- Transactions can not see **uncommitted changes** in the database.

► Durability

- Changes are written to a **disk** before a database commits a transaction so that committed data cannot be lost through a power **failure**.

The Third Database Revolution

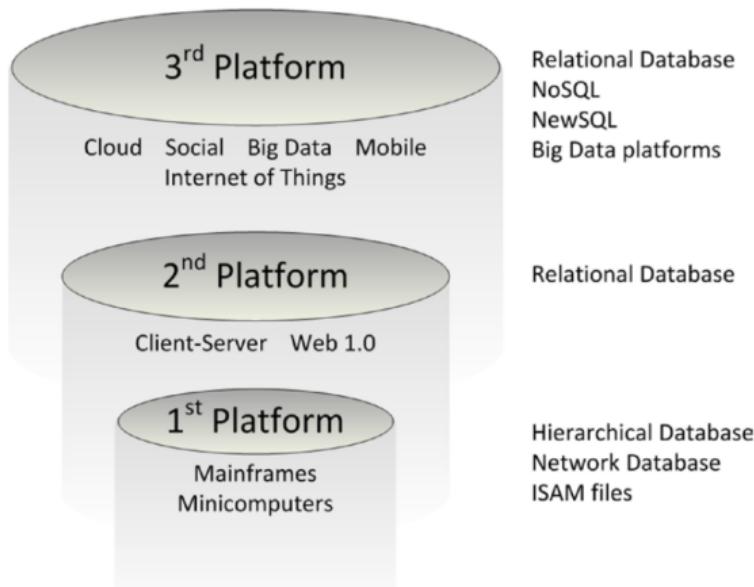
- ▶ NoSQL databases: BASE instead of ACID.
- ▶ NewSQL databases: scalable performance of NoSQL + ACID.



[<http://ithare.com/nosql-vs-sql-for-mogs>]



Three Waves of Database Technology



[Guy Harrison, Next Generation Databases: NoSQLAnd Big Data, 2015]



SQL vs. NoSQL Databases

Relational SQL Databases

- ▶ The **dominant** technology for storing **structured** data in web and business applications.
- ▶ **SQL** is good
 - Rich language and toolset
 - Easy to use and integrate
 - Many **vendors**
- ▶ They promise: **ACID**





SQL Databases Challenges

- ▶ Web-based applications caused spikes.
 - Internet-scale data size
 - High read-write rates
 - Frequent schema changes



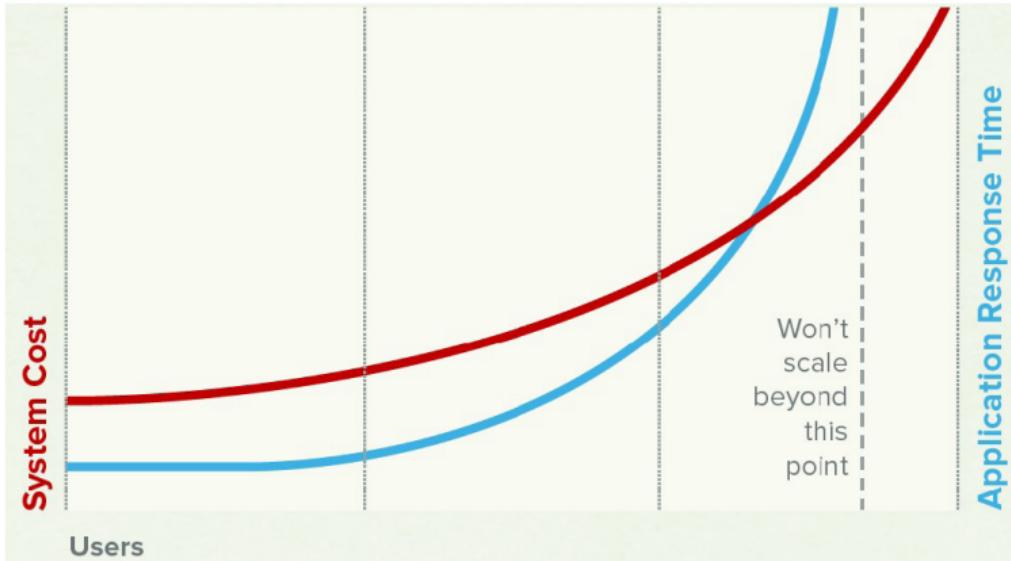


SQL Databases Challenges

- ▶ Web-based applications caused spikes.
 - Internet-scale data size
 - High read-write rates
 - Frequent schema changes
- ▶ RDBMS were **not** designed to be **distributed**.



Scaling SQL Databases is Expensive and Inefficient



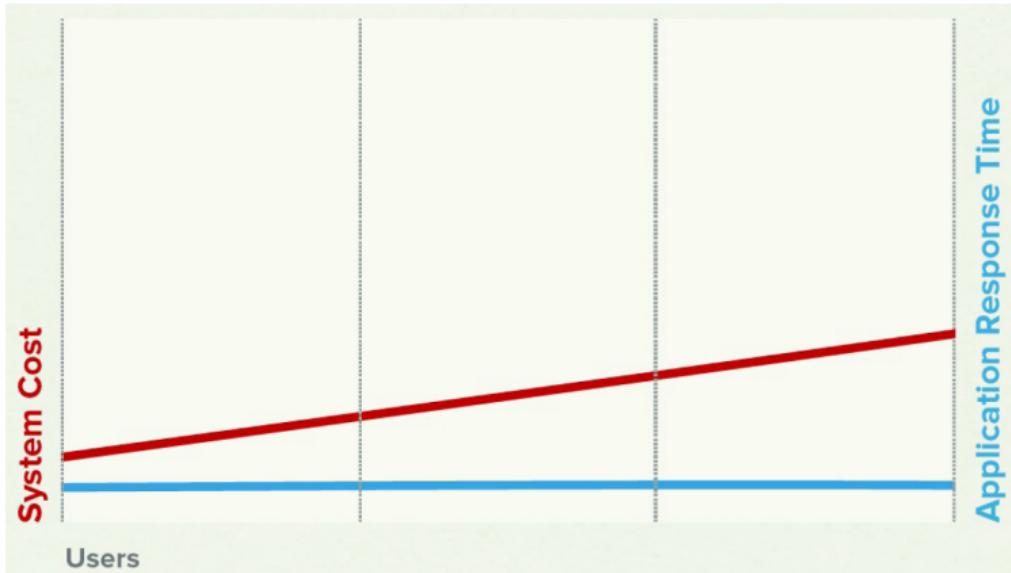
[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf>]

NoSQL

- ▶ Avoids:
 - Overhead of ACID properties
 - Complexity of SQL query
- ▶ Provides:
 - Scalability
 - Easy and frequent changes to DB
 - Large data volumes

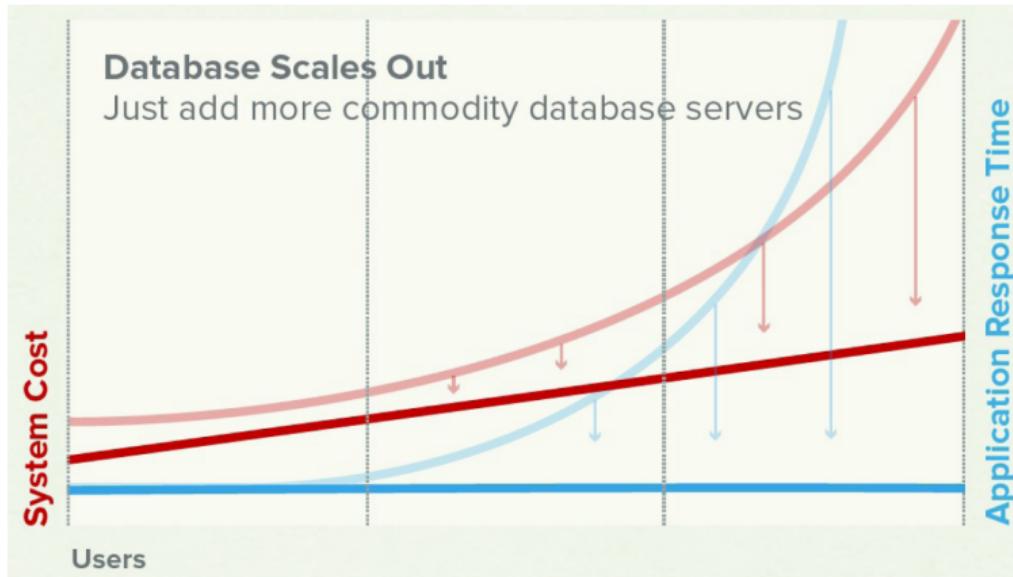


NoSQL Cost and Performance



[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf>]

SQL vs. NoSQL



[<http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQLWhitepaper.pdf>]



ACID vs. BASE

Availability

- ▶ Replicating data to improve the availability of data.
- ▶ Data replication
 - Storing data in more than one site or node



Consistency

► Strong consistency

- After an update completes, any subsequent access will return the updated value.



