

COMP5338 – Advanced Data Models

Week 1: Big Data and NoSQL Database

Dr. Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

■ The Value and limitations of SQL

■ Typical Scale-Out Options

■ NoSQL Database Systems

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

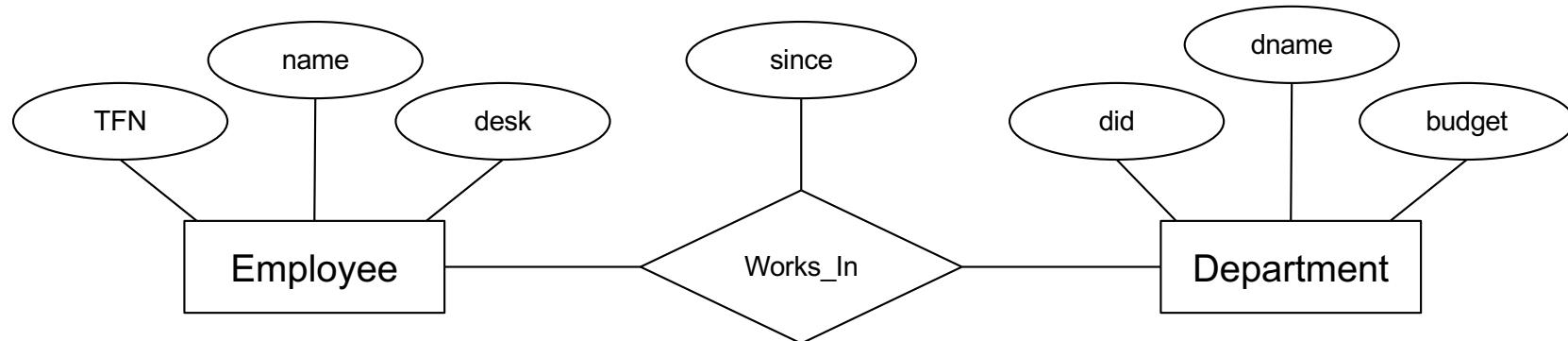
The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



The Relational Model of Data

- Entity-Relationship (ER) Data Model describes data as
 - ▶ Entities – distinct objects in the domain
 - ▶ Relationships – between two or more entities
- Entities – described using a set of attributes
- Relationships – have attributes too
- Used for Conceptual Database Design
 - ▶ Translated to final database implementation



The Rational RDBMS

- Commercial vendors: Oracle, IBM, Microsoft, ...
- Open source systems: MySQL, PostgreSQL, ...
- Common features
 - ▶ Disk-oriented storage;
 - ▶ Table stored row-by-row on disk
 - ▶ B-trees as indexing mechanism
 - ▶ Dynamic locking as the concurrency-control mechanism
 - ▶ A write-ahead log, or WAL for crash recovery
 - ▶ SQL as the access language
 - ▶ A “row-oriented” query optimizer and executor

<http://cacm.acm.org/magazines/2011/6/108651-10-rules-for-scalable-performance-in-simple-operation-datastores/fulltext>



The Value of Relational Databases

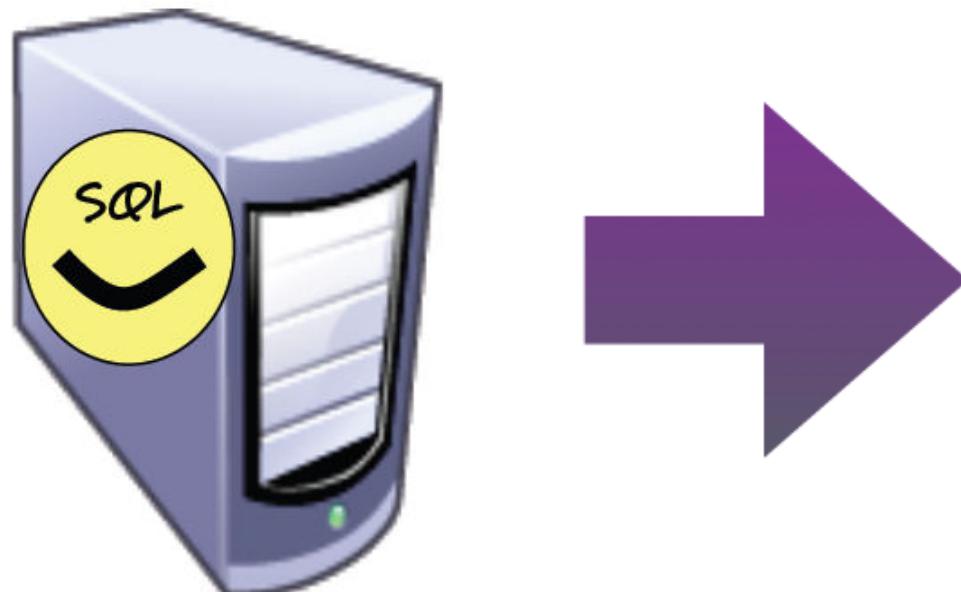
- Store persistent data
 - ▶ Storing large amounts of data on disks, while allowing applications to grab the bits they need through queries
- Application Integration
 - ▶ Many applications in an enterprise need to share information, which might happen at the database level
- Concurrency Control
 - ▶ Database provide transactions to ensure consistent interaction when many users access the same information at the same time
- Mostly Standard
 - ▶ Relational model is widely used and understood.
 - ▶ SQL is the standard language.
- Reporting
 - ▶ SQL's simple data model and standardization has made it a foundation for many reporting tools

<http://martinfowler.com/articles/nosql-intro.pdf>



The Scaling Problem of SQL

Relational databases are designed to run on a single machine, so to scale, you need buy a bigger machine or increase capacity of existing server (**scale up**)



But it's cheaper and more effective to **scale out** by buying lots of machines.



<http://martinfowler.com/articles/nosql-intro.pdf>



The Fixed Schema Problem of SQL

- In a relational database
 - ▶ Table structure are predefined
 - ▶ Tables are related with relationships, which are predefined as well
- Schema evolution in RDBMS has large impact on queries and applications
- Example
 - ▶ MediaWiki had been through 171 schema versions between April 2003 and November 2007.
 - MySQL backend
 - ~ 34 tables, ~242 columns, ~700GB in wikipedia (note: 2008 data)
 - ▶ Schema change has big impact on queries
 - Large number of queries could fail due to schema change.

<http://yellowstone.cs.ucla.edu/schema-evolution/documents/curino-schema-evolution.pdf>



World of Big Data

- Big Data are **high-volume**, **high-velocity**, and/or **high-variety** information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization.

[Gartner 2012 report]
- Leaders in database research community identified “big data as a defining challenging of our time” in a 2013 meeting.
- Three major trends behind big data
 - “It has become much cheaper to generate a wide variety of data, due to inexpensive storage, sensors, smart devices, social software, multiplayer games and the Internet of Things, ...”
 - “It has become much cheaper to process large amount data, due to advances in multicore CPUs, solid state storage, inexpensive cloud computing, and open source software”
 - “data management has become democratized. The process of generating, processing, and consuming data is no longer just for database professionals. Decision makers, domain scientists, .. and everyday consumers now routinely do it”

D. Abadi, et al. “The Beckman report on database research”. *Commun. ACM* 59, 2 (January 2016), 92-99



Schema Change is Unavoidable

■ News paper site example

- ▶ Early days, for each news article, we may only record the following information
 - Title, author, publishing date and time, actual content
- ▶ Gradually we may want to record more about an article
 - Keywords, views, geotags, comments, who “favoured” it, who emailed it, who twittered it ... etc

■ Evolution of an application is inevitable

- ▶ Accept it, incorporate it in the long-term plan for the system
- ▶ Pick a system that allows schema evolution or have a strategy



Outline

- The Value and limitations of SQL

- Typical Scale Out Options

- NoSQL Storage Systems



When Scalability Becomes an Issue

- “**Scalability** is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged in order to accommodate that growth.”

[wikipedia: <https://en.wikipedia.org/wiki/Scalability>]

- In database context, the need for scaling occurs when the **size of the database** and/or the **traffic against it** grows to the point of crossing an optimal level of performance
 - ▶ Scale up
 - ▶ Scale out



Scalability Scenario I

■ Persistent Storage Requirements

- ▶ Medium data size (can fit in one server)
- ▶ Typical query workload consists of **large number of read request** and **relatively low number of write request**

■ Example: wikipedia

- ▶ Only article meta data such as article revision history, articles relations, user account and setting are stored in core relational database system (MySQL)
- ▶ Article text and images are stored separately

■ Key challenge

- ▶ Scale to maintain reasonable **read latency**

<http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf>

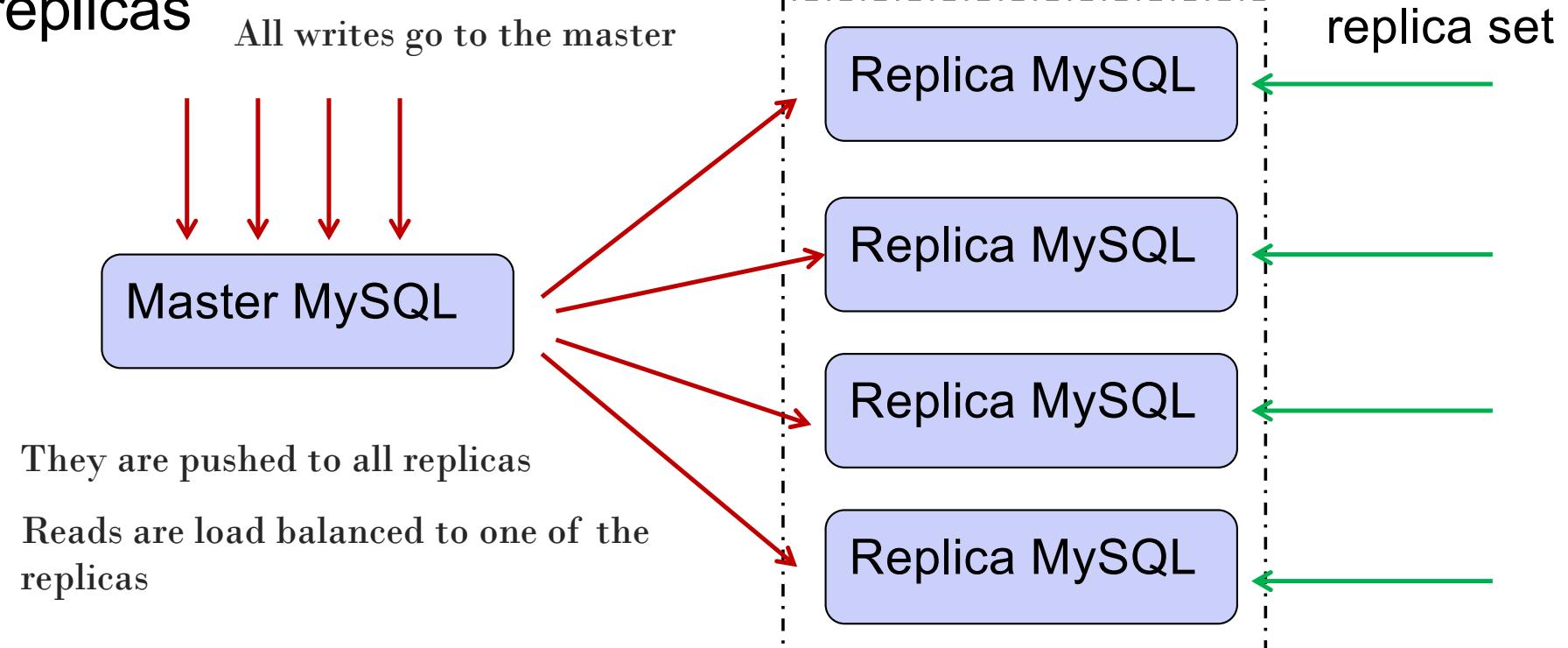


Scalability Scenario I -- Solution

■ Master-Slave Replication

- ▶ Adopted by many companies
- ▶ Also a typical approach to ensure durability

■ Example: Wikipedia has one Master database and many replicas



<http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf>



Scalability Scenario I -- Implications

- When the master dies
 - ▶ One of the replica can be elected as the new master
- Some read may return old data if the latest value has not been pushed from the master
 - ▶ It is possible to let Master handle read request for data requiring strong consistency
- Relatively easy to setup in most RDBMS



Scalability Scenario II

■ Persistent Storage Requirements

- ▶ Medium or large data size (cannot fit in one server)
- ▶ Typical query workload consists of **large number of read request** and **large number of write request**

■ Example: **flickr.com**

- ▶ Heavy write traffic: upload photos, adding tags, adding favourite, ...
- ▶ Over 400,000 photos being added every day. (**note**: 2007 data)
- ▶ More than 4 billion queries per day. (**note**: 2007 data)
- ▶ Uses MySQL as backend storage

■ Key challenge

- ▶ Scale to maintain both **read and write latency**

<http://highscalability.com/flickr-architecture>

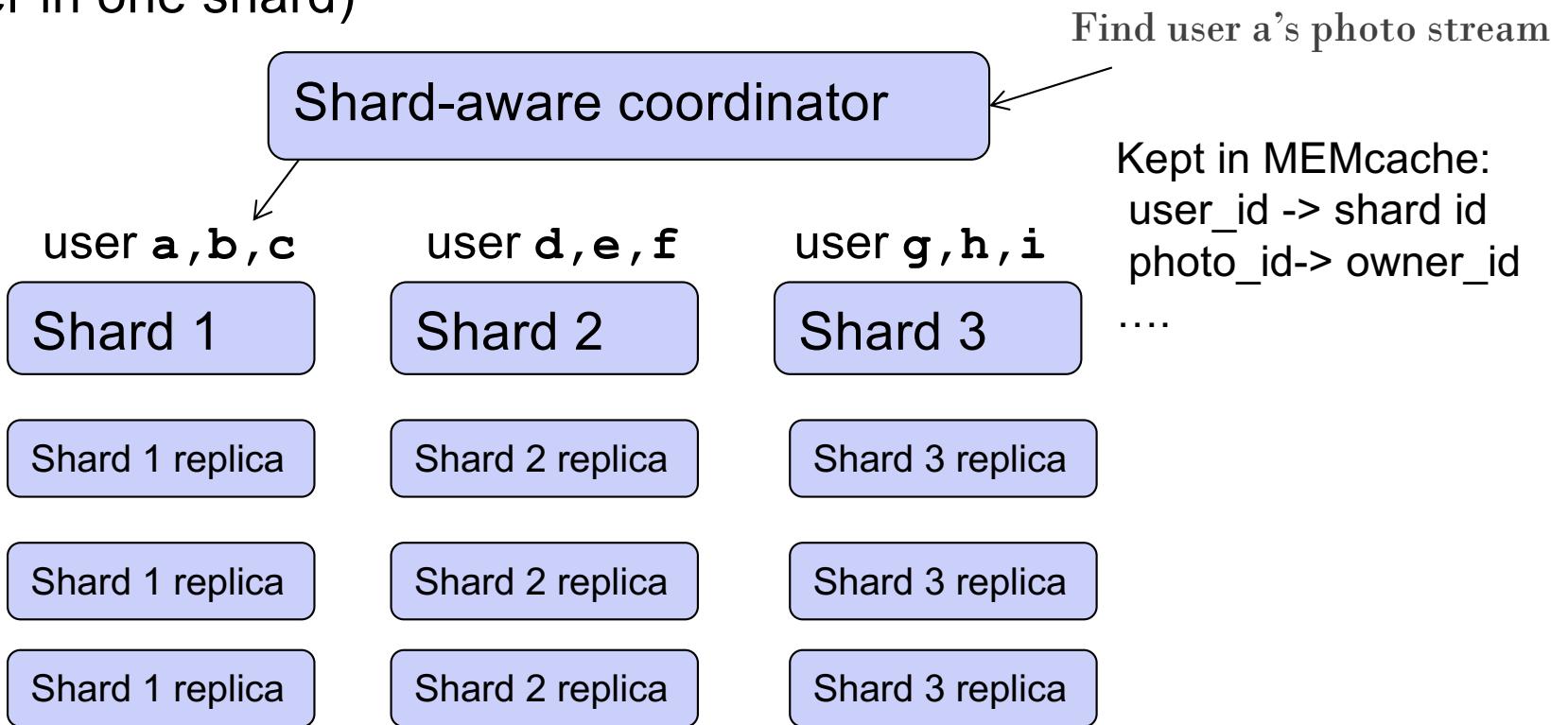
<http://mysqldba.blogspot.com.au/2008/04/mysql-uc-2007-presentation-file.html>



Scalability Scenario II -- Solution

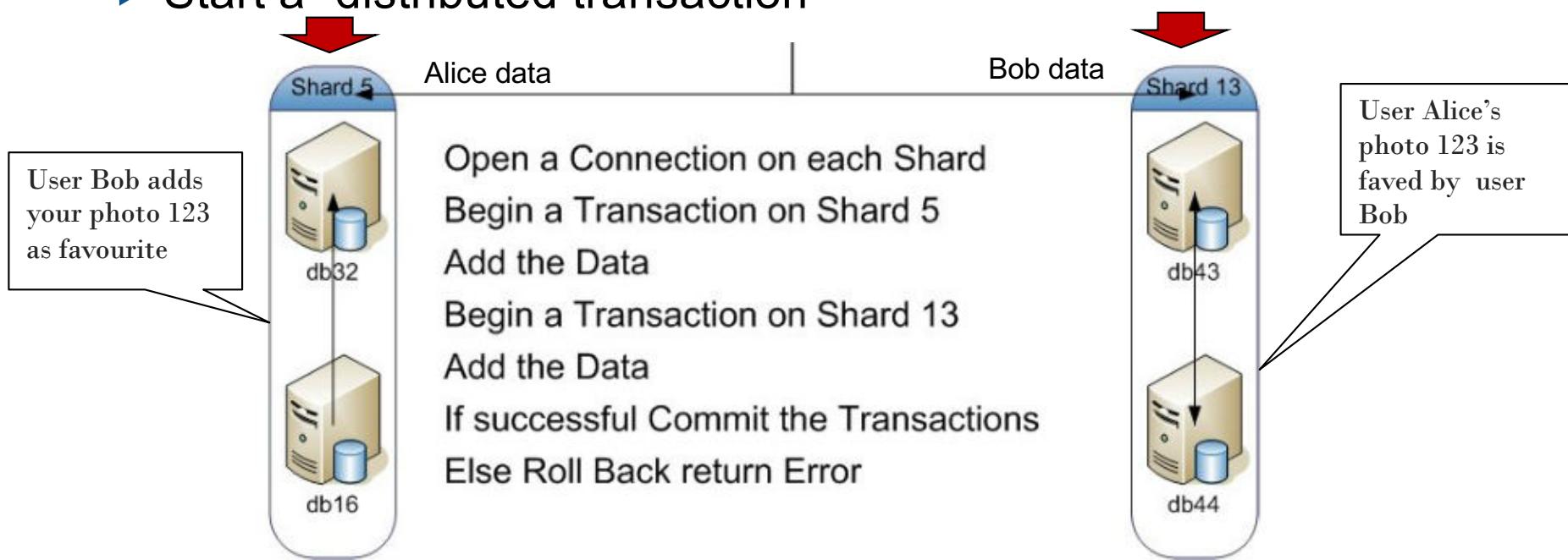
Database Sharding

- ▶ The process of slicing a database across multiple machines
- ▶ Most likely horizontally (e.g., store all data related with a particular user in one shard)



Scalability Scenario II – Flickr Example

- User Bob adds User Alice's photo 123 as “favourite”
 - ▶ Pulls the photo (123) owner's account from cache (“Alice”), to get the shard location
 - SHARD-5
 - ▶ Pulls Bob's information from cache, to get Bob's shard location
 - SHARD-13
 - ▶ Start a “distributed transaction”



<http://mysqldb.blogspot.com.au/2008/04/mysql-uc-2007-presentation-file.html>



Scalability Scenario II -- Implications

- Data have to be de-normalized
 - ▶ E.g. in the previous example, the “fav” relation is stored in both Bob and Alice’s record.
 - ▶ Join is too expensive when data are sharded
 - ▶ Sometimes join can’t be avoided, e.g. building friends network
- Re-balancing or Re-Sharding is hard
 - ▶ What to do when data do not fit in one shard
- Deciding on a partition factor/plan is hard
 - ▶ May generate hotspots
 - See the twitter example on next slides
- Sharding is largely managed outside RDBMS
 - ▶ Recent version of RDBMS may provide limited support for sharding



Scalability Scenario II – Twitter Example

■ Twitter's problem

- ▶ To store 250 million tweets a day using MySQL

■ Twitter's original Tweet Store:

- ▶ Sharding based on time
- ▶ Range partition (timestamp range)
- ▶ The benefits: shards are filled one by one sequentially
- ▶ The downsides: very unbalanced traffic
 - Shards with old tweets do not get any traffic

■ Twitter's new Tweet Store:

- ▶ Sharding based on random partition (id based)
- ▶ A set of in-house systems to manage shards on top of MySQL

<http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html>



Outline

- The Value and limitations of SQL
- Handling Scalability
- NoSQL Storage Systems



The Coming of NoSQL Storage Systems

- There is no standard definition of NoSQL, the term came up during a workshop on 2009 with presentations from **Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB and MongoDB**
 - ▶ Means “Not Only SQL”
- Typical features
 - ▶ They don't use the relational data model and thus don't use the SQL language
 - ▶ They don't have fixed schema, allowing you to store any data in any record
 - ▶ Many of them are designed to run on a cluster
 - Manage “sharding”, fault-tolerance, etc. efficiently
 - ▶ Many of them can be integrated with big data processing framework such as MapReduce

<http://martinfowler.com/articles/nosql-intro.pdf>



NoSQL Ecosystem -- Scalability

■ Distributed NoSQL systems

- ▶ Designed to run on a cluster
- ▶ Support automatically partitioning data across multiple machines
- ▶ Machines can add or leave a *running* cluster
- ▶ Handles failover, fault-tolerance

■ Example Distributed NoSQL systems

- ▶ HBase, Cassandra,...

■ Non-distributed NoSQL systems

- ▶ Designed to run on a single machine
- ▶ Some has limited support for replication and sharding
- ▶ Schema-less and “object” support

■ Example

- ▶ MongoDB, Neo4j, etc..

<http://www.rackspace.com/blog/nosql-ecosystem/>



NoSQL Ecosystem – Data Models

- Document store
 - ▶ Has “table” like concept
 - ▶ Each “record” in a “table” is a semi-structured document
 - ▶ Examples: MongoDB, CouchDB
- Key Value Store
 - ▶ Inspired by Amazon’s Dynamo storage
 - ▶ The overall storage is structured like a big hash table
 - ▶ May or may not have a “table” concept
 - ▶ Redis, Memcached, Voldemort, S3, Cassandra, *DynamnoDB*
- Graph model
 - ▶ Storage is organized as nodes and edges
 - ▶ Neo4j
- Column based store
 - ▶ Inspired by Google’s Bigtable structure
 - ▶ Has “table” like concept
 - ▶ Storage is organized around column family instead of “row”
 - ▶ Examples: Hbase, Cassandra



AWS Database Services

Database

Amazon Aurora

High performance managed relational database

Amazon DocumentDB (with MongoDB compatibility)

Fully managed document database

Amazon DynamoDB

Managed NoSQL database

Amazon ElastiCache

In-memory caching service

Amazon Neptune

Fully managed graph database service

Amazon Quantum Ledger Database (QLDB)

Fully managed ledger database

Amazon RDS

Managed relational database service for MySQL, PostgreSQL, Oracle, SQL Server, and MariaDB



Microsoft Azure Database Service

Databases

Support rapid growth and innovate faster with secure, enterprise-grade and fully managed database services.

Azure Cosmos DB

Globally distributed, multi-model database for any scale

SQL Database

Managed, relational SQL Database as a service

Azure Database for MySQL

Managed MySQL database service for app developers

Azure Database for PostgreSQL

Managed PostgreSQL database service for app developers

Azure Database for MariaDB

Managed MariaDB database service for app developers

SQL Server on virtual machines

Host enterprise SQL Server apps in the cloud

Azure Database Migration Service

Simplify on-premises database migration to the cloud

Azure Cache for Redis

Power applications with high throughput, low latency data access

SQL Server Stretch Database

Dynamically stretch on-premises SQL Server databases to Azure

Table storage

NoSQL key-value store using semi-structured datasets



Google Cloud Platform Database Services

DATABASE TYPE	COMMON USES	GCP PRODUCT
Relational	Compatibility Transactions Complex queries Joins	Cloud Spanner Cloud SQL
NoSQL / Nonrelational	Time series Streaming Mobile Web IoT Offline sync Caching Low latency	Cloud Bigtable Cloud Firestore Firebase Realtime Database Cloud Memorystore



The reality

- The rise of NoSQL databases marks the end of the era of relational database dominance
- But NoSQL databases will not become the new dominators. Relational will still be popular, and used in the majority of situations. They, however, will no longer be the automatic choice.

<http://martinfowler.com/nosql.html>

359 systems in ranking, August 2020

Rank			DBMS	Database Model	Score		
Aug 2020	Jul 2020	Aug 2019			Aug 2020	Jul 2020	Aug 2019
1.	1.	1.	Oracle 	Relational, Multi-model 	1355.16	+14.90	+15.68
2.	2.	2.	MySQL 	Relational, Multi-model 	1261.57	-6.93	+7.89
3.	3.	3.	Microsoft SQL Server 	Relational, Multi-model 	1075.87	+16.15	-17.30
4.	4.	4.	PostgreSQL 	Relational, Multi-model 	536.77	+9.76	+55.43
5.	5.	5.	MongoDB 	Document, Multi-model 	443.56	+0.08	+38.99
6.	6.	6.	IBM Db2 	Relational, Multi-model 	162.45	-0.72	-10.50
7.	↑ 8.	↑ 8.	Redis 	Key-value, Multi-model 	152.87	+2.83	+8.79
8.	↓ 7.	↓ 7.	Elasticsearch 	Search engine, Multi-model 	152.32	+0.73	+3.23
9.	9.	↑ 11.	SQLite 	Relational	126.82	-0.64	+4.10
10.	↑ 11.	↓ 9.	Microsoft Access	Relational	119.86	+3.32	-15.47

<https://db-engines.com/en/ranking>



Social Commerce Application Example

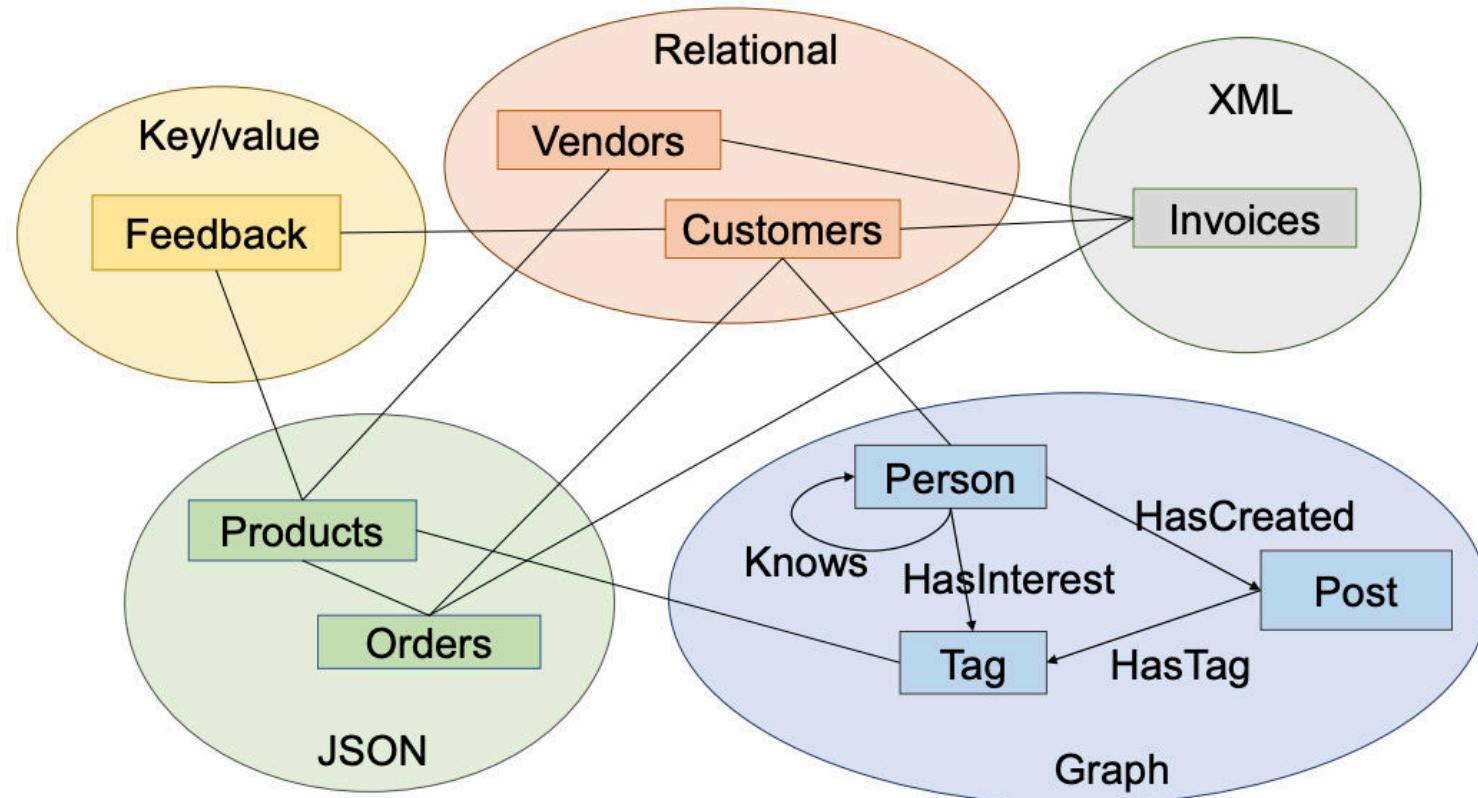


Fig. 1: Unibench Data Model

[UniBench: A Benchmark for Multi-Model Database Management Systems](#)



Polyglot Persistence vs Multi-Model Database

■ Polyglot Persistence

- ▶ Use multiple data storage technologies, chosen based upon the way data is being used by individual applications.

■ Multi-Model Database

- ▶ A single database that supports multiple data model abstraction
- ▶ It usually has a base or primary model with extensions to support other models
- ▶ Object relational database management system is an early version of multi-model database
- ▶ Typical models supported include spatial model, graph model, time series model, etc..
- ▶ Separate abstract model (high level model) from physical model (low level representation)
- ▶ Provide special supports for those extended models



References

- Paramod J. Sadalage and Martin Fowler, “NoSQL Distilled”, Addison-Wesley Professional; 1 edition (August 18, 2012)
 - ▶ Chapter 1
- Daniel Abadi, Rakesh Agrawal, et, al 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (January 2016), 92-99.
DOI=<http://dx.doi.org/10.1145/2845915>
- Curino, Carlo A., Hyun J. Moon, Letizia Tanca, and Carlo Zaniolo. "Schema Evolution In Wikipedia." ICEIS, 2008.
 - ▶ <http://yellowstone.cs.ucla.edu/schema-evolution/documents/curino-schema-evolution.pdf>
- High Scalability post: “How Twitter Stores 250 Million Tweets A Day Using MySQL”, 2011
 - ▶ <http://highscalability.com/blog/2011/12/19/how-twitter-stores-250-million-tweets-a-day-using-mysql.html>
- Dathan Pattishall, “Federation at Flickr: Doing Billions of Queries per Day”, 2007
 - ▶ <http://www.scribd.com/doc/2592098/DVPmysqlFederation-at-Flickr-Doing-Billions-of-Queries-Per-Day>
- Chao Zhang, Jiaheng Lu, Pengfei Xu, Yuxing Chen. ["UniBench: A Benchmark for Multi-Model Database Management Systems"](#) (PDF). TPCTC 2018.



COMP5338 – Advanced Data Models

Week 2: Document Store: Data Model and Simple Query

Dr. Ying Zhou

School of Computer Science



Outline

■ Overview of Document Store

■ MongoDB Data Model

■ MongoDB CRUD Operations

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



Structured and Unstructured Data

- Relational Database System is designed to store **structured data** in tabular format, e.g. each piece of data is stored in a predefined field (attribute)

Supplier Table:

SupplID Name Phone

8703	Heinz	0293514287
8731	Edgell	0378301294
8927	Kraft	0299412020
9031	CSR	0720977632

- **Unstructured data** does not follow any predefined “model” or “format” that is aware to the underlying system. Examples include data stored in various files, e.g word document



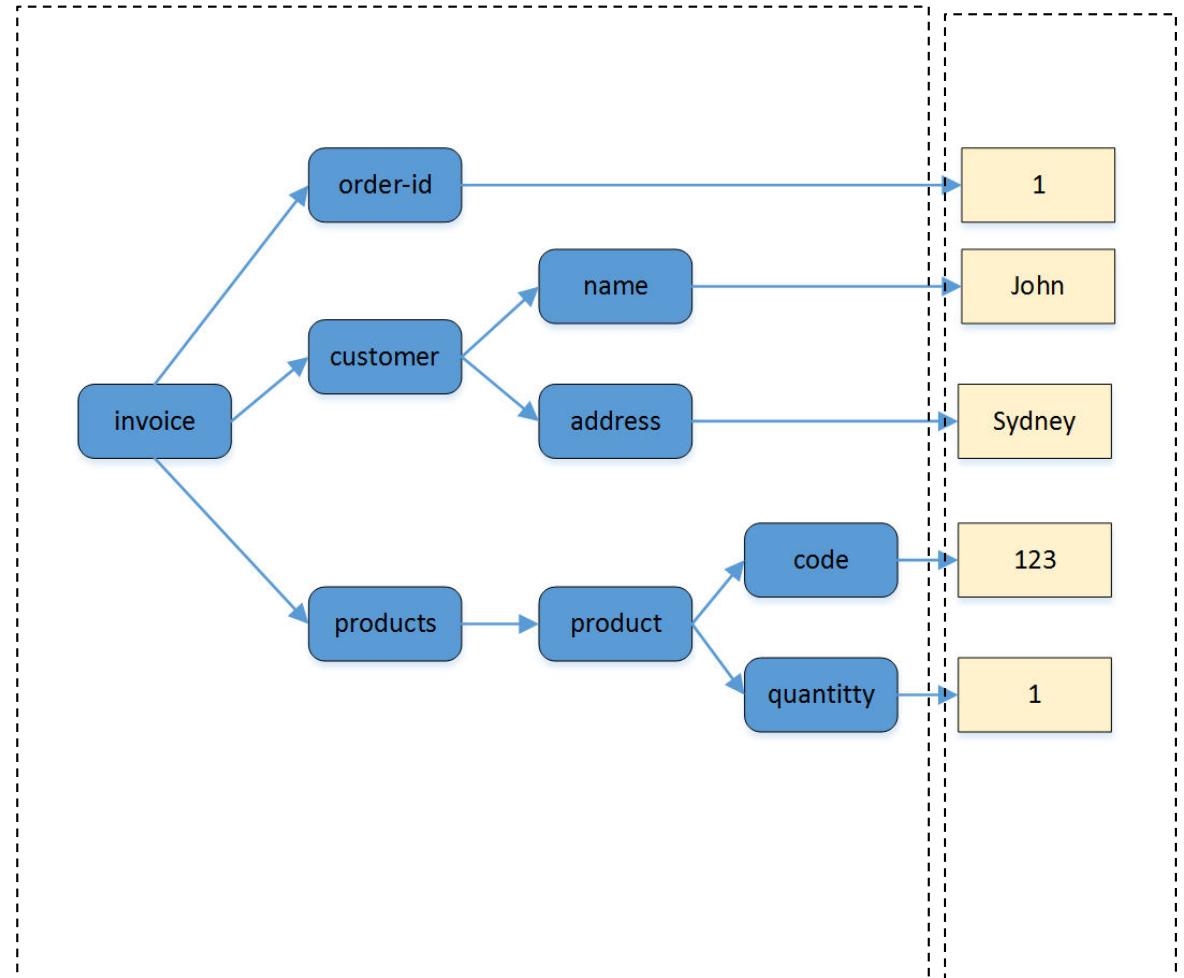
Semi-structured Data

- Many data have some structure but should not be constrained by a *predefined* and *rigid* schema
 - ▶ E.g. if some suppliers have *multiple* phone numbers, it is hard to capture such information in a classic relational model effectively
- **Self-describing** capability is the key feature of semi-structured data
 - ▶ schema/structure is an integral part of the data, instead of a separate declaration
 - ▶ in relational database system, the structure is “**declared**” when a table is created. All rows in the table need to follow the structure.
- **XML** and **JSON** are two types of semi-structured data
 - ▶ Both provide a way to incorporate the structure as part of the data



A Self-describing XML document

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice>
  <order-id> 1</order-id>
  <customer>
    <name> John</name>
    <address> Sydney</address>
  </customer>
  <products>
    <product>
      <code>123</code>
      <quantity>1</quantity>
    </product>
  </products>
</invoice>
```



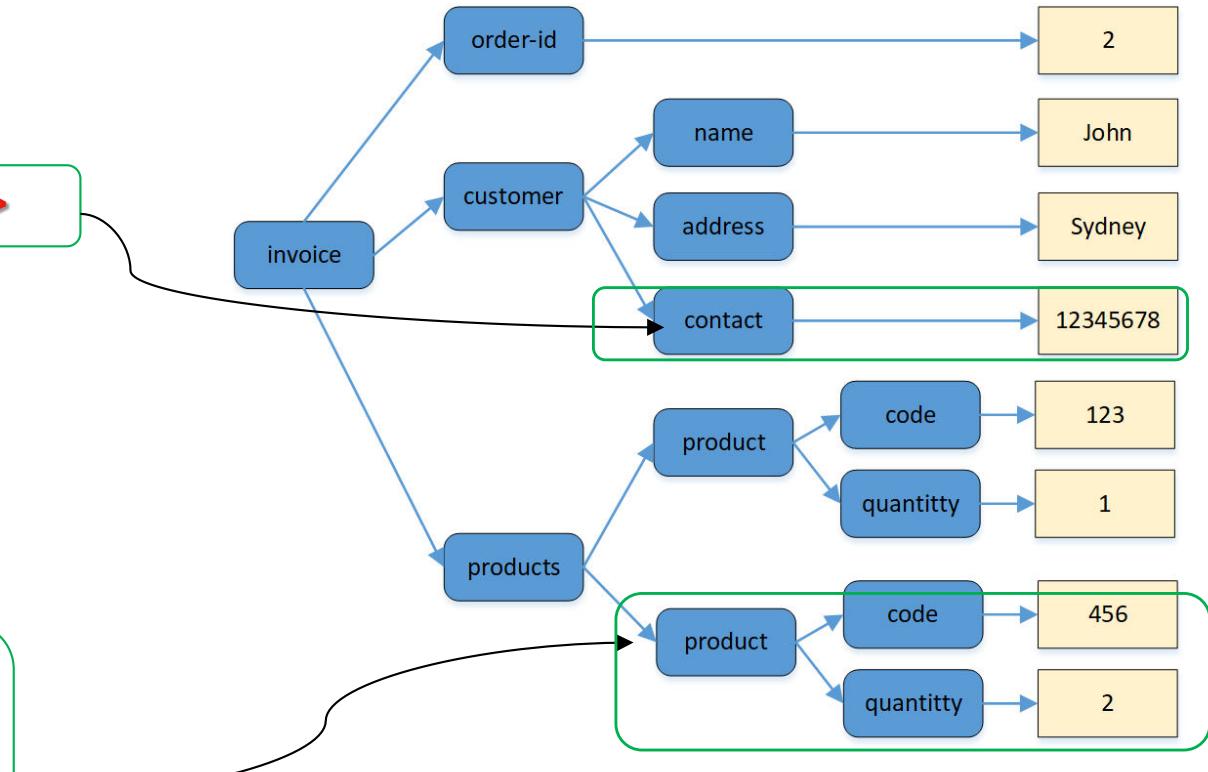
metadata/structure information

data



Another invoice with slightly different structure