Consistency

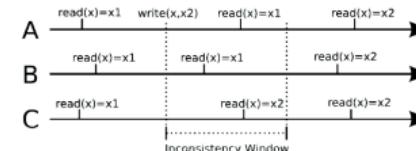
► Strong consistency

- After an update completes, any subsequent access will return the **updated value**.



► Eventual consistency

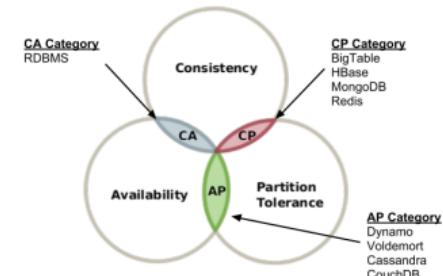
- Does **not guarantee** that subsequent accesses will return the **updated value**.
- **Inconsistency window**.
- If no new updates are made to the object, **eventually** all accesses will return the last updated value.



CAP Theorem

► Consistency

- Consistent state of data after the execution of an operation.



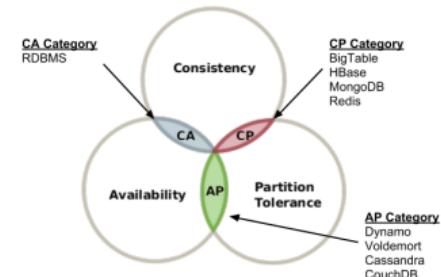
CAP Theorem

► Consistency

- Consistent state of data after the execution of an operation.

► Availability

- Clients can always read and write data.



CAP Theorem

► Consistency

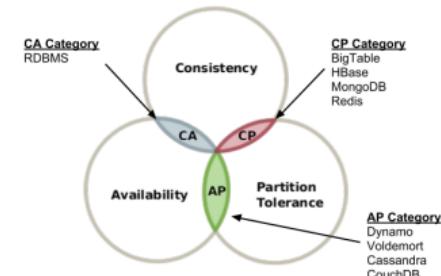
- Consistent state of data after the execution of an operation.

► Availability

- Clients can always read and write data.

► Partition Tolerance

- Continue the operation in the presence of network partitions.



CAP Theorem

► Consistency

- Consistent state of data after the execution of an operation.

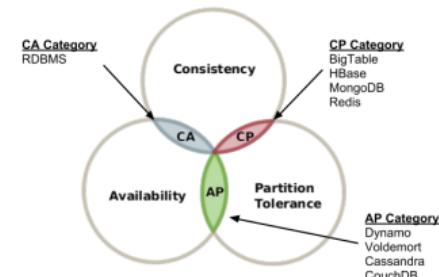
► Availability

- Clients can always read and write data.

► Partition Tolerance

- Continue the operation in the presence of network partitions.

► You can choose only two!





Consistency vs. Availability

- ▶ The large-scale applications have to be **reliable**: availability, consistency, partition tolerance
- ▶ Not possible to achieve with **ACID** properties.
- ▶ The **BASE** approach forfeits the ACID properties of **consistency** and **isolation** in favor of **availability** and performance.



BASE Properties

► Basic Availability

- Possibilities of faults but not a fault of the whole system.



BASE Properties

► Basic Availability

- Possibilities of faults but not a fault of the whole system.

► Soft-state

- Copies of a data item may be inconsistent



BASE Properties

► Basic Availability

- Possibilities of faults but not a fault of the whole system.

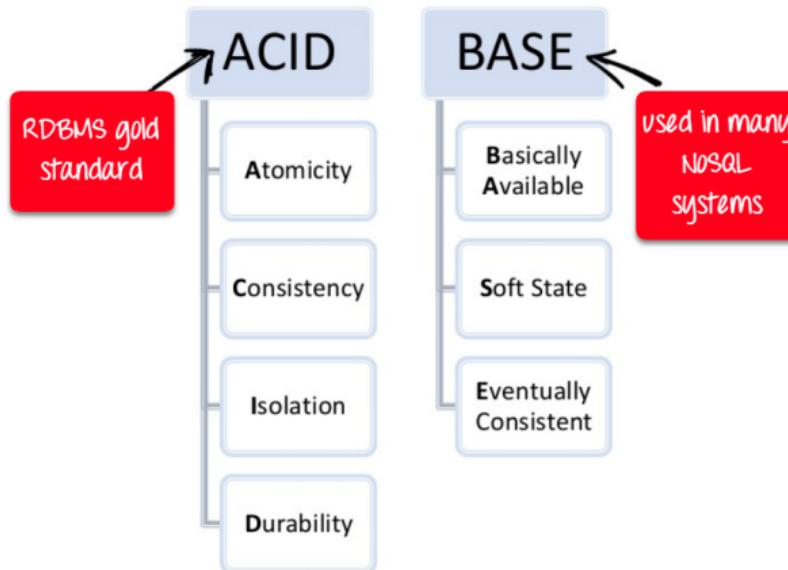
► Soft-state

- Copies of a data item may be inconsistent

► Eventually consistent

- Copies becomes consistent at some later time if there are no more updates to that data item

ACID vs. BASE

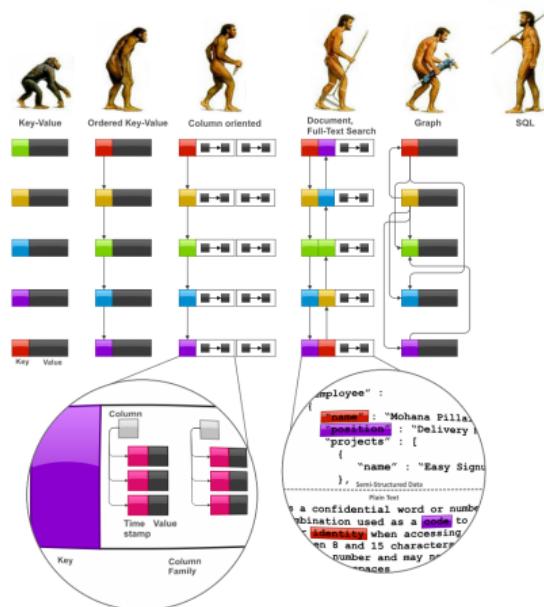


[<https://www.guru99.com/sql-vs-nosql.html>]



NoSQL Data Models

NoSQL Data Models



[<http://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques>]



Key-Value Data Model

- ▶ Collection of **key/value** pairs.
- ▶ **Ordered Key-Value:** processing over **key ranges**.
- ▶ **Dynamo, Scalaris, Voldemort, Riak, ...**

Column-Oriented Data Model

- ▶ Similar to a **key/value** store, but the **value** can have multiple **attributes** (Columns).
- ▶ **Column**: a set of data **values** of a particular **type**.
- ▶ Store and process data by **column** instead of **row**.
- ▶ **BigTable, Hbase, Cassandra, ...**





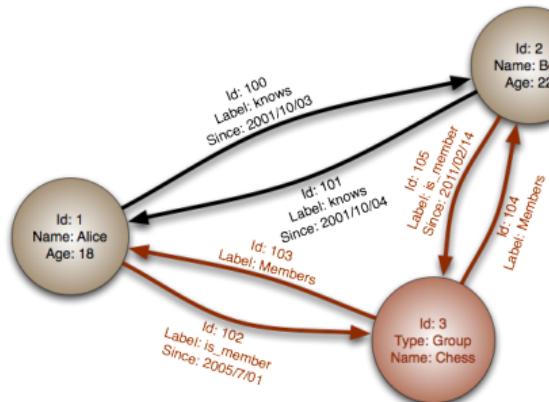
Document Data Model

- ▶ Similar to a **column-oriented** store, but values can have **complex documents**.
- ▶ Flexible schema (XML, YAML, JSON, and BSON).
- ▶ **CouchDB, MongoDB, ...**

```
{  
    FirstName: "Bob",  
    Address: "5 Oak St.",  
    Hobby: "sailing"  
}  
  
{  
    FirstName: "Jonathan",  
    Address: "15 Wanamassa Point Road",  
    Children: [  
        {Name: "Michael", Age: 10},  
        {Name: "Jennifer", Age: 8},  
    ]  
}
```

Graph Data Model

- ▶ Uses **graph** structures with **nodes**, **edges**, and **properties** to represent and store data.
- ▶ Neo4J, InfoGrid, ...