```
<?xml version="1.0" encoding="UTF-8"?>
<invoice>
    <order-id> 2</order-id>
    <customer>
        <name> John</name>
        <address> Sydney</address>
        <contact>12345678</contact>
    </customer>
    <products>
        <product>
            <code>123</code>
            <quantity>1</quantity>
        </product>
        <product>
            <code>456</code>
            <quantity>2</quantity>
        </product>
    </products>
</invoice>
```



JSON Data Format

- JSON (JavaScript Object Notation) is a simple way to represent JavaScript objects as strings.
 - ▶ There are many tools to serialize objects in other programming language as JSON
- JSON was introduced in 1999 as an alternative to XML for data exchange.
- Each JSON object is represented as a list of property names and values contained in curly braces, in the following format:

```
{ propertyName1 : value1, propertyName2 : value2 }
```

- Arrays are represented in JSON with square brackets in the following format:

```
[ value1, value2, value3 ]
```



JSON format example

```
Invoice _1= {  
    order-id: 1,  
    customer: {name: "John", address: "Sydney"},  
    products: [ { code: "123", quantity: 1}] }  
  
array  
  
Invoice _3= {  
    order_id: 3,  
    customer: {name: "Smith",  
               address: "Melbourne",  
               contact: "12345"},  
    products: [ { code: "123", quantity: 20},  
               { code: "456", quantity: 2} ]  
    delivery: "express"  
}
```



Document Store

- Document store or document oriented database stores data in semi-structured documents
 - ▶ Document structure is *flexible*
- Provide own query syntax (different to standard SQL)
- Usually has powerful index support
- Examples:
 - ▶ XML based database
 - ▶ JSON based database: MongoDB



Outline

■ Overview of Document Databases

■ MongoDB Data Model

■ MongoDB CRUD operations



Matching Terms between SQL and MongoDB

MongoDB is a general purpose document store.

SQL	MongoDB
Database	Database
Table	Collection
Row	BSON document
Column	BSON field
Primary key	<code>_id</code> field

<https://www.mongodb.com/json-and-bson>



MongoDB Document Model

users table in RDBMS

TFN	Name	Email	age
12345	Joe Smith	joe@gmail.com	30
54321	Mary Sharp	mary@gmail.com	27

Column name is part of **schema**

Defined **once** during
table creation

} two rows

Field name is part
of **data**

```
{ _id: 12345,  
  name: "Joe Smith",  
  email: "joe@gmail.com",  
  age: 30  
}
```

Repeated in every
document

```
{ _id: 54321,  
  name: "Mary Sharp",  
  email: "mary@gmail.com",  
  age: 27  
}
```

} two documents

users collection in MongoDB



Native Support for Array

```
{ _id: 12345,  
  name: "Joe Smith",  
  emails: ["joe@gmail.com", "joe@ibm.com"],  
  age: 30  
}
```

```
{ _id: 54321,  
  name: "Mary Sharp",  
  email: "mary@gmail.com",  
  age: 27  
}
```

<u>TFN</u>	Name	Email	age
12345	Joe Smith	joe@gmail.com , joe@ibm.com ??	30
54321	Mary Sharp	mary@gmail.com	27



Native Support for Embedded Document

```
{ _id: 12345,  
  name: "Joe Smith",  
  email: ["joe@gmail.com", "joe@ibm.com"],  
  age: 30  
}
```

```
{ _id: 54321,  
  name: "Mary Sharp",  
  email: "mary@gmail.com",  
  age: 27,  
  address: { number: 1,  
             name: "cleveland street",  
             suburb: "chippendale",  
             zip: 2008  
  }  
}
```

TFN	Name	Email	age	address
12345	Joe Smith	joe@gmail.com	30	
54321	Mary Sharp	mary@gmail.com	27	1 cleveland street, chippendale, NSW 2008



MongoDB data types

■ Primitive types

- ▶ String, integer, boolean (true/false), double, Null

■ Predefined special types

- ▶ Date, object id, binary data, regular expression, timestamp, and a few more
- ▶ DB Drivers implement them in language-specific way

■ Array and object

■ Field name is of **string** type with certain restrictions

- ▶ “_id” is reserved for primary key
- ▶ cannot start with “\$”, cannot contain “.” or null

<http://docs.mongodb.org/manual/reference/bson-types/>



Data Modelling

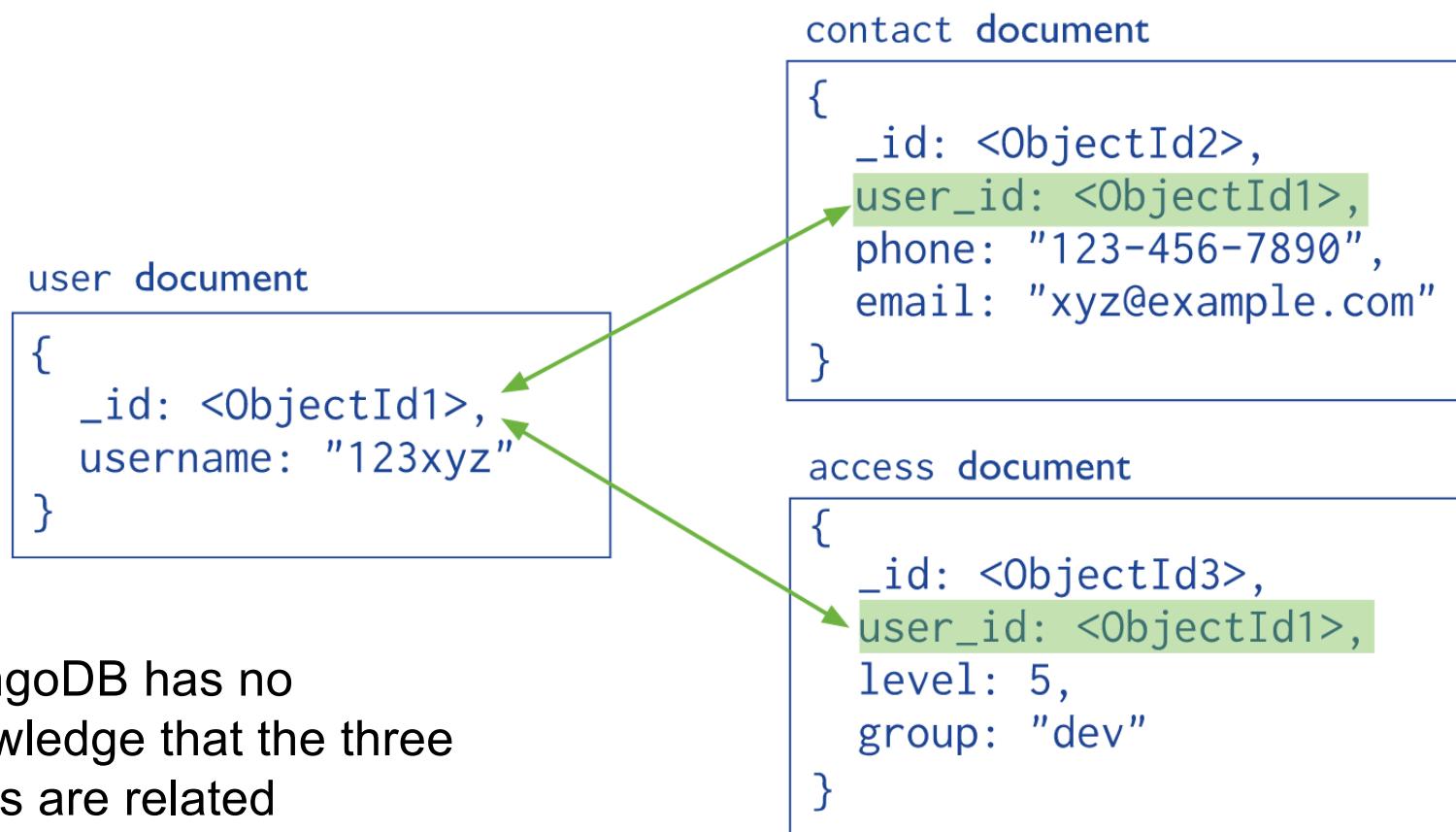
- Key design decision in MongoDB data modelling involves how to represent relationship between data
 - ▶ How many collections should we use
 - ▶ What is the rough document structure in each collection
- Embedding or Referencing
 - ▶ Which object should have its own Collection
 - And reference the `_id` in other collection
 - ▶ Which object can or should be embedded in other object
- As the database system does not keep schema information, the relationship is “remembered” and “managed” externally by developers

<http://www.mongodb.org/display/DOCS/Schema+Design>



Referencing

- References store the relationships between data by including links or *references* from one document to another.



Embedding

- Embedded documents capture relationships between data by storing related data in a single document structure.

```
{  
  _id: <ObjectId1>,  
  username: "123xyz",  
  contact: {  
    phone: "123-456-7890",  
    email: "xyz@example.com"  
  },  
  access: {  
    level: 5,  
    group: "dev"  
  }  
}
```

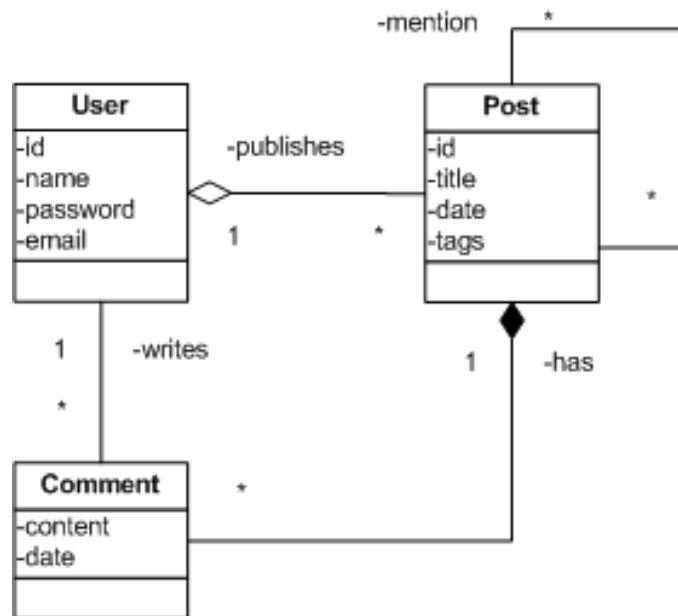
Embedded sub-document

Embedded sub-document

_id is not required
for contact and
access document
now



“Schema” Design Example



■ A fully normalized relational model would have the following tables:

- ▶ User
- ▶ Post
- ▶ Comment
- ▶ PostLink

<http://docs.mongodb.org/manual/applications/data-models/>



MongoDB schema design

■ Using **three** collections

- ▶ **User** collection
- ▶ **Post** collection (with links to **User** and **Post** itself)
- ▶ **Comment** Collection (with links to **User** and **Post**)

■ Using **two** collections

- ▶ **User** collection
- ▶ **Post** collection (with embedded **Comment** object and links to **User** and **Post** itself)



Two Collections Schema

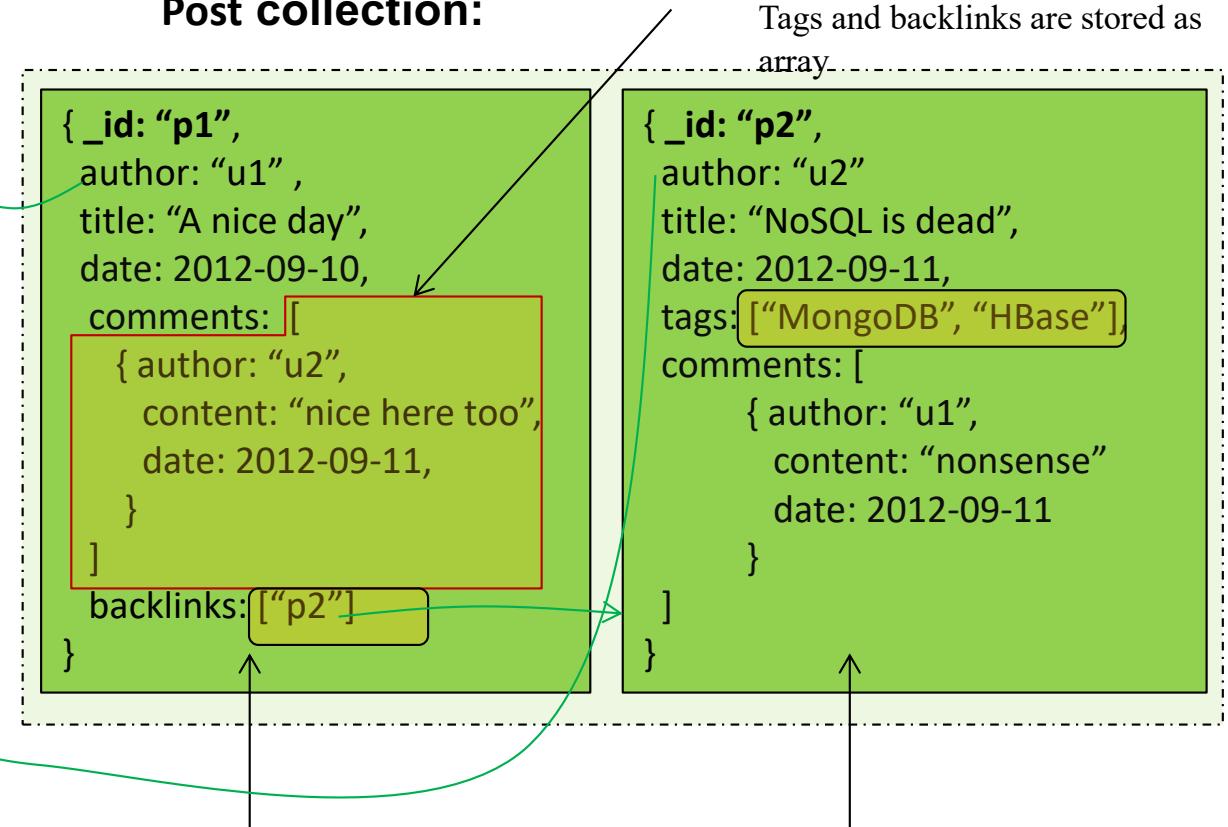
- Two collections
 - ▶ User collection
 - ▶ Post collection (with *embedded Comment object* and links to User and Post itself)

User collection:



Each user profile is saved as a JSON document

Post collection:

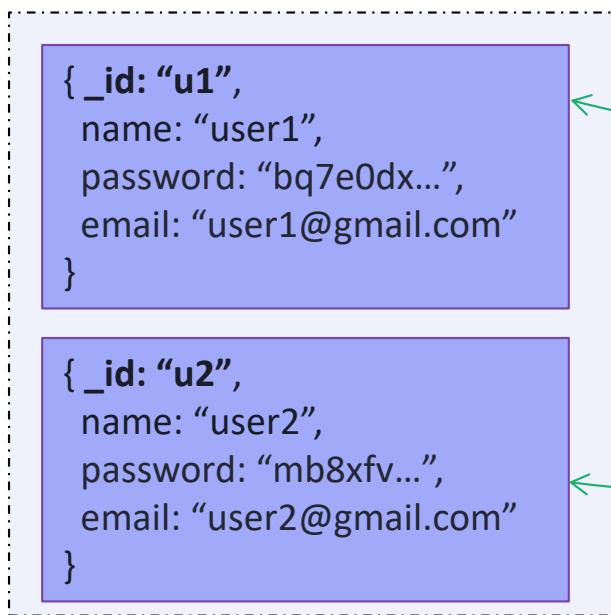


Three Collections Schema

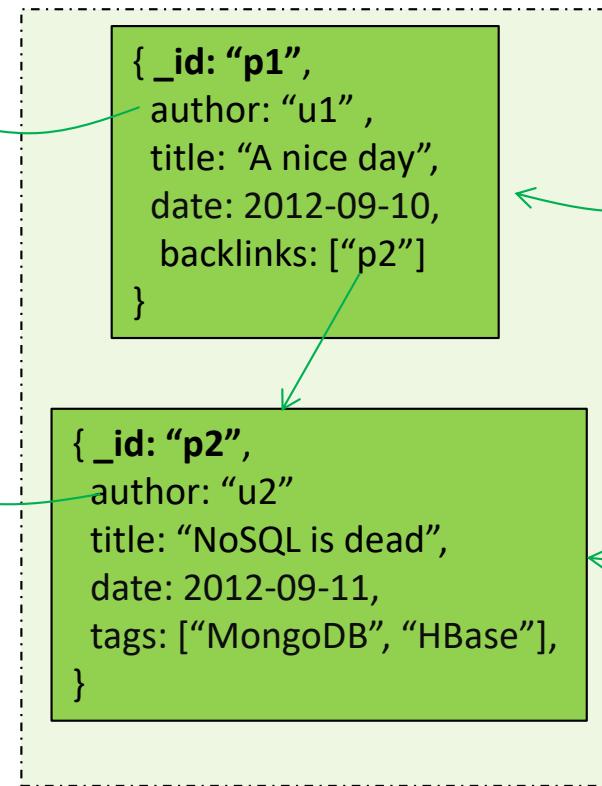
■ Three collections

- ▶ User collection
- ▶ Post collection (with links to User and Post itself)
- ▶ Comment Collection (with links to User and Post)

User collection:



Post collection:



Comment collection:



Two Collections vs. Three Collections

■ Which one is better?

- ▶ Hard to tell by schema itself, we need to look at the actual application to understand
 - Typical data feature
 - What would happen if a post attracts lots of comments?
 - Typical queries
 - Do we want to show all comments when showing a post, or only the latest few, or not at all?
 - Are most comments made in a short period of time?
 - Atomicity consideration
 - Is there “all or nothing” update requirement with respect to post and comment

■ Other design variation?

- ▶ In three collection schema, store post-comment link information in **Post** collection instead of **Comment** collection?
- ▶ Embed the recent comments in **Post**?
- ▶ One **User** collection with embedded **Post** and **Comment** objects? 
- ▶ One **User** collection with **user**, **post** and **comment** documents? 



General Schema Design Guideline

- Depends on data and intended use cases
 - ▶ “independent” object should have its own collection
 - ▶ **composition** relationship are generally modelled as embedded relation
 - Eg. ShoppingOrder and LineItems, Polygon and Points belonging to it
 - BUT, other features need to be considered
 - *Post* and *Comment* have a composition relationship, but it might be beneficial to model them as separate documents
 - ▶ **aggregation** relationship are generally modelled as links (references) with the link data modelled in the ‘part’ object.
 - Eg. *Department* and *Employee*
 - ▶ **Many-to-Many** relationship are generally modelled as links (references)
 - Eg. Course and Students enrolled in a course



Outline

- Overview of Document Databases
- MongoDB Data Model
- MongoDB CRUD Operations



MongoDB Queries

- In MongoDB, a **read** query targets a specific collection. It specifies **criteria**, and may include a **projection** to specify fields from the matching documents; it may include **modifier** to *limit*, *skip*, or *sort* the results.
- A **write** query may *create*, *update* or *delete* data. One query modifies the data of a single collection. Update and delete query can specify query **criteria**

<http://docs.mongodb.org/manual/core/crud-introduction/>



Read Operation Interface

■ db.collection.find()

```
db.users.find(  
  { age: { $gt: 18 } },  
  { name: 1, address: 1 }  
).limit(5)
```

← collection
← query criteria
← projection
← cursor modifier

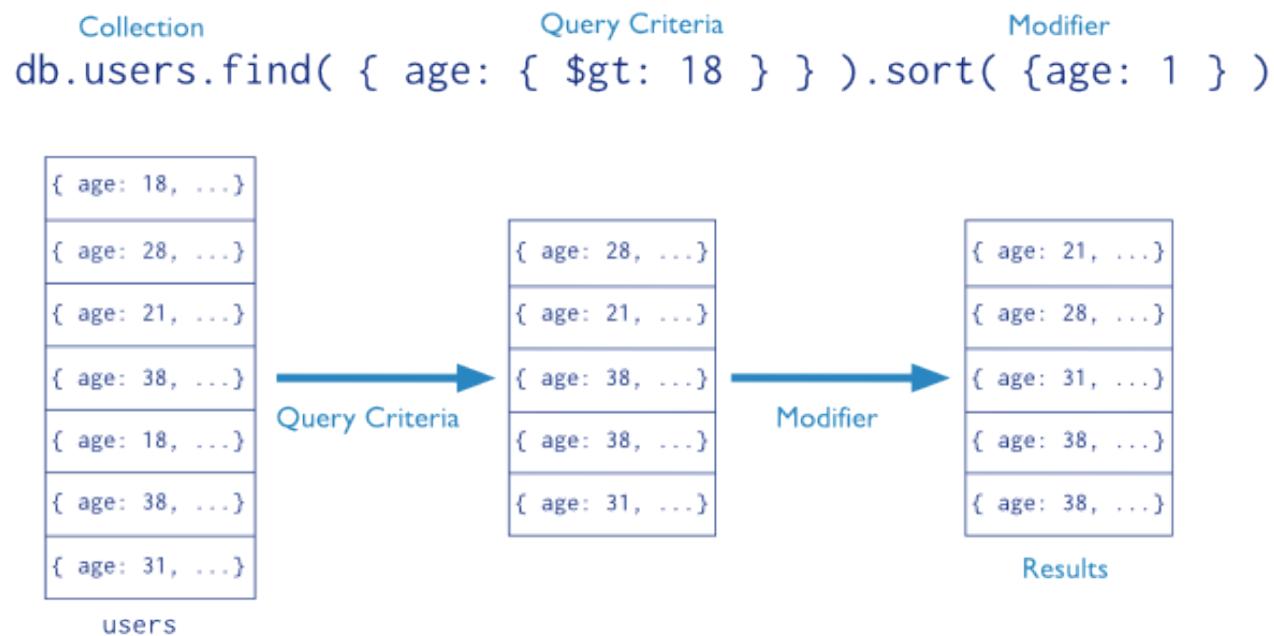
Find at most 5 documents in the `users` collection with `age` field greater than 18, return only the `name` and `address` field of each document.

```
SELECT _id, name, address  
FROM   users  
WHERE  age > 18  
LIMIT  5
```

← projection
← table
← select criteria
← cursor modifier



Read Query Example



Find documents in the **users** collection with **age** field greater than 18, sort the results in ascending order by **age**



Read Query Features

- Users can find data using any criteria in MongoDB
 - ▶ Does not require indexing
 - ▶ Indexing can improve performance (week 4)
- Query **criteria** are expressed as BSON/JSON document (query object)
 - ▶ Individual condition is expressed using predefined selection operator, eg. `$gt` is the operator for “greater than”
- Query **projection** are expressed as BSON/JSON document as well

SQL	MongoDB Query in Shell
<code>select * from user</code>	<code>db.user.find()</code> or <code>db.user.find({})</code>
<code>select name, age from user</code>	<code>db.user.find({}, {name:1, age:1, _id:0})</code>
<code>select * from user where name = "Joe Smith"</code>	<code>db.user.find({name: "Joe Smith"})</code>
<code>select * from user where age < 30</code>	<code>db.user.find({age: {\$lt:30}})</code>



Querying Array field

- MongoDB provide various features for querying array field
 - ▶ <https://docs.mongodb.com/manual/tutorial/query-arrays/>
- The syntax are similar to querying simple type field
 - ▶ db.users.find({emails: "joe@gmail.com"})
 - Find user(s) whose email include "joe@gmail.com".
 - ▶ db.users.find({"emails.0": "joe@gmail.com"})
 - Find user(s) whose first email is "joe@gmail.com".
 - ▶ db.users.find({emails: {\$size:2}})
 - Find user(s) with 2 emails

```
{ _id: 12345,  
  name: "Joe Smith",  
  emails: ["joe@gmail.com", "joe@ibm.com"],  
  age: 30}
```

```
{ _id: 54321,  
  name: "Mary Sharp",  
  email: "mary@gmail.com",  
  age: 27}
```



Querying Embedded Document

- Embedded Document can be queried as a **whole**, or by **individual field**, or by **combination of individual fields**

- ▶ db.user.find({address: {number: 1, name: "pine street", suburb: "chippendale", zip: 2008}})
- ▶ db.user.find({"address.suburb": "chippendale"})
- ▶ db.user.find({"address.name": "pine street", "address.suburb": "chippendale"})

```
{ _id: 12345,  
  name: "Joe Smith", email: ["joe@gmail.com", "joe@ibm.com"], age: 30,  
  address: {number: 1, name: "pine street", suburb: "chippendale", zip: 2008 }  
}
```

```
{ _id: 54321,  
  name: "Mary Sharp", email: "mary@gmail.com", age: 27,  
  address: { number: 1, name: "cleveland street", suburb: "chippendale", zip: 2008 }  
}
```

<http://docs.mongodb.org/manual/tutorial/query-documents/#embedded-documents>



Write Query- Insert Operation

```
db.users.insertOne(      ← collection
  {
    name: "sue",        ← field: value
    age: 26,           ← field: value
    status: "pending"  ← field: value
  }
)
```

Insert a new document in **users** collection.



Insert Example

- db.user.insertOne({**_id: 12345**, name: “Joe Smith”, emails: [“joe@gmail.com”, “joe@ibm.com”], age: 30})
- db.user.insertOne({ **_id: 54321**, name: “Mary Sharp”, email: “mary@gmail.com”, age: 27, address: { number: 1, name: “cleveland street”, suburb: “chippendale”, zip: 2008}})

user collection

```
{ _id: 12345, name: "Joe Smith",
  emails: ["joe@gmail.com", "joe@ibm.com"],
  age: 30
}

{ _id: 54321,
  name: "Mary Sharp", email: "mary@gmail.com", age: 27,
  address: { number: 1,
    name: "cleveland street",
    suburb: "chippendale",
    zip: 2008
  }
}
```



Insert Behavior

- If the collection does not exist, the operation will create one
- If the new document does not contain an “`_id`” field, the system will adds an “`_id`” field and assign a unique value to it.
- If the new document does contain an “`_id`” field, it should have a unique value
- Two other operations:
 - ▶ `insertMany`
 - Insert many documents
 - ▶ `Insert`
 - Major language APIs only support `insertOne` and `insertMany`



Write Query – Update Operation

```
db.users.updateMany(  
  { age: { $lt: 18 } },  
  { $set: { status: "reject" } }  
)
```

← collection
← update filter
← update action

Has the same effect as the following SQL:

```
UPDATE users  
SET status = "reject"  
WHERE age < 18
```

← table
← update action
← update criteria

Two other operations: **updateOne**, **replaceOne**



Updates operators

■ Modifying simple field: \$set, \$unset

- ▶ db.user.updateOne({_id: 12345}, {\$set: {age: 29}})
- ▶ db.user.updateOne({_id:54321}, {\$unset: {email:1}}) // remove the field

■ Modifying array elements: \$push, \$pull, \$pullAll

- ▶ db.user.updateOne({_id: 12345}, {\$push: {emails: "joe@hotmail.com"}})
- ▶ db.user.updateOne({_id: 54321},
{\$push: {emails: {\$each: ["mary@gmail.com", "mary@microsoft.com"]}}})
- ▶ db.user.updateOne({_id: 12345}, {\$pull: {emails: "joe@ibm.com"}})

```
{ _id: 12345,  
  name: "Joe Smith",  
  emails: ["joe@gmail.com", "joe@ibm.com"],  
  age: 30}
```

```
{ _id: 54321,  
  name: "Mary Sharp",  
  email: "mary@gmail.com",  
  age: 27}
```

```
{ _id: 12345,  
  name: "Joe Smith",  
  emails: ["joe@gmail.com", "joe@hotmail.com"],  
  age: 29}
```

```
{ _id: 54321,  
  name: "Mary Sharp",  
  emails: ["mary@gmail.com", "mary@microsoft.com"]  
  age: 27}
```

https://docs.mongodb.com/manual/reference/operator/update/push/#up._S_push



Write Operation - Delete

- `db.user.deleteMany();`
 - ▶ Remove all documents in user collection
- `db.user.deleteMany({age: {$gt:18}})`
 - ▶ Remove all documents matching a certain condition
- `db.user.deleteOne({_id: 12345})`
 - ▶ Remove one document matching a certain condition



Atomicity of write operation (single document)

- The modification of a single document is always atomic
 - ▶ It does not leave a document as partially updated.
 - ▶ A concurrent read will not see a partially updated document
 - ▶ This is true even if the operation modifies multiple embedded documents *within* a single document

<https://docs.mongodb.com/manual/core/read-isolation-consistency-recency/>



Single Document Atomicity

```
db.inventory.insertMany( [  
    { item: "canvas", qty: 100, size: { h: 28, w: 35.5, uom: "cm" }, status: "A" },  
    { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },  
    { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" }]);
```

```
db.inventory.updateOne(  
    { item: "paper" },  
    { $set: { "size.uom": "cm", status: "P" } }  
)
```

```
db.inventory.find({item: "paper"})
```

```
{ item: "paper", qty: 100,  
    size: { h: 8.5, w: 11, uom: "in" },  
    status: "D" }]);
```

```
{ item: "paper", qty: 100,  
    size: { h: 8.5, w: 11, uom: "cm" },  
    status: "D" }]);
```



```
{ item: "paper", qty: 100,  
    size: { h: 8.5, w: 11, uom: "cm" },  
    status: "P" }]);
```



Atomicity of write operation (multi documents)

- If a write operation modifies multiple documents (**insertMany**, **updateMany**, **deleteMany**), the operation as a whole is not atomic, and other operations may interleave.
- Multi-Document Transactions is supported in version 4.0
- Other mechanisms were used in earlier versions
 - ▶ The **\$isolated** operator can prevents a write operation that affects multiple documents from yielding to other reads or writes once the first document is written
- All those mechanisms have great performance impact and are recommended to avoid if possible, document embedding is recommended as an alternative



Write Operation – interleaving Scenario

A write query comes

```
db.users.updateMany(  
  { age: { $gt: 18 } },  
  { $set: { status: "A" } }  
)
```

{age: 21, status: "U"}
{age: 23, status: "S"}
{age: 17, status: "E"}
{age: 25, status: "R"}
{age: 15, status: "S"}
{age: 16, status: "C"}
{age: 19, status: "O"}
{age: 22, status: "L"}

users collection

{age: 21, status: "U"}
{age: 23, status: "S"}
{age: 25, status: "R"}
{age: 19, status: "O"}
{age: 22, status: "L"}

{age: 21, status: "A"}
{age: 23, status: "A"}
{age: 25, status: "A"}
{age: 19, status: "O"}
{age: 22, status: "L"}

{age: 21, status: "A"}
{age: 23, status: "A"}
{age: 25, status: "A"}
{age: 19, status: "A"}
{age: 22, status: "A"}

write is on going, a
read query comes

```
db.users.find(  
  { age: { $gt: 20 } }  
)
```

{age: 21, status: "A"}
{age: 23, status: "A"}
{age: 25, status: "A"}
{age: 22, status: "L"}

Write finishes

Read returned documents



References

■ MongoDB online documents:

- ▶ Mongo DB Data Models
 - <http://docs.mongodb.org/manual/core/data-modeling-introduction/>
- ▶ MongoDB CRUD Operations
 - <http://docs.mongodb.org/manual/core/crud-introduction/>
- ▶ Pramod J. Sadalage, Martin Fowler NoSQL distilled, Addison-Wesley Professional; 1 edition (August 18, 2012)
 - <https://www.amazon.com/NoSQL-Distilled-Emerging-Polyglot-Persistence/dp/0321826620>



COMP5338 – Advanced Data Models

Week 3: MongoDB – Aggregation Framework

Dr. Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

■ Null type

■ Aggregation

► Single collection aggregation

► Aggregation pipeline with multiple collection

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Null, empty string and related operators

- Null (or null) is a special data type
 - ▶ Similar to `None`, `Null` or `Nil` in any programming language
 - ▶ It has a singleton value expressed as `null`
 - ▶ Indicating no value is here
- The interpretation of null is different depending on where it appears
- It might represents
 - ▶ The field exists, but has no value
 - ▶ The field does not exits
 - ▶ Or both
- This is different to giving a field an empty string “” as value

<https://docs.mongodb.com/manual/tutorial/query-for-null-fields/index.html>

Null query example

■ Collection revisions document sample

```
{ "_id" : ObjectId("5799843ee2cbe65d76ed919b"),  
  "title" : "Hillary_Clinton",  
  "timestamp" : "2016-07-23T02  
  "revid" : 731113635,  
  "user" : "BD2412",  
  "parentid" : 731113573,  
  "size" : 251742,  
  "minor" : ""}  
  
1 {  
2   "_id" : ObjectId("5d42869d0c84336545f9b2d3"),  
3   "title" : "Hillary_Clinton",  
4   "timestamp" : ISODate("2016-07-01T19:33:39.000Z"),  
5   "parsedcomment" : "The religious affiliation of Hillary  
6   "revid" : 727871391,  
7   "user" : "Theologicalmess",  
8   "parentid" : 727749878,  
9   "size" : 246162  
10 }
```

- We need a field to indicate if a revision is *minor* or not. The original schema uses a field with empty string value to indicate a minor revision; a document without this field would be a non-minor revision or major revision.

<https://docs.mongodb.com/manual/tutorial/query-for-null-fields/>

Querying for null or field existence

■ Queries

- ▶ `db.revisions.find({minor:{$exists:true}})`
 - Find all documents that has a field called `minor`
- ▶ `db.revisions.find({minor:""})`
 - Find all documents whose `minor` field has a value of "", empty string
- ▶ `db.revisions.find({minor: {$ne: null}})`
 - Find all documents whose `minor` field's value is not `null`
- ▶ `db.revisions.find({minor:null})`
 - Find all documents that do not have a `minor` field or the value of `minor` field is null
- ▶ `db.revisions.find({minor:{$exists:false}})`
 - Find all documents that does not have a `minor` field

It is possible to set the value to null

```
db.revisions.insertOne({title:"nulltest",
  "timestamp" : "2018-08-14T02:02:06Z",
  "revid" : NumberLong(7201808141159),
  "user" : "BD2412",
  "parentid" : 731113573,
  "size" : NumberInt(251900),
  "minor":null})
```

```
db.revisions.insertOne({title:"nulltest",
  "timestamp" : "2018-08-14T02:02:06Z",
  "revid" : NumberLong(201808141157),
  "user" : "BD2412",
  "parentid" : NumberLong(731113573),
  "size" : NumberInt(251800)})
```

db.revisions.find({minor:null}) would return both documents

db.revisions.find({minor:{\$exists:false}}) can differentiate the two

Outline

■ Null type

■ Aggregation

- ▶ **Single collection aggregation pipeline**
- ▶ **Aggregation pipeline with multiple collection**

Aggregation

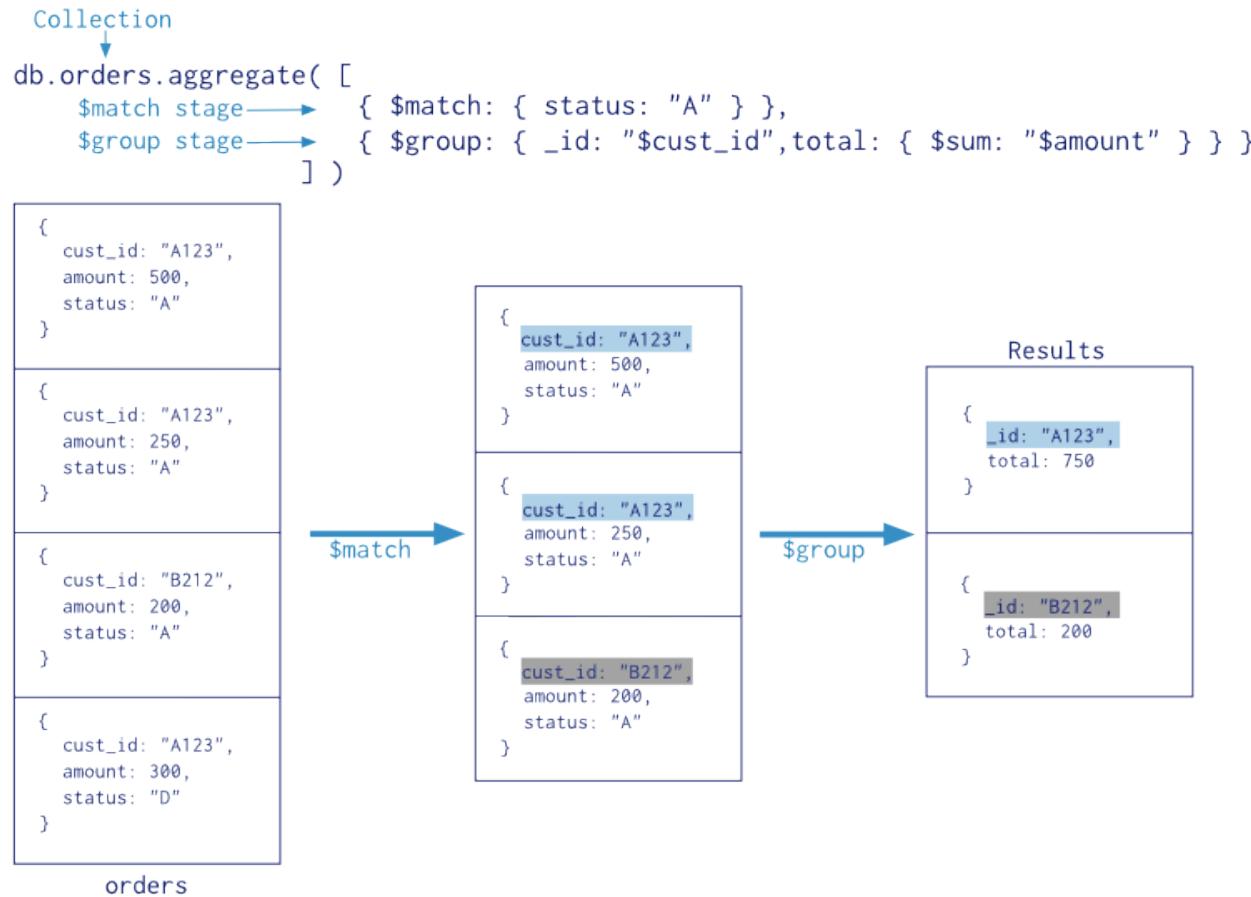
- Simple and relatively standard data analytics can be achieved through **aggregation**
 - ▶ Grouping, summing up value, counting, sorting, etc
 - ▶ Running on the DB engine instead of application layer
- Several options
 - ▶ Aggregation Pipeline
 - ▶ MapReduce
 - Through JavaScript Functions
 - Is able to do customized aggregations

Aggregation Pipeline

- Aggregation pipeline consists of multiple stages
 - ▶ Each stage transforms the incoming documents as expressed in the stage object
 - ▶ The stage object is enclosed in a pair of curly braces
 - ▶ The pipeline is an array of many stage objects.