[http://en.wikipedia.org/wiki/Graph_database]



BigTable

BigTable

- ▶ Lots of (semi-)structured data at Google.
 - URLs, per-user data, geographical locations, ...
- ▶ Distributed multi-level map
- ▶ CAP: strong consistency and partition tolerance





Data Model



Data Model (1/6)

- ▶ Column-Oriented data model
- ▶ Similar to a key/value store, but the value can have multiple attributes (Columns).

Data Model (1/6)

- ▶ **Column-Oriented** data model
- ▶ Similar to a **key/value** store, but the **value** can have multiple **attributes** (Columns).
- ▶ **Column**: a set of data **values** of a particular **type**.
- ▶ Store and process data by **column** instead of **row**.





Data Model (2/6)

- ▶ Table
- ▶ Distributed multi-dimensional sparse map



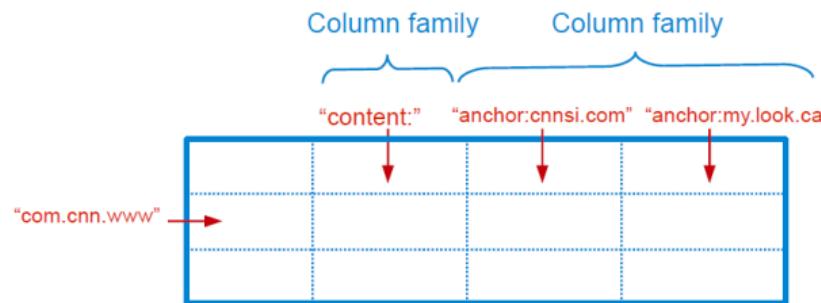
Data Model (3/6)

- ▶ Rows
- ▶ Every read or write in a **row** is **atomic**.
- ▶ Rows sorted in **lexicographical** order.



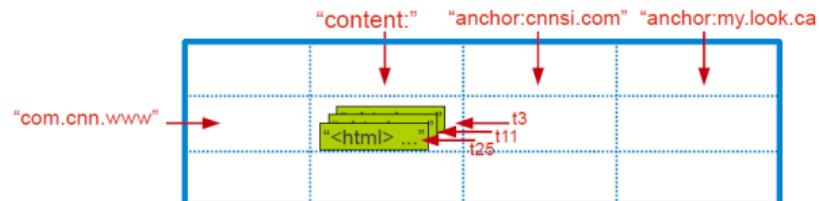
Data Model (4/6)

- ▶ Column
- ▶ The **basic unit** of data access.
- ▶ Column families: group of (the same type) column keys.
- ▶ Column key naming: **family:qualifier**



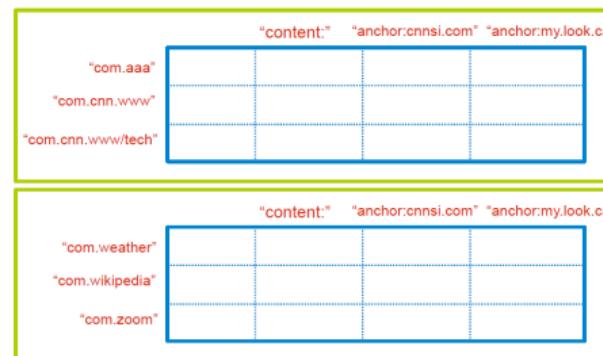
Data Model (5/6)

- ▶ **Timestamp**
- ▶ Each column value may contain multiple **versions**.



Data Model (6/6)

- ▶ **Tablet**: contiguous ranges of rows stored together.
- ▶ Tablets are **split** by the system when they become too large.
- ▶ Each **tablet** is served by exactly one **tablet server**.

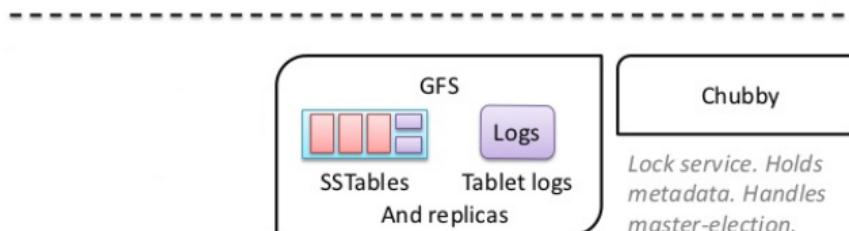
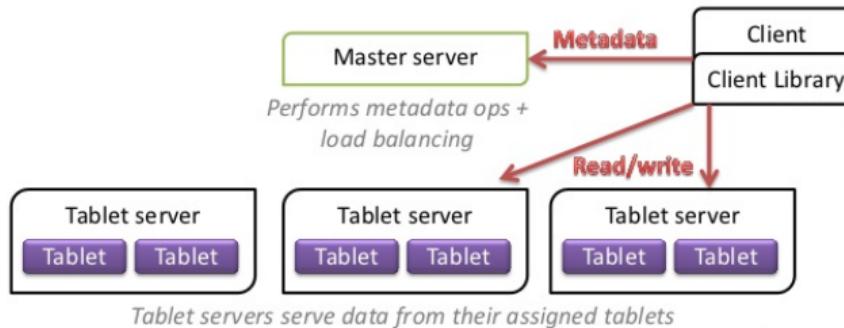




System Architecture



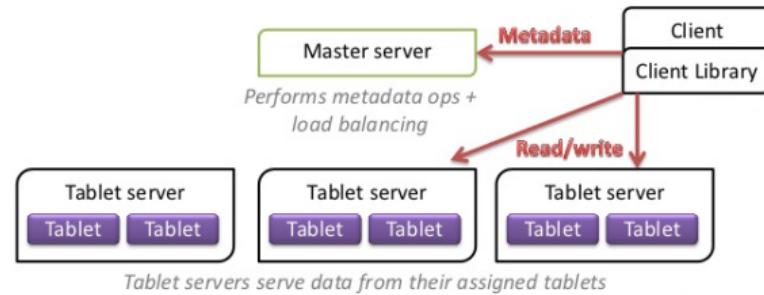
BigTable System Structure



[<https://www.slideshare.net/GrishaWeintraub/cap-28353551>]

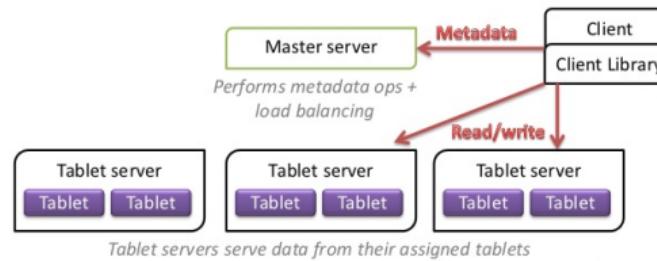
Main Components

- ▶ Master
- ▶ Tablet server
- ▶ Client library



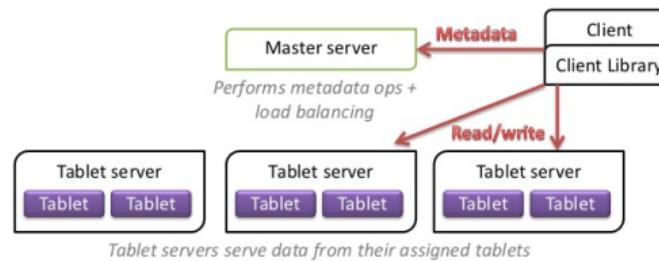
Master

- ▶ Assigns tablets to tablet server.