```
db.collection.aggregate( [  
    { stage name: {expression,..., expression} },  
    { stage name: {expression,..., expression} },  
    ...  
]
```

Aggregation Example



```
select cust_id as _id, SUM(amount) as total  
from orders  
where status = "A"  
group by cust_id
```

Typical aggregation stages

- **\$match**
- **\$group**
- **\$project**
- **\$unwind**
- **\$sort**
- **\$skip**
- **\$limit**
- **\$count**
- **\$sample**
- **\$out**
- **\$lookup**

\$match stage

- **\$match:** filters the incoming documents based on given conditions

- Format:

```
{$match: {<query>}}
```

- The query document is the same as those in the **find** query

- Example:

```
db.revisions.aggregate([{$match:{size :{$lt: 250000 }}}])
```

Has the same effect as

```
db.revisions.find({size :{$lt: 250000 }})
```

\$group stage

- **\$group:** groups incoming documents by some specified expression and outputs to the next stage a document for each distinct group

- Format:

```
{ $group:  
    {_id:<expression>, // Group By Expression  
     <field1>:{accumulator: <expression>},  
     ...}  
}
```

- ▶ The `_id` field of the output document has the value of the group key for each group
- ▶ The other fields usually represent the statistics you want to produce for each group
- ▶ One statistics per field
 - Total amount, average price, group size

\$group stage (cont'd)

- <Expression> in {_id:<expression>},
 - ▶ null value, to specify the whole collection as a group
 - ▶ field path to specify one or many fields as grouping key
 - Field name prefixed with \$ sign in a pair of quotes
 - "\$title", or "\$address.street"
- {accumulator: <expression>}
 - ▶ There are predefined accumulators: \$sum, \$avg, \$first, \$last, etc
 - ▶ User defined accumulators can be used as well
 - ▶ Field path will be used in the <expression> if the accumulator returns value based on field values in the incoming document

\$group stage example

- Find the earliest revision time in the whole collection

```
db.revisions.find({}, {timestamp:1, _id:0})
```

```
.sort({timestamp:1})
```

Sort in ascending order of timestamp

```
.limit(1)
```

Accumulator Field path as expression

```
db.revisions.aggregate([
```

```
  {$group: {_id:null, earliest: {$min: "$timestamp"}}}
```

```
])
```

Returns a single document

- Find the earliest revision time of each page in the collection

```
db.revisions.aggregate([
```

```
  {$group: {_id:"$title", earliest: {$min: "$timestamp"}}}
```

```
])
```

field path

Returns a document for each distinct title

\$group stage example (cont'd)

- Find the number of revisions made on each page by each individual user
 - This would require grouping based on two fields: title and user
 - We need to specify these two as the `_id` field of the output document

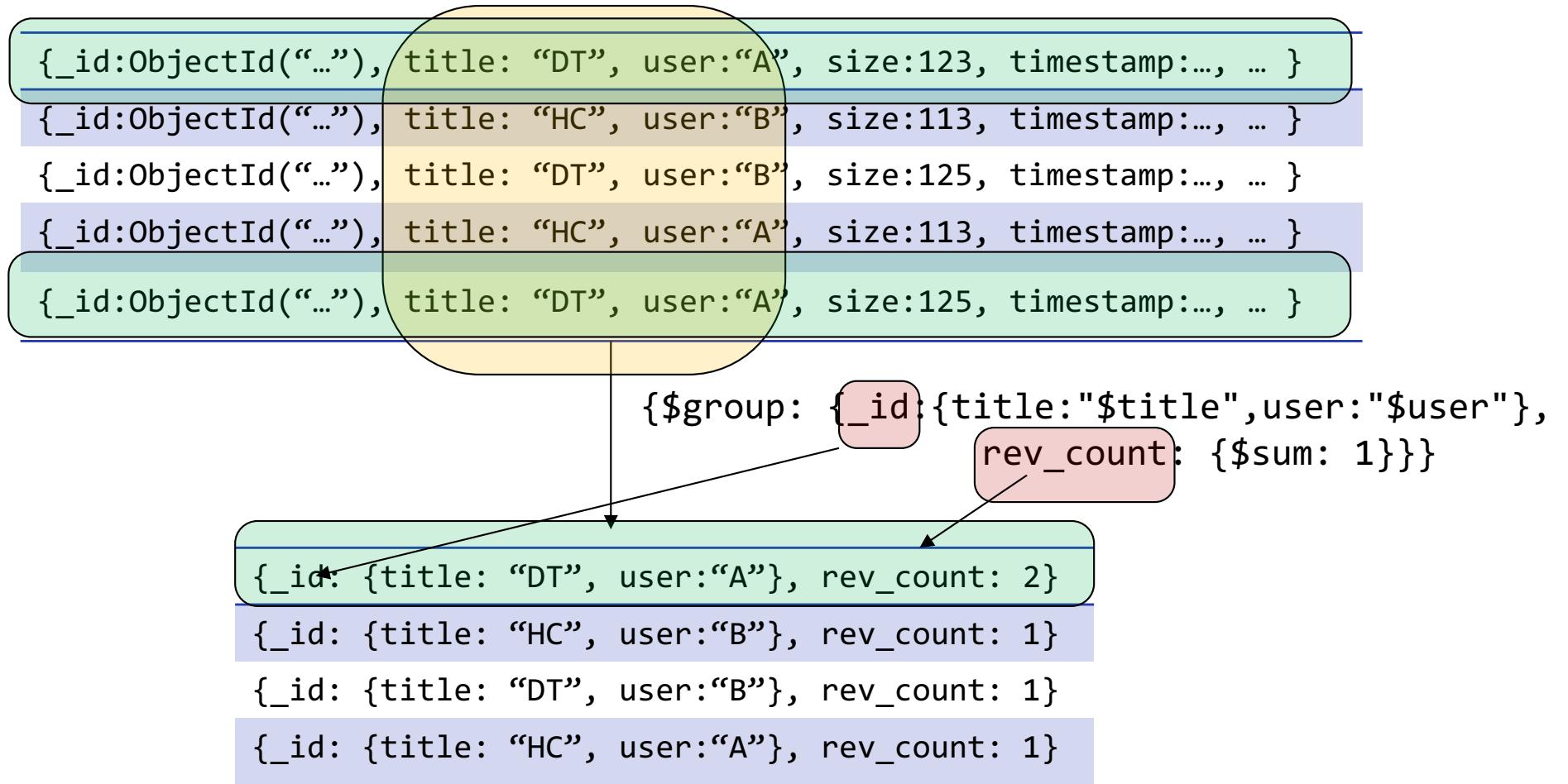
Composite type as `_id`

```
db.revisions.aggregate([
  {$group: {_id: {title: "$title", user: "$user"}},
   rev_count: {$sum: 1}}])
])
```

accumulator number literal

Same effect as count

\$group by more than one field



\$group examples (cont'd)

- Accumulators do not *just* return a single value, we can use accumulators to create an array to hold data from incoming documents
- Example of two commands:

```
db.revisions.aggregate([  
  {$group: {_id:"$title",  
            revs: {$push:{user:"$user",timestamp:"$timestamp"}}}  
}])
```

```
db.revisions.aggregate([  
  {$group: {_id:"$title",rev_users: {$addToSet:"$user"}}}  
])
```

They have the same group key: \$title

They have another field in addition to the group key

The other field is created with different accumulators

\$push accumulator

```
{_id:ObjectId("..."), title: "DT", user:"A", size:123, timestamp:..., ... }  
{_id:ObjectId("..."), title: "HC", user:"B", size:113, timestamp:..., ... }  
{_id:ObjectId("..."), title: "DT", user:"B", size:125, timestamp:..., ... }  
{_id:ObjectId("..."), title: "HC", user:"A", size:113, timestamp:..., ... }  
{_id:ObjectId("..."), title: "DT", user:"A", size:125, timestamp:..., ... }
```

```
db.revisions.aggregate([  
  {$group:  
    {_id:"$title",  
     revs:{$push:{user:"$user",timestamp:"$timestamp"}}}  
  }])
```

```
{ _id: "DT",  
  revs:[  
    {user:"A",timestamp:...},  
    {user:"B",timestamp:...},  
    {user:"A",timestamp:...}  
  ]}  
{ _id:"HC",  
  revs:[  
    {user:"A", timestamp:...},  
    {user:"B", timestamp:...}  
  ]}
```

\$addToSet accumulator

```
{_id:ObjectId("..."), title: "DT", user:"A", size:123, timestamp:..., ... }  
{_id:ObjectId("..."), title: "HC", user:"B", size:113, timestamp:..., ... }  
{_id:ObjectId("..."), title: "DT", user:"B", size:125, timestamp:..., ... }  
{_id:ObjectId("..."), title: "HC", user:"A", size:113, timestamp:..., ... }  
{_id:ObjectId("..."), title: "DT", user:"A", size:125, timestamp:..., ... }
```

```
db.revisions.aggregate([  
    {$group: {_id:"$title",  
              rev_users:{$addToSet:"$user"}}}  
])
```

```
{ _id: "DT",  
  rev_users:["A", "B"]  
}
```

```
{ _id:"HC",  
  rev_users:["A", "B"]  
}
```

\$project stage

■ \$project

- ▶ **Restructure** the document by including/excluding field, adding new fields, resetting the value of existing field
- ▶ More powerful than the *project* argument in **find** query
- ▶ Format

{\$project: {<specification(s)>}}

- ▶ The specification can be an existing field name followed by a single value indicating the inclusion (1) or exclusion (0) of fields
- ▶ Or it can be a field name (existing or new) followed by an expression to compute the value of the field

<field>: <expression>

- ▶ In the expression, existing field from incoming document can be accessed using field path: “*\$fieldname*”

\$project examples

- Find the **age** of each title in the collection, where the **age** is defined as the duration between the last and the first revision of that title, assuming the timestamp is of ISODate type

```
db.revisions.aggregate([
  {$group: {_id:"$title",
    first: {$min:"$timestamp"},
    last: {$max:"$timestamp"} }},
  {$project: {_id: 0,
    title: "$_id",
    age: {$subtract:["$last","$first"]}}}
])
```



Arithmetic expression operator, part of a large group of **Aggregation pipeline operator**

<https://docs.mongodb.com/manual/reference/operator/aggregation/#arithmetic-expression-operators>

\$group then \$project

```
{_id:ObjectId("..."), title: "DT", timestamp:"2016-07-01 00:03:46.000Z", ... }  
{_id:ObjectId("..."), title: "HC", timestamp:"2016-07-01 00:55:44.000Z", ... }  
{_id:ObjectId("..."), title: "DT", timestamp:"2016-07-15 12:22:35.000Z", ... }  
{_id:ObjectId("..."), title: "HC", timestamp:"2016-07-28 00:03:58.000Z", ... }  
{_id:ObjectId("..."), title: "DT", timestamp:"2016-07-28 00:20:19.000Z", ... }
```

```
↓  
{$group: {_id:"$title",  
           first: {$min:"$timestamp"},  
           last: {$max:"$timestamp"} }},
```

```
{_id:"DT", first:"2016-07-01 00:03:46.000Z", last:"2016-07-28 00:20:19.000Z"}  
{_id:"HC", first:"2016-07-01 00:55:44.000Z", last:"2016-07-28 00:03:58.000Z"}
```

```
↓  
{$project: {_id: 0,  
            title: "$_id",  
            age: $subtract:[ "$last", "$first" ]}}}
```

```
{title: "DT", age:2333793000}  
{title: "HC", age:2329694000}
```

We can combine multiple operators

```
db.revisions.aggregate([
  {$group: {_id: "$title",
    first: {$min: "$timestamp"},
    last: {$max: "$timestamp"} }},
  {$project: {_id: 0,
    title: "$_id",          ($last-$first)/86400000
    age: {$divide:
      [{ $subtract: ["$last", "$first"] },
       86400000]} }
    age_unit: {$literal: "day"} }})
])
```

Dealing with data of array type

- To aggregate (e.g. grouping) values in an array field, it is possible to flatten the array to access individual value
- **\$unwind** stage flattens an array field from the input documents to output a document for *each* element. Each output document is the input document with the value of the array field replaced by the element.
 - ▶ { **\$unwind**: <field path> } or
 - ▶ {
 \$unwind:
 {
 path: <field path>,
 includeArrayIndex: <string>,
 preserveNullAndEmptyArrays: <boolean>
 }
}

\$unwind example

■ Default behaviour

- ▶ Input document:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L" ] }
```

- ▶ After \$unwind:"\$sizes"

- ▶ Becomes 3 output documents:

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
```

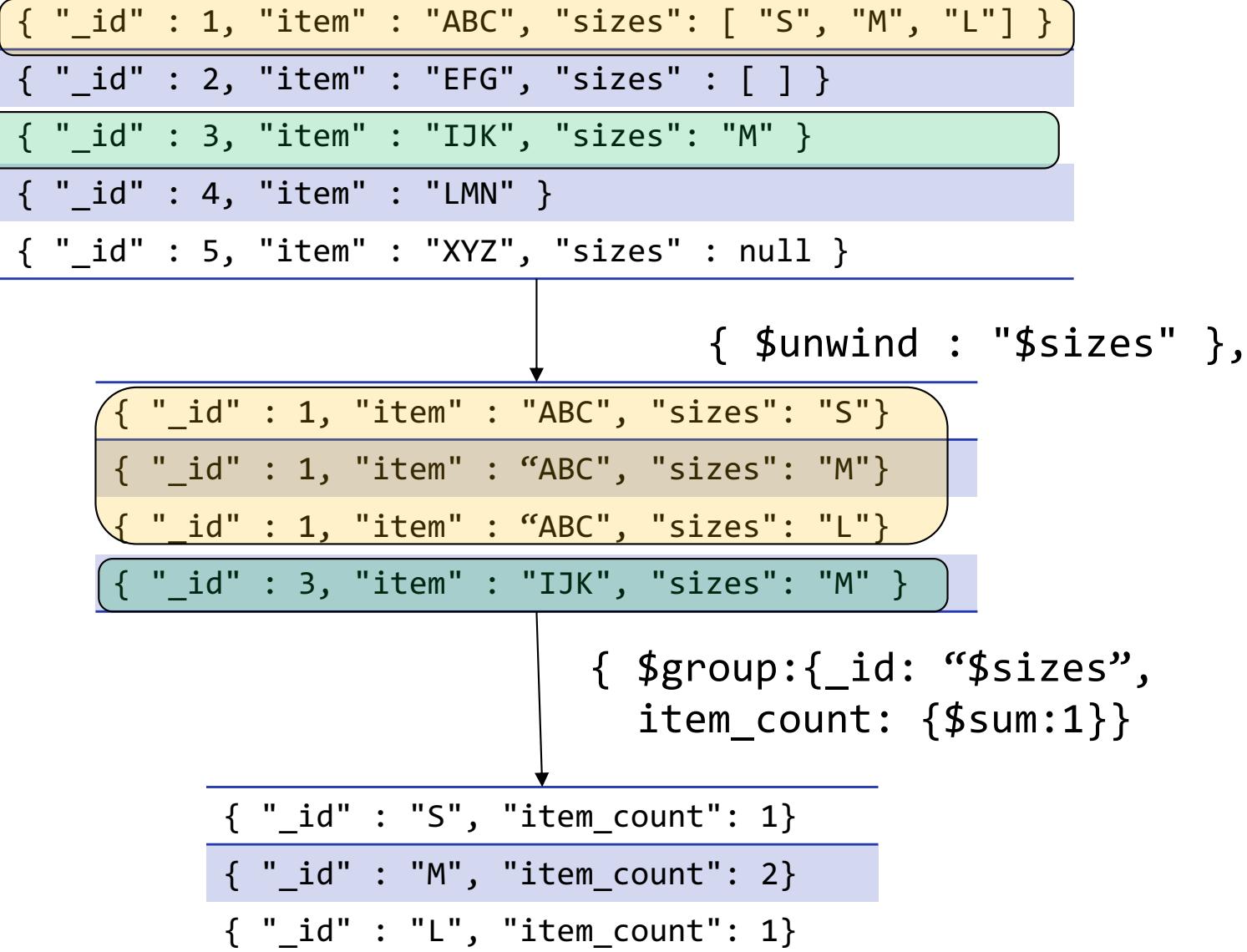
```
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
```

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

■ Find the number of items that are available in each size

```
db.inventory.aggregate( [  
    { $unwind : "$sizes" },  
    { $group:{_id: "$sizes", item_count: {$sum:1}} }  
] )
```

\$unwind then \$group



\$sort, \$skip, \$limit and \$count stages

- **\$sort** stage sorts the incoming documents based on specified field(s) in ascending or descending order
 - ▶ The function and format is similar to the sort modifier in **find** query
 - ▶ { **\$sort**: { <field1>: <sort order>, <field2>: <sort order> ... } }
- **\$skip** stage skips over given number of documents
 - ▶ The function and format is similar to the skip modifier in **find** query
 - ▶ { **\$skip**: <positive integer> }
- **\$limit** stage limits the number of documents passed to the next stage
 - ▶ The function and format is similar to the limit modifier in **find** query
 - ▶ { **\$limit**: <positive integer> }
- **\$count** stage counts the number of documents passing to this stage
 - ▶ The function and format is similar to the count modifier in **find** query
 - ▶ { **\$count**: <string> }
 - ▶ String is the name of the field representing the count

\$sample and \$out stages

- The **\$sample** stage randomly selects given number of documents from the previous stage
 - ▶ { **\$sample**: { **size**: <positive integer> } }
 - ▶ Different sampling approaches depending on the location of the stage and the size of the sample and the collection
 - ▶ May fail due to memory constraints
- The **\$out** stage writes the documents in a given collection
 - ▶ should be the last one in the pipeline
 - ▶ { **\$out**: "<output-collection>" }

Aggregation Operators

- A few aggregation stages allow us to add new fields or to give existing fields new values based on expression
 - ▶ In **\$group** stage we can use various *operators* or *accumulators* to compute values for new fields
 - ▶ In **\$project** stage we can use operators to compute values for new or exiting fields
- There are many predefined operators for various data types to carry out common operations in that data type
 - ▶ Arithmetic operators: **\$mod**, **\$log**, **\$sqrt**, **\$subtract**, ...
 - ▶ String operators: **\$concat**, **\$split**, **\$indexofBytes**, ...
 - ▶ Comparison operators: **\$gt**, **\$gte**, **\$lt**, **\$lte**,...
 - ▶ Set operators: **\$setEquals**, **\$setIntersection**, ...
 - ▶ Boolean operators: **\$and**, **\$or**, **\$not**, ...
 - ▶ Array operators: **\$in**, **\$size**, ..

Aggregation vs. Query operators

- There is another set of operators that can be used in **find/update/delete** queries or the **\$match** stage of an aggregation
 - ▶ E.g. **\$gt**, **\$lt**, **\$in**, **\$all**...
- The set is smaller and are different to the operators used in **\$group** or **\$project** stage
- Some operators look the same but have different syntax and slightly different interpretation in query and in aggregation.
 - ▶ E.g. **\$gt** in find query looks like

```
{age: {$gt:18}}
```
 - ▶ **\$gt** in **\$project** stage looks like:

```
{over18: {$gt:[“$age”, 18]}}
```

Returns true or false

Outline

■ Null type

■ Aggregation

- ▶ Single collection aggregation pipeline
- ▶ Aggregation pipeline with multiple collections

\$lookup stage

- \$lookup stage is added since 3.2 to perform left outer join between two collections
 - ▶ The collection already in the pipeline (maybe after a few stages)
 - ▶ Another collection (could be the same one)
- For each incoming document from the pipeline, the \$lookup stage adds a new **array field** whose elements are the matching documents from the other collection.
- A few different forms
 - ▶ Equality match
 - ▶ Join with other conditions
 - ▶ Join with uncorrelated sub-queries

\$lookup stage (cont'd)

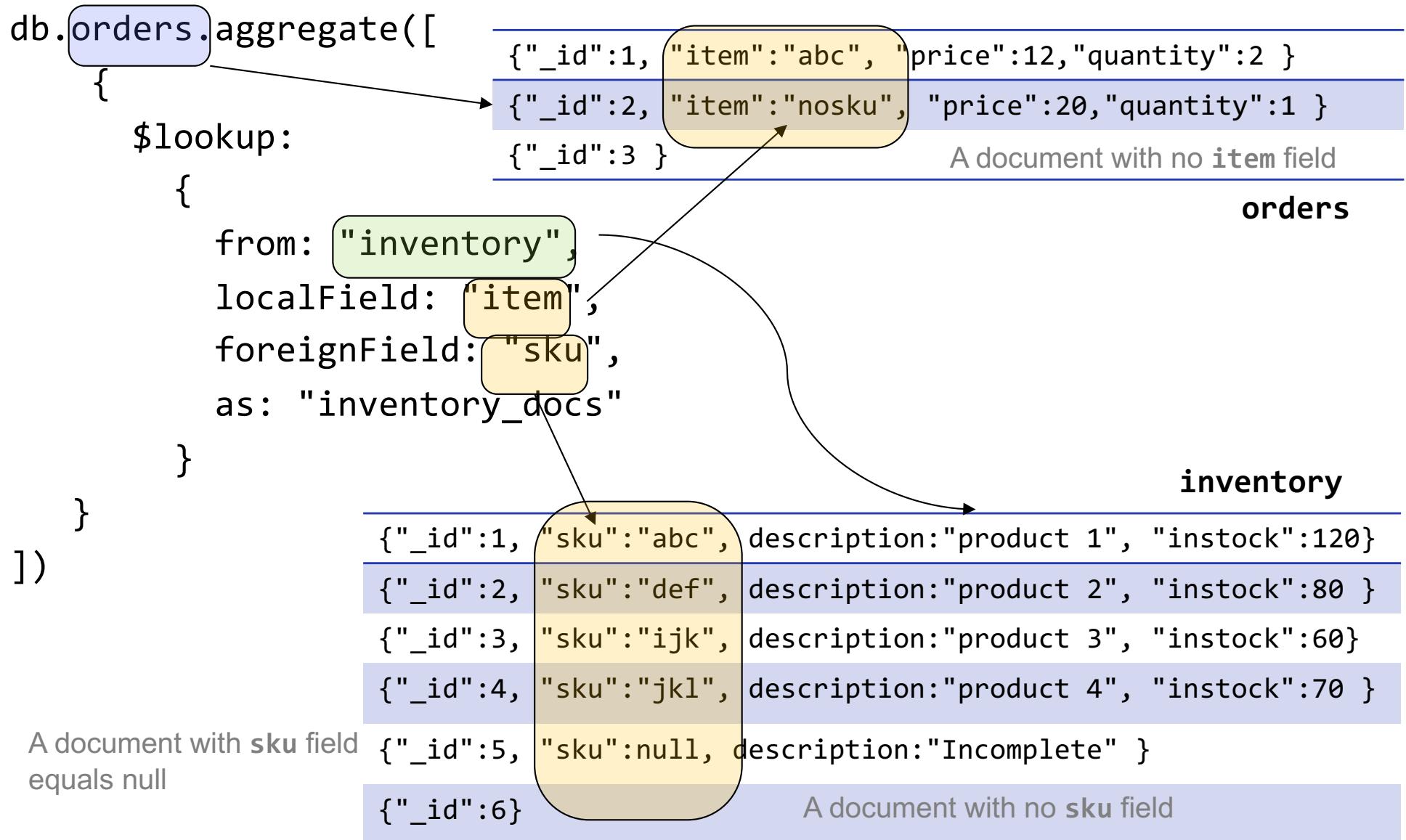
- The output of `$lookup` stage has the same number of documents as the previous stage
- Each document is augmented with an array field storing matching document(s) from the other collection
- The array could contain any number of documents depending on the match, including zero
- Missing local or foreign field is treated as having `null` value

Equality Match \$lookup

```
{$lookup:  
  { from: <collection to join>,  
    localField: <field from the input documents>,  
    foreignField: <field from the documents of the "from" collection>,  
    as: <output array field>  
  }  
}
```

```
SELECT *, <output array field>  
FROM collection  
WHERE <output array field> IN (SELECT *  
                      FROM <collection to join>  
                      WHERE <foreignField>= <collection.localField>);
```

Equality match \$lookup example



https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#pipe._S_lookup



Equality match \$lookup example (cont'd)

```
{"_id":1, "item":"abc", "price":12,"quantity":2 }
```

```
{"_id":2, "item":"nosku", "price":20,"quantity":1 }
```

```
{"_id":3 }
```

local

```
{"_id":1, "sku":"abc", description:"product 1", "instock":120}
```

```
{"_id":2, "sku":"def", description:"product 2", "instock":80 }
```

```
{"_id":3, "sku":"ijk", description:"product 3", "instock":60}
```

```
{"_id":4, "sku":"jkl", description:"product 4", "instock":70 }
```

```
{"_id":5, "sku":null, description:"Incomplete" }
```

foreign

```
{"_id":6}
```

Non exists field matches null and non exists field

```
{"_id":1, "item":"abc", "price":12,"quantity":2,
```

```
"inventory_docs": [
```

```
 { "_id":1, "sku":"abc", description:"product 1", "instock":120 }] }
```

```
{"_id":2, "item":"nosku", "price":20,"quantity":1,
```

```
"inventory_docs" : [] }
```

An empty array for no matching from other collection

```
{"_id":3, "inventory_docs" : [
```

```
 { "_id" : 5, "sku" : null, "description" : "Incomplete" },  
 { "_id" : 6 } ] }
```

output

Other format of \$lookup

```
{  
  $lookup:  
  {  
    from: <collection to join>,  
    let: { <var_1>: <expression>, ..., <var_n>: <expression> },  
    pipeline: [ <pipeline to execute on the collection to join> ],  
    as: <output array field>  
  }  
}
```

let: Optionally specifies variables to use in the [pipeline](#) field stages. Most likely the variable(s) may refer to field(s) in the local collection already in the pipeline

pipeline: Specifies the pipeline to run on the joined collection. The [pipeline](#) determines the resulting documents from the joined collection.

Multiple Joint Condition Example

orders collection

```
{ "_id" : 1, "item" : "almonds", "price" : 12, "ordered" : 2 }
```

```
{ "_id" : 2, "item" : "pecans", "price" : 20, "ordered" : 1 }
```

```
{ "_id" : 3, "item" : "cookies", "price" : 10, "ordered" : 60 }
```

warehouses collection

```
{ "_id" : 1, "stock_item" : "almonds", warehouse: "A", "instock" : 120 }
```

```
{ "_id" : 2, "stock_item" : "pecans", warehouse: "A", "instock" : 80 }
```

```
{ "_id" : 3, "stock_item" : "almonds", warehouse: "B", "instock" : 60 }
```

```
{ "_id" : 4, "stock_item" : "cookies", warehouse: "B", "instock" : 40 }
```

```
{ "_id" : 5, "stock_item" : "cookies", warehouse: "A", "instock" : 80 }
```

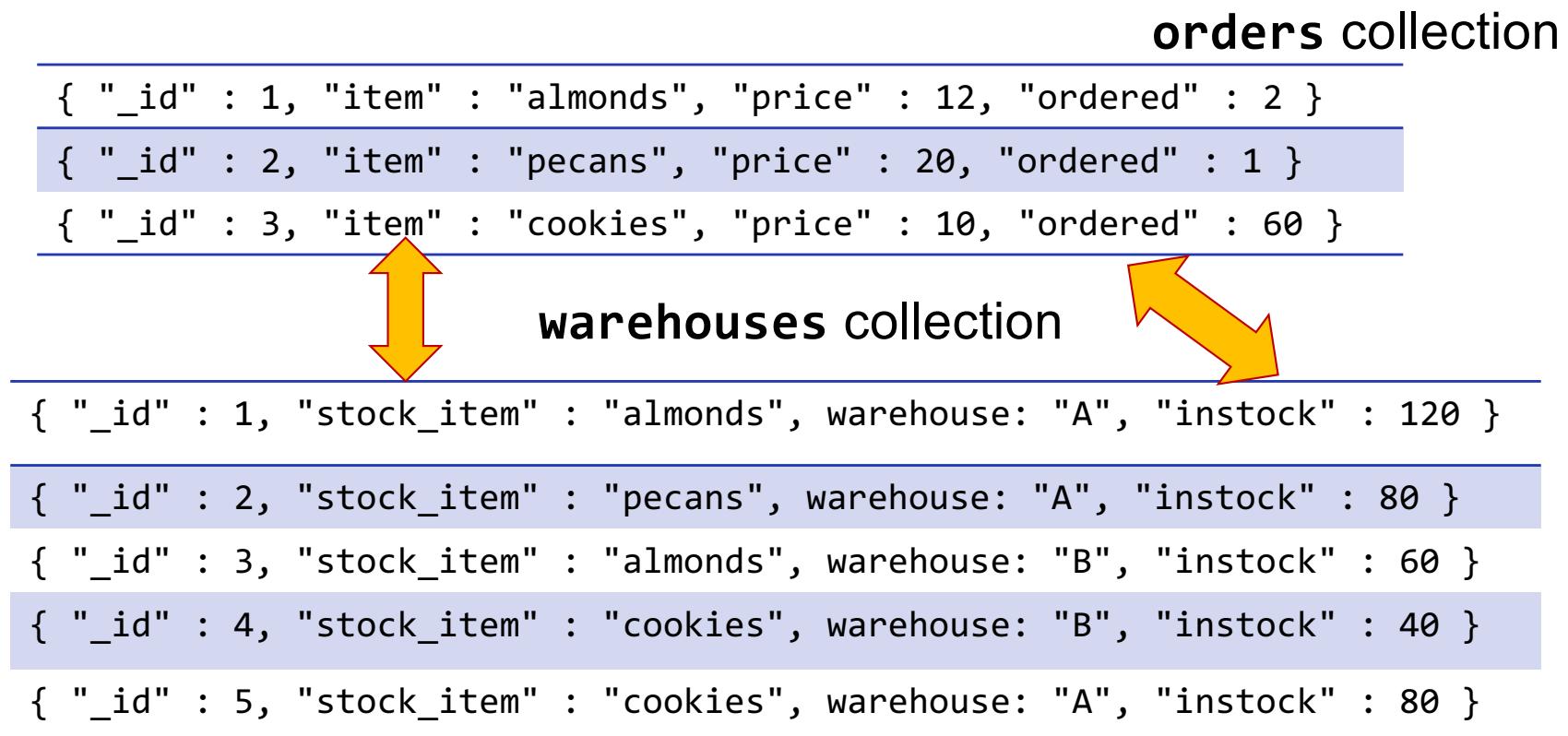
An ordered item may be stocked in multiple warehouses;

We want to find for each ordered item the warehouse with sufficient stock to cover the order

Multiple Joint Condition

- This query involves comparing two fields of the local and foreign documents:

- “item” in **orders** should match “stock_item” in **warehouses**
- “ordered” in **orders** should be less than or equal to “instock” in **warehouses**



Multiple Joint Condition \$lookup

```
db.orders.aggregate([
  {
    $lookup:
      {
        from: "warehouses",
        let: { order_item: "$item", order_qty: "$ordered" },
        pipeline: [
          { $match:
              { $expr:
                  { $and:
                      [
                        { $eq: [ "$stock_item", "$$order_item" ] },
                        { $gte: [ "$instock", "$$order_qty" ] }
                      ]
                  }
            }
          },
          { $project: { stock_item: 0, _id: 0 } }
        ],
        as: "stockdata"
      }
  }
])
```

This is the way to specify multiple condition

This is the way to let the pipeline access local fields: use variable **order_item** to access the local document's **item** field; use variable **order_qty** to access the local document's **ordered** field

variables are accessed using “\$\$” prefix

\$lookup by default includes the entire matched foreign document in the array, we can use \$project stage to get rid of some field

Matching document after the pipeline stage will be stored in this variable

Results

```
{ "_id" : 1,
  "item" : "almonds",
  "price" : 12,
  "ordered" : 2,
  "stockdata" : [
    { "warehouse" : "A", "instock" : 120 },
    { "warehouse" : "B", "instock" : 60 }
  ]
}
```

```
{ "_id" : 2,
  "item" : "pecans",
  "price" : 20,
  "ordered" : 1,
  "stockdata" : [
    { "warehouse" : "A", "instock" : 80 }
  ]
}
```

```
{ "_id" : 3,
  "item" : "cookies",
  "price" : 10,
  "ordered" : 60,
  "stockdata" : [
    { "warehouse" : "A", "instock" : 80 }
  ]
}
```

Uncorrelated Subquery Example

absences collection

```
{  
  "_id" : 1,  
  "student" : "Ann Aardvark",  
  "sickdays": [ "2018-05-01", 2018-08-23"]  
}
```

```
{  
  "_id" : 2,  
  "student" : "Zoe Zebra",  
  "sickdays": ["2018-02-01", 2018-05-23") ]  
}
```

holidays collection

```
{ "_id" : 1, year: 2018, name: "New Years", date: "2018-01-01" }  
{ "_id" : 2, year: 2018, name: "Pi Day", date: 2018-03-14" }  
{ "_id" : 3, year: 2018, name: "Ice Cream Day", date: "2018-07-15"}  
{ "_id" : 4, year: 2017, name: "New Years", date: "2017-01-01" }  
{ "_id" : 5, year: 2017, name: "Ice Cream Day", date: "2017-07-16"}
```

We want to include all 2018 public holidays in the **absences** collection

Uncorrelated Subquery \$lookup

```
db.absences.aggregate([
  {
    $lookup:
    {
      from: "holidays",
      pipeline: [
        { $match: { year: 2018 } },
        { $project: { _id: 0,
                     date: { name: "$name", date: "$date" } } },
        { $replaceRoot: { newRoot: "$date" } }
      ],
      as: "holidays"
    }
  }
])
```

The inner pipeline selects documents from **holidays** collection based on a condition unrelated with the local collection

The **\$project** and **\$replaceRoot** change the structure of the inner pipeline output documents

Has the same effect as: `{ $project: { _id: 0, year:0 } }` in this case

\$replaceRoot and similar “project” like stage are useful for promoting an embedded document at root level

Results

```
{  
    "_id" : 1,  
    "student" : "Ann Aardvark",  
    "sickdays: [ "2018-05-01", 2018-08-23"],  
    "holidays" : [  
        { "name" : "New Years", "date" : " 2018-01-01" ),  
        { "name" : "Pi Day", "date" : "2018-03-14" ),  
        { "name" : "Ice Cream Day", "date" : "2018-07-15" }  
    ]  
}  
  
{  
    "_id" : 2,  
    "student" : "Zoe Zebra",  
    "sickdays: [“2018-02-01”, 2018-05-23” ) ],  
    "holidays" : [  
        { "name" : "New Years", "date" : " 2018-01-01" ),  
        { "name" : "Pi Day", "date" : "2018-03-14" ),  
        { "name" : "Ice Cream Day", "date" : "2018-07-15" }  
    ]  
}
```

References

■ BSON types

- ▶ <https://docs.mongodb.com/manual/reference/bson-types/>

■ Querying for Null or Missing Field

- ▶ <https://docs.mongodb.com/manual/tutorial/query-for-null-fields/index.html>

■ Aggregation Pipelines

- ▶ <https://docs.mongodb.com/manual/core/aggregation-pipeline/>

■ Aggregation operators

- ▶ <https://docs.mongodb.com/manual/reference/operator/aggregation/>

COMP5338 – Advanced Data Models

Week 4: MongoDB Indexing

Dr. Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

■ Database Indexing

■ MongoDB Indexes

■ MongoDB Query Execution

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

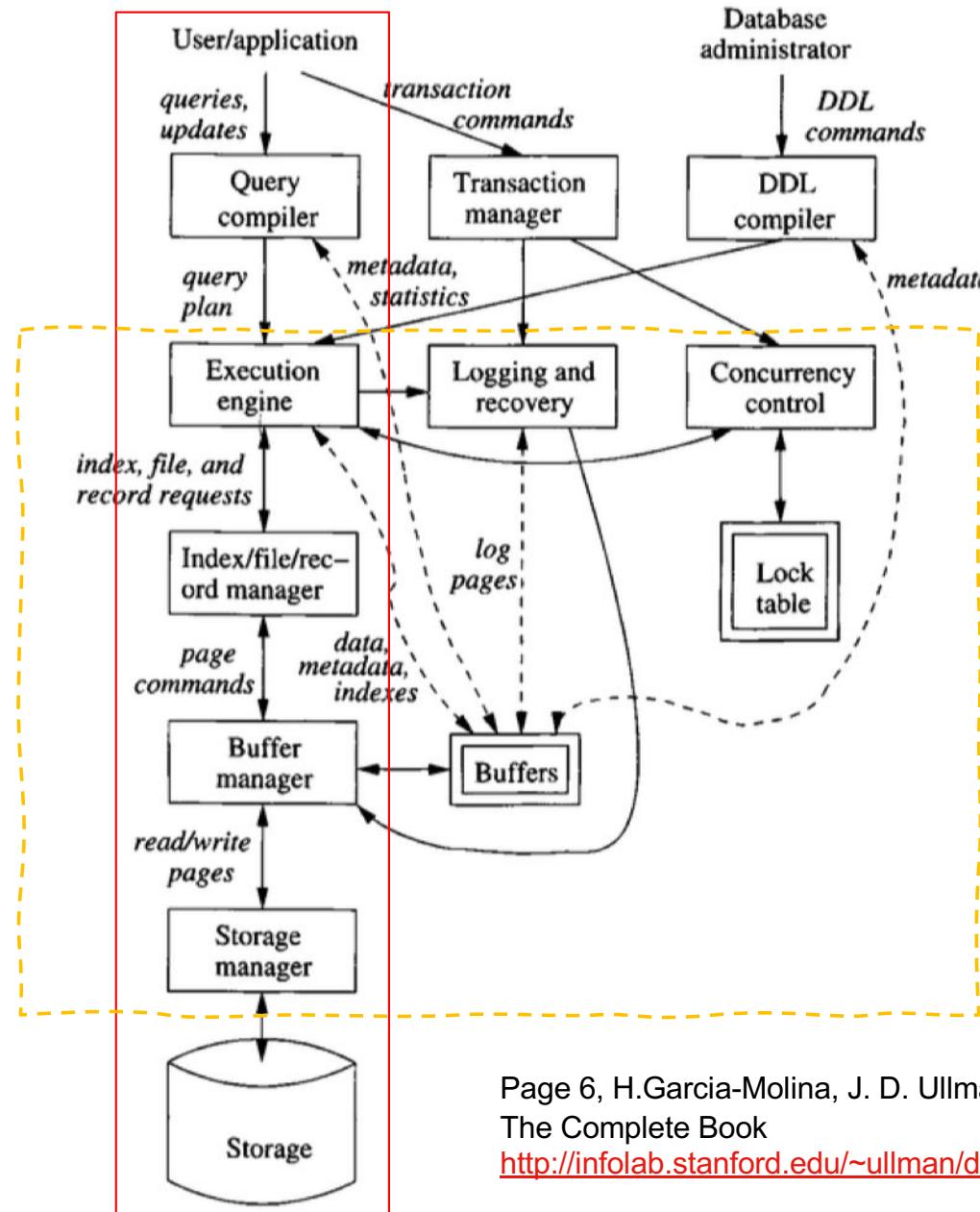
WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Review: DBMS Components



Disk based
database
system

Storage engine is
responsible for
managing how data
is store in memory
and disk

Page 6, H.Garcia-Molina, J. D. Ullman, J. Wildom, Databsae Systems
The Complete Book
<http://infolab.stanford.edu/~ullman/dscb.html>

The primitive operations of a query

- Read query
 - ▶ Load the element of interest from disk to main-memory buffer(s) if it is not already there
 - ▶ Read the content to client's address space
- Write query
 - ▶ The new value is created in the client's address space
 - ▶ It is copied to the appropriate buffers representing the database in the memory
 - ▶ The buffer content is flushed to the disk
- Both operations involve data movement between disk and memory and between memory spaces
- Typically disk access is the predominant performance cost in single node settings. Network communication contributes to the cost in cluster setting
- One design goal of database system is to reduce the amount of disk I/Os in read and write queries

Typical Solutions to minimize Disk I/O

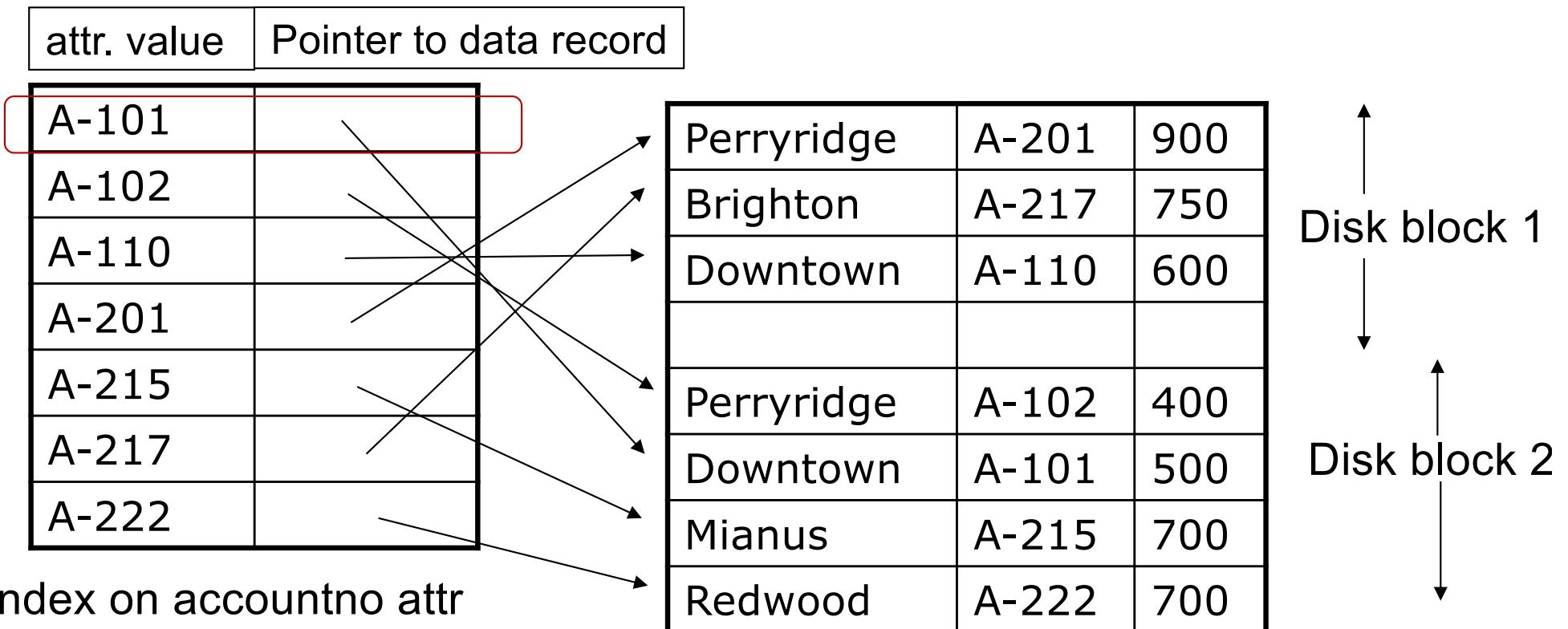
- Queries involve reading data from the database
 - ▶ Minimize the amount of data need to be moved from disk to memory
 - ▶ Use index and data distribution information to decide on a query plan
- Queries involve writing data to the database
 - ▶ Minimize the amount of disk I/O in the write path
 - Avoid flushing memory content to disk immediately after each write
 - Push non essential write out the of write path, e.g. do those asynchronously
 - ▶ To ensure durability, write ahead log/journal/operation log is always necessary
 - Appending to logs are much faster than updating the actual database file
 - The DB system may acknowledge once the data is updated in memory and appended in the WAL
 - Update to replicas can be done asynchronously, e.g. not in the write path

Textbook architecture of storage engine

- Data is stored in disk blocks with row-based format
- B-Tree primary and secondary indexes
- ACID transaction support
- Row based locking
- MVCC (multi-version concurrency control)

Indexing

- An index on an attribute **A** of a table is a data structure that makes it efficient to find those rows(document) that have a required value for attribute/field **A**.
- An index consists of records (called index entries) each of which has a value for the attribute(s) of the form



Indexing (con'td)

- Index entries are sorted by the attribute (search key) value
- Index files are typically much smaller than the original file

db.revisions.stats({scale:1024})		
🕒 0.005 sec.		
Key	Value	Type
▼ (1)	{ 11 fields }	Object
ns	wikipedia.revisions	String
count	623	Int32
size	188	Int32
avgObjSize	309	Int32
storageSize	204	Int32
capped	false	Boolean
wiredTiger	{ 13 fields }	Object
nindexes	1	Int32
totalIndexSize	28	Int32
▼ indexSizes	{ 1 field }	Object
id	28	Int32
ok	1.0	Double

Outline

- Database Indexing
- MongoDB Indexes
- MongoDB Query Execution

MongoDB Storage Engine

- MongoDB supports multiple storage engines
 - ▶ **WiredTiger** is the default one since version 3.2
- Some prominent features of WiredTiger
 - ▶ Provide both B-tree and Log Structured Merge tree index
 - ▶ Document level concurrency
 - ▶ Multi Version Concurrency Control (MVCC)
 - Snapshots are provided at the start of operation using timestamp
 - Snapshots are written to disk (creating checkpoints) at intervals of 60 seconds (or 2GB of journal data)
 - ▶ Journal
 - Write-ahead transaction log
 - ▶ Compression

MongoDB Basic Indexes

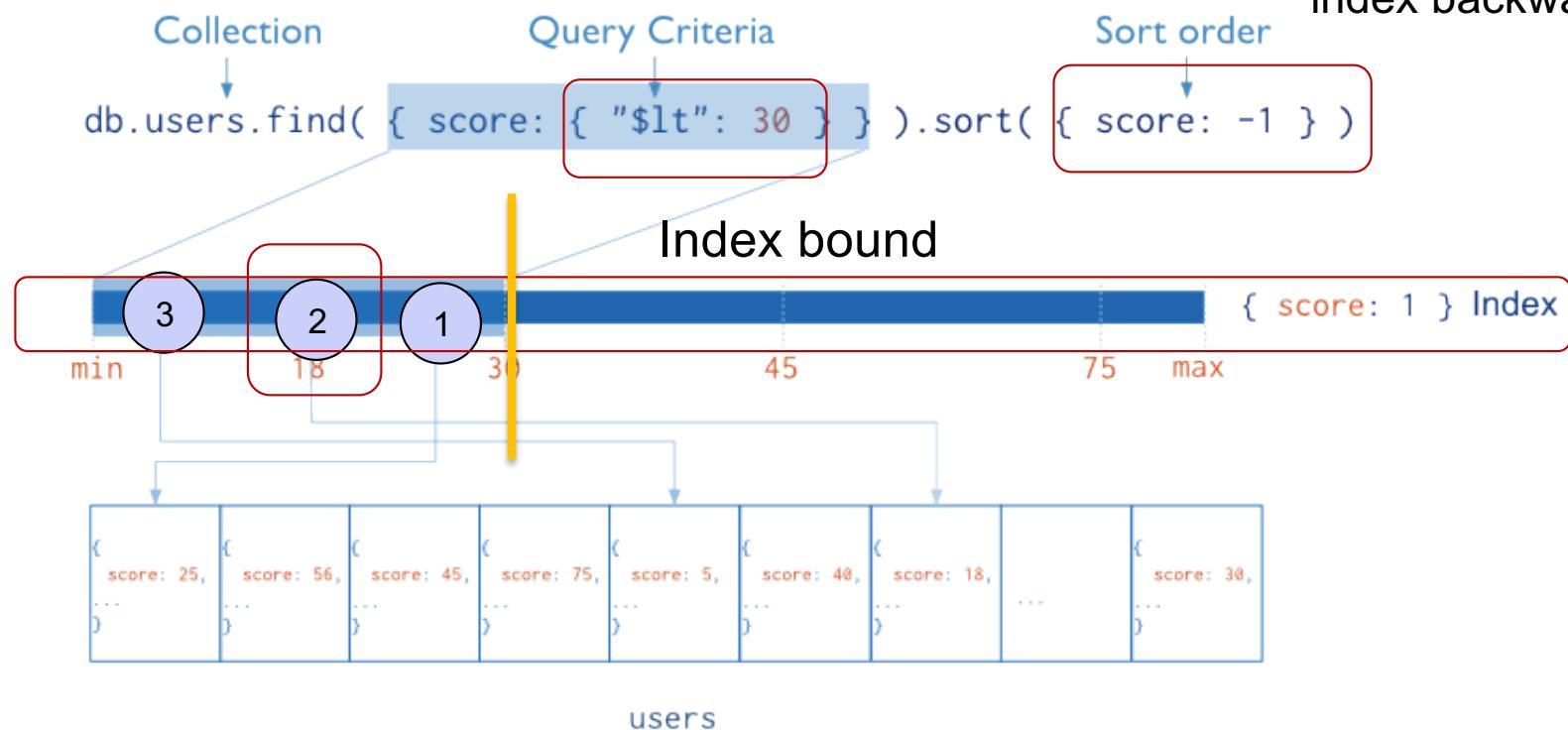
- The `_id` index
 - ▶ `_id` field is automatically indexed for all collections
 - ▶ The `_id` index enforces uniqueness for its keys
- Indexing on other fields
 - ▶ Index can be created on any other field or combination of fields
 - `db.<collectionName>.createIndex ({<fieldName>:1}) ;`
 - `fieldName` can be a simple field, array field or field of an embedded document (using dot notation)
 - `db.blog.createIndex({author:1})`
 - `db.blog.createIndex({tags:1})`
 - `db.blog.createIndex({"comments.author":1})`
 - the number specifies the direction of the index (1: ascending; -1: descending)
 - ▶ Additional properties can be specified for an index
 - **Sparseness, uniqueness, background, ..**
- Most MongoDB indexes are organized as **B-Tree** structure by default

<http://www.mongodb.org/display/DOCS/Indexes>

Single field Index

`db.users.createIndex({ score: 1 })`

Descending order means traversing the index backwards

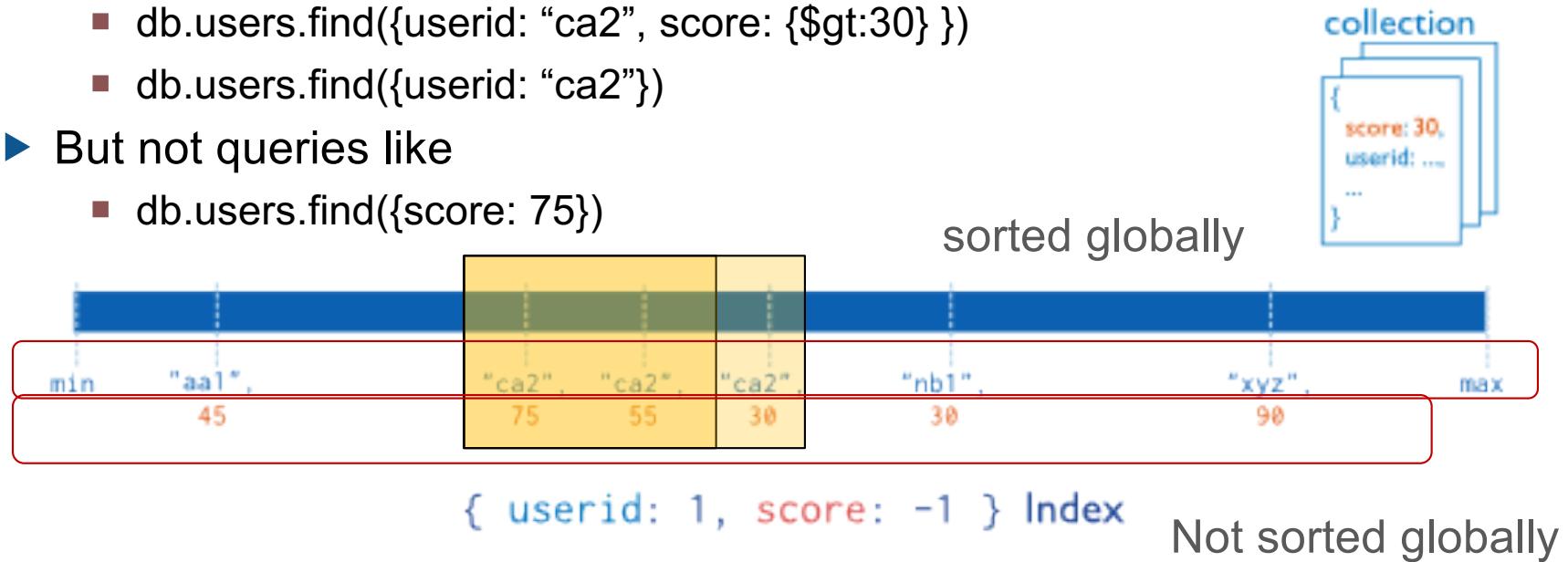


Single entry with filed value and pointer to document

<https://docs.mongodb.com/manual/core/index-single/>

Compound Index

- Compound Index is a single index structure that holds references to multiple fields within a collection
- The order of field in a compound index is very important
 - ▶ The index entries are sorted by the value of the first field, then second, third...
 - ▶ If we have a compound index: {userid:1, score:-1}
 - ▶ It supports queries like
 - db.users.find({userid: "ca2", score: {\$gt:30} })
 - db.users.find({userid: "ca2"})
 - ▶ But not queries like
 - db.users.find({score: 75})



<https://docs.mongodb.com/manual/core/index-compound/>

Use Index to Sort (single field)

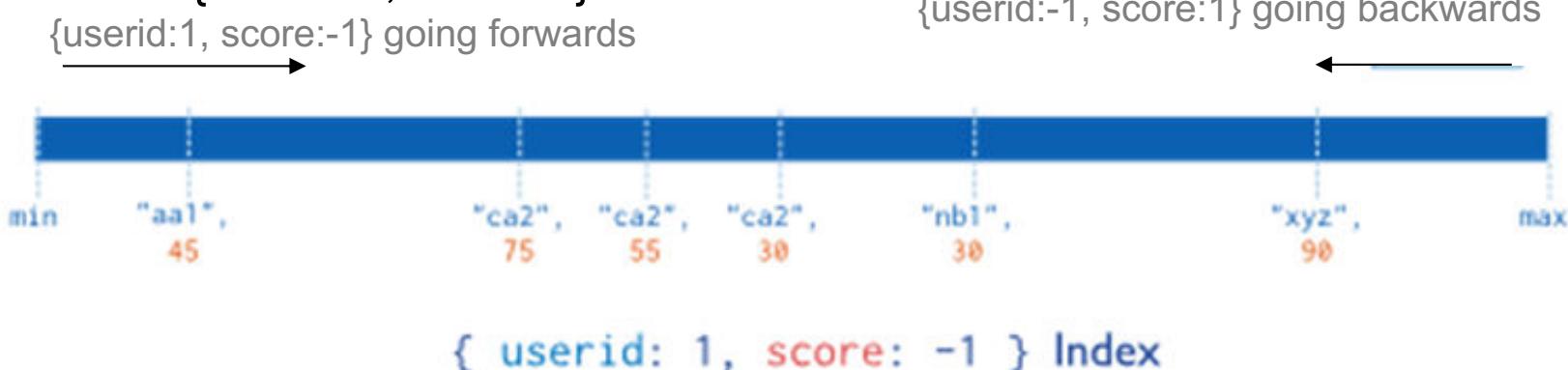
- Sort operation may obtain the order from index or sort the result in memory
- Index can be traversed in either direction
- Sort with a single field index
 - ▶ For single field index, sorting by that field can always use the index regardless of the sort direction
 - ▶ E.g. `db.records.createIndex({ a: 1 })` supports both
 - `db.records.find().sort({a:1})` and
 - `db.records.find().sort({a: -1})`

<https://docs.mongodb.com/manual/tutorial/sort-results-with-indexes/>

Use Index to Sort (multiple fields)

Sort on multiple fields

- ▶ Compound index may be used on sorting multiple fields.
- ▶ There are constraints on fields and direction
 - Sort key should have the same order as they appear in the index
 - All field sort have same sort direction, either going forwards or backwards the index
 - E.g. `{userid:1, score:-1}` and `{userid:-1, score:1}` can use the index, but not `{userid:1, score:1}`



Use Index to Sort (multiple fields)

■ Sort and Index Prefix

- ▶ If the sort keys correspond to the index keys or an *index prefix*, MongoDB can use the index to sort the query results.

- E.g. `db.data.createIndex({ a:1, b: 1, c: 1, d: 1 })`

- Supported query:

- `db.data.find().sort({ a: -1 })`

- `db.data.find().sort({ a: 1, b: 1 })`

- `db.data.find({a:{ $gt: 4}}).sort({ a: 1, b: 1 })`

Results can be obtained using the same index

■ Sort and Non-prefix Subset of an Index

- ▶ An index can support sort operations on a non-prefix subset of the index key pattern if the query include **equality** conditions on all the prefix keys that precede the sort keys. **Sort keys**

- `db.data.find({ a: 5 }).sort([b: 1, c: 1])`

- `db.data.find({ a: 5, b: { $lt: 3 } }).sort([b: 1])`

Equality condition on a

Equality condition on a

Sort key

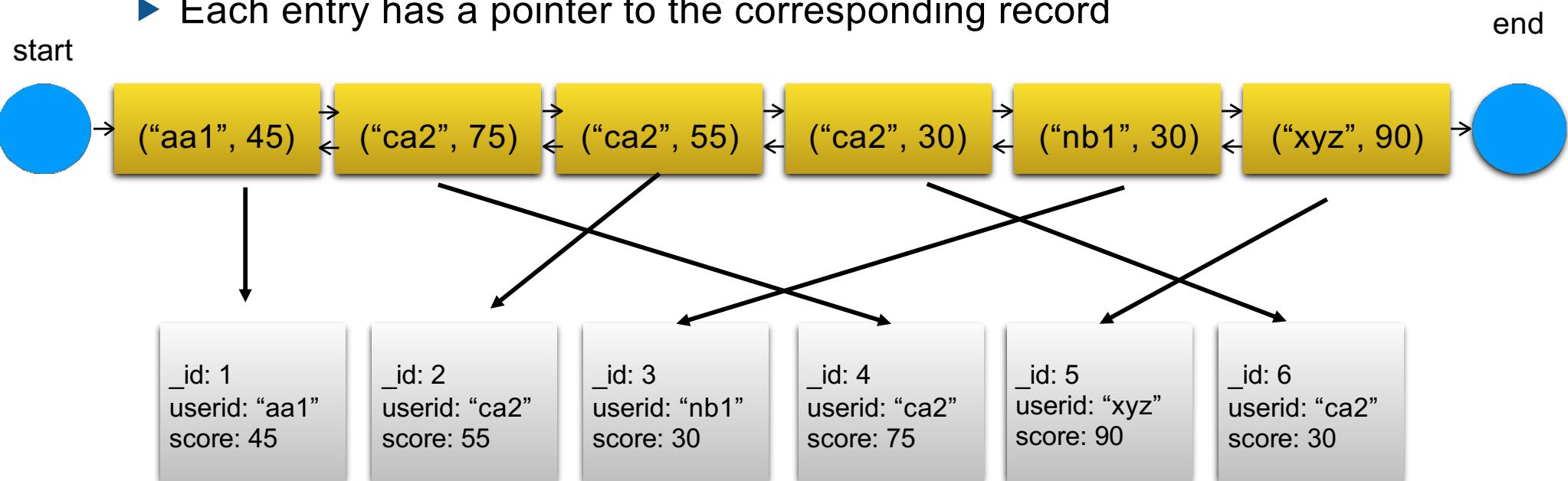
Running Example

- Suppose we have a **users** collection with the following 6 documents stored in the order of `_id` values

<code>_id: 1</code> userid: "aa1" score: 45	<code>_id: 2</code> userid: "ca2" score: 55	<code>_id: 3</code> userid: "nb1" score: 30	<code>_id: 4</code> userid: "ca2" score: 75	<code>_id: 5</code> userid: "xyz" score: 90	<code>_id: 6</code> userid: "ca2" score: 30
---	---	---	---	---	---

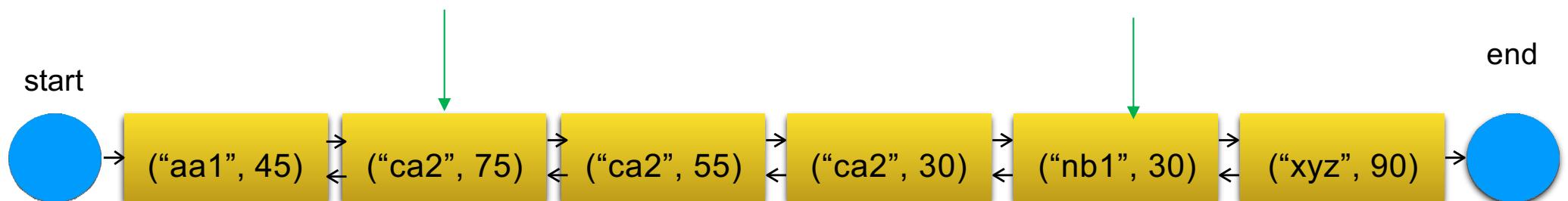
Index Entries

- Now we create a compound index on **userid** and **score** fields :
`db.users.createIndex(userid:1, score:-1)`
- With the current data, the index has six entries because we have 6 records in the collection
 - The entries are sorted in descending (**userid**, **score**) value
 - the index entries usually form a doubly linked list to facilitate bi-directional traversal
 - Each entry has a pointer to the corresponding record



Using index to find documents

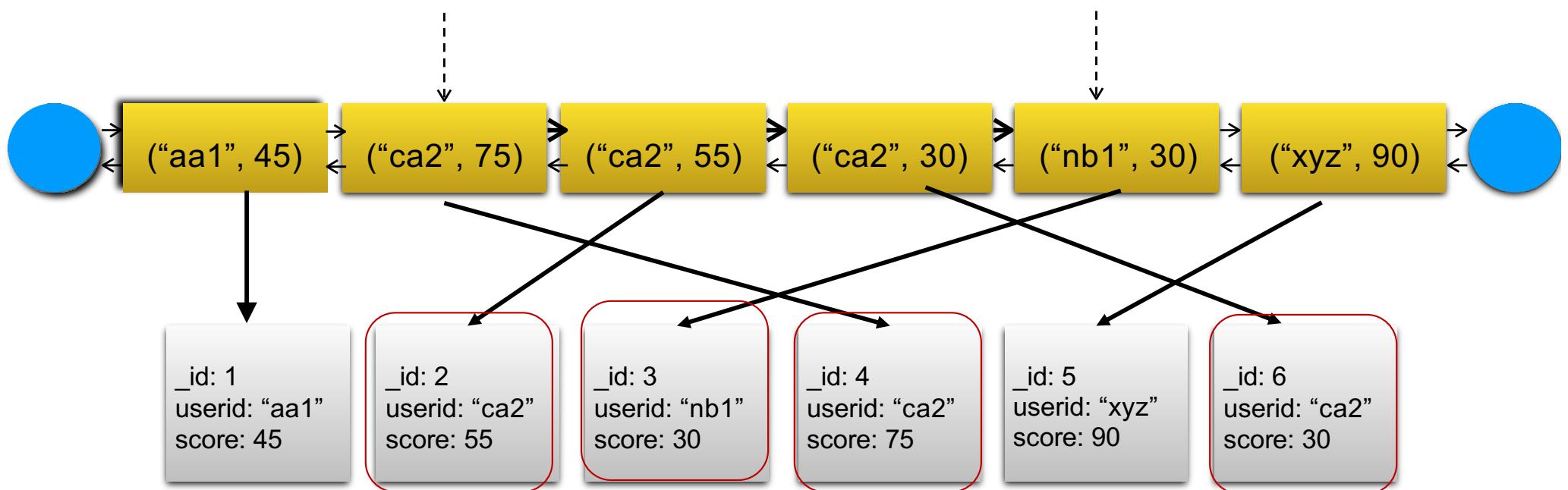
- For queries that are able to use index, the first step is to find the boundary entries on the list based on given query condition
- Example query
 - ▶ `db.users.find({userid:{$gt: "b", $lt:"s"}})`
- This query is able to use the compound index and the two bounds are:("ca1", 75) and ("nb1",30) inclusive at both ends



Using index to find documents

- The four documents with `_id` equals: 4, 2, 6 and 3 are the result of the above query

```
db.users.find({userid:{$gt: "b", $lt:"s"}})
```

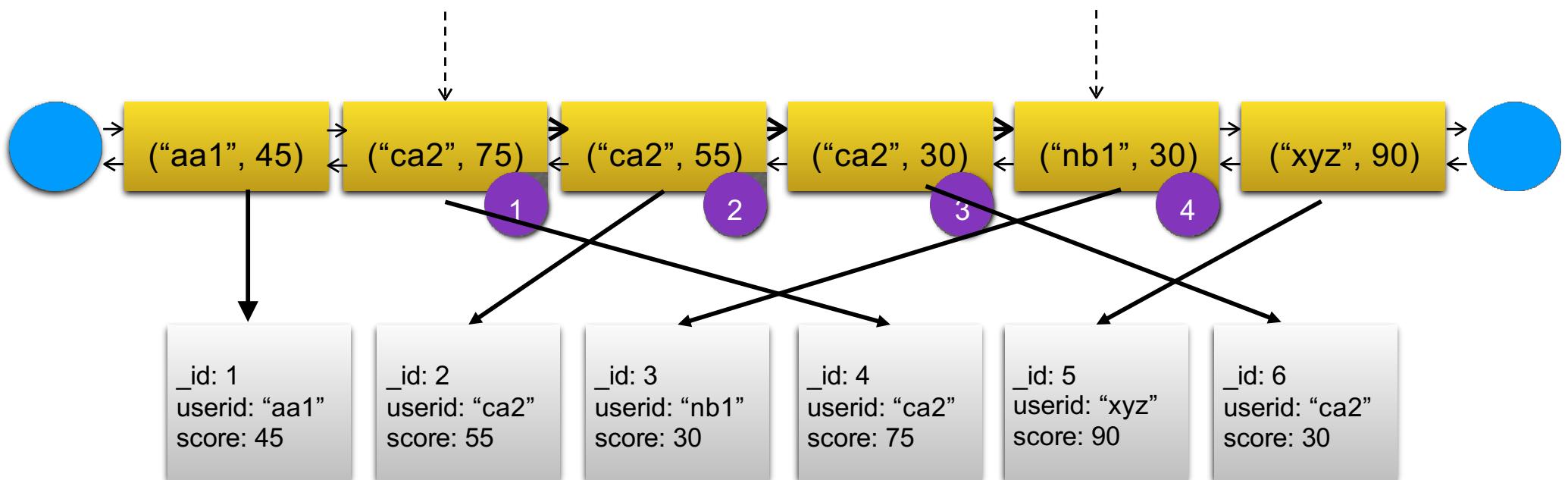


Using Index to sort

- If our queries include a sorting criteria

```
db.users.find(  
    {userid:{$gt: "b", $lt:"s"}},  
    ).sort({userid:1, score:-1})
```

- The engine will start from the **lower bound**, following the forward links to the **upper bound** and return all documents pointed by the entries



Sorting that cannot use index

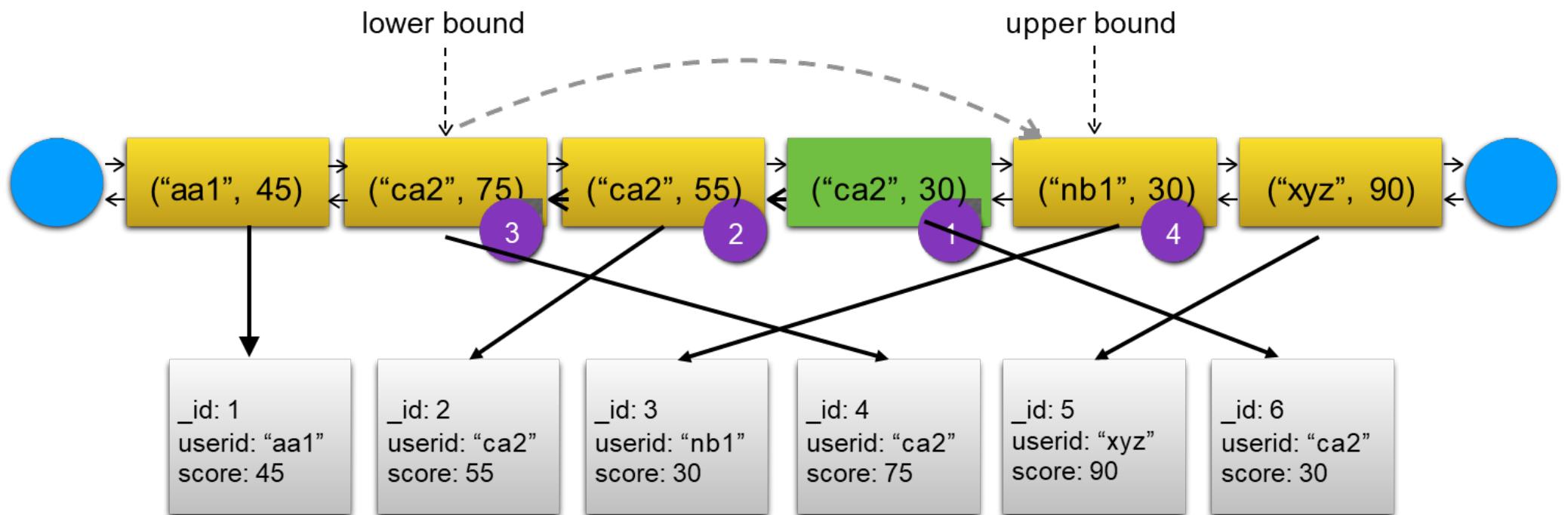
- If our query includes yet another sorting criteria

```
db.users.find(  
    {userid:{$gt: "b", $lt:"s"}  
}).sort({userid:1, score:1})
```

- We can still use the index to find the bounds and the four documents satisfying the query condition, but we are not able to follow a single forward or backward link to get the correct order of the documents

Sorting that cannot use index

- If we want to use the index entry list to obtain the correct, we would start from a mysterious position (“ca2”,30), follow the backward links to (“ca2”,75), and make a magic jump to the entry (“nb1”, 30).
 - complexity involved:
 - how do we find the start point in between lower and upper bound?
 - how do we decide when and where to jump in another direction?
 - The complexity of such algorithm makes it less optimal than a memory sort of the actual documents.

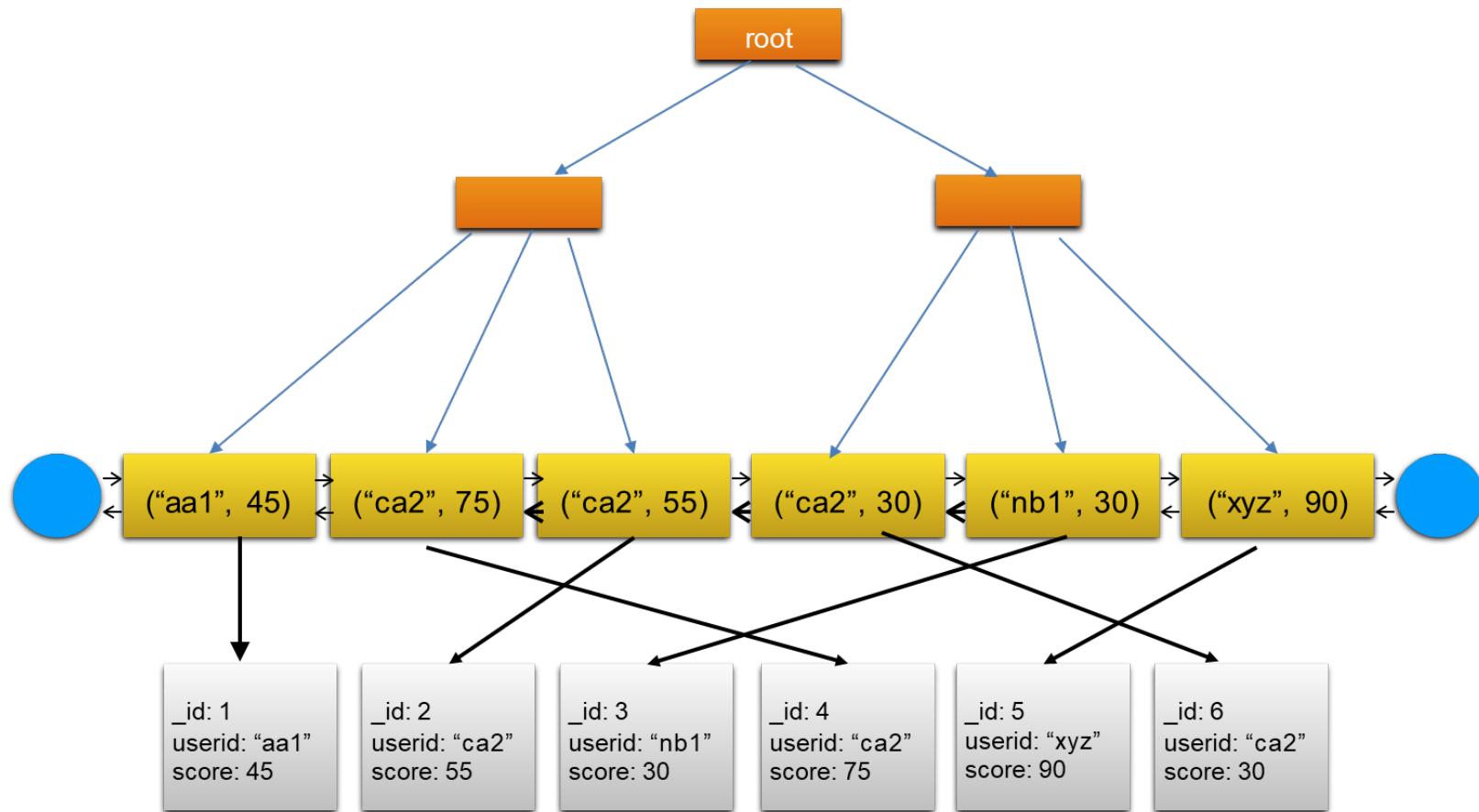


General rules

- If you are able to traverse the list between the upper and lower bounds as determined by your query condition in one direction to obtain the correct order as specified in the sort condition, the index will be used to sort the result
- Otherwise you may still use index to obtain the results but have to sort them in memory

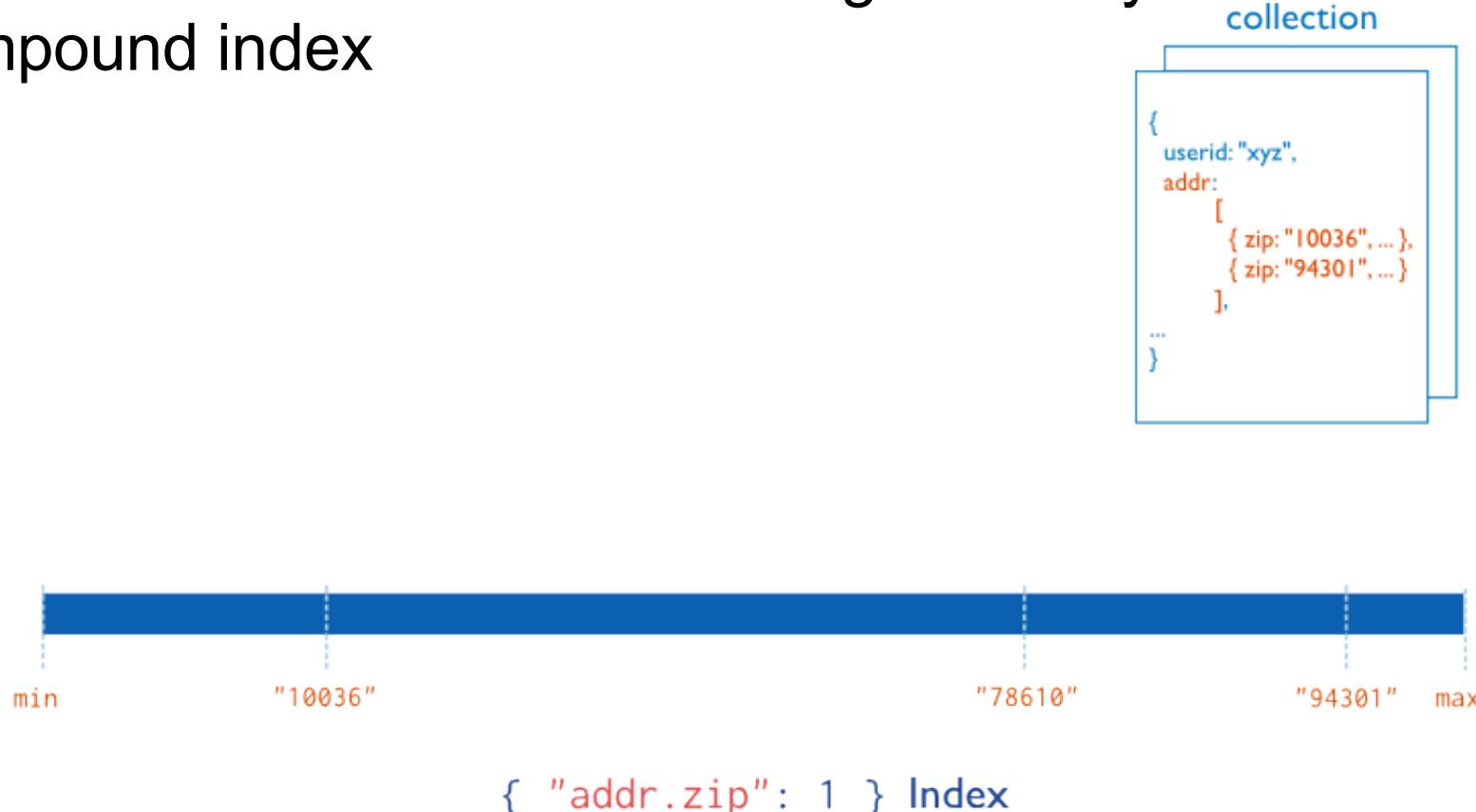
BTree motivation

- Finding the boundaries could be time consuming if we only have the list structure and can only start from one of the two ends
- B-Tree structure is built on top of the index values to accelerate the process of locating the boundary.



Multi key index

- Index can be created on array field, the key set includes each element in the array. It behaves the same as single index field otherwise
- There are restrictions on including multi key index in compound index



Text Indexes

- Text indexes support efficient text search of string content in documents of a collection
- To create a text index
 - ▶ `db.<collectionName>.createIndex({<fieldName>:'text'});`
 - ▶ text index tokenizes and stems the terms in the indexed fields for the index entries.
- To perform text query
 - ▶ `db.find($text:{$search:<search string>})`
 - No field name is specified
- Restrictions:
 - ▶ A collection can have at most one text index, but it can include text from multiple fields
 - ▶ Different field can have different weights in the index, results can be sorted using text score based on weights
 - ▶ Sort operations cannot obtain sort order from a text index

Other Indexes

■ Geospatial Index

- ▶ MongoDB can store and query spatial data in a flat or spherical surface
 - 2d indexes and 2dsphere indexes

■ Hash indexes

- ▶ Index the hash value of a field
- ▶ Only support equality match, but not range query
- ▶ Mainly used in hash based sharding

Indexing properties

- Similar to index in RDBMS, extra properties can be specified for index
- We can enforce the *uniqueness* of a field by create a unique indexes
 - ▶ `db.members.createIndex({ "user_id": 1 }, { unique: true })`
- We can reduce the index storage by specifying index as *sparse*
 - ▶ Only documents with the indexed field will have entries in the index
 - ▶ By default, non-sparse index contain entries for all documents. Documents without the indexed field will be considered as having `null` value.
- MongoDB also supports TTL indexes and partial index
 - ▶ Not all documents will be indexed, based on time or based on given condition

Indexing strategy

■ Indexing cost

- ▶ Storage, memory, write latency

■ Performance consideration

- ▶ In general, MongoDB only uses one index to fulfil specific queries
 - `$or` query on different fields may use different indexes
 - MongoDB may use intersection of multiple indexes
- ▶ When index fits in memory, you get the most performance gain

■ Build index if the performance gain can justify the cost

- ▶ Understand the query
- ▶ Understand the index behaviour

Outline

- Database Indexing
- MongoDB Indexes
- MongoDB Query Execution

Performance Monitoring Tools

■ Profiler

- ▶ Collects execution information about queries running on a database
- ▶ It can be used to identify various underperforming queries
 - Slowest queries
 - Queries not using any index
 - Queries running slower than some threshold
 - Custom tagged queries, e.g. by commenting
 - And more

■ Explain method

- ▶ Collect detailed information about a particular query
 - How the query is executed
 - What execution plans are evaluated
 - Detailed execution statistics, e.g. how many index entries or documents have been examined

<https://studio3t.com/knowledge-base/articles/mongodb-query-performance/>

MongoDB Query Execution

- MongoDB supports querying any field in a collection
 - ▶ Including non-existent field
- When index exists on a query field
 - ▶ It uses the index to find intermediate or final results
- Otherwise
 - ▶ It performs a full collection scan and exams every document to find the results
- When multiple indexes can be used for a query
 - ▶ The query optimizer evaluates different plans and determine the best one
 - Usually the one with high selectivity that can narrow the results most using index
- The `explain` method output many information about a particular query.

Using the explain method

- The method can be added on both **find** and **aggregate** command
- Explain find command:
 - ▶ db.collection.find(...).explain(...) or
 - ▶ db.collection.explain(...).find(...)
- Explain aggregation command:
 - ▶ db.collection.aggregate(...).explain(...)
 - ▶ db.collection.explain(...).aggregate(...)
- The Robo 3T shell only supports **explain after find** and **explain before aggregate**
- Other shell (e.g. VS + Mongo Ext) supports all four options.

Explain Verbosity Modes

- Showing only the query plan
 - ▶ Use it without any parameter
- Showing also the execution statistics of the chosen plan
 - ▶ `explain("executionStats")`
- Showing execution statistics of all candidate plans
 - ▶ `explain("allPlansExecution")`