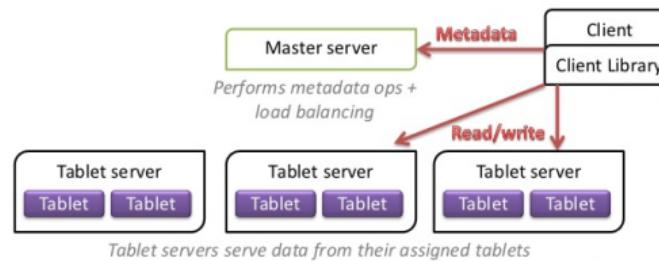
Master

- ▶ Assigns tablets to tablet server.
- ▶ Balances tablet server load.



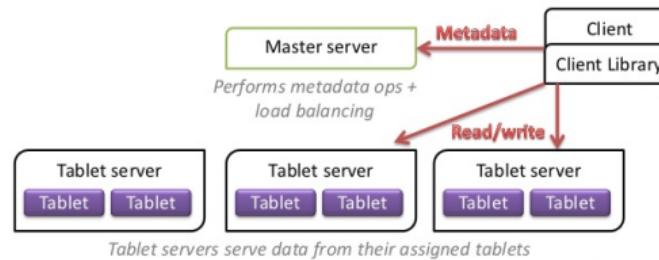
Master

- ▶ Assigns tablets to tablet server.
- ▶ Balances tablet server load.
- ▶ Garbage collection of unneeded files in GFS.



Master

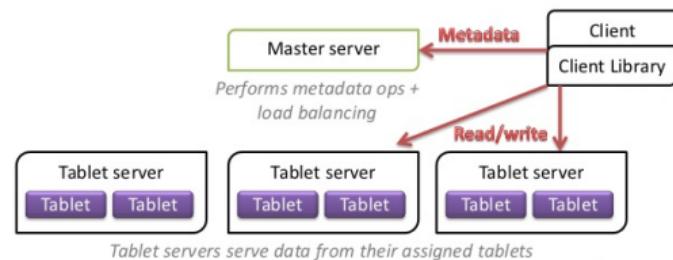
- ▶ Assigns tablets to tablet server.
- ▶ Balances tablet server load.
- ▶ Garbage collection of unneeded files in GFS.
- ▶ Handles schema changes, e.g., table and column family creations





Tablet Server

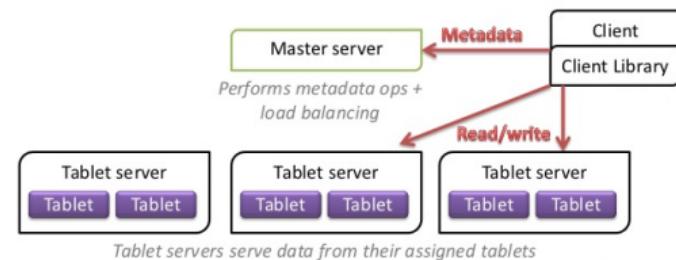
- ▶ Can be added or removed dynamically.





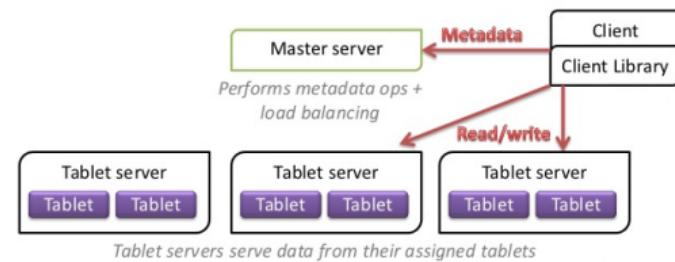
Tablet Server

- ▶ Can be **added** or **removed** **dynamically**.
- ▶ Each **manages** a set of tablets (typically 10-1000 tablets/server).



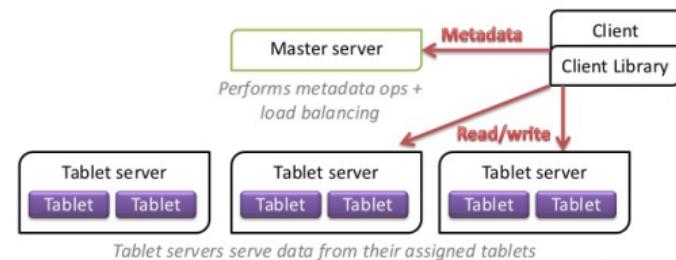
Tablet Server

- ▶ Can be **added** or **removed** **dynamically**.
- ▶ Each **manages** a set of tablets (typically 10-1000 tablets/server).
- ▶ Handles **read/write** requests to tablets.



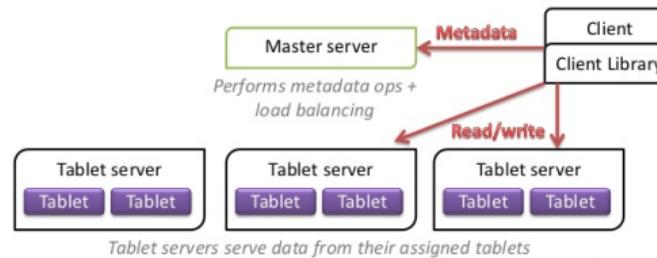
Tablet Server

- ▶ Can be **added** or **removed** **dynamically**.
- ▶ Each **manages** a set of tablets (typically 10-1000 tablets/server).
- ▶ Handles **read/write** requests to tablets.
- ▶ **Splits tablets** when too large.



Client Library

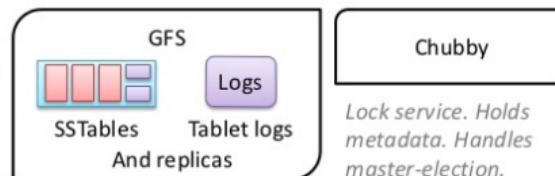
- ▶ **Library** that is linked into every client.
- ▶ Client **data** does not move though the **master**.
- ▶ Clients communicate **directly** with **tablet servers** for **reads/writes**.



Building Blocks

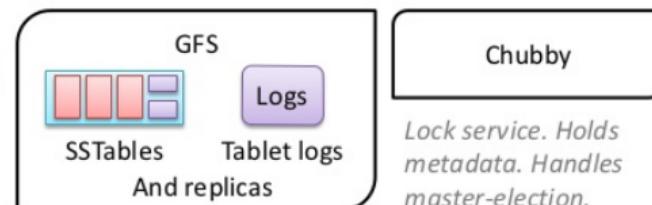
- ▶ The building blocks for the BigTable are:

- Google File System (GFS)
- Chubby
- SSTable



Google File System (GFS)

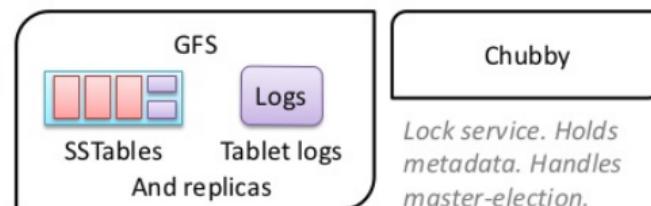
- ▶ Large-scale **distributed file system**.
- ▶ Store **log** and **data** files.





Chubby Lock Service

- ▶ Ensure there is only **one active master**.
- ▶ Store **bootstrap location** of BigTable data.
- ▶ **Discover** tablet servers.
- ▶ Store BigTable **schema** information and **access control lists**.





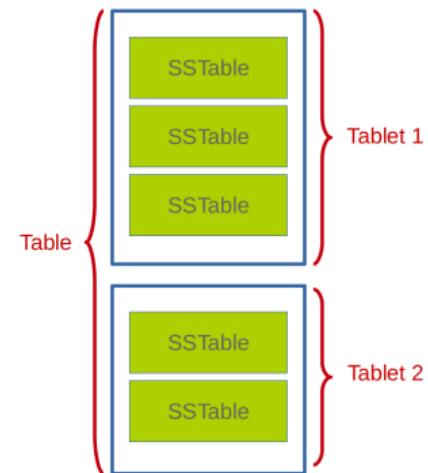
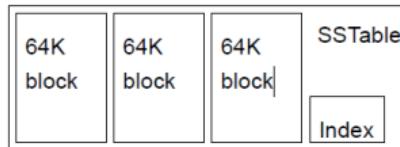
SSTable

- ▶ **SSTable** file format used internally to store BigTable data.