```
1 db.revisions.find({"_id" : ObjectId("5f53204cc89636b3c92e0851")}).explain("executionStats")
```

⌚	0.002 sec.	
Key	Value	Type
▼ (1)	{ 4 fields }	Object
▶ queryPlanner	{ 6 fields }	Object
▶ executionStats	{ 6 fields }	Object
▶ serverInfo	{ 4 fields }	Object
## ok	1.0	Double

No Index Query Plan

1 db.revisions.find({random: {\$exists: true}}).explain("executionStats")	
⌚ 0.009 sec.	
Key	Value
▼ (1)	{ 4 fields }
▼ queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisions
indexFilterSet	false
► parsedQuery	{ 1 field }
► winningPlan	{ 3 fields }
stage	COLLSCAN ←
filter	{ 1 field }
random	{ 1 field }
\$exists	true
direction	forward
► rejectedPlans	[0 elements]
► executionStats	{ 6 fields }
executionSuccess	true
nReturned	0
executionTimeMillis	5
totalKeysExamined	0
totalDocsExamined	623
► executionStages	{ 13 fields }
► serverInfo	{ 4 fields }
ok	1.0

There is only one candidate plan: whole collection scan!

And every document will be checked to see if a field called **random** exists

The query searches for documents with a field “random”, The current collection does not have any document with that field

Two Collections with same data

```
db.revisions.aggregate(  
  [  
    {$out: "revisionsWI"}  
  ]  
)
```

```
db.revisionsWI.createIndex({user:1})  
db.revisionsWI.createIndex({timestamp:1})  
db.revisionsWI.createIndex({title:1})  
db.revisionsWI.createIndex({parsedcomment:"text"})
```

Query with one indexed field

1 db.revisionsWI.find({user: "Muboshgu"}, { title: 1}).explain("executionStats")	
⌚ 0.001 sec.	
Key	Value
▼ (1)	{ 4 fields }
queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisionsWI
indexFilterSet	false
parsedQuery	{ 1 field }
winningPlan	{ 3 fields }
stage	PROJECTION_SIMPLE
transformBy	{ 1 field }
title	1.0
inputStage	{ 2 fields }
stage	FETCH
inputStage	{ 11 fields }
stage	IXSCAN
keyPattern	{ 1 field }
indexName	user_1
isMultiKey	false
multiKeyPaths	{ 1 field }
isUnique	false
isSparse	false
isPartial	false
indexVersion	2
direction	forward
indexBounds	{ 1 field }
user	[1 element]
[0]	["Muboshgu", "Muboshgu"]
	2nd stage, projection
	1st stage index based search
	executionStats { 6 fields }
	executionSuccess true
	nReturned 5
	executionTimeMillis 0
	totalKeysExamined 5
	totalDocsExamined 5
	totalBytesExamined 140.6112

Multiple Query Plans

```
db.revisionsWI.find(  
  {  
    "title": "Donald_Trump",  
    "user": "ThiefOfBagdad"  
  }  
).explain(["executionStats"])
```

The query document contains two condition on two fields

There are three possible plans to execute the query

We can find all revision documents for “Donald Trump” page using the **title** index; then check if the revision is made by “ThiefOfBagdad”

We can find all revision documents made by “ThiefOfBagdad” using the **user** index; then check if the title is “Donald Trump”

We can find revision documents made by “ThiefOfBagdad” using the **user** index; then find revision documents for “Donald Trump” page using the **title** index. The intersection of these two sets is the query result.

Query Plans by MongoDB

```
1 db.revisionsWI.find(  
2   {  
3     "title": "Donald_Trump",  
4     "user": "ThiefOfBagdad"  
5   }  
6 ).explain("executionStats")
```

⌚ 0.001 sec.

Key	Value
▼ (1)	{ 4 fields }
▼ queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisionsWI
indexFilterSet	false
► parsedQuery	{ 1 field }
► winningPlan	{ 3 fields }
► rejectedPlans	[2 elements]

Winning Plan

```
1 db.revisionsWI.find(  
2   {  
3     "title": "Donald_Trump",  
4     "user": "ThiefOfBagdad"  
5   }  
6 ).explain("executionStats")
```

Key	Value	
winningPlan	{ 3 fields }	
stage	FETCH	
filter	{ 1 field }	Then filter by title
title	{ 1 field }	
inputStage	{ 11 fields }	
stage	IXSCAN	Search user index
keyPattern	{ 1 field }	
indexName	user_1	
isMultiKey	false	
multiKeyPaths	{ 1 field }	
isUnique	false	
isSparse	false	
isPartial	false	
indexVersion	2	Index bound is the given user value
direction	forward	
indexBounds	{ 1 field }	
user	[1 element]	
[0]	["ThiefOfBagdad", "ThiefOfBagdad"]	

First Rejected Plan

▼ [1] rejectedPlans	[2 elements]
▼ [0]	{ 3 fields }
stage	FETCH
▼ filter	{ 1 field }
▼ user	{ 1 field }
\$eq	ThiefOfBagdad
▼ inputStage	{ 11 fields }
stage	IXSCAN
► keyPattern	{ 1 field }
indexName	title_1
isMultiKey	false
► multiKeyPaths	{ 1 field }
isUnique	false
isSparse	false
isPartial	false
indexVersion	2
direction	forward
▼ indexBounds	{ 1 field }
▼ title	[1 element]
[0]	["Donald_Trump", "Donald_Trump"]

3. the second step is to check the user field value of documents returned from index scan using

1. The Input stage uses index on title field

2. Index value used is "Donald_Trump"

Second Rejected Plan

▼ [L] rejectedPlans	[2 elements]
► [L] [0]	{ 3 fields }
▼ [L] [1]	{ 3 fields }
└ stage	FETCH
└ filter	{ 1 field }
└ \$and	[2 elements]
└ inputStage	{ 2 fields }
└ stage	AND_SORTED
└ inputStages	[2 elements]
▼ [L] [0]	{ 11 fields }
└ stage	IXSCAN
└ keyPattern	{ 1 field }
└ indexName	user_1
└ isMultiKey	false
└ multiKeyPaths	{ 1 field }
└ isUnique	false
└ isSparse	false
└ isPartial	false
└ indexVersion	2
└ direction	forward
└ indexBounds	{ 1 field }
▼ [L] [1]	{ 11 fields }
└ stage	IXSCAN
└ keyPattern	{ 1 field }
└ indexName	title_1
└ isMultiKey	false
└ multiKeyPaths	{ 1 field }
└ isUnique	false
└ isSparse	false
└ isPartial	false
└ indexVersion	2
└ direction	forward
└ indexBounds	{ 1 field }



Candidate Plan Selection

- Theoretically it depends on selectivity of plan
 - ▶ Using user index can narrow the result to 68 documents, while using title index can only narrow the result to 434 documents
- How does the database build such knowledge
 - ▶ Using various statistics to calculate cost (most RDBMS)
 - ▶ Run each plan partially to estimate cost and cache the plan for future use (MongoDB's current approach)

Check index usage on sort

- Index usage on SORT is not *explicitly* included in `explain()` result and the indication of sort usage in the `explain()` result varies in different MongoDB version
- In the current version, the inclusion of a stage called SORT means we cannot obtain sort order from index and sort need to be handled separately

Sort not supported by index

```
1 db.revisionsWI.find(  
2 {  
3   "user": "ThiefOfBagdad"  
4 }  
5 ).sort({"timestamp":1}).explain("executionStats")
```

0.001 sec.

Key	Value
queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisionsWI
indexFilterSet	false
parsedQuery	{ 1 field }
winningPlan	{ 3 fields }
stage	SORT
sortPattern	{ 1 field }
timestamp	1.0
inputStage	{ 2 fields }
stage	SORT_KEY_GENERATOR
inputStage	{ 2 fields }
stage	FETCH
inputStage	{ 11 fields }
stage	IXSCAN
keyPattern	{ 1 field }
indexName	user_1
isMultiKey	false
multiKeyPaths	{ 1 field }
isUnique	false
isSparse	false
isPartial	false
indexVersion	2
direction	forward
indexBounds	{ 1 field }
user	[1 element]
[0]	["ThiefOfBagdad", "ThiefOfBagdad"]

Search field is
'user', sort field is
'timestamp'

Extra SORT stage
in the query plan,
Sort by 'timestamp'

The results to be
sorted are obtained
using index on
'user' field.

Index
entry to
be
examined

Sort not supported by index

```
1 db.revisionsWI.find(  
2   {  
3     "user": "ThiefOfBagdad"  
4   }  
5 ).sort({"timestamp":1}).explain("executionStats")
```



0.001 sec.

Key	Value
▼ (1)	{ 4 fields }
► queryPlanner	{ 6 fields }
▼ executionStats	{ 6 fields }
executionSuccess	true
nReturned	68
executionTimeMillis	0
totalKeysExamined	68
totalDocsExamined	68
▼ executionStages	{ 14 fields }
stage	SORT
nReturned	68
executionTimeMillisEstimate	0
works	140
advanced	68
needTime	70
needYield	0
saveState	2
restoreState	2
isEOF	1
► sortPattern	{ 1 field }
memUsage	17298
memLimit	33554432
► inputStage	{ 11 fields }

Extra execution stage
for sort and its
memory usage

No sort comparison

```
1 db.revisionsWI.find(  
2   {  
3     "user": "ThiefOfBagdad"  
4   }  
5 ).explain("executionStats")
```

Key	Value
⌚ 0.001 sec.	
queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisionsWI
indexFilterSet	false
parsedQuery	{ 1 field }
winningPlan	{ 2 fields }
stage	FETCH
inputStage	{ 11 fields }
rejectedPlans	[0 elements]
executionStats	{ 6 fields }
executionSuccess	true
nReturned	68
executionTimeMillis	0
totalKeysExamined	68
totalDocsExamined	68
executionStages	{ 13 fields }
stage	FETCH
nReturned	68
executionTimeMillisEstimate	0
works	69
advanced	68
needTime	0
needYield	0
saveState	0
restoreState	0
isEOF	1
docsExamined	68
alreadyHasObj	0
inputStage	{ 24 fields }

No sort modifier
in the query,
only search by
'user' field

Only a FETCH stage

SORT supported by index

```
1 db.revisionsWI.find(  
2   {  
3     "title": "Donald_Trump",  
4     "timestamp":{  
5       $gte: ISODate("2016-07-01"),  
6       $lte: ISODate("2016-07-02")  
7     }  
8   }  
9 ).sort({"timestamp":1}).explain("executionStats")
```

⌚ 0.001 sec.	
Key	Value
⌚ (1)	{ 4 fields }
queryPlanner	{ 6 fields }
plannerVersion	1
namespace	wikipedia.revisionsWI
indexFilterSet	false
parsedQuery	{ 1 field }
winningPlan	{ 3 fields }
stage	FETCH
filter	{ 1 field }
title	{ 1 field }
\$eq	Donald_Trump
inputStage	{ 11 fields }
stage	IXSCAN
keyPattern	{ 1 field }
indexName	timestamp_1
isMultiKey	false
multiKeyPaths	{ 1 field }
isUnique	false
isSparse	false
isPartial	false
indexVersion	2
direction	forward
indexBounds	{ 1 field }
timestamp	[1 element]
[0]	[new Date(1467331200000), new Date(1467417600000)]
rejectedPlans	[1 element]

Search fields are 'title' and 'timestamp'
sort field is 'timestamp'

Theoretically, if we use index on "timestamp" to find the results, we can obtain sort order from the index

This is the chosen plan with a FETCH stage. No extra sort is needed

In fact, the winning plan is exactly the same as the one without sort modifier. Try it yourself.

Only one rejected plan in this case

SORT supported by index (rejected plan)

```
1 db.revisionsWI.find(  
2   {  
3     "title": "Donald_Trump",  
4     "timestamp":{  
5       $gte: ISODate("2016-07-01"),  
6       $lte: ISODate("2016-07-02")  
7     }  
8   }  
9 ).sort({"timestamp":1}).explain("executionStats")
```

⌚ 0.001 sec.	
Key	Value
▶ [x] winningPlan	{ 3 fields }
▼ [x] rejectedPlans	[1 element]
▼ [x] [0]	{ 3 fields }
[!][x] stage	SORT
▼ [x] sortPattern	{ 1 field }
## timestamp	1.0
▼ [x] inputStage	{ 2 fields }
[!][x] stage	SORT_KEY_GENERATOR
▼ [x] inputStage	{ 3 fields }
[!][x] stage	FETCH
► [x] filter	{ 1 field }
▼ [x] inputStage	{ 11 fields }
[!][x] stage	IXSCAN
► [x] keyPattern	{ 1 field }
## indexName	title_1
[T/F] isMultiKey	false
► [x] multiKeyPaths	{ 1 field }
[T/F] isUnique	false
[T/F] isSparse	false
[T/F] isPartial	false
## indexVersion	2
[!][x] direction	forward
► [x] indexBounds	{ 1 field }

Use index on
“title” to find the
result, and sort
based on
“timestamp”

Index Usage in Aggregation Pipeline

- Index can be used in some pipeline stages under certain conditions:
 - ▶ `$match` stage if it is the first in the pipeline
 - ▶ `$sort` stage if the original document has not been changed, e.g. there is no `$project`, `$unwind` or `$group` stage in front
 - ▶ `$group` stage if the grouping key is sorted right before and if the grouping accumulator is `$first`
 - ▶ A few other stages under respective conditions
- The output of explain on information on pipe stage is limited
- See lab question for details

<https://docs.mongodb.com/manual/core/aggregation-pipeline/#aggregation-pipeline-operators-and-performance>

References

- MongoDB documentation on indexes
 - ▶ <https://docs.mongodb.com/manual/indexes/>
- MongoDB documentation on **explain()** method
 - ▶ <https://docs.mongodb.com/manual/reference/explain-results/>

COMP5338 – Advanced Data Models

Week 5: MongoDB – Replication and Sharding

Dr. Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

■ Replication

- ▶ **Replica Set**
- ▶ **Write Concern**
- ▶ **Read Preference**
- ▶ **Read Concern**

■ Sharding

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Replication in MongoDB

- MongoDB uses replication to achieve durability, availability and/or read scalability.
- A basic replication component in MongoDB is called a **replica set**
- A replica set contains several data bearing nodes and optionally one **arbiter** node.
- The data bearing nodes are organized following traditional master/slave replication mechanism
 - ▶ One primary and one to multiple secondary members
- The arbiter node does not contain data, it is only used for primary election to establish majority

Primary and Secondary Members

Primary and secondary contain the same data set

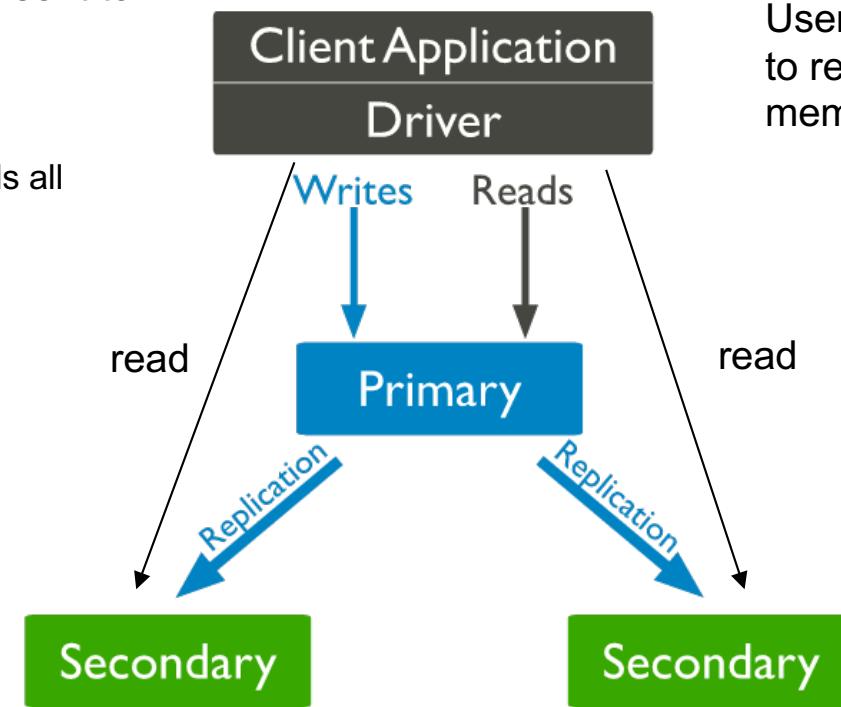
By default all reads/writes are sent to primary member only

In write operation, primary records all changes to its data set in `oplog` (operation log)

The operation log is then sent to secondary member

Secondary member uses the operation log to update its own data set

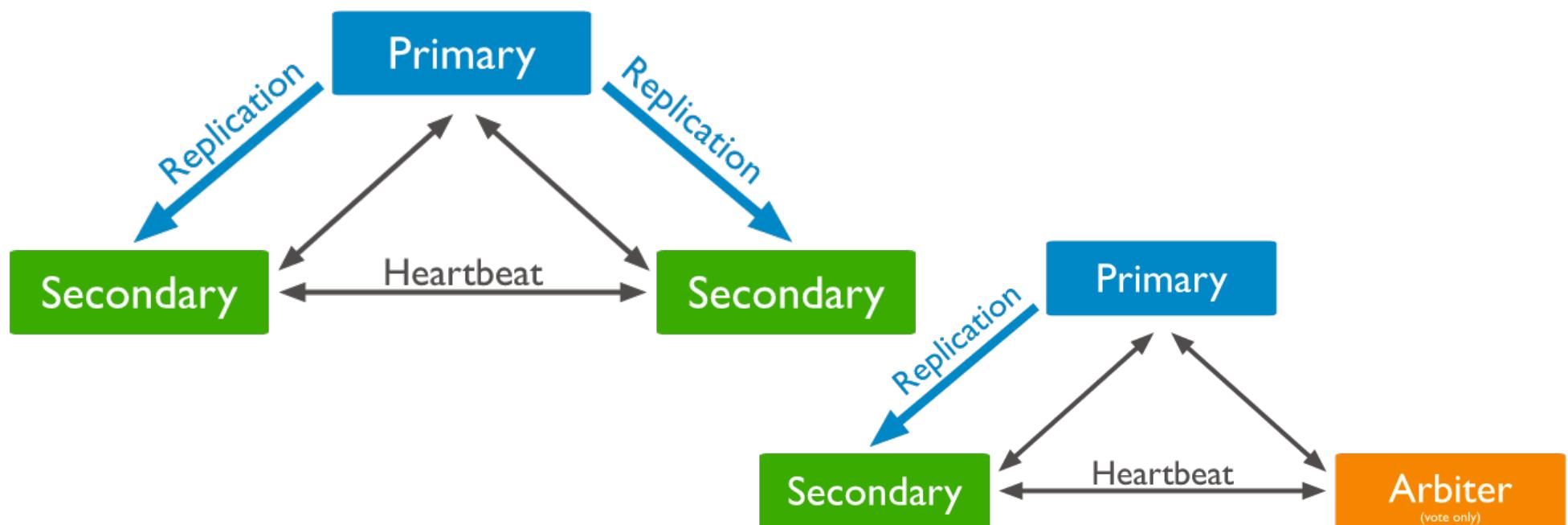
User may indicate that it is safe to read from secondary member;



There is replication lag between primary and secondary. The data set on secondary may not reflect the latest change

Member Communication

- Replica set members send heartbeats (pings) to each other every two seconds. If a heartbeat does not return within 10 seconds, the other members mark the delinquent member as inaccessible.



<https://docs.mongodb.com/manual/core/replica-set-elections/#replica-set-elections>

Fault Tolerance on Primary

- When a primary is deemed inaccessible, other members will hold a primary election
- The replica set cannot process write operations until the election completes successfully.
 - ▶ One secondary is elected as the primary
- Network Partition
 - ▶ A network partition may segregate a **primary** into a partition with a *minority* of nodes.
 - ▶ When the primary detects that it can only see a minority of nodes in the replica set, the primary steps down as primary and becomes a secondary.
 - ▶ Independently, a member in the partition that can communicate with a majority of the nodes (including itself) holds an election to become the new primary.

Replica Set Read/Write Semantics

- By default, read operations are answered by the primary member
- Client can specify “**Read Preference**” to read from different members
 - ▶ Primary(default), secondary, nearest, etc
- Client can also specify “**Read Concern**” to indicate the consistency level they expect.
- By default, write operation is considered successful when it is written on the primary member
- To maintain consistency requirements, client can specify different levels of “**Write Concern**”

Write Concern

- “**Write concern** describes *the level of acknowledgment* requested from MongoDB for write operations to a **standalone mongod** or to **replica sets** or to **sharded clusters**.”
- The specification is attached to a write query with the format
 - ▶ { **w**: <value>, **j**: <boolean>, **wtimeout**: <number> }
 - **w**: how many or whose acknowledgements receive before replying clients
 - The value can be a number, ‘majority’ or custom name
 - **j**: send the acknowledgement before or after logging to disk
 - true, false or unspecified
 - **wtimeout** : time limit in ms to prevent write operations from blocking indefinitely
- The complete signature of updateMany is:

```
db.collection.updateMany(  
  <filter>,  
  <update>,  
  {  
    upsert: <boolean>,  
    writeConcern: <document>,  
    collation: <document>,  
    arrayFilters: [ <filterdocument1>, ... ],  
    hint: <document|string>      // Available starting in MongoDB 4.2.1  
  }  
)
```

Write Concern Specification

- Write concern values control how soon a write request will be returned to the client
- Remember any database write operation involves
 - ▶ In memory update
 - ▶ log appending
 - ▶ data file update
 - happens after the write acknowledgement in MongoDB
- The write concern specifies
 - ▶ when should each db instance send acknowledgement
 - controlled by **j** value
 - ▶ How many db instances should acknowledge before acknowledging the client
 - controlled by **w** value
 - ▶ The maximum time client should wait for an acknowledgement.
 - Controlled by **wttimeout** value

Write Acknowledge Behavior

- The meaning of `j` value
 - ▶ true: the db instance waits until log is successfully appended to send acknowledgement
 - ▶ false: the db instance sends acknowledge once the memory is updated
 - ▶ *Unspecified*: the meaning depends on db environment and other setting
- The meaning of `w` value
 - ▶ 0: no acknowledgement is required
 - ▶ 1: the standalone db instance need to acknowledge or the primary member needs to acknowledge
 - ▶ > 1: only applicable in replica set, require the specified member(s) to acknowledge
 - ▶ ‘majority’: require majority of the members in replica set to acknowledge; or it can override `j` setting in standalone setting.

Write Concern Behaviour

- Default write concern is {w=1}
 - ▶ Standalone: acknowledge after in memory update
 - ▶ Replica set: acknowledge after in memory update in primary
- **wtimeout** is only applicable for w values greater than 1
- **wtimeout** causes write operations to return with an error after the specified limit, **but** it does not necessarily mean the write operation is failed. It only indicates that the required number of acknowledgements are not received within the limit.
 - ▶ MongoDB will not perform any roll back
 - ▶ Applications should not interpret time out error as fail and act accordingly
- Unspecified wtimeout may block the write operation indefinitely

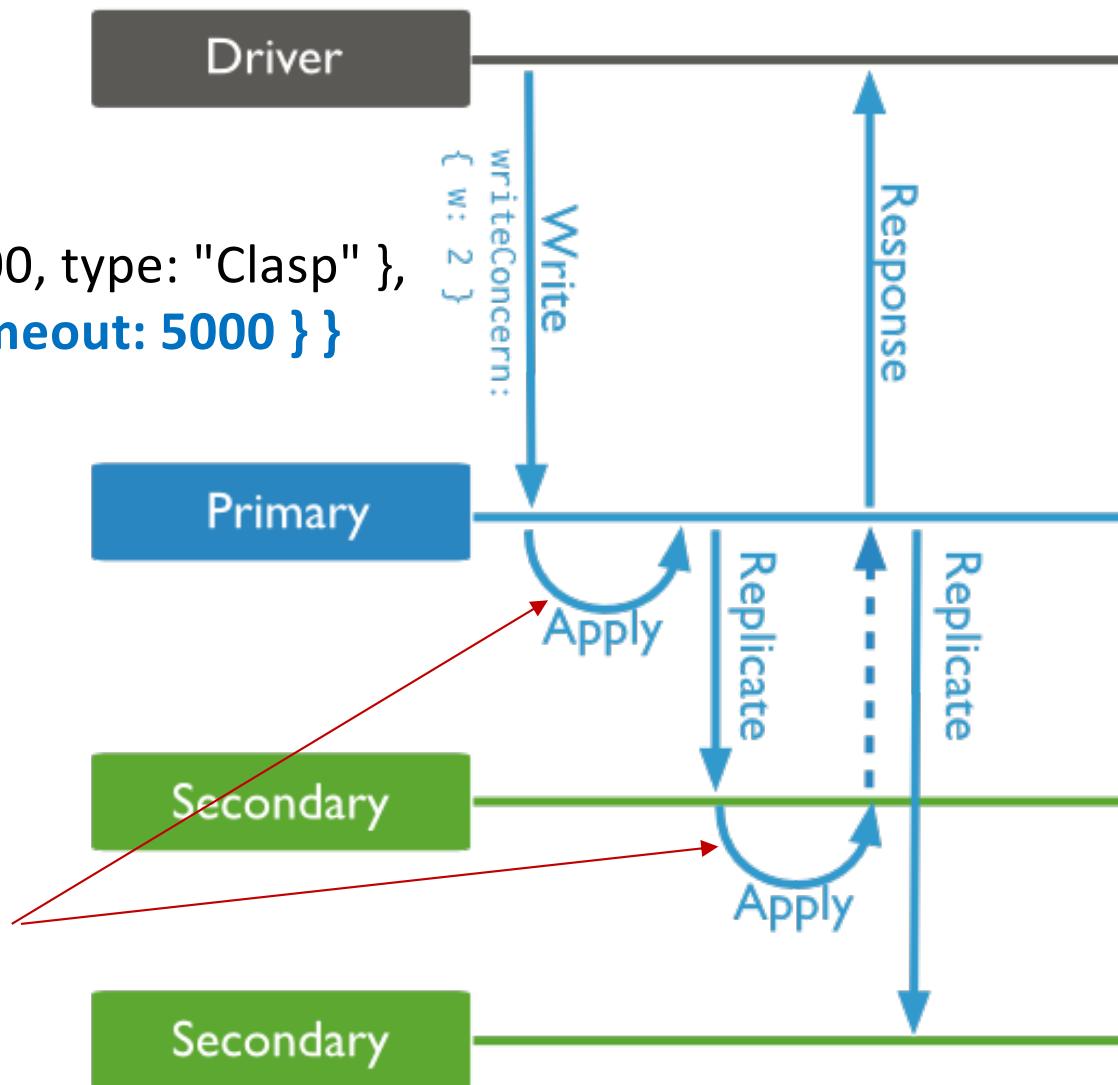
Replica Set Write Concern Example

```
db.products.insert(  
  { item: "envelopes", qty : 100, type: "Clasp" },  
  { writeConcern: { w: 2, wtimeout: 5000 } }  
)
```

The **j** value is unspecified

Acknowledgment behaviour is equivalent to **j=false**

Member send acknowledgement after applying the in memory update



<http://docs.mongodb.org/manual/core/replica-set-write-concern/>

Read Preference

- Read preference describes how MongoDB clients route read operations to the members of a replica set.

Mode	Description
Primary	Default one. All operations read from the primary node
PrimaryPreferred	In most situations, operations read from the primary but if it is unavailable, operations read from secondary members.
Secondary	All operations read from the secondary members of the replica set.
SecondaryPreferred	In most situations, operations read from secondary members but if no secondary members are available, operations read from the primary.
nearest	Operations read from member of the replica set with the least network latency, irrespective of the member's type.

Read Isolation (Read Concern)

- How read operation is carried out inside db engine to control the consistency and availability
- There are many levels
- New release may introduce new level(s) to satisfy growing consistency requirement
- To understand what sort of consistency you will get, all three properties need to be looked at
 - ▶ Write Concern
 - ▶ Read Preference
 - ▶ Read Concern

Read Concern Levels

- local: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members (i.e. may be rolled back)
 - ▶ Default for read against primary, or reads against secondaries if the reads are associated with causally consistent sessions
- available: the query returns data from the instance with no guarantee that the data has been written to a majority of the replica set members
 - ▶ Default for read against secondaries if the reads are **not** associated with causally consistent sessions
- majority: The query returns the data that has been acknowledged by a majority of the replica set members. The documents returned by the read operation are durable, even in the event of failure.
- linearizable
- Snapshot
- Setting read concern level in find query:
`db.collection.find().readConcern(<level>)`

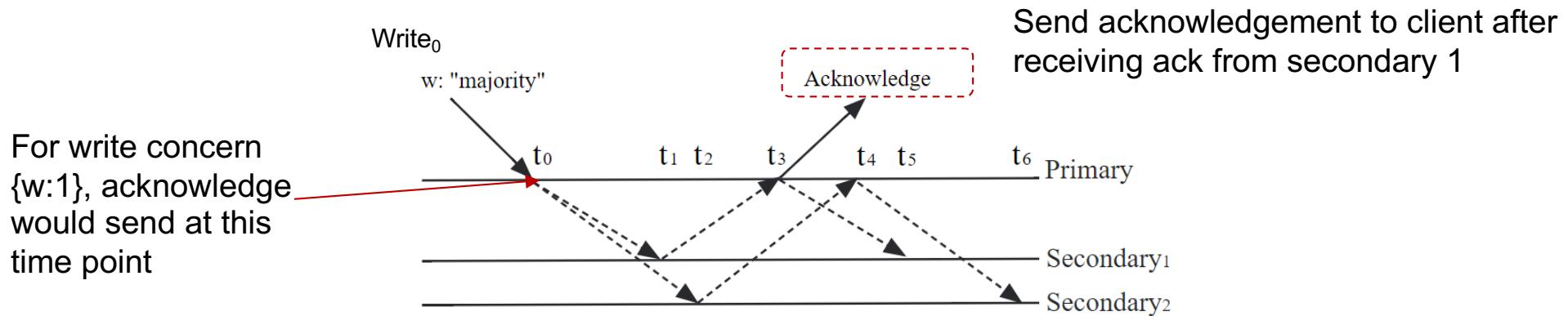
Replica Set Default Behaviour

- Write concern:
 - ▶ Write is considered successful when it is written on the **memory** of primary member
 - `w:1, j:false`
 - ▶ replication to the secondary members happen asynchronously
- Read Preference:
 - ▶ primary: All read operations are sent to the primary
- Read Concern
 - ▶ Local: returns data from the instance (in this case, the primary) with no guarantee that the data has been written to a majority of the replica set members
- Read Uncommitted
 - ▶ Client can see the results of writes before the writes are durable
 - Concurrent read may see the result of a write before the write is acknowledged to the client
 - Client may see data that are subsequently rolled back during replica set failovers

Customized Behaviour: Write: majority

Write concern: “majority”

- Requests acknowledgement that write operations have propagated to the majority of data bearing nodes, including the primary

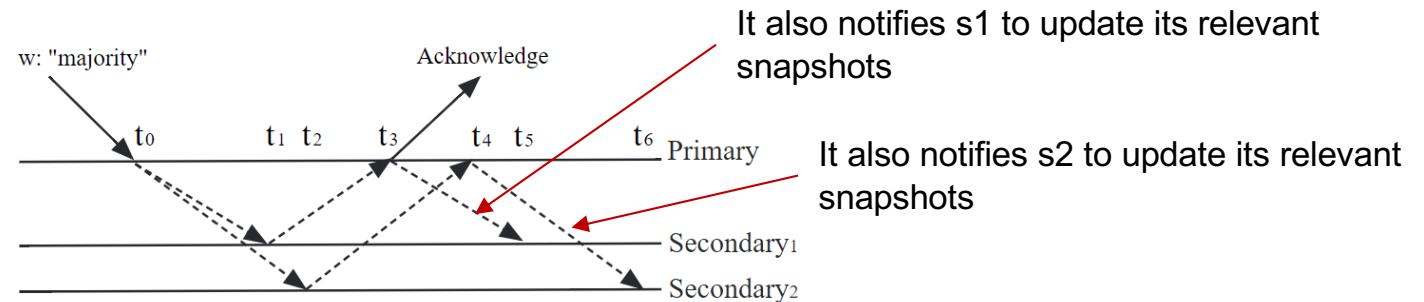


Assume the following:

- All writes prior to Write_0 have been successfully replicated to all members.
- $\text{Write}_{\text{prev}}$ is the previous write before Write_0 .
- No other writes have occurred after Write_0 .

<https://docs.mongodb.com/manual/reference/read-concern-local/#readconcern.%22local%22>

Write: majority example case

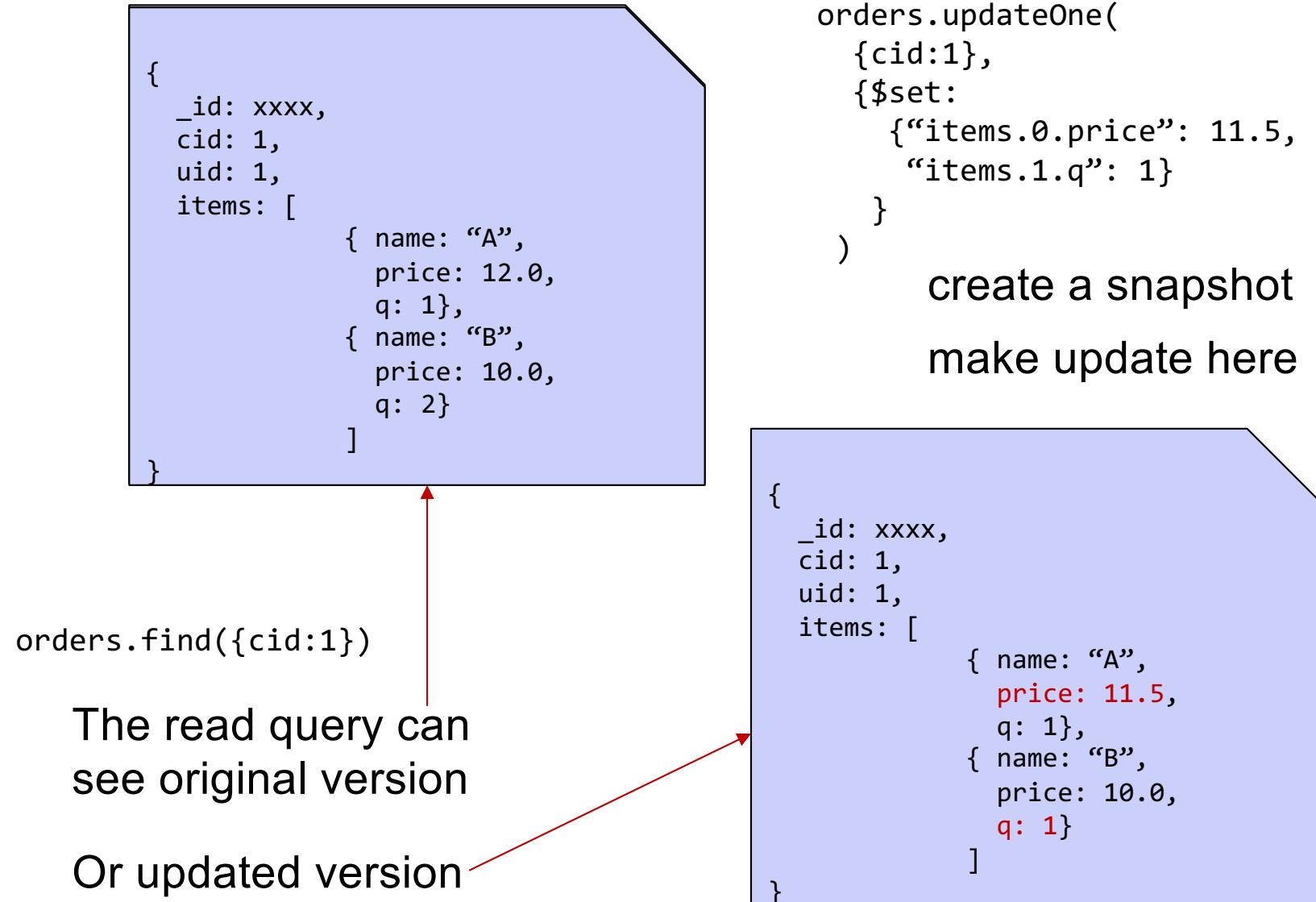


Time	Event	Most Recent Write	Most Recent w: "majority" write
t_0	Primary applies Write_0	Primary: Write_0 Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$	Primary: $\text{Write}_{\text{prev}}$ Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_1	Secondary ₁ applies write_0 Send acknowledgement to primary	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : $\text{Write}_{\text{prev}}$	Primary: $\text{Write}_{\text{prev}}$ Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_2	Secondary ₂ applies write_0 Send acknowledgement to primary	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: $\text{Write}_{\text{prev}}$ Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_3	Primary is aware of successful replication to Secondary ₁ and sends acknowledgement to client	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: Write_0 Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_4	Primary is aware of successful replication to Secondary ₂	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: Write_0 Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_5	Secondary ₁ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : $\text{Write}_{\text{prev}}$
t_6	Secondary ₂ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: "majority" write	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0

Multiple Versions of Data Item

- MongoDB storage engine uses **Multi Version Concurrency Control (MVCC)** to provide concurrent access to the database.
- “When an MVCC database needs to update a piece of data, it will not overwrite the original data item with new data, but instead creates a newer version of the data item. Thus there are multiple versions stored”
[https://en.wikipedia.org/wiki/Multiversion_concurrency_control]
- Read operation can specify which version to use
- MongoDB achieves document level atomicity
 - ▶ At the start of write operation, WiredTiger provides a snapshot of the document to and update that snapshot accordingly

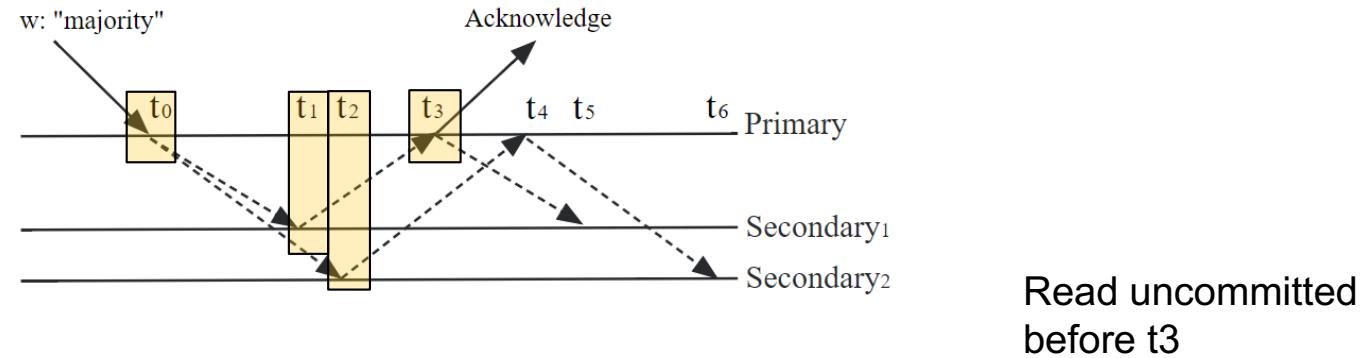
Concurrent Access



Read Concern: *local* example

Read Preference:

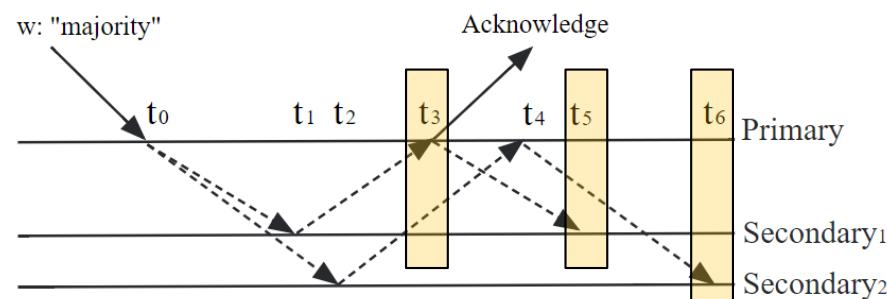
Primary,
PrimaryPreferred,
SecondaryPreferred,
Nearest



Read Target	Time T	State of Data
Primary	After t ₀	Data reflects Write ₀ .
Secondary ₁	Before t ₁	Data reflects Write _{prev}
Secondary ₁	After t ₁	Data reflects Write ₀
Secondary ₂	Before t ₂	Data reflects Write _{prev}
Secondary ₂	After t ₂	Data reflects Write ₀

Read Concern: available has similar behaviour

Read Concern: *majority* example



Primary has the most recent update Write_0 since t_0 , but before t_3 it knows that majority of the replica has the previous value $\text{Write}_{\text{prev}}$

Read Target	Time T	State of Data
Primary	Before t_3	Data reflects $\text{Write}_{\text{prev}}$
Primary	After t_3	Data reflects Write_0
Secondary ₁	Before t_5	Data reflects $\text{Write}_{\text{prev}}$
Secondary ₁	After t_5	Data reflects Write_0
Secondary ₂	Before or at t_6	Data reflects $\text{Write}_{\text{prev}}$
Secondary ₂	After t_6	Data reflects Write_0

t_2	Secondary ₂ applies write_0	Primary: Write_0 Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : Write_0	Primary: $\text{Write}_{\text{prev}}$ Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$
t_3	Primary is aware of successful replication to Secondary ₁ and sends acknowledgement to client	Primary: Write_0 Secondary ₁ : Write_0 Secondary ₂ : Write_0	Primary: Write_0 Secondary ₁ : $\text{Write}_{\text{prev}}$ Secondary ₂ : $\text{Write}_{\text{prev}}$

Consequence

- Read concern “*local*” returns the latest value as soon as it is applied locally, it has the danger of **read uncommitted**, e.g. return a value before the write is durable, the value may not exist if rolled back
- Read concern “*majority*” will return old value some time after the write happens even if the target is set to primary node; it does not return uncommitted value regardless of the target node.
- Customized setting will have better scalability by allowing read to happen at the secondary node
 - ▶ There are various trade offs depending on the actual setting

Outline

■ Replication

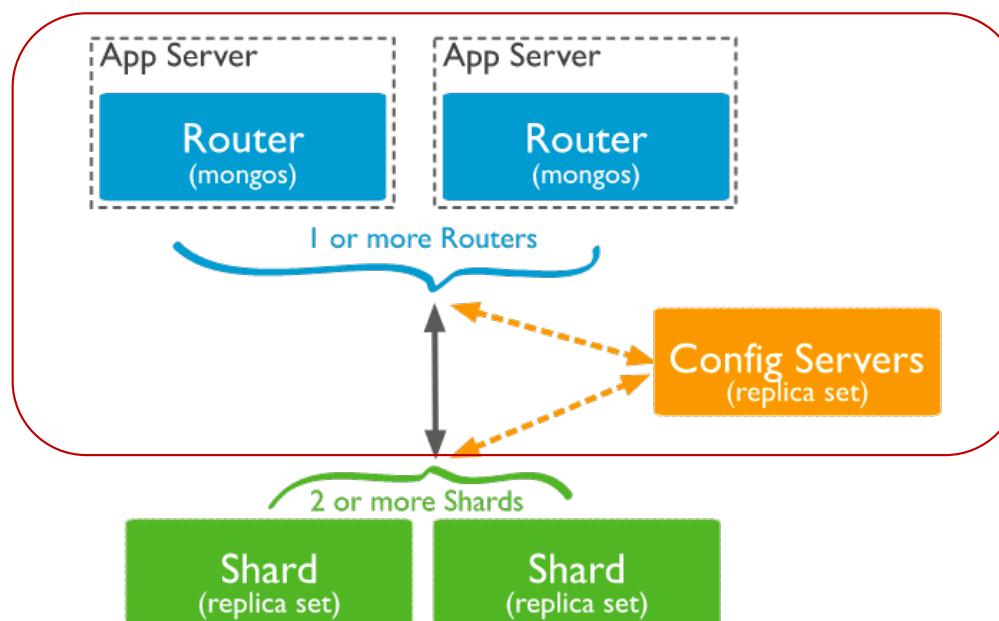
■ Sharding

When Both Data Size and Traffic Grow

- Replication solves certain scalability issue by distributing read traffic to secondary members
- When data size grows beyond the capacity of a single node
 - ▶ Sharding/partition is needed
- Sharding is a method for distributing data across multiple machines.
- MongoDB uses horizontal sharding
 - ▶ E.g. instead of putting all documents of a collection in a single node, we can put subsets of documents in different nodes.
 - ▶ Users indicate how to divide the full collection into subsets of documents
 - ▶ System manages the actual data partition and query

MongoDB Sharding

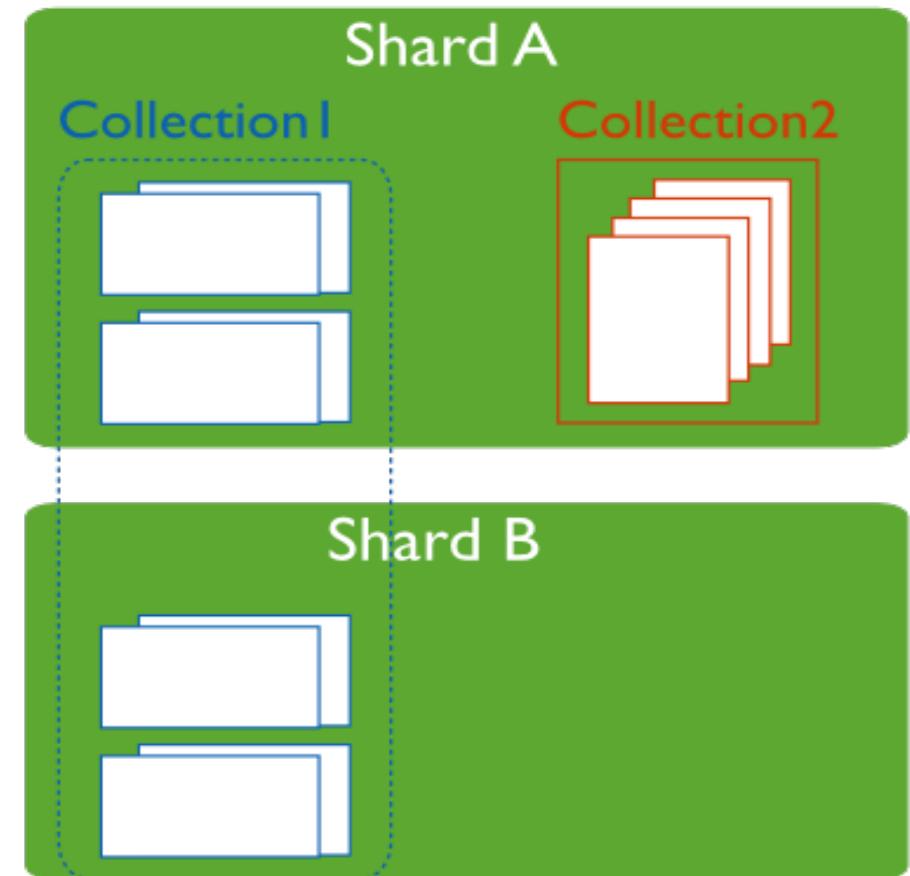
- The main database engine **mongod** is not distributed
 - ▶ Running on a single node
- Sharding is achieved by running an extra coordinator service **mongos** together with a set of **config servers** on top of **mongod**



<https://docs.mongodb.com/manual/sharding/#sharded-cluster>

Shard

- Each shard is a standalone mongod server or a replica set (with one primary and a few secondary members)
 - ▶ Shard must be a replica set since version 3.6
- Each shard stores some portion of a sharded cluster's total data set.
- Primary Shard
 - ▶ Every database has a primary shard that holds all unsharded collections for a database



Data Partitioning with Chunks

- Data stored in each shard are organized as fixed sized **chunks** (default 64MB, but configurable)
- A chunk contains partition of documents belonging to the same collection
- Document-chunk distribution is determined by sharding key and sharding strategy
 - ▶ Sharding key is specified by use as single or compound field
- A shard may contain many chunks of a collection
- A cluster balancer is responsible to ensure chunks of a sharded collection are evenly distributed among the shards

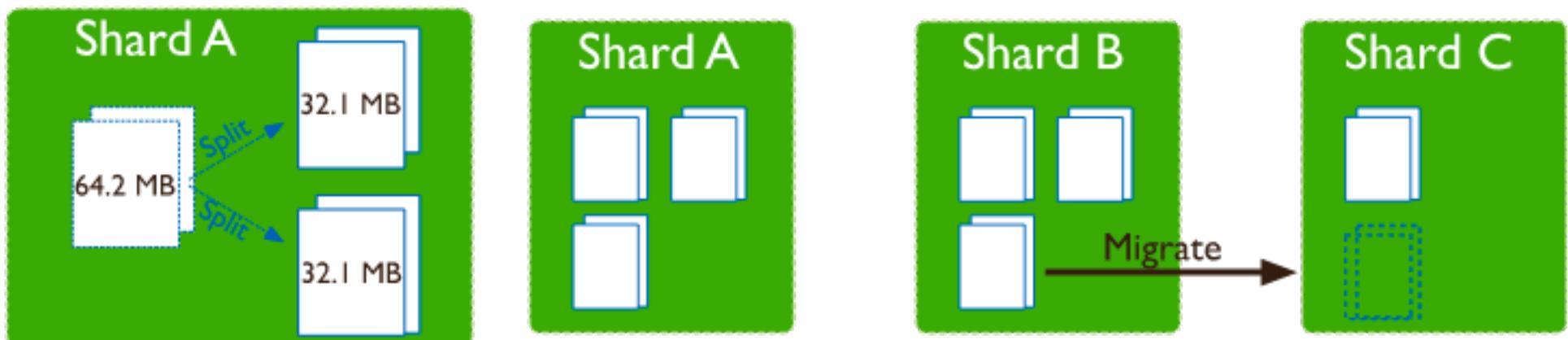
<https://docs.mongodb.com/manual/core/sharding-data-partitioning/>

Initial Chunk creation

- Initial chunks may be created for populated collections or for empty ones
- Populated collection
 - ▶ The sharding operation creates the initial chunk(s) to cover the entire range of the shard key values. The number of chunks created depends on the chunk size and collection size
- Empty collection
 - ▶ One or many empty chunks may be created and placed on shards depending on the various settings
- Write operations would cause chunk size to vary along the time

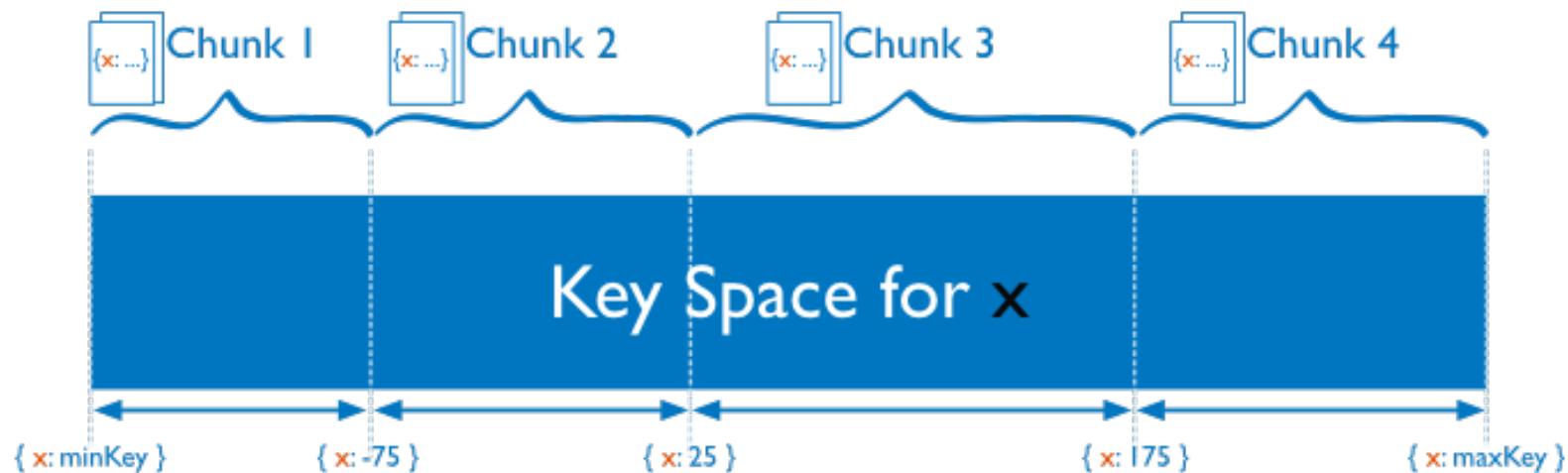
Chunk Split and Migration

- When a chunk grows beyond the specified size, it will be split
- When chunks of the same collection are not distributed evenly among shards, some chunk will migrate between shards
- In some cases, chunks can grow beyond the specified chunk size but cannot undergo a split. The most common scenario is when a chunk represents a single shard key value. Since the chunk cannot split, it continues to grow beyond the chunk size, becoming a **jumbo** chunk.



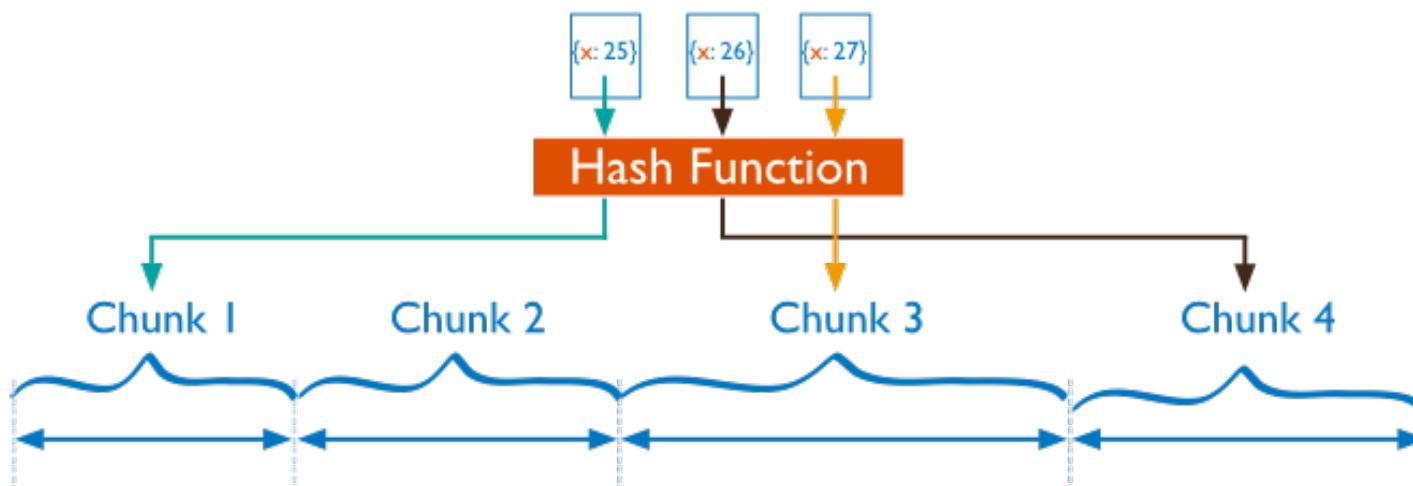
Sharding Strategy: Range Sharding

- Chunk is created based on actual sharding key's value range
- Each chunk represents a range of the sharding key value
- Contiguous sharding key values are likely to be stored on the same shard



Sharding Strategy: Hash Sharding

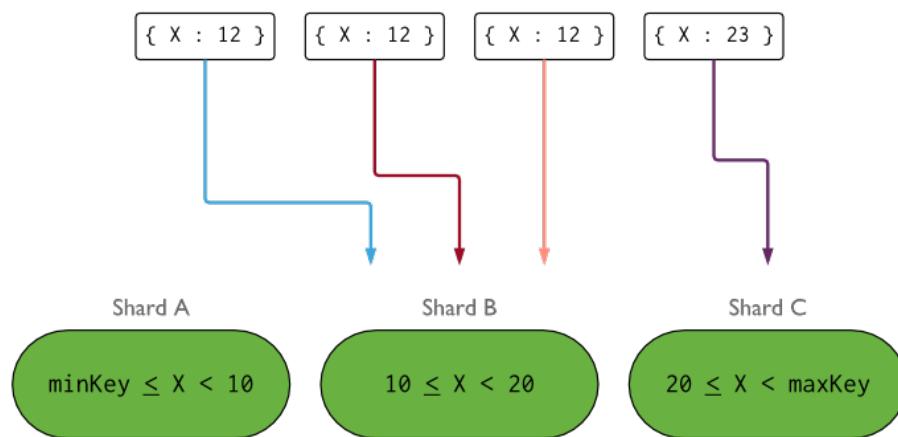
- Chunk is created based on the hash value of sharding key
- Each chunk represent a range of the hash value
- Contiguous sharding key values are likely to be distributed in different chunks



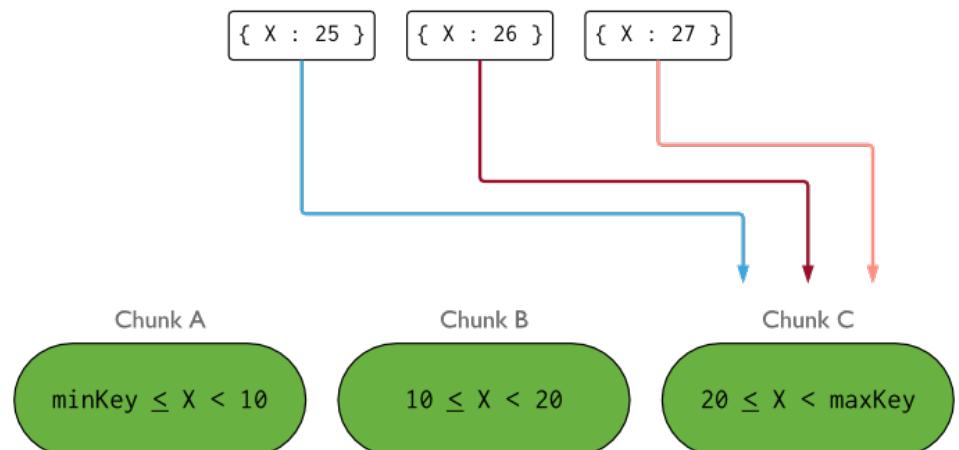
Shard Key Selection

- The ideal shard key should distribute data and query evenly in shards
 - ▶ High cardinality
 - Gender is not a good sharding key candidate
 - ▶ Distribution not skewed
 - Key with zipf value distribution is not a good sharding key candidate
 - ▶ Change pattern
 - Timestamp is perhaps not a very good shard key candidate

Shard key with skewed distribution would create query hot spot



Monotonically increasing shard key would create insert hot spot



Example of Good sharding Key

Machine 1	Machine 2	Machine 3
Alabama → Arizona a chunk	Colorado → Florida	Arkansas → California
Indiana → Kansas	Idaho → Illinois	Georgia → Hawaii
Maryland → Michigan	Kentucky → Maine	Minnesota → Missouri
Montana → Montana	Nebraska → New Jersey	Ohio → Pennsylvania
New Mexico → North Dakota	Rhode Island → South Dakota	Tennessee → Utah
	Vermont → West Virginia	Wisconsin → Wyoming

user collection partitioned by field “**state**” as shard key

Indexing on Sharded Collection

- There is no global index structure on sharded collections
- When an index is created on a sharded collection
 - ▶ Local index structure is created by each shard for the data portion it is responsible
 - ▶ Shards do not communicate with each other
- Certain index properties cannot be maintained
 - ▶ Uniqueness can only be guaranteed for sharding key
 - ▶ No other index should be created with the uniqueness property
 - A `users` collection sharded by `userid` can guarantee each document has a unique `userid` but cannot guarantee no duplication of `TFN`
- MongoDB also require index support for sharding key

Config Server

- Config servers are deployed as a replica set
 - ▶ Admin and shard metadata are stored as MongoDB collections
- Config servers maintain an admin database and a config database
 - ▶ The admin database stores data related to the authentication and authorization and other system internal information
 - ▶ Config database stores data about chunks and their locations in shard

user collection partitioned by field
“**name**” as shard key and are
stored as chunks in different
shards

collection	minkey	maxkey	location
users	{ name : 'Miller' }	{ name : 'Nessman' }	shard ₂
users	{ name : 'Nessman' }	{ name : 'Ogden' }	shard ₄
...			

Config Servers Read/Write

- Users are not supposed to read/write collections maintained by config servers
 - ▶ Various MongoDB components will read/write data from them
- MongoDB writes data to the config database when the metadata changes, such as after a chunk migration or a chunk split.
 - ▶ When writing to the config servers, MongoDB uses a write concern of "majority".
- Admin and config database are read by various MongoDB components for authentication data and collection distribution data
 - ▶ When reading from the config servers, MongoDB uses a read concern of “majority”

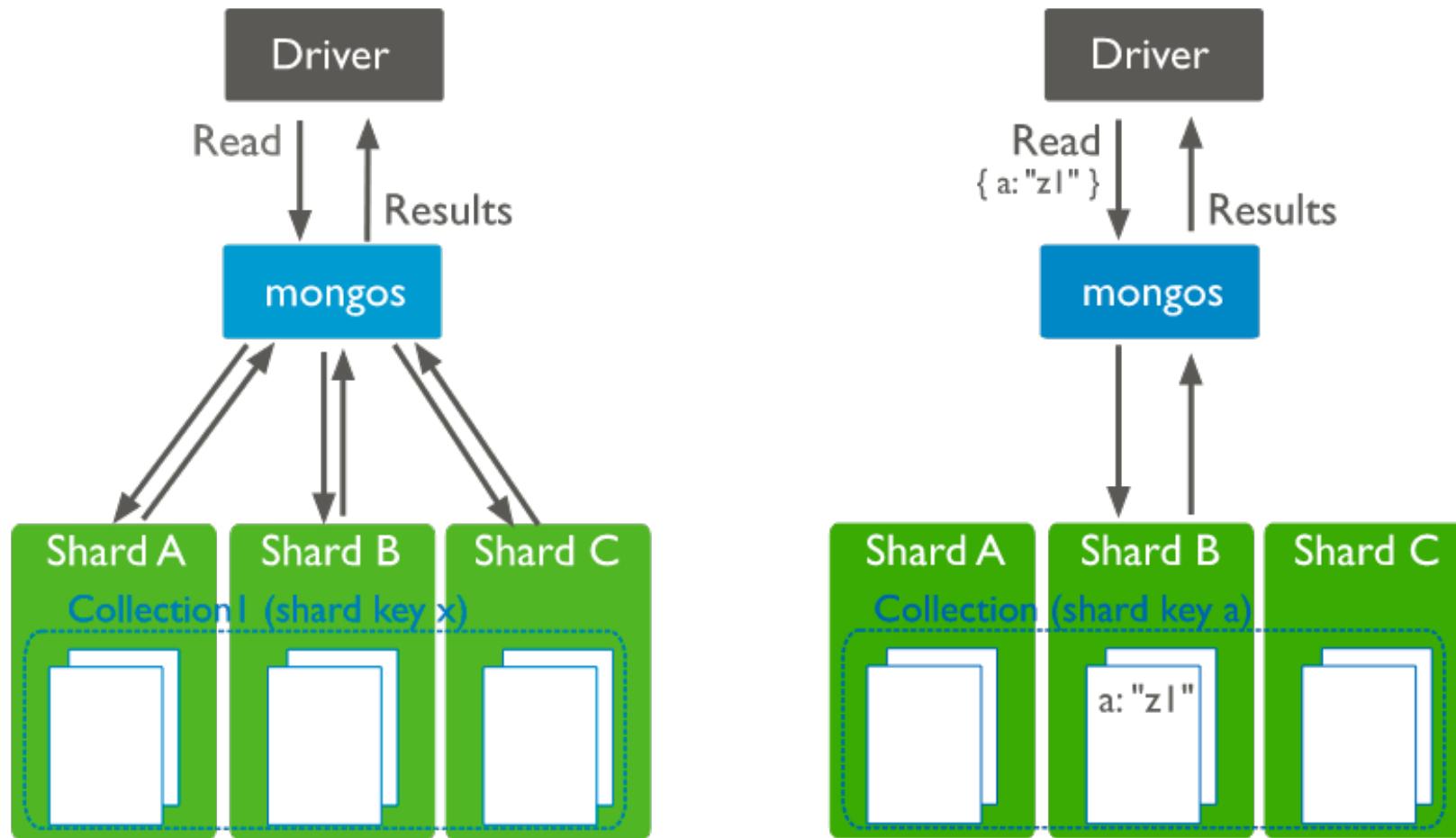
Config Server Availability

- If the config server loses its primary, the cluster's metadata becomes *read only*. You can still read and write data from the shards, but no chunk migration or chunk splits will occur until a new primary is elected.
- If all config servers become unavailable, the cluster can become inoperable.

Routing Processes -- **mongos**

- In a sharded cluster, **mongos** is the front end for client request
 - ▶ When receiving client requests, the **mongos** process routes the request to the appropriate server(s) and merges any results to be sent back to the client
 - ▶ It has no persistent state, the meta data are pulled from **config servers**
 - ▶ There is no limits on the number of **mongos** processes. They are independent to each other
- Query types
 - ▶ Targeted at a single shard or a limited group of shards based on the **shard key**.
 - ▶ Broadcast to all shards in the cluster that hold documents in a collection.

Targeted and Broadcast Operations



Targeted and Broadcast Operation Examples

Assuming shard key is field x

Operation	Type	Execution
db.food.find({x:300})	Targeted	Query a single shard
db.foo.find({ x : 300, age : 40 })	Targeted	Query a single shard
db.foo.find({ age : 40 })	Global	Query all shards
db.foo.find()	Global	Query all shards, sequential
db.foo.find(...).count()	Variable	Same as the corresponding find() operation
db.foo.count()	Global	Parallel counting on each shard, merge results on mongos
db.foo.insert(<object>)	Targeted	Insert on a single shard
db.foo.createIndex(...)	Global	Parallel indexing on each shard

Summary

■ MongoDB is a general purpose NoSQL storage system

- ▶ Lots of resemblance with RDBMS
 - Indexing, queries on any field
 - It supports spatial and graph queries
- ▶ Single document update is always atomic
- ▶ Later version has support for multi-document transaction

■ Key Features

- ▶ Flexible schema
 - Collection and Document
 - Documents are stored in binary JSON format
 - Natural support for object style query (array and dot notation)
- ▶ Scalability
 - Sharding and Replication
- ▶ Various consistency levels achieved through write concern, read preference and read concern property combination

References

■ MongoDB Replication

- ▶ <https://docs.mongodb.com/manual/replication/>

■ MongoDB Replica Set Read and Write Semantics

- ▶ <https://docs.mongodb.com/manual/applications/replication/>

■ MongoDB Write Concern

- ▶ <https://docs.mongodb.com/manual/reference/write-concern>

■ MongoDB Read Isolation (Read Concern)

- ▶ [https://docs.mongodb.com/manual/reference/read-concern/](https://docs.mongodb.com/manual/reference/read-concern)

COMP5338 – Advanced Data Models

Week 6: Graph Data and Neo4j Introduction

Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

■ Brief Review of Graphs

- ▶ Examples of Graph Data
- ▶ Modelling Graph Data

■ Property Graph Model

■ Cypher Query

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

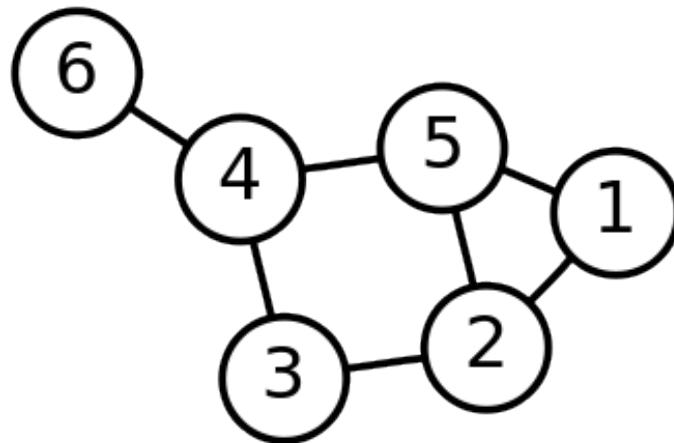
The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



Graphs

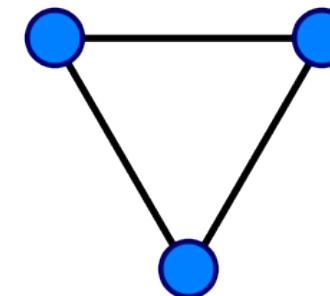
- A graph is just a collection of *vertices* and *edges*
 - ▶ Vertex is also called Node
 - ▶ Edge is also called Arc/Link



Type of Graphs

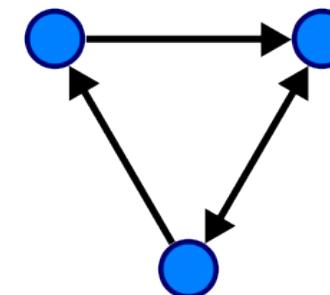
■ Undirected graphs

- ▶ Edges have no orientation (direction)
- ▶ (a, b) is the same as (b, a)



■ Directed graphs

- ▶ Edges have orientation (direction)
- ▶ (a, b) is not the same as (b, a)

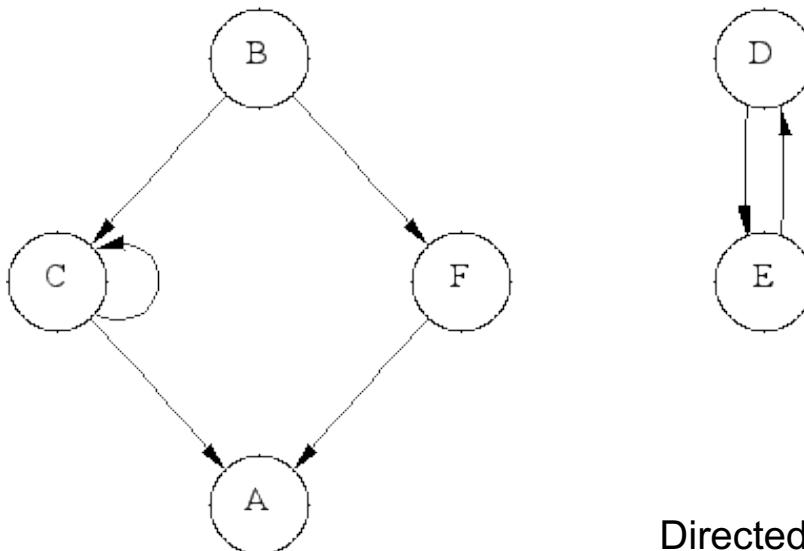


Representing Graph Data

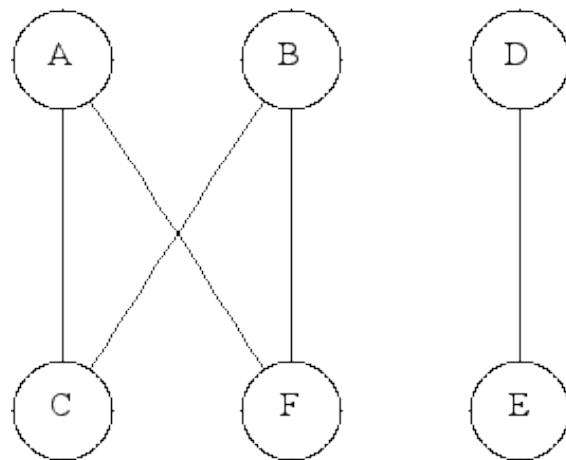
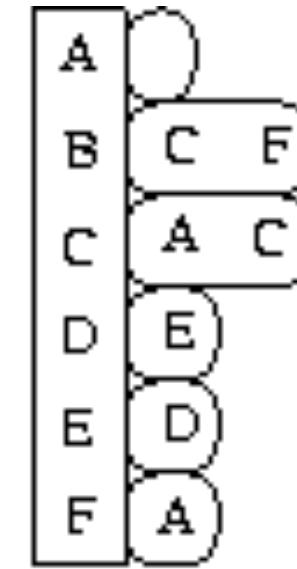
- Data structures used to store graphs in programs
 - ▶ Adjacency list
 - ▶ Adjacency matrix



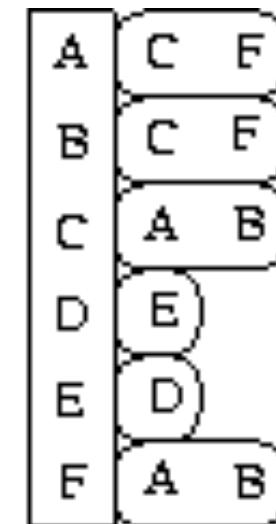
Adjacency List



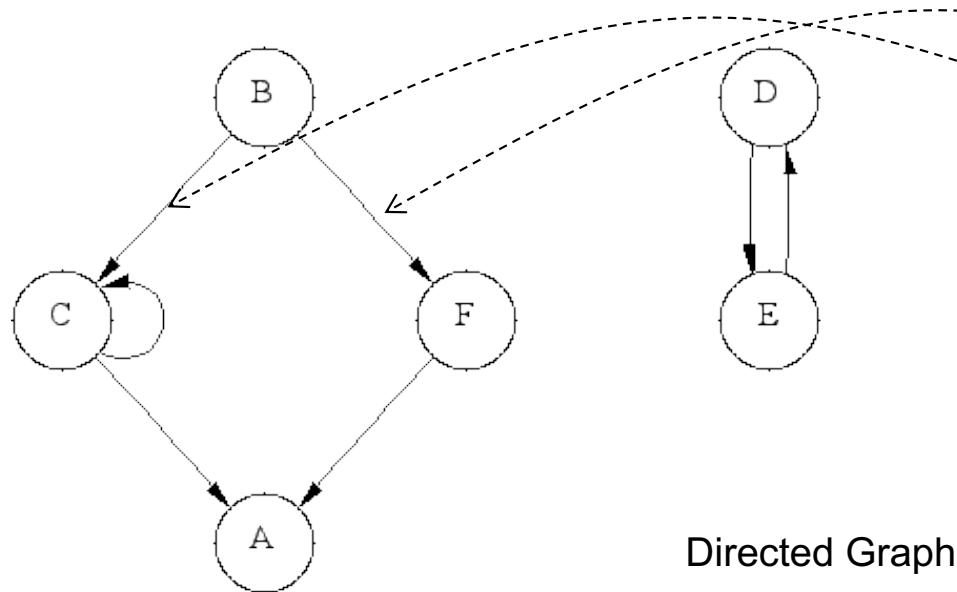
Directed Graph



Undirected Graph

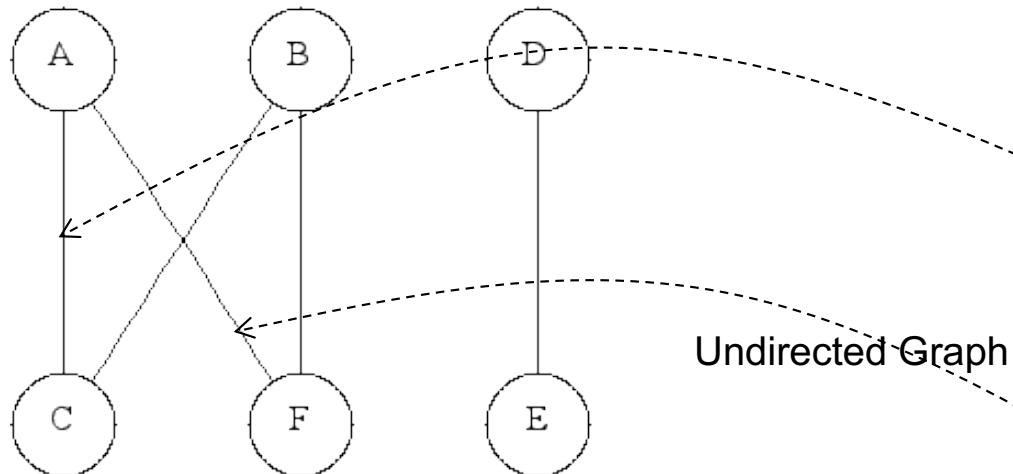


Adjacency matrix



Directed Graph

	A	B	C	D	E	F
A	0	0	0	0	0	0
B	0	0	1	0	0	1
C	1	0	1	0	0	0
D	0	0	0	0	1	0
E	0	0	0	1	0	0
F	1	0	0	0	0	0



Undirected Graph

	A	B	C	D	E	F
A	0					
B	0	0				
C	1	1	0			
D	0	0	0	0		
E	0	0	0	1	0	
F	1	1	0	0	0	0



Outline

■ Brief Review of Graphs

- ▶ Examples of Graph Data
- ▶ Modelling Graph Data

■ Introduction to Neo4j

■ Cypher Query

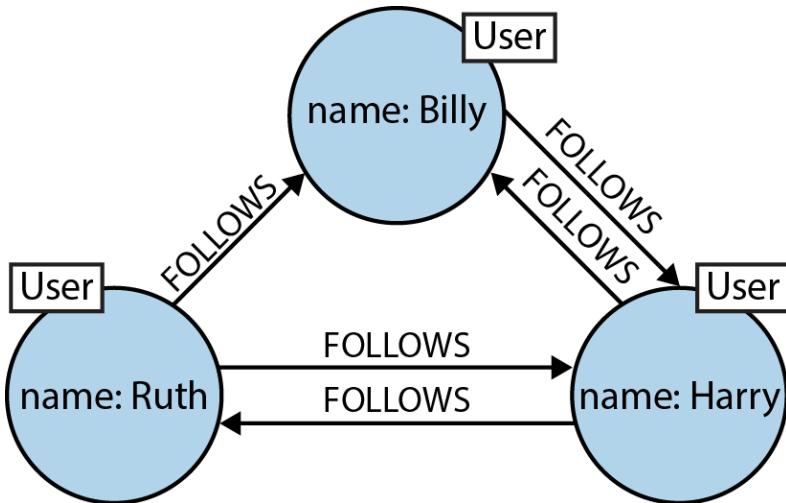


Examples of graphs

- Social graphs
 - ▶ Organization structure
 - ▶ Facebook, LinkedIn, etc.
- Computer Network topologies
 - ▶ Data centre layout
 - ▶ Network routing tables
- Road, Rail and Airline networks

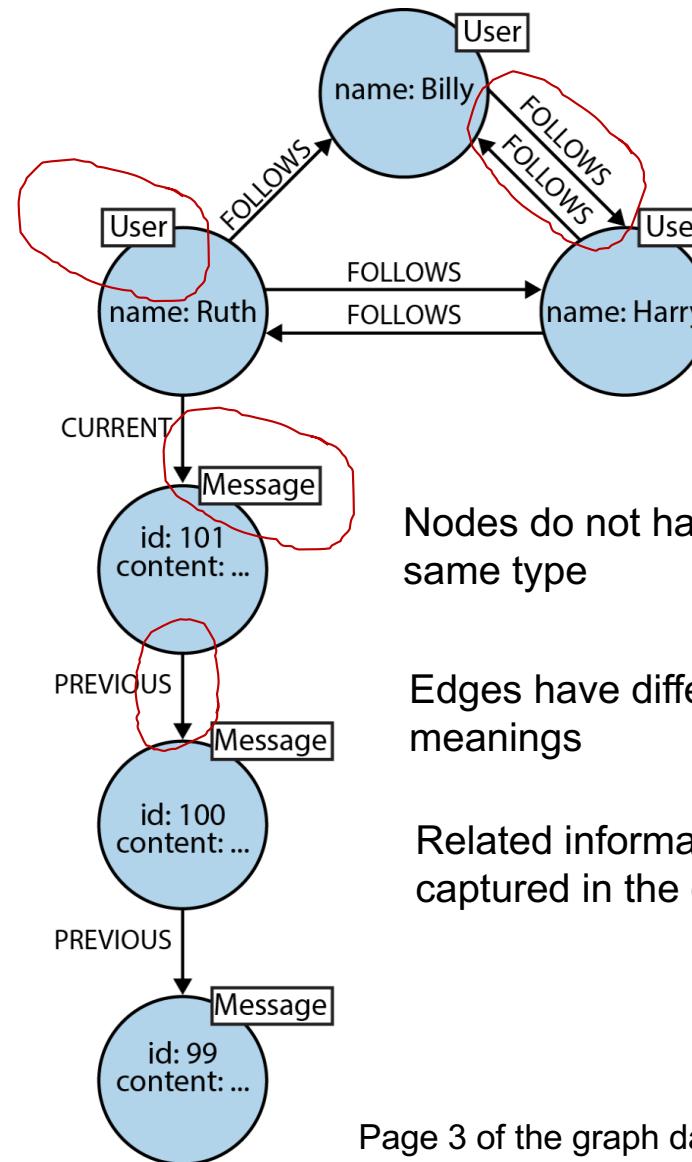


Social Graphs and extension



A small social graph

Page 2 of the graph database book



Nodes do not have to be of same type

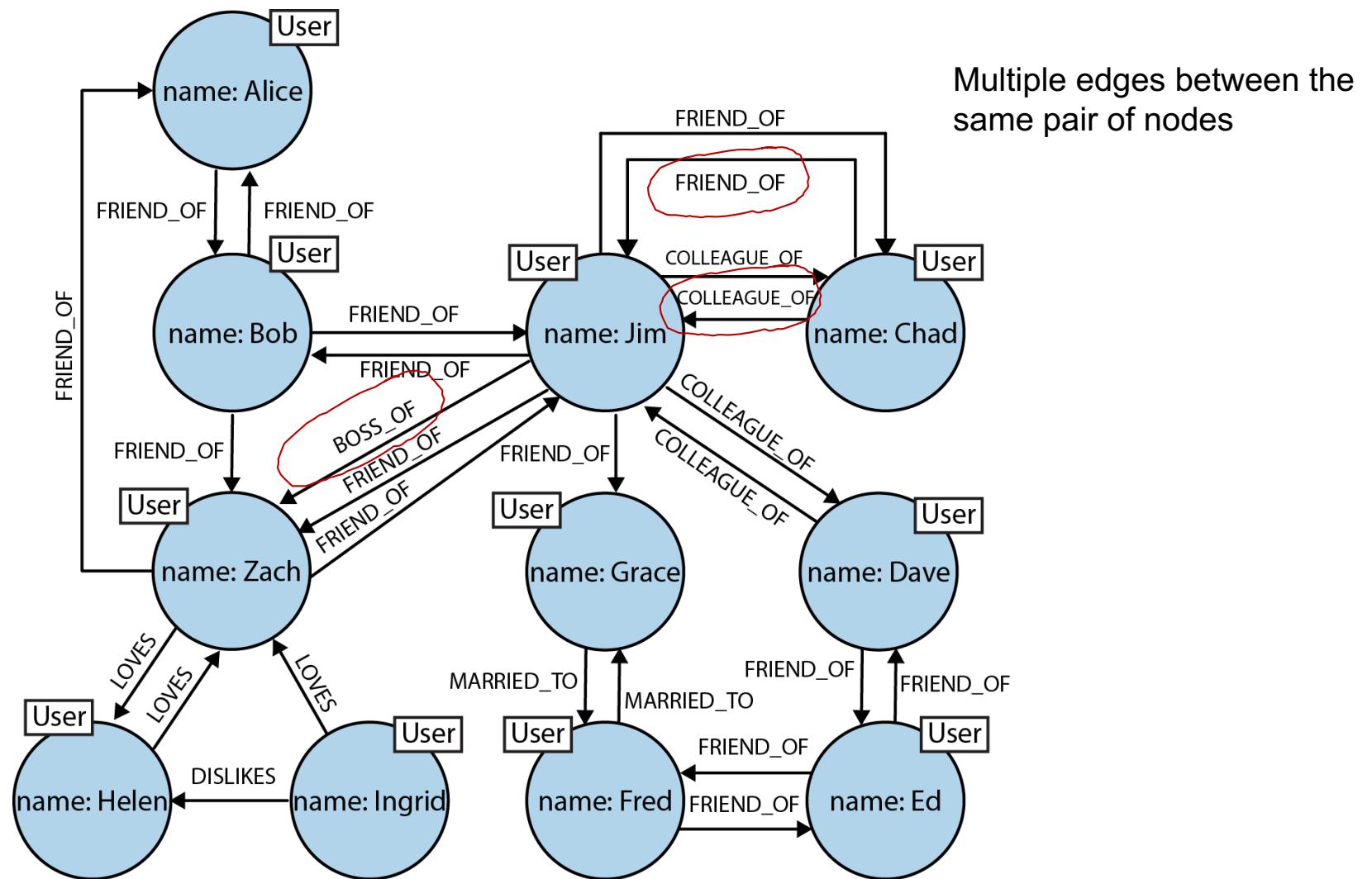
Edges have different meanings

Related information can be captured in the graph

Page 3 of the graph database book



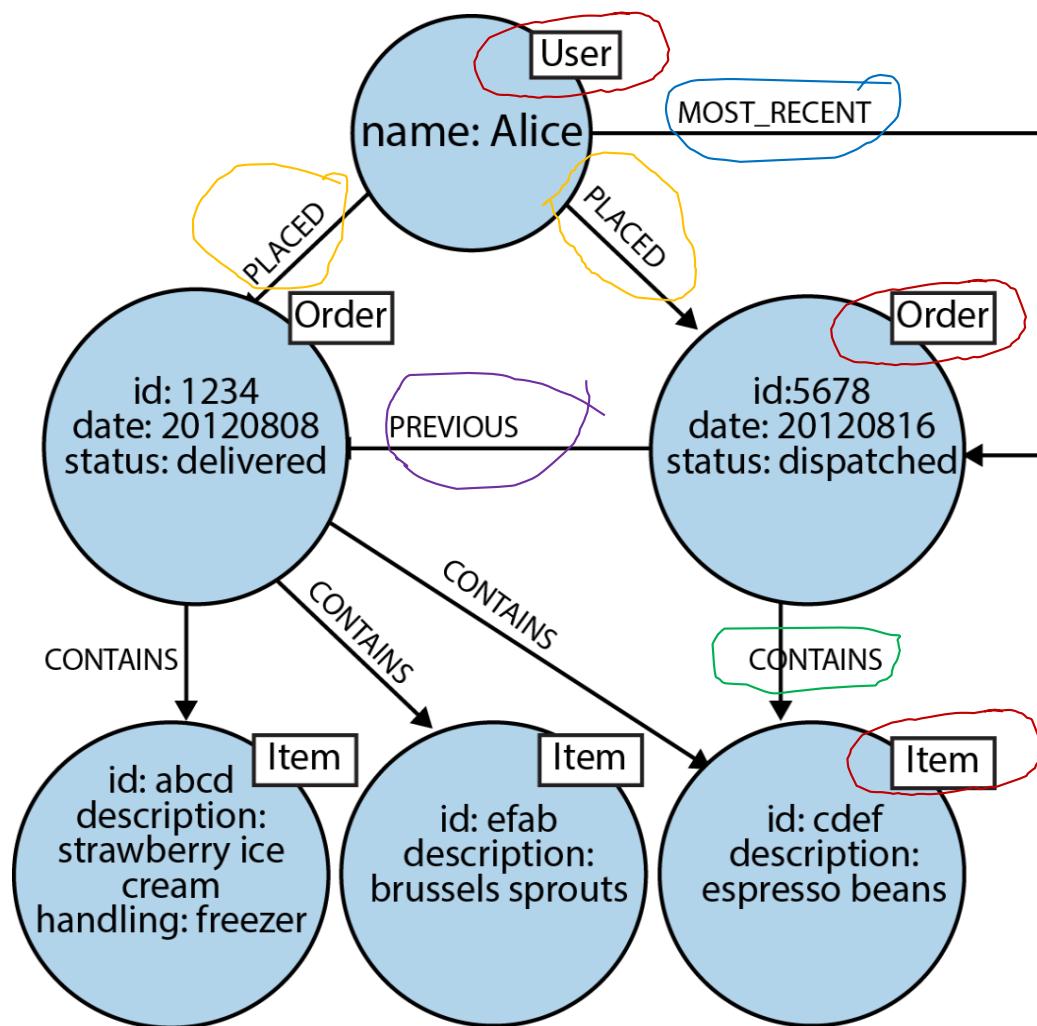
Social Graph with Various Relationships



Page 19 of the graph database book



Transaction information



This graph shows three types of entities

And various types of relationships among those entities

User PLACED order

User MOST_RECENT order

Order CONTAINS Item

Order PREVIOUS Order

Outline

■ Brief Review of Graphs

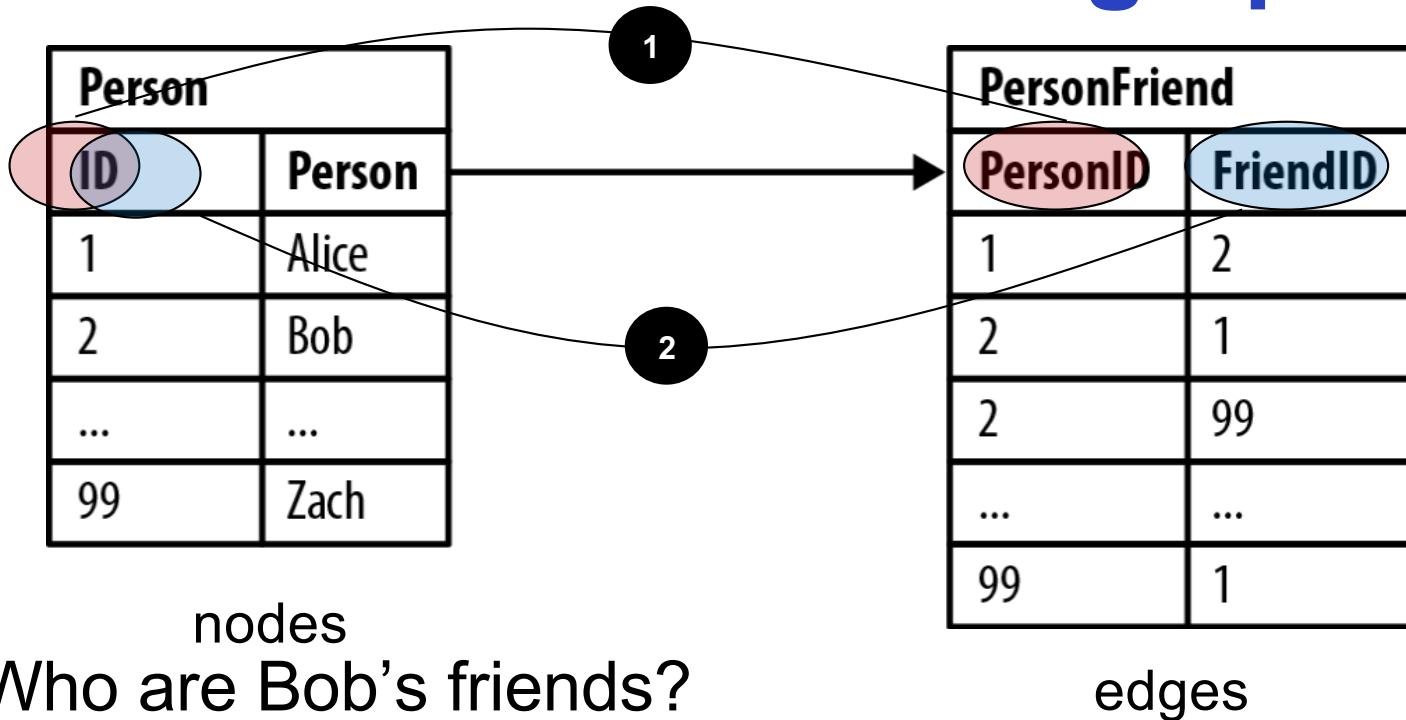
- ▶ Examples of Graph Data
- ▶ Modelling Graph Data

■ Property Graph Model

■ Cypher Query



RDBMS to store graph



■ Who are Bob's friends?

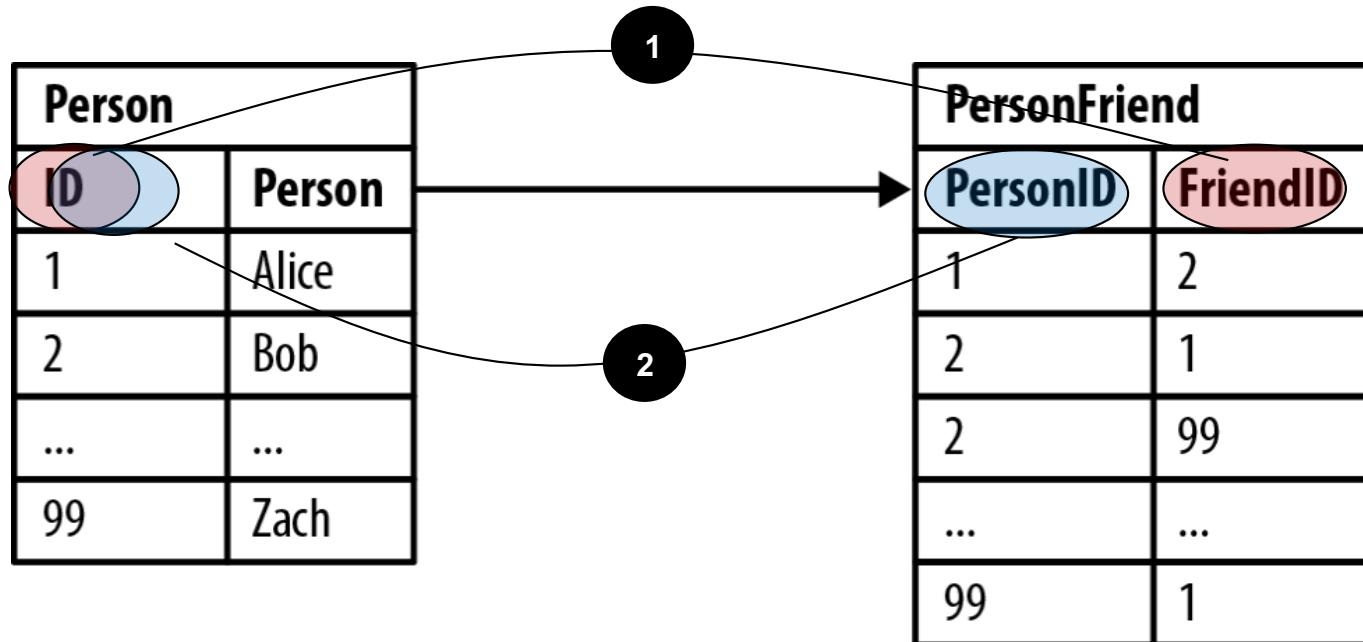
```
SELECT p1.Person  
FROM Person p1 JOIN PersonFriend pf ON pf.FriendID = p1.ID  
JOIN Person p2 ON pf.PersonID = p2.ID  
WHERE p2.Person = "Bob"
```

Page 13 of the graph database book



RDBMS to store Graphs

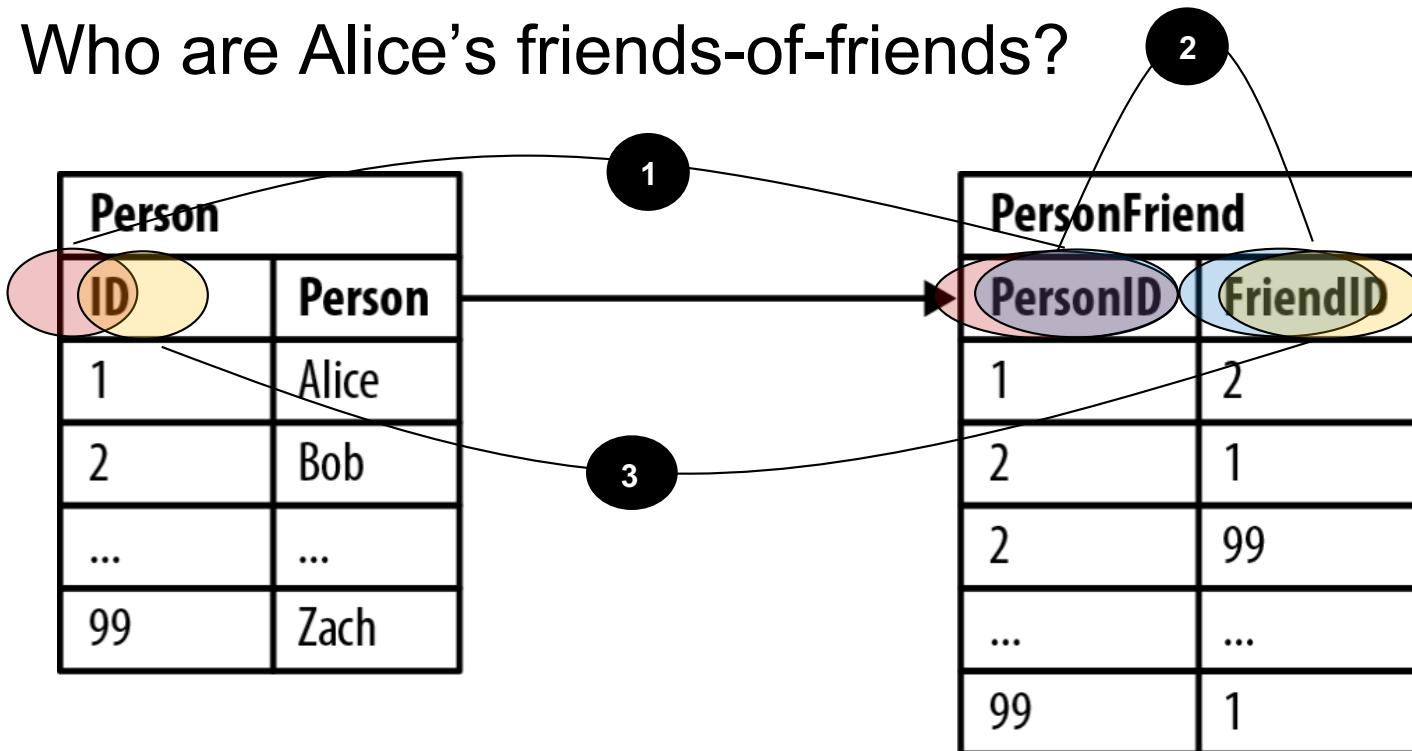
- Who are friends with Bob?