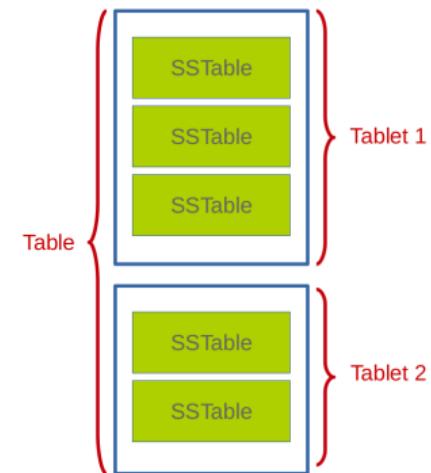
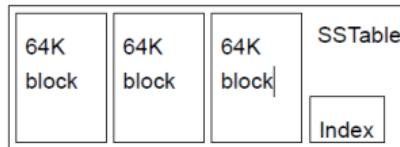
SSTable

- ▶ **SSTable** file format used internally to store BigTable data.
- ▶ Chunks of **data** plus a **block index**.



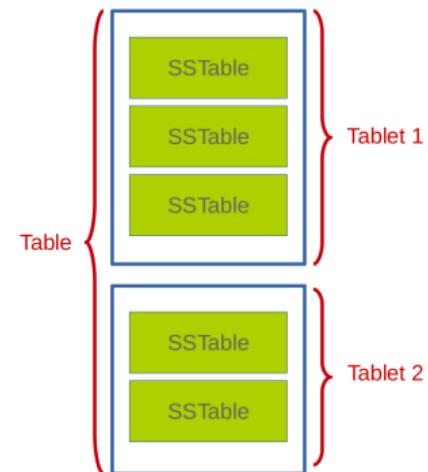
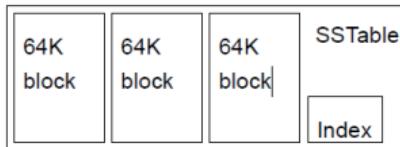
SSTable

- ▶ **SSTable** file format used internally to store BigTable data.
- ▶ Chunks of **data** plus a **block index**.
- ▶ **Immutable**, sorted file of **key-value** pairs.



SSTable

- ▶ **SSTable** file format used internally to store BigTable data.
- ▶ Chunks of **data** plus a **block index**.
- ▶ **Immutable**, sorted file of **key-value** pairs.
- ▶ Each SSTable is stored in a **GFS** file.





Tablet Serving



Master Startup

- ▶ The **master** executes the following steps at **startup**:



Master Startup

- ▶ The **master** executes the following steps at **startup**:
 - Grabs a unique master **lock in Chubby**, which prevents **concurrent master instantiations**.



Master Startup

- ▶ The **master** executes the following steps at **startup**:
 - Grabs a unique master **lock in Chubby**, which prevents **concurrent master instantiations**.
 - **Scans the servers directory** in Chubby to find the live servers.



Master Startup

- ▶ The **master** executes the following steps at **startup**:
 - Grabs a unique master **lock in Chubby**, which prevents **concurrent master instantiations**.
 - **Scans the servers directory** in Chubby to find the live servers.
 - **Communicates** with every live tablet server to discover what tablets are already assigned to each server.



Master Startup

- ▶ The **master** executes the following steps at **startup**:
 - Grabs a unique master **lock in Chubby**, which prevents **concurrent master** instantiations.
 - **Scans the servers directory** in Chubby to find the live servers.
 - **Communicates** with every live tablet server to discover what tablets are already assigned to each server.
 - **Scans the METADATA** table to learn the set of tablets.



Tablet Assignment

- ▶ 1 tablet → 1 tablet server.



Tablet Assignment

- ▶ 1 tablet → 1 tablet server.
- ▶ Master uses **Chubby** to keep tracks of **live tablet servers** and **unassigned tablets**.
 - When a **tablet server starts**, it creates and acquires an **exclusive lock** in Chubby.



Tablet Assignment

- ▶ 1 tablet → 1 tablet server.
- ▶ Master uses **Chubby** to keep tracks of **live tablet servers** and **unassigned tablets**.
 - When a **tablet server starts**, it creates and acquires an **exclusive lock** in Chubby.
- ▶ Master detects the **status of the lock** of each tablet server by checking periodically.

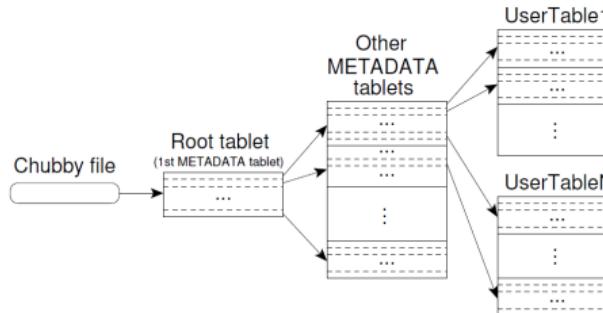


Tablet Assignment

- ▶ 1 tablet → 1 tablet server.
- ▶ Master uses **Chubby** to keep tracks of **live tablet servers** and **unassigned tablets**.
 - When a **tablet server starts**, it creates and acquires an **exclusive lock** in Chubby.
- ▶ Master detects the **status of the lock of each tablet server** by checking periodically.
- ▶ Master is responsible for finding when tablet server is **no longer serving its tablets** and **reassigning** those tablets as soon as possible.

Finding a Tablet

- ▶ Three-level hierarchy.
- ▶ The first level is a **file stored in Chubby** that contains the location of the root tablet.
- ▶ Root tablet contains **location of all tablets** in a special **METADATA** table.
- ▶ **METADATA** table contains location of each **tablet** under a row.
- ▶ The client library **caches tablet locations**.



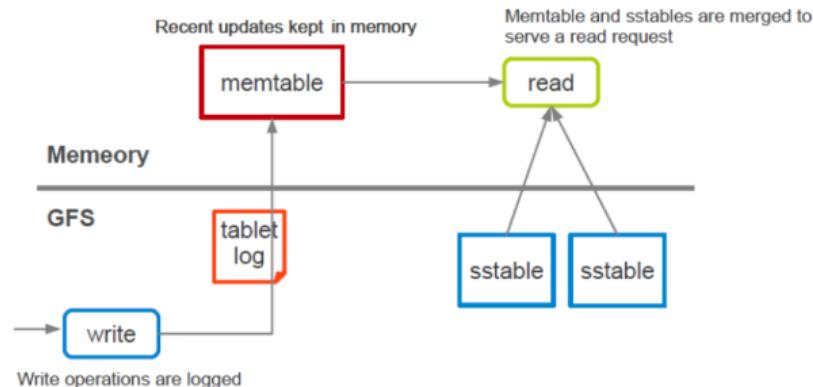


Loading Tablets

- ▶ To load a tablet, a **tablet server** does the following:
- ▶ Finds **location of tablet** through its **METADATA**.
 - Metadata for a tablet includes **list of SSTables** and set of redo points.
- ▶ Read **SSTables index blocks** into memory.
- ▶ Read the **commit log** since the redo point and reconstructs the **memtable**.

Tablet Serving (1/2)

- ▶ Updates committed to a **commit log**.
- ▶ Recently committed updates are stored in **memory - memtable**
- ▶ Older updates are stored in a sequence of **SSTables**.





Tablet Serving (2/2)

- ▶ Strong consistency

- Only one tablet server is responsible for a given piece of data.
- Replication is handled on the GFS layer.



Tablet Serving (2/2)

- ▶ Strong consistency
 - Only one tablet server is responsible for a given piece of data.
 - Replication is handled on the GFS layer.
- ▶ Trade-off with availability
 - If a tablet server fails, its portion of data is temporarily unavailable until a new server is assigned.