```
SELECT p1.Person  
FROM Person p1 JOIN PersonFriend pf ON pf.PersonID = p1.ID  
JOIN Person p2 ON pf.FriendID = p2.ID  
WHERE p2.Person = "Bob"
```

RDBMS to store Graphs

■ Who are Alice's friends-of-friends?



```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1 ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2 ON pf2.PersonID = pf1.FriendID
JOIN Person p2 ON pf2.FriendID = p2.ID
WHERE p1.Person = "Alice" AND pf2.FriendID <> p1.ID
```



MongoDB to store Graph

persons collection

```
{ _id: 1,  
  person: "Alice",  
  friends:[2]  
}
```

```
{ _id: 2,  
  person: "Bob",  
  friends:[1,99]  
}
```

```
{ _id: 99,  
  person: "Zach",  
  friends:[1]  
}
```

- Who are Bob's friends?
 - ▶ Find out Bob's friends' ID
 - db.persons.find({person:"Bob"},{friends:1})
 - ▶ For each id, find out the actual person
 - db.persons.find({_id: 1},{person:1}),
db.persons.find({_id: 99},{person:1}),
■ db.persons.find({_id:{\$in:[1,99]}}, {person:1})
- Who are friends with Bob?
 - ▶ Find out Bob's id
 - db.persons.find({person:"Bob"})
 - ▶ Find out the persons that are friends with Bob
 - db.persons.find({friends: 2}, {person:1})
- Who are Alice's friends-of-friends?
 - ▶ Find out Alice's friends ID
 - db.persons.find({person:"Alice"},{friends:1})
 - ▶ For each id, find out the friends ID again
 - db.persons.find({_id:{\$in:[2]}}, {friends:1})
 - ▶ For each id, find out the actual person
 - db.persons.find({_id:{\$in:[1,99]}}, {person:1})
- The MongoDB 3.4 and later has a new aggregation stage called \$graphLookup



\$graphLookup

```
db.persons.aggregate([
  {$match:{person:"Alice"}},
  {$graphLookup:{
    from: "persons",
    startWith: "$friends",
    connectFromField:"friends",
    connectToField:"_id",
    maxDepth: 1,
    as: "friendsnetwork"}}
])
```

```
{"_id" : 1,
  "person" : "Alice",
  "friends" : [ 2],
  "friendsnetwork" : [
    {"_id" : 99.0,
      "name" : "Zach",
      "friends" : [ 1, 3],
      "depth" : 1 },
    {"_id" : 1,
      "name" : "Alice",
      "friends" : [ 2],
      "depth" : 1 },
    {"_id" : 2,
      "name" : "Bob",
      "friends" : [1, 99],
      "depth" : 0}
  ]
}
```