BigTable vs. HBase

BigTable	HBase
GFS	HDFS
Tablet Server	Region Server
SSTable	StoreFile
Memtable	MemStore
Chubby	ZooKeeper



HBase Example

```
# Create the table "test", with the column family "cf"
create 'test', 'cf'
```



HBase Example

```
# Create the table "test", with the column family "cf"
create 'test', 'cf'
```

```
# Use describe to get the description of the "test" table
describe 'test'
```



HBase Example

```
# Create the table "test", with the column family "cf"
create 'test', 'cf'
```

```
# Use describe to get the description of the "test" table
describe 'test'
```

```
# Put data in the "test" table
put 'test', 'row1', 'cf:a', 'value1'
put 'test', 'row2', 'cf:b', 'value2'
put 'test', 'row3', 'cf:c', 'value3'
```



HBase Example

```
# Create the table "test", with the column family "cf"
create 'test', 'cf'
```

```
# Use describe to get the description of the "test" table
describe 'test'
```

```
# Put data in the "test" table
put 'test', 'row1', 'cf:a', 'value1'
put 'test', 'row2', 'cf:b', 'value2'
put 'test', 'row3', 'cf:c', 'value3'
```

```
# Scan the table for all data at once
scan 'test'
```



HBase Example

```
# Create the table "test", with the column family "cf"
create 'test', 'cf'
```

```
# Use describe to get the description of the "test" table
describe 'test'
```

```
# Put data in the "test" table
put 'test', 'row1', 'cf:a', 'value1'
put 'test', 'row2', 'cf:b', 'value2'
put 'test', 'row3', 'cf:c', 'value3'
```

```
# Scan the table for all data at once
scan 'test'
```

```
# To get a single row of data at a time, use the get command
get 'test', 'row1'
```



Cassandra

Cassandra

- ▶ A **column-oriented** database
- ▶ It was created for **Facebook** and was later **open sourced**
- ▶ **CAP**: availability and **partition tolerance**





Borrowed From BigTable

- ▶ Data model: **column oriented**
 - **Keyspaces** (similar to the schema in a relational database), **tables**, and **columns**.



Borrowed From BigTable

- ▶ Data model: **column oriented**
 - **Keyspaces** (similar to the schema in a relational database), **tables**, and **columns**.
- ▶ **SSTable** disk storage
 - Append-only commit log
 - Memtable (buffering and sorting)
 - Immutable sstable files

Data Partitioning (1/2)

- ▶ Key/value, where values are stored as objects.
- ▶ If size of data exceeds the capacity of a single machine: partitioning



Data Partitioning (1/2)

- ▶ Key/value, where values are stored as **objects**.
- ▶ If size of data **exceeds the capacity** of a single machine: **partitioning**
- ▶ **Consistent hashing** for partitioning.





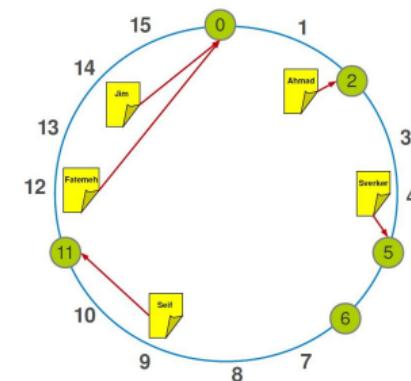
Data Partitioning (2/2)

- ▶ Consistent hashing.
- ▶ Hash both **data** and **node ids** using the **same hash function** in a **same id space**.
- ▶ `partition = hash(d) mod n`, **d**: data, **n**: the size of the id space

Data Partitioning (2/2)

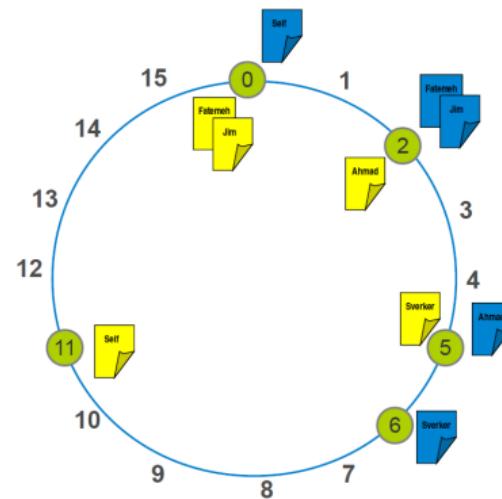
- ▶ Consistent hashing.
- ▶ Hash both data and node ids using the same hash function in a same id space.
- ▶ `partition = hash(d) mod n`, d: data, n: the size of the id space

```
id space = [0, 15], n = 16
hash("Fatemeh") = 12
hash("Ahmad") = 2
hash("Seif") = 9
hash("Jim") = 14
hash("Sverker") = 4
```



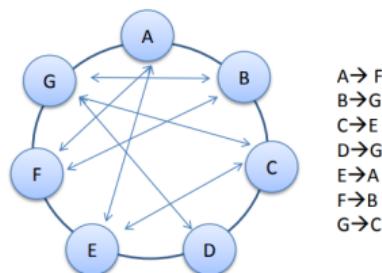
Replication

- To achieve high availability and durability, data should be replicated on multiple nodes.



Adding and Removing Nodes

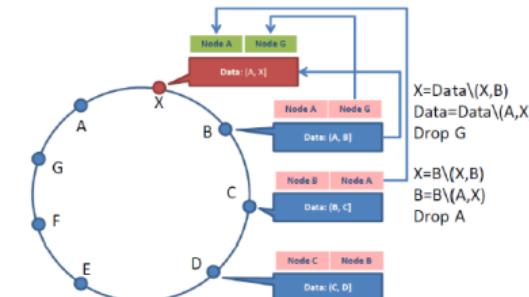
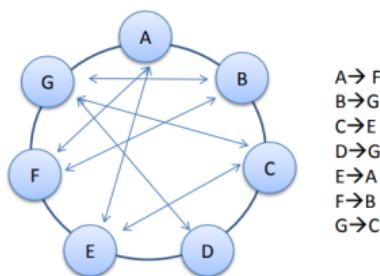
- ▶ Gossip-based mechanism: periodically, each node contacts another randomly selected node.



$A \rightarrow F$
 $B \rightarrow G$
 $C \rightarrow E$
 $D \rightarrow G$
 $E \rightarrow A$
 $F \rightarrow B$
 $G \rightarrow C$

Adding and Removing Nodes

- ▶ Gossip-based mechanism: periodically, each node contacts another randomly selected node.





Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```



Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
# Print the list of keyspaces
describe keyspaces;
```



Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
# Print the list of keyspaces
describe keyspaces;
```

```
# Navigate to the "test" keyspace
use test
```



Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
# Print the list of keyspaces
describe keyspaces;
```

```
# Navigate to the "test" keyspace
use test
```

```
# Create the "words" table in the "test" keyspace
create table words (word text, count int, primary key (word));
```



Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
# Print the list of keyspaces
describe keyspaces;
```

```
# Navigate to the "test" keyspace
use test
```

```
# Create the "words" table in the "test" keyspace
create table words (word text, count int, primary key (word));
```

```
# Insert a row
insert into words(word, count) values('hello', 5);
```



Cassandra Example

```
# Create a keyspace called "test"
create keyspace test
with replication = {'class': 'SimpleStrategy', 'replication_factor': 1};
```

```
# Print the list of keyspaces
describe keyspaces;
```

```
# Navigate to the "test" keyspace
use test
```

```
# Create the "words" table in the "test" keyspace
create table words (word text, count int, primary key (word));
```

```
# Insert a row
insert into words(word, count) values('hello', 5);
```

```
# Look at the table
select * from words;
```



Neo4j



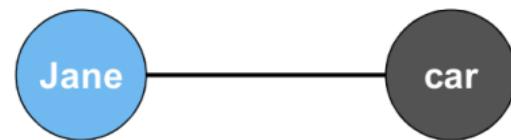
- ▶ A graph database
- ▶ The relationships between data is **equally important** as the **data itself**
- ▶ **Cypher**: a declarative query language similar to SQL, but **optimized for graphs**
- ▶ **CAP**: **strong consistency** and **availability**



Data Model (1/4)

► Node (Vertex)

- The **main data element** from which graphs are constructed.
- A waypoint along a **traversal route**



Data Model (2/4)

- ▶ Relationship (Edge)
- ▶ May contain
 - Direction
 - Metadata, e.g., weight or relationship type



Data Model (3/4)

► Label

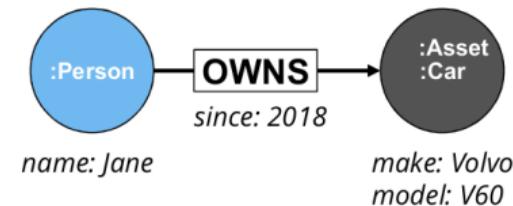
- Define **node category** (optional)
- Can have **more than one**



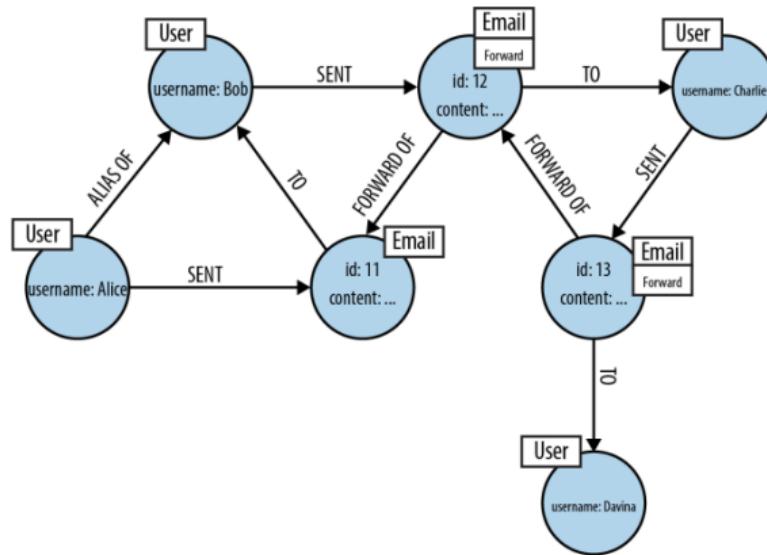
Data Model (4/4)

► Properties

- Enrich a node or relationship

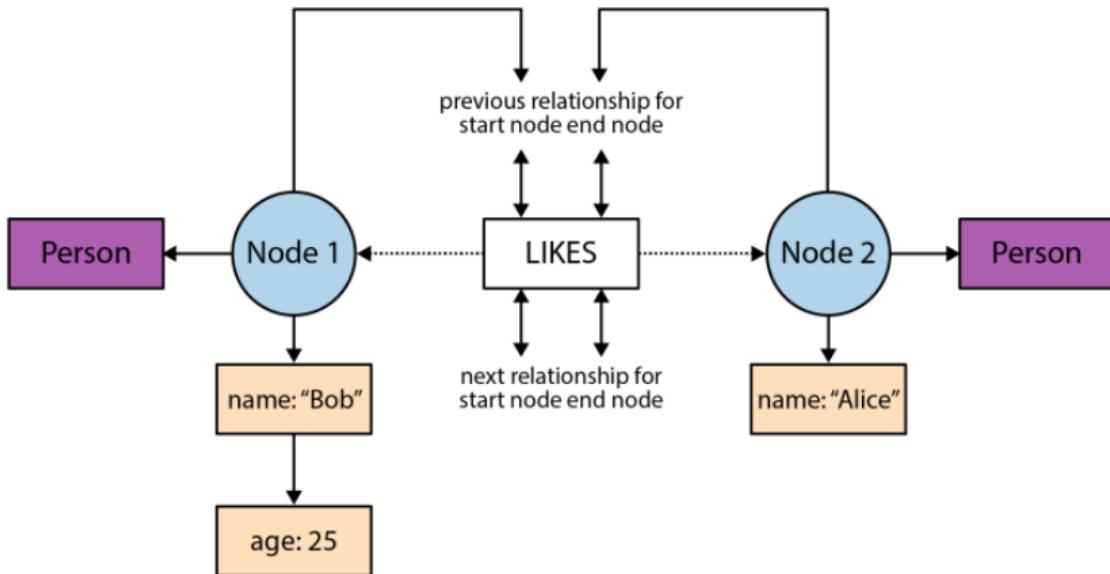


Example



[Ian Robinson et al., Graph Databases, 2015]

How a Graph is Stored in Neo4j? (1/2)

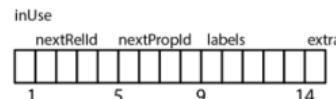


[Ian Robinson et al., Graph Databases, 2015]

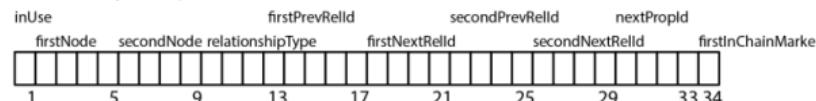
How a Graph is Stored in Neo4j? (2/2)

- ▶ Neo4j stores graph data in a number of different **store files**.

Node (15 bytes)



Relationship (34 bytes)

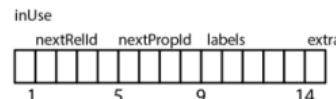


[Ian Robinson et al., Graph Databases, 2015]

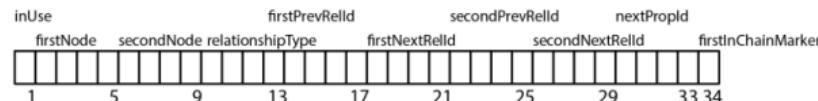
How a Graph is Stored in Neo4j? (2/2)

- ▶ Neo4j stores graph data in a number of different **store files**.
- ▶ Each **store file** contains the data for a **specific part** of the graph.
 - Separate stores for **nodes**, **relationships**, **labels**, and **properties**.

Node (15 bytes)



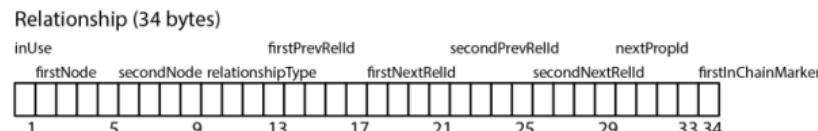
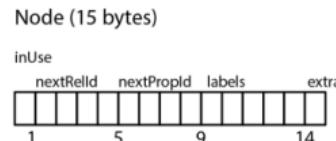
Relationship (34 bytes)



[Ian Robinson et al., Graph Databases, 2015]

How a Graph is Stored in Neo4j? (2/2)

- ▶ Neo4j stores graph data in a number of different **store files**.
- ▶ Each **store file** contains the data for a **specific part** of the graph.
 - Separate stores for **nodes**, **relationships**, **labels**, and **properties**.
- ▶ The division of storage responsibilities **facilitates** performant graph traversals.



[Ian Robinson et al., Graph Databases, 2015]



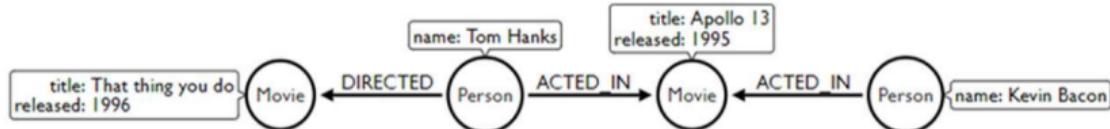


What is Cypher?

- ▶ Declarative query language
- ▶ (): Nodes
- ▶ []: Relationships
- ▶ {}: Properties

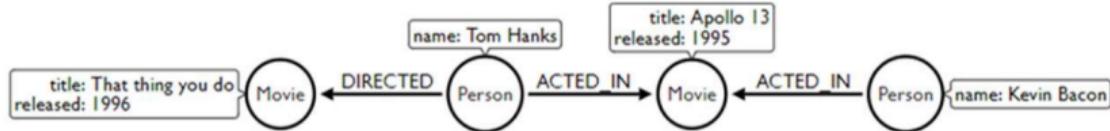


Cypher Example (1/4)



```
// Match all nodes
MATCH (n)
RETURN n;
```

Cypher Example (1/4)

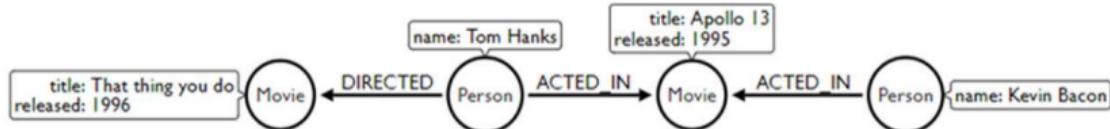


```
// Match all nodes
MATCH (n)
RETURN n;
```

```
// Match all nodes with a Person label
MATCH (n:Person)
RETURN n;
```



Cypher Example (1/4)



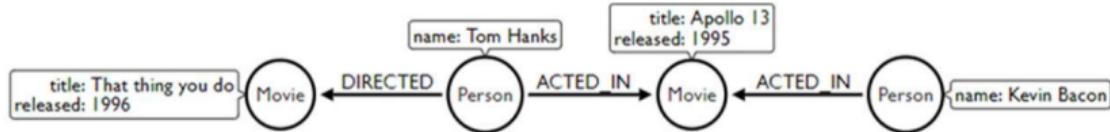
```
// Match all nodes
MATCH (n)
RETURN n;
```

```
// Match all nodes with a Person label
MATCH (n:Person)
RETURN n;
```

```
// Match all nodes with a Person label and property name is 'Tom Hanks'
MATCH (n:Person {name: 'Tom Hanks'})
RETURN n;
```

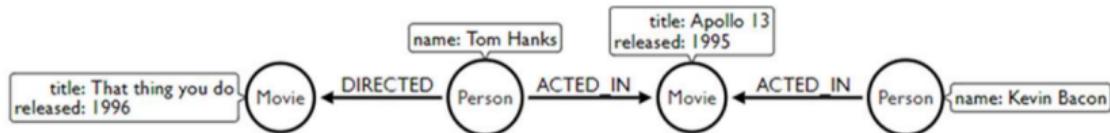


Cypher Example (2/4)



```
// Return nodes with label Person and name property equals 'Tom Hanks'  
MATCH (p:Person)  
WHERE p.name = 'Tom Hanks'  
RETURN p;
```

Cypher Example (2/4)

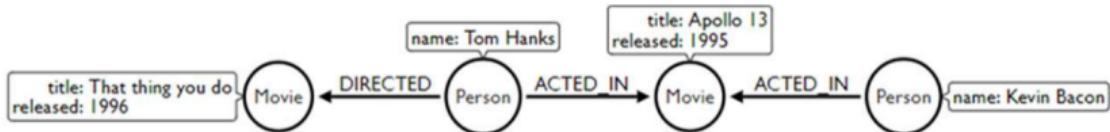


```
// Return nodes with label Person and name property equals 'Tom Hanks'  
MATCH (p:Person)  
WHERE p.name = 'Tom Hanks'  
RETURN p;
```

```
// Return nodes with label Movie, released property is between 1991 and 1999  
MATCH (m:Movie)  
WHERE m.released > 1990 AND m.released < 2000  
RETURN m;
```



Cypher Example (2/4)

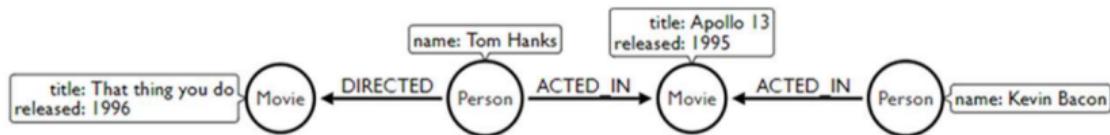


```
// Return nodes with label Person and name property equals 'Tom Hanks'  
MATCH (p:Person)  
WHERE p.name = 'Tom Hanks'  
RETURN p;
```

```
// Return nodes with label Movie, released property is between 1991 and 1999  
MATCH (m:Movie)  
WHERE m.released > 1990 AND m.released < 2000  
RETURN m;
```

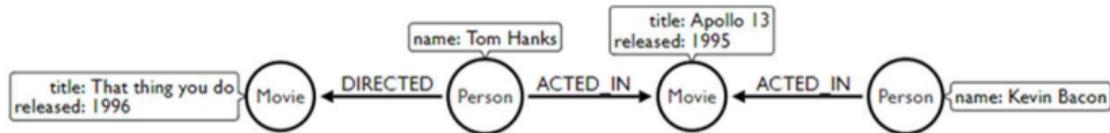
```
// Find all the movies Tom Hanks acted in  
MATCH (:Person {name: 'Tom Hanks'})-[:ACTED_IN]->(m:Movie)  
RETURN m.title;
```

Cypher Example (3/4)



```
// Find all the movies Tom Hanks directed and order by latest movie
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(m:Movie)
RETURN m.title, m.release ORDER BY m.release DESC;
```

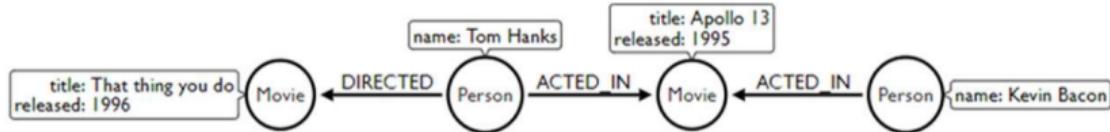
Cypher Example (3/4)



```
// Find all the movies Tom Hanks directed and order by latest movie
MATCH (:Person {name: 'Tom Hanks'})-[:DIRECTED]->(m:Movie)
RETURN m.title, m.release ORDER BY m.release DESC;
```

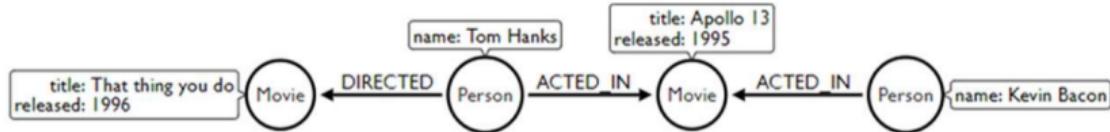
```
// Find all of the co-actors Tom Hanks has ever worked with
MATCH (:Person {name: 'Tom Hanks'})-->(:Movie)<-[:ACTED_IN]-(coActor:Person)
RETURN coActor.name;
```

Cypher Example (4/4)



```
// Find nodes with an ACTED_IN relationship
MATCH (p)-[:ACTED_IN]->()
RETURN p
```

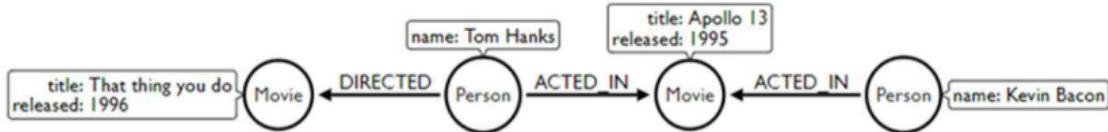
Cypher Example (4/4)



```
// Find nodes with an ACTED_IN relationship
MATCH (p)-[:ACTED_IN]->()
RETURN p
```

```
// Find Person nodes with an ACTED_IN or DIRECTED_IN relationship
MATCH (p:Person)-[:ACTED_IN|DIRECTED]->()
RETURN p
```

Cypher Example (4/4)



```
// Find nodes with an ACTED_IN relationship
MATCH (p)-[:ACTED_IN]->()
RETURN p
```

```
// Find Person nodes with an ACTED_IN or DIRECTED_IN relationship
MATCH (p:Person)-[:ACTED_IN|DIRECTED]->()
RETURN p
```

```
// Find Person nodes who do not have an ACTED_IN relationship
MATCH (p:Person)
WHERE NOT (p)-[:ACTED_IN]->()
RETURN p
```



Summary



Summary

- ▶ NoSQL data models: key-value, column-oriented, document-oriented, graph-based
- ▶ Sharding and consistent hashing
- ▶ ACID vs. BASE
- ▶ CAP (Consistency vs. Availability)



Summary

- ▶ BigTable
- ▶ Column-oriented
- ▶ Main components: master, tablet server, client library
- ▶ Basic components: GFS, SSTable, Chubby
- ▶ CP



Summary

- ▶ Cassandra
- ▶ Column-oriented (similar to BigTable)
- ▶ Consistency hashing
- ▶ Gossip-based membership
- ▶ AP



Summary

- ▶ Neo4j
- ▶ Graph-based
- ▶ Cypher
- ▶ CA



References

- ▶ F. Chang et al., Bigtable: A distributed storage system for structured data, ACM Transactions on Computer Systems (TOCS) 26.2, 2008.
- ▶ A. Lakshman et al., Cassandra: a decentralized structured storage system, ACM SIGOPS Operating Systems Review 44.2, 2010.
- ▶ I. Robinson et al., Graph Databases (2nd ed.), O'Reilly Media, 2015.



Questions?

Acknowledgements

Some content of the Neo4j slides were derived from Ljubica Lazarevic's slides.