```
{ _id: 1,
  person: "Alice",
  friends:[2]
}

{ _id: 2,
  person: "Bob",
  friends:[1,99]
}

{ _id: 99,
  person: "Zach",
  friends:[1]
}
```



In Summary

■ It is possible to store graph data in various storage systems

▶ Shallow traversal

- Relatively easy to implement
- Performance OK

▶ Deep traversal or traversal in other direction

- Complicated to implement
 - Multiple joins or multiple queries or full table scan
- Less efficient
- Error prone



Outline

■ Brief Review of Graphs

■ Property Graph Model

■ Cypher Query



Graph Technologies

■ Graph Processing

- ▶ take data in any input format and perform graph related operations
- ▶ OLAP – OnLine Analysis Processing of graph data
- ▶ Google Pregel, Apache Giraph

■ Graph Databases

- ▶ manage, query, process graph data
- ▶ support high-level query language
- ▶ native storage of graph data
- ▶ OLTP – OnLine Transaction Processing possible
- ▶ OLAP – also possible



Graph Data Models

■ RDF (Resource Description Framework) Model

- ▶ Express node-edge relation as “subject, predicate, object” triple (RDF statement)
- ▶ SPARQL query language
- ▶ Examples: AllegroGraph, Apache Jena

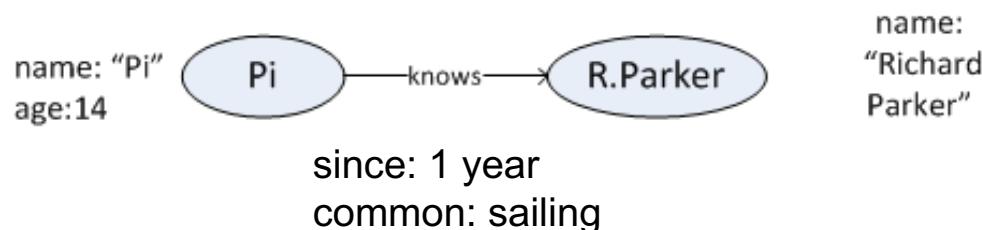
■ Property Graph Model

- ▶ Express node and edge as object like entities, both can have properties
- ▶ Various query language
- ▶ Examples
 - Apache Titan
 - Support various NoSQL storage engine: BerkeleyDB, Cassandra, HBase
 - Structural query language: Gremlin
 - Neo4j
 - Native storage manager for graph data (Index-free Adjacency)
 - Declarative query language: Cypher query language



Property Graph Model

- Proposed by **Neo** technology
- No standard definition or specification
- Both Node and Edges can have property
 - ▶ RDF model cannot express edge property in a natural and easy to understand way
- The actual storage varies
- The query language varies



Neo4j

- Native graph storage using **property graph model**
- Index-free Adjacency
 - ▶ Nodes and Edges are stored based on graph structure
- Supports indexes
- **Cypher** – query language
- Replication
 - ▶ Traditional master/slave replication mechanism
- Neo4j also introduced a sharded graph mechanism since 4.0
 - ▶ Neo4j Fabric



Property Graph Model as in Neo4j

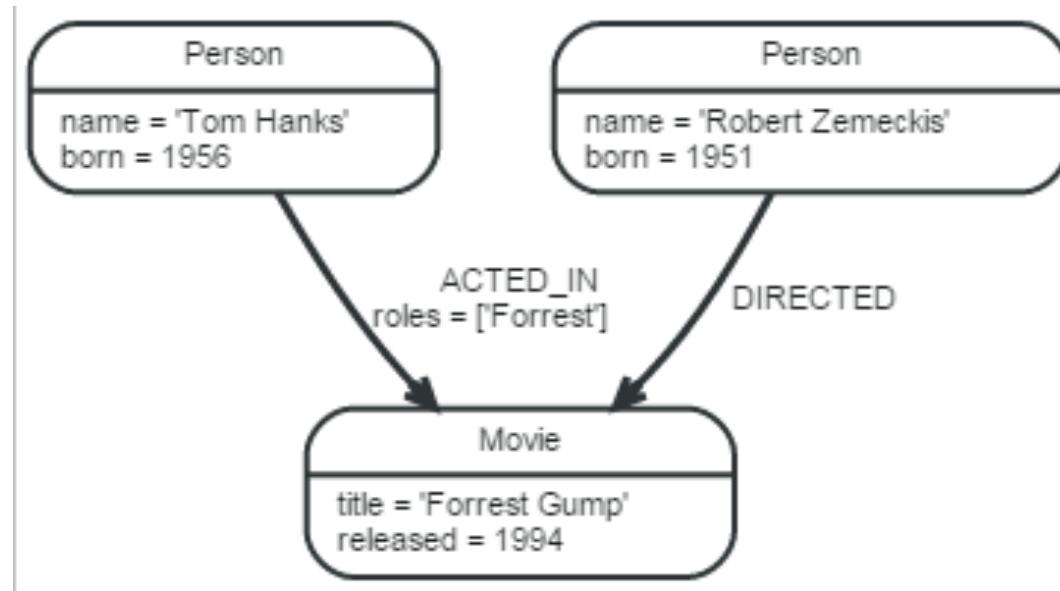
■ Property graph has the following characteristics

- ▶ It contains nodes and relationships
- ▶ Nodes contain properties
 - Properties are stored in the form of key-value pairs
 - A node can have labels (classes)
- ▶ Relationships connect nodes
 - Has a *direction*, an optional *type*, a *source node* and a *target node*
 - No dangling relationships (can't delete node with a relationship)
- ▶ Properties
 - Both nodes and relationships have properties
 - Useful in modeling and querying based on properties of relationships

<https://neo4j.com/developer/guide-data-modeling/>



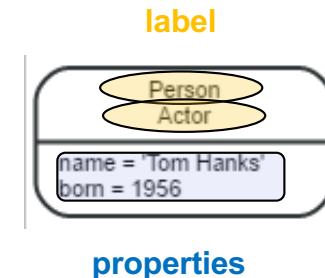
Property Graph Model Example



It models a graph with three entities: two **person** and one **movie**, each with a set of properties;
It also models the relationship among them: one person acted in the movie with a role, another person directed the movie

Property Graph Model: Nodes

- Nodes are often used to represent entities, e.g. objects
 - ▶ It has properties
 - ▶ It can have labels
- A label is a way to group similar nodes
 - ▶ It acts like the ‘class’ concept in programming world
- Label is a dynamic and flexible feature
 - ▶ It can be added or removed during run time
 - ▶ It can be used to tag node temporarily
 - E.g. :Suspend, :OnSale, etc

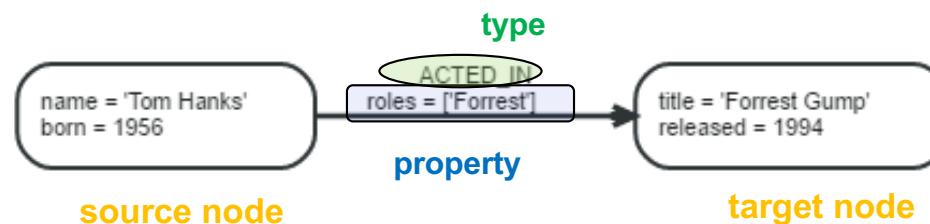


A node with two labels and two properties



Property Graph Model: Relationships

- A relationship connects two nodes: source node and target node
 - ▶ The source and the target node can be the same one
- It always has a direction
 - ▶ But traversal can happen in either direction
- It can have a type
- It can have properties

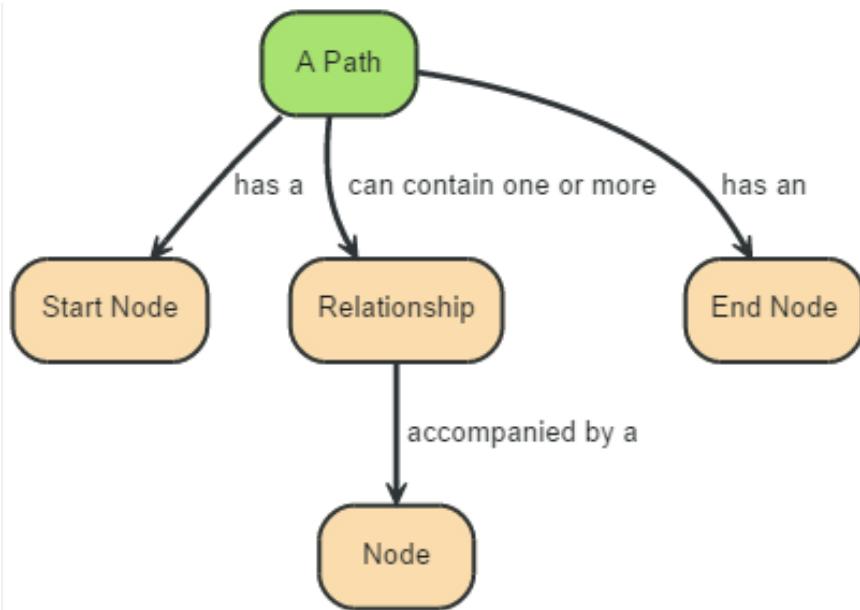


Property Graph Model: Properties

- A property is a pair of property key and property value
- The property value can be of simple type:
 - ▶ Number: Integer and Float
 - ▶ String
 - ▶ Boolean
 - ▶ Spatial Type: Point
 - ▶ Temporal Type
- The property value can also have homogeneous list of simple types as type
 - ▶ e.g. a list of integers or strings
- It cannot have heterogeneous list or other complex types with many levels of embedding



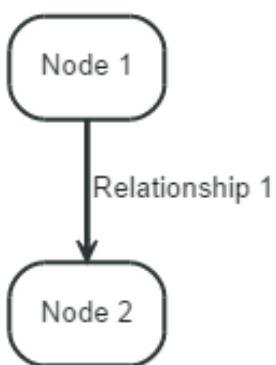
Property Graph Model: Paths



- A path is one or more nodes with connecting relationships, typically retrieved as a query or traversal result.



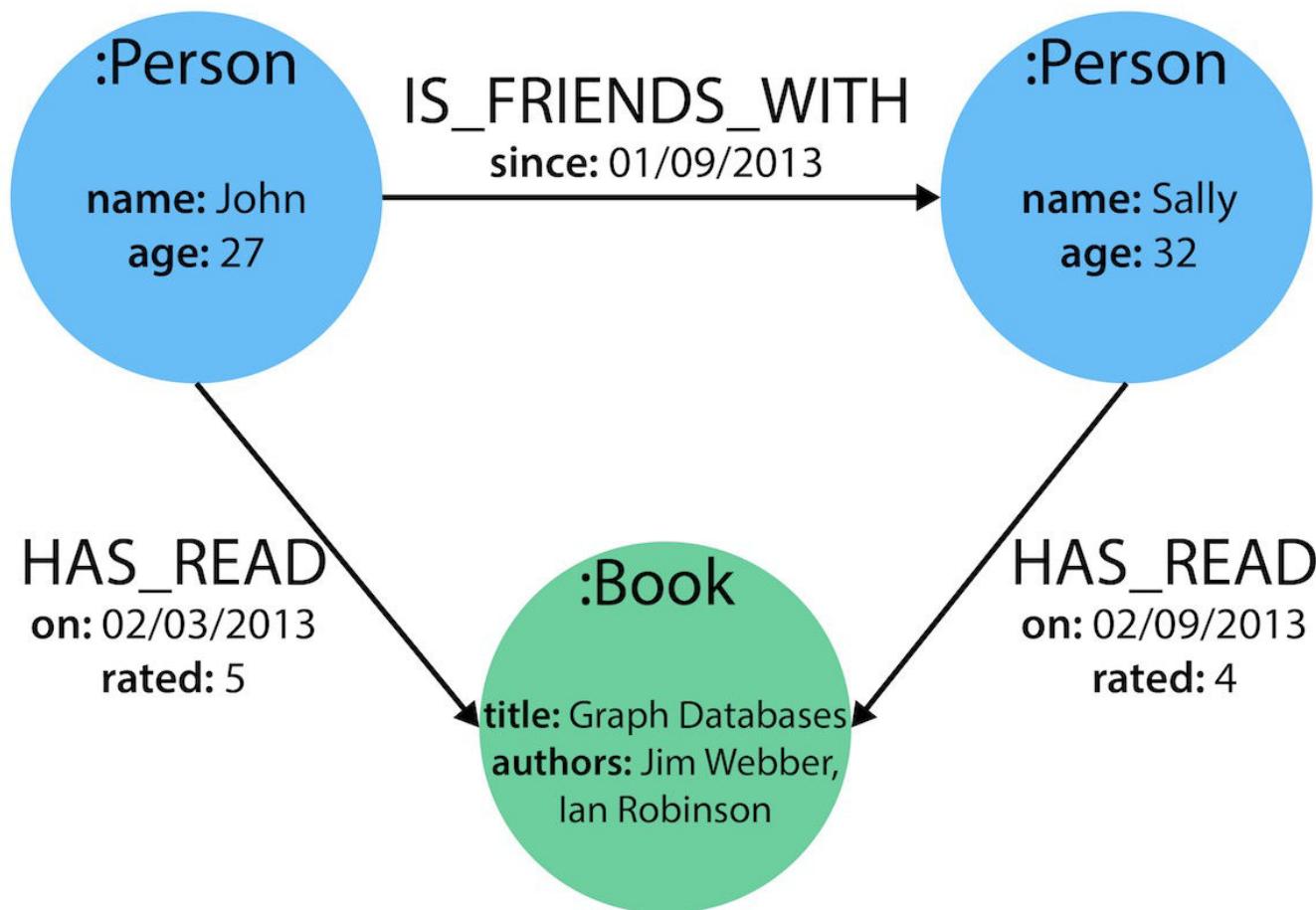
A path of length zero



A path of length one



Another Example



Outline

- Brief Review of Graphs
- Property Graph Model
- Cypher Query
 - ▶ Patterns and basic clauses
 - ▶ Subclause, subquery and functions



Cypher

- Cypher is a query language specific to Neo4j
- Easy to read and understand
- It uses **patterns** to represent core concepts in the property graph model
 - ▶ E.g. a pattern may represent that a user node is having a transaction with the item “*formula*” in it.
 - ▶ There are basic pattern representing nodes, relationships and path
- It uses **clauses** to build queries; Certain clauses and keywords are inspired by SQL
 - ▶ A query may contain multiple clauses
- **Functions** can be used to perform aggregation and other types of analysis



Cypher patterns: node

■ A single node

- ▶ A node is described using a pair of parentheses, and is typically given an identifier (variable)
- ▶ E.g.: **(n)** means a node **n**
- ▶ The variable's scope is restricted in a single query statement

■ Labels

- ▶ Label(s) can be attached to a node
- ▶ E.g.: **(a:User)** or **(a:User:Admin)**

■ Specifying properties

- ▶ Properties are a list of name value pairs enclosed in a curly brackets
- ▶ E.g.: **(a { name: "Andres", sport: "Brazilian Ju-Jitsu" })**

<https://neo4j.com/docs/developer-manual/current/cypher/syntax/patterns/>



Cypher patterns: relationships

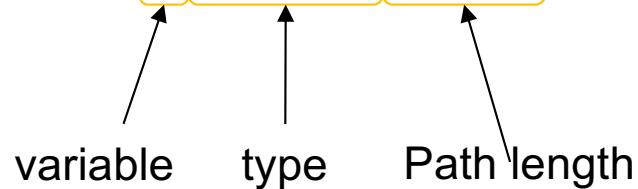
- Relationship is expressed as a pair of dashes (--)
 - ▶ Arrowhead can be added to indicate direction
 - ▶ Relationship always need a source and a target node.
- Basic Relationships
 - ▶ Directions are not important: `(a)--(b)`
 - ▶ Named relationship: `(a)-[r]->(b)`
 - ▶ Named and typed relationship: `(a)-[r:REL_TYPE]->(b)`
 - ▶ Specifying Relationship that may belong to one of a set of types:
`(a)-[r:TYPE1|TYPE2]->(b)`
 - ▶ Typed but not named relationship: `(a)-[:REL_TYPE]->(b)`
- Whether to not to name a node/relation depends on if we want to refer to them later in the query



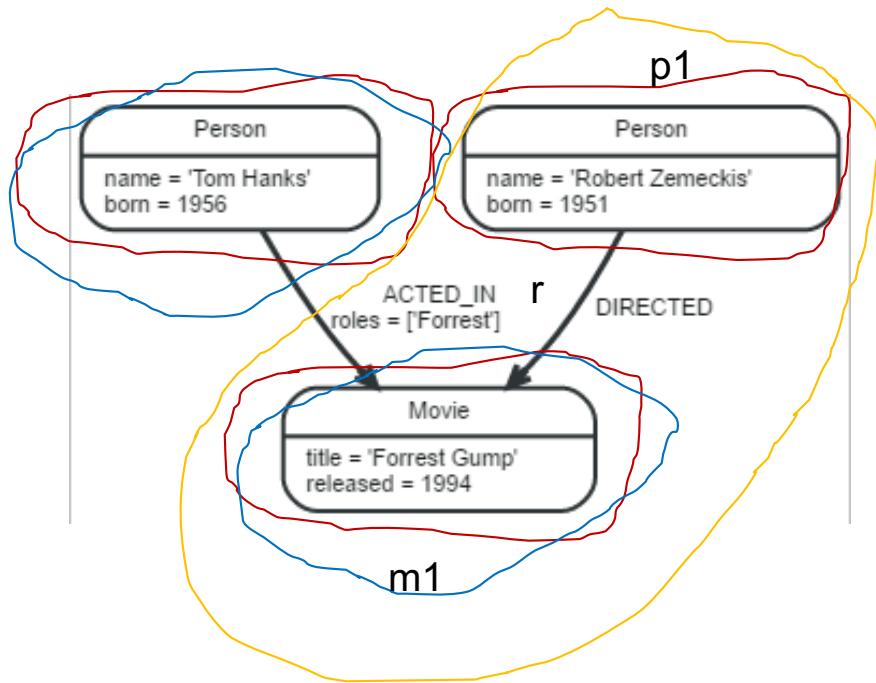
Relationship of variable lengths

- **(a)-[*2]->(b)** describes a path of length 2 between node a and node b
 - ▶ This is equivalent to **(a)-->()->(b)**
- **(a)-[*3..5]->(b)** describes a path of minimum length of 3 and maximum length of 5 between node a and node b
- Either bound can be omitted **(a)-[*3..]->(b)**, **(a)-[*..5]->(b)**
- Both bounds can be omitted as well **(a)-[*]->(b)**
- They can be named and typed as well

▶ **(a) - [r:KNOWS *1..2] ->(b)**

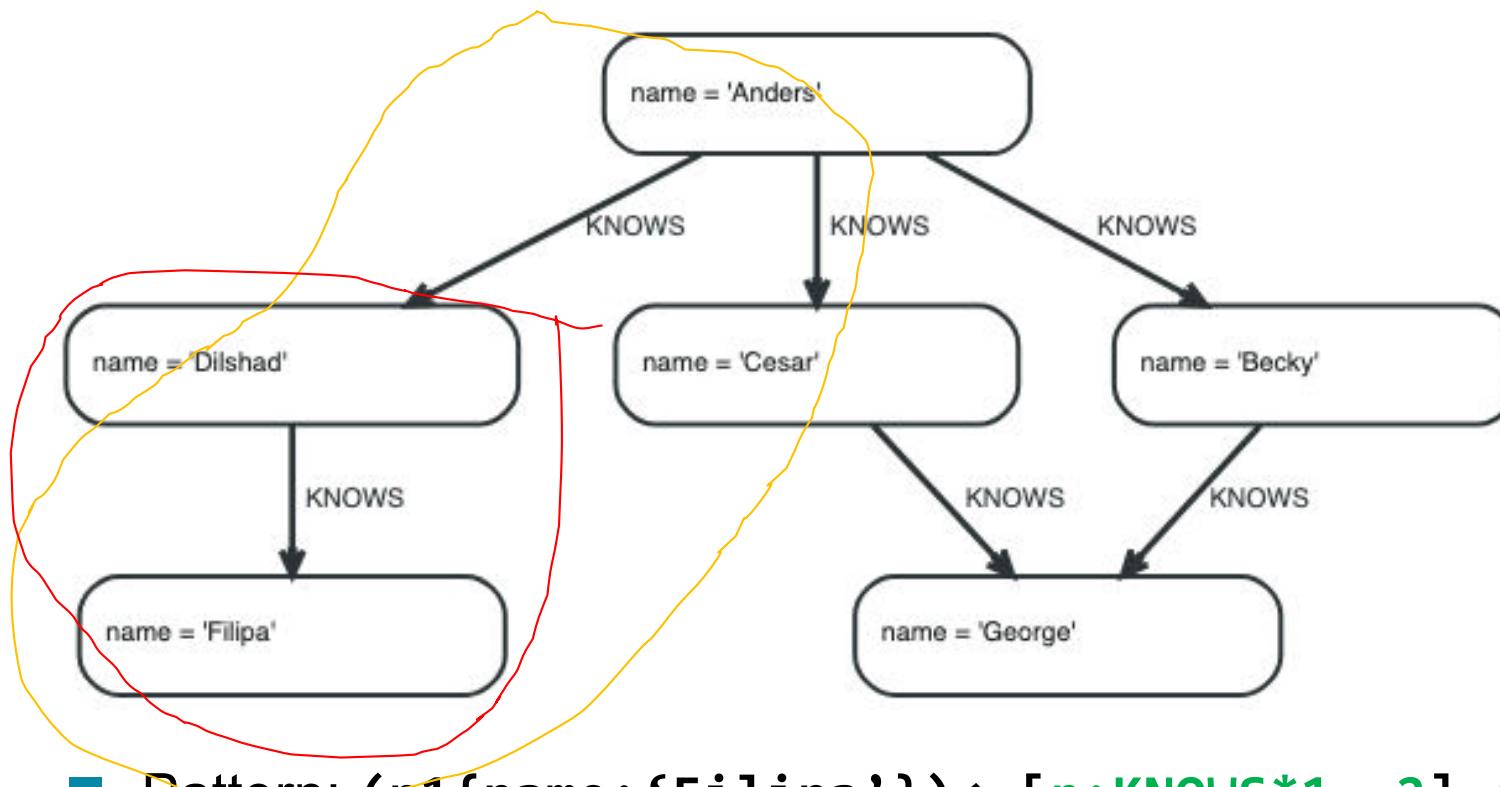


Pattern Examples



- Pattern: **(n)**
 - Matches all nodes in the graph
- Pattern: **(m:Movie)**
 - Matches the movie node in the graph
- Pattern: **(p:{name: 'Tom Hanks'})**
 - Matches the person node with name 'Tom Hanks' in the graph
- Pattern: **(p1)-[r:DIRECTED]->(m1)**
 - Matches the path from person Robert Zemeckis to movie "Forrest Gump"

Pattern Examples



- Pattern: `(p1{name: 'Filipa'})-<-[r:KNOWS*1..2]-()`
- Matches
 - ▶ the path from Dilshad to Filipa (length 1)
 - ▶ The path from Anders to Filipa (length 2)

<https://neo4j.com/docs/cypher-manual/4.1/syntax/patterns/>



Create Clause

■ CREATE *pattern*

- ▶ Create nodes or relationships with properties

■ Create a node **matrix1** with the label **Movie**

```
CREATE (matrix1:Movie {title:'The Matrix', released:1999,  
tagline:'Welcome to the Real World'})
```

We give the node an identifier so we can refer to the particular node later in the same query

■ Create a node **keanu** with the label **Actor**

```
CREATE (keanu:Actor {name:'Keanu Reeves', born:1964})
```

■ Create a relationship **ACTS_IN**

```
CREATE (keanu)-[:ACTS_IN {roles:'Neo'}]->(matrix1)
```

The identifier "Keanu" and "matrix1" are used in the this create clause.

We did not give the relationship a name/identifier.

We need to write the three clauses in a single query statement to be able to use those variables



Read Clause

■ MATCH *pattern*

RETURN *var-expression*

- ▶ MATCH is the main reading clause
- ▶ RETURN is a projecting clause
- ▶ They are chained to make a query

■ Return all nodes:

```
MATCH (n) RETURN n
```

■ Return all nodes with a given label: select * from movie

```
MATCH (movie:Movie) RETURN movie
```

■ Return all actors' name in the movie "The Matrix"

We give the Actor node an identifier "a" so we can use refer to in the RETURN sub-clause

```
MATCH (a:Actor) -[:ACTS_IN] -> (:Movie{title:"The Matrix"})  
RETURN a.name
```

We do not need to return the relationship so we did not give an identifier to it

We do not need to give an identifier to the Movie node too,



Update Clause

■ **MATCH** *pattern*

SET/REMOVE *properties/Labels*

■ Set the age property for all actor nodes

```
MATCH (n:Actor)  
SET n.age = 2014 - n.born  
RETURN n
```

■ Remove a property

```
MATCH (n:Actor)  
REMOVE n.age  
RETURN n
```

■ Remove a label

```
MATCH (n:Actor{name:"Keanu Reeves"})  
REMOVE n:Actor  
RETURN n
```



MERGE Clause: basic form

■ MERGE clause acts like an upsert:

- ▶ updating an existing pattern when there is a match or create a new one when there is no match

■ Simplest form is

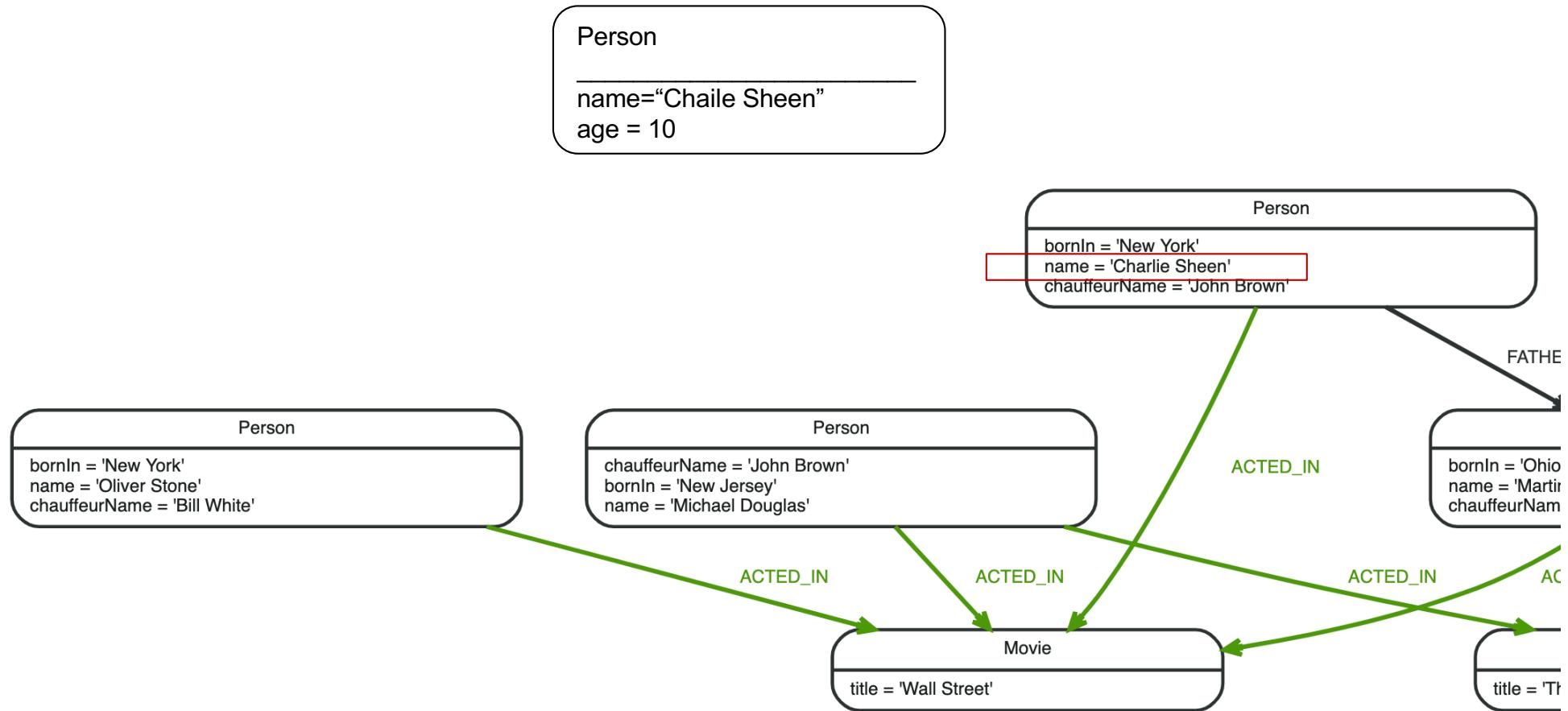
- ▶ **MERGE *pattern***
- ▶ Example:
- ▶ **MERGE (charlie { name: 'Charlie Sheen', age: 10 })
RETURN Charlie**
- ▶ Create a new node if we could not find a node with all matching properties in the current graph

<https://neo4j.com/docs/cypher-manual/4.1/clauses/merge/>



Example Graph

```
MERGE (charlie { name: 'Charlie Sheen', age: 10 })
RETURN Charlie
```

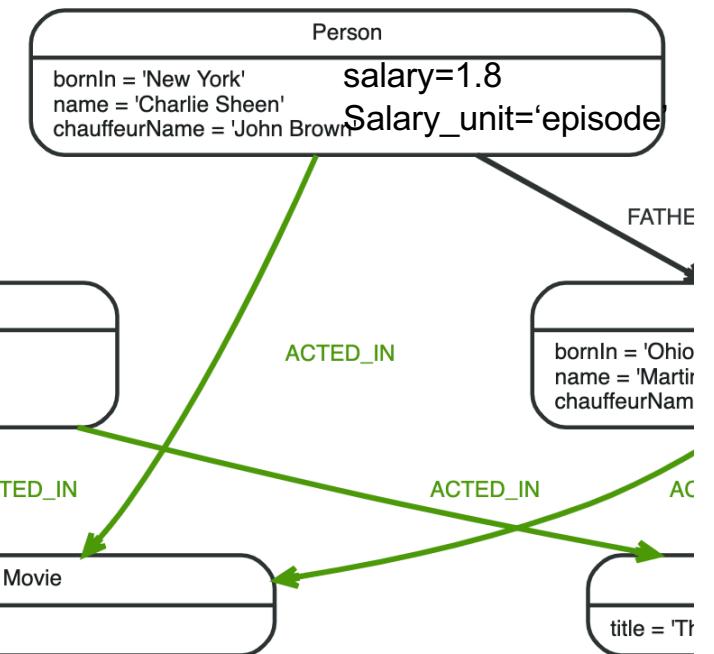


MERGE Clause: property

■ MERGE pattern

SET properties/Labels

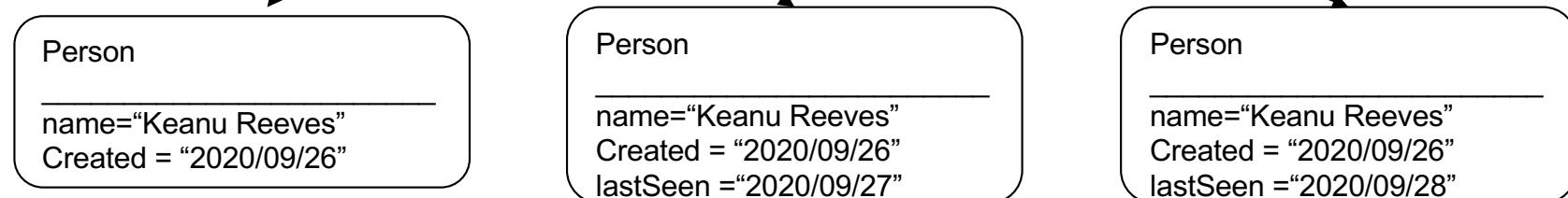
```
MERGE (c{ name: 'Charlie Sheen' })
SET c.salary = 1.8,
    c.salary_unit='episode'
```



MERGE Clause: property

Specifying different actions on insert and update

```
MERGE (keanu:Person { name: 'Keanu Reeves' })
ON CREATE SET keanu.created = timestamp()
ON MATCH SET keanu.lastSeen = timestamp()
RETURN keanu.name, keanu.created, keanu.lastSeen
```



MERGE Clause: relationship

- **MATCH** *node_pattern(s)*
MERGE *relationship_pattern*
- Example:

```
MATCH (charlie:Person { name: 'Charlie Sheen' }), (wallStreet:Movie { title: 'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN charlie.name, type(r), wallStreet.title
```

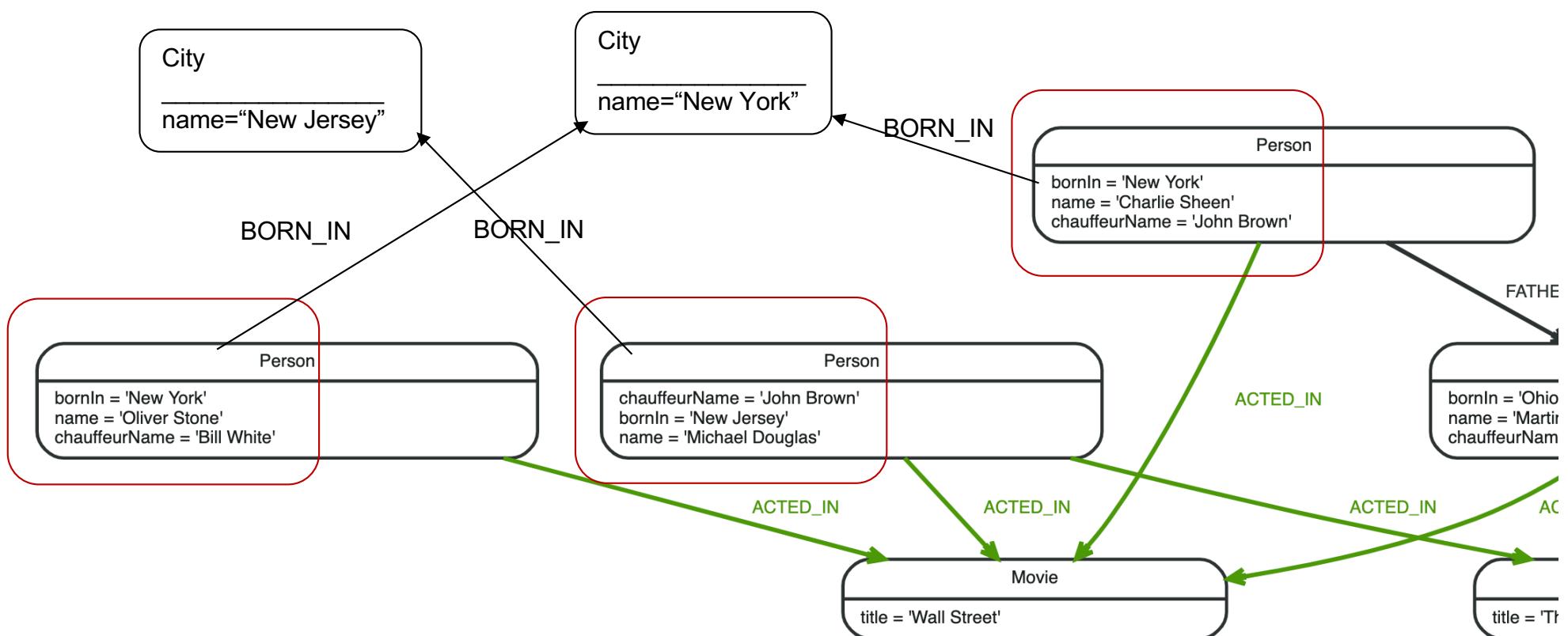
- **MATCH** *node_pattern*
MERGE *node_pattern*
MERGE *relationship_pattern*

```
MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city
```

Example Graph

```

MATCH (person:Person)
MERGE (city:City { name: person.bornIn })
MERGE (person)-[r:BORN_IN]->(city)
RETURN person.name, person.bornIn, city
  
```



```

MATCH (charlie:Person { name: 'Charlie Sheen' }), (wallStreet:Movie { title: 'Wall Street' })
MERGE (charlie)-[r:ACTED_IN]->(wallStreet)
RETURN charlie.name, type(r), wallStreet.title
  
```



MERGE Clause: Usage and Performance

- One major use case of **MERGE** is to create graph model from source data
 - ▶ CSV, JSON, XML, ..
- There are always gaps between source data format and the desirable graph model
 - ▶ Properties need to be extracted from columns and assigned
 - ▶ Relationships need to be built across different lines
- **MERGE** will be called repeatedly in building graph from raw data
 - ▶ Call **MERGE** multiple times per line of source data
- It is very important to build index before bulk loading with **MERGE**



Cypher - Delete

- **MATCH *pattern* DELETE *var-expression***

- Delete relationship

```
MATCH (n{name:"Keanu Reeves"})-[r:ACTS_IN]->()
DELETE r
```

- Delete a node and all possible relationship

```
MATCH (m{title:'The Matrix'})-[r]-()
DELETE m,r
```



Outline

■ Brief Review of Graphs

■ Property Graph Model

■ Cypher Query

- ▶ Patterns and basic clauses
- ▶ Subclause, function and Subqueries



MATCH: sub-clauses

- The WHERE sub clause can be used to specify various query conditions

- ▶ Boolean operators AND, OR, NOT, XOR can be used

```
MATCH (n)
WHERE n.age <30 AND n.employ>=3
RETURN n.name
```

- ▶ It can be used to chain an existential sub queries, but you may find an easier way of writing the same query

```
MATCH (person:Person)
WHERE EXISTS {
    MATCH (person)-[:HAS_DOG]->(dog :Dog)
    WHERE person.name = dog.name
}
RETURN person.name as name
```

```
1 MATCH (person:Person)-[:HAS_DOG]→(dog :Dog)
2 WHERE person.name = dog.name
3 RETURN person.name as name
```



Functions

- Functions may appear in various clauses

- ▶ Build-in and user-defined functions

- Build-in functions

- ▶ **Predicate functions**
 - ▶ Scalar functions
 - ▶ **Aggregation functions**
 - ▶ List functions
 - ▶ Mathematical functions
 - ▶ String functions
 - ▶ Temporal functions
 - ▶ Spatial Functions



Predicate Functions

■ They are boolean functions that return true or false for a given set of non-null input. They are most commonly used to filter out subgraphs in the WHERE part of a query.

- ▶ `all()`, `any()`, `exists()`,`none()`,`single()`

■ `all()` usage

- ▶ `all(variable IN list WHERE predicate)`

Assign a variable to the entire path

```
MATCH p =(a)-[*1..3]->(b)
WHERE a.name = 'Alice' AND b.name = 'Daniel' AND ALL (x IN nodes(p) WHERE x.age > 30)
RETURN p
```

- ▶ All nodes in the returned paths should have an age property of at least '30'.

A function returns all nodes of a path

■ `any()`,`single()`, and `none()` have similar signature but different meanings



Predicate Functions

■ `exists()` usage

- ▶ `exists(pattern-or-property)`

```
MATCH (n)
WHERE EXISTS (n.name)
RETURN n.name AS name, EXISTS ((n)-[:MARRIED]->()) AS is_married
```

- ▶ The names of all nodes with the name property are returned, along with a boolean true / false indicating if they are married.
- ▶ The first `EXISTS()` function does filtering because it is used in WHERE clause
- ▶ The second `EXISTS()` does not filter anything because it is used in the return clause, it only computes and returns value



Aggregating Functions

- GROUP BY feature in Neo4j is achieved using aggregating functions
 - ▶ E.g. `count()`, `sum()`, `avg()`, `max()`, `min()` and so on
- The grouping key is implied in the RETURN clause
 - ▶ None aggregate expression in the return clause is the grouping key
 - ▶ `RETURN n, count(*)`
 - `n` is a variable declared in a previous clause, and it is the grouping key
 - ▶ `MATCH(n:Person) RETURN n.gender, COUNT(*)`
 - Count the number of nodes representing each gender in the graph
 - A person's gender is the grouping key
- A grouping key is not always necessary, the aggregation function can apply to all results returned
 - ▶ `MATCH (n:Person) RETURN COUNT(*)`
 - To count the number of Person nodes in the graph



Aggregation Examples

- To find out the earliest year a Person was born in the data set

```
MATCH (n:Person) RETURN min (n.born)
```

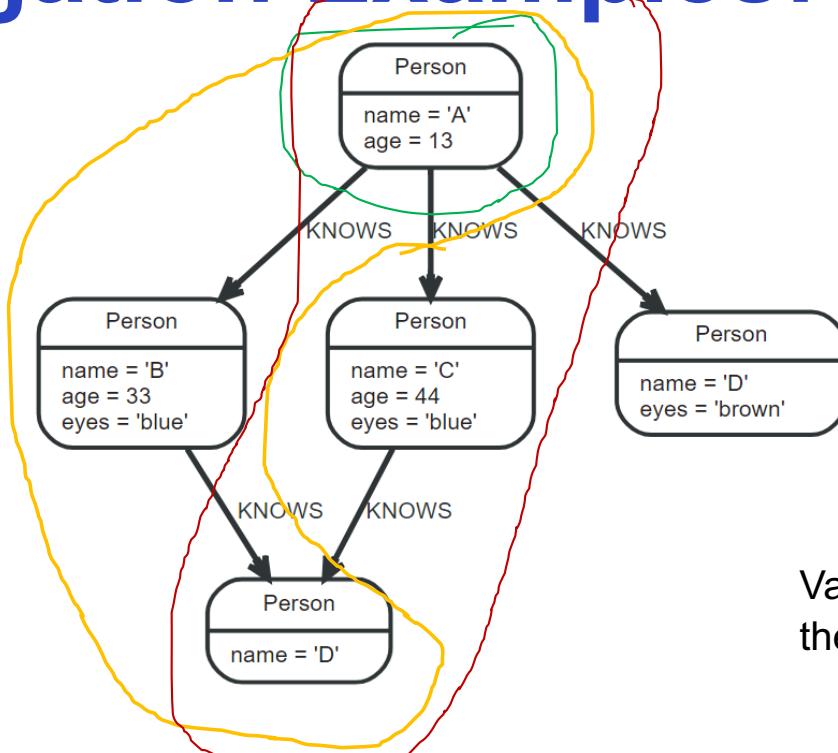
- To find out the distribution of relationship types belonging to nodes with certain feature

```
MATCH (n { name: 'A' })-[r]->()
RETURN type(r), count(*)
```

The grouping key is `type(r)` which is a scalar function, returns the type of relationship in the matching results



Aggregation Examples: DISTINCT



Variable **friend_of_friend** refers to the same node in both matches

MATCH (me:Person)-->(friend:Person)-->(**friend_of_friend:Person**)

WHERE me.name = 'A'

RETURN count(DISTINCT friend_of_friend), count(friend_of_friend)

count(DISTINCT friend_of_friend)	count(friend_of_friend)
1	2
1 row	



MATCH: subqueries

- The **WITH** clause can chain different query parts together in a pipeline style
 - ▶ Used to apply conditions on aggregation result
 - ▶ Used to modify (order, limiting, etc) the results before collecting them as a list
- Examples
 - ▶ Find the person who has directed 3 or more movies

```
MATCH (p:Person)-[r:DIRECTED]->(m:Movie)  
WITH p, count(*) as movies  
WHERE movies >= 3  
RETURN p.name, movies
```
 - ▶ Return the oldest 3 person as a list

```
MATCH (n:Person)  
WITH n  
ORDER by n.age DESC LIMIT 3  
RETURN collect(n.name)
```

```
MATCH (n:Person)  
RETURN n.name  
ORDER by n.age DESC LIMIT  
3
```



Dealing with Array type

- Array literal is written in a similar way as it is in most programming languages
 - ▶ examples
 - An array of integer: [1, 2, 3]
 - An array of string: ["Sydney", "University"]
- Both node and relationship can have property of array type
 - ▶ Example: create an relationship with array property

```
create (Keanu)-[:ACTED_IN {roles: ['Neo']}]->(TheMatrix)
```
 - ▶ Example: update an existing node with array property

```
MATCH (n:Person{name: "Tom Hanks"})  
set n.phone=["0123456789", "93511234"]
```



Dealing with Array type (cont'd)

■ Querying array property

- ▶ The **IN** operator: check if a value is in an array

- Example: find out who has played 'Neo' in which movie

```
MATCH (a:Person) -[r:ACTED_IN]->(m:Movie)  
WHERE 'Neo' IN r.roles  
RETURN a , m
```

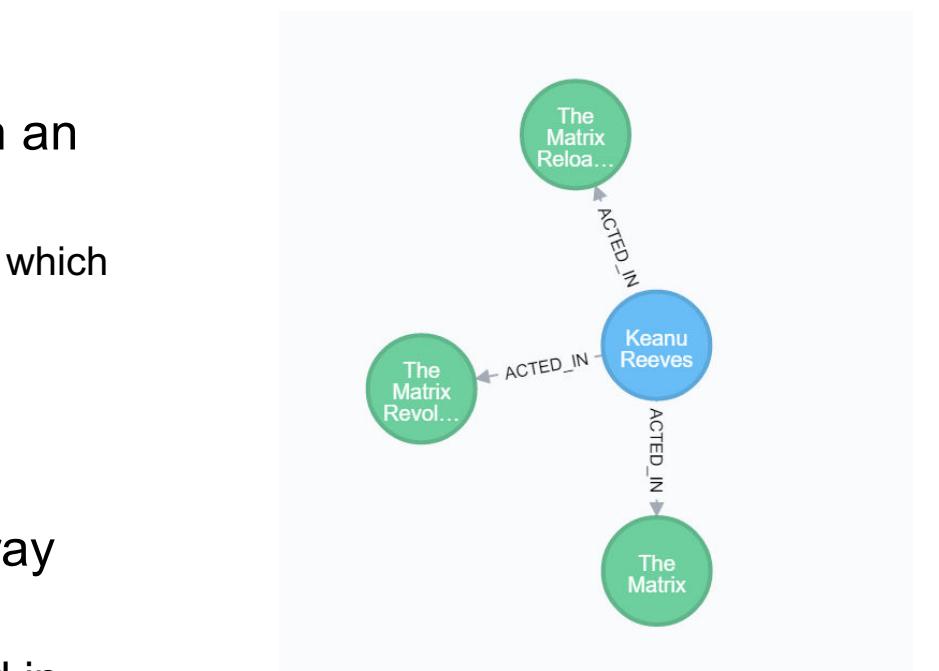
- ▶ The **UNWIND** operator: flatten an array into multiple rows

- Example: find all the movies released in 1999 or in 2003

```
UNWIND [1999,2003] as year  
MATCH (m: Movie)  
WHERE m.released = year  
RETURN m.title, m.released
```

This is equivalent to

```
MATCH(m: Movie)  
WHERE m.released IN [1999,2003]  
RETURN m.title, m.released
```



\$ UNWIND [1999,2003] as year MATCH (m: Movie) WHERE m.releas...

Rows	m.title	m.released
A	The Matrix	1999
Text	Snow Falling on Cedars	1999
	The Green Mile	1999
	Bicentennial Man	1999
	The Matrix Reloaded	2003
	The Matrix Revolutions	2003
	Something's Gotta Give	2003

Returned 7 rows in 37 ms.



Dealing with Array Type (cont'd)

■ A relatively complex query

- ▶ Update another node

```
MATCH (n:Person{name: "Meg Ryan"}) set n.phone=["0123456789"]
```

- ▶ Run a query to see who shares any phone number with Tom Hanks

```
MATCH (n:Person{name: "Tom Hanks"})
```

```
WITH n.phone as phones, n
```

```
UNWIND phones as phone
```

```
MATCH (m:Person)
```

```
WHERE phone in m.phone and n<>m
```

```
RETURN m.name
```

Where to find more about cypher query:

Developer's guide: <http://neo4j.com/docs/developer-manual/current/cypher/>

Reference card: <https://neo4j.com/docs/cypher-refcard/current/>



Indexing

- Neo4j supports index on properties of labelled node
- Index has similar behaviour as those in relational systems
- Create Index
 - ▶ **CREATE INDEX ON :Person(name)**
- Drop Index
 - ▶ **DROP INDEX ON :Person(name)**
- Storage and query execution will be covered in week 7



References

- Ian Robinson, Jim Webber and Emil Eifrem, *Graph Databases*, Second Edition, O'Reilly Media Inc., June 2015
 - ▶ You can download this book from the Neo4j site, <http://www.neo4j.org/learn> will redirect you to <http://graphdatabases.com/>
- The Neo4j Document
 - ▶ The Neo4j Graph Database Concept (<http://neo4j.com/docs/stable/graphdb-neo4j.html>)
 - ▶ Cypher manual (<https://neo4j.com/docs/cypher-manual/current/introduction/>)
- Noel Yuhanna, *Market Overview: Graph Databases*, Forrester White Paper, May, 2015
- Renzo Angeles, *A Comparison of Current Graph Data Models*, ICDE Workshops 2013 (DOI-10.1109/ICDEW.2012.31)
- Renzo Angeles and Claudio Gutierrez, *Survey of Graph Database Models*, ACM Computing Surveys, Vol. 40, N0. 1, Article 1, February 2008 (DOI-10.1145/1322432.1322433)



COMP5338 – Advanced Data Models

Week 7: Neo4j Internal and Data Modelling

Dr. Ying Zhou
School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

- Neo4j Storage
- Neo4j Query Plan and Indexing
- Neo4j – Data Modeling

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Materials adapted by permission from *Graph Databases (2nd Edition)* by Ian Robinson et al (O'Reilly Media Inc.). Copyright 2015 Neo Technology, Inc



Property Graph Model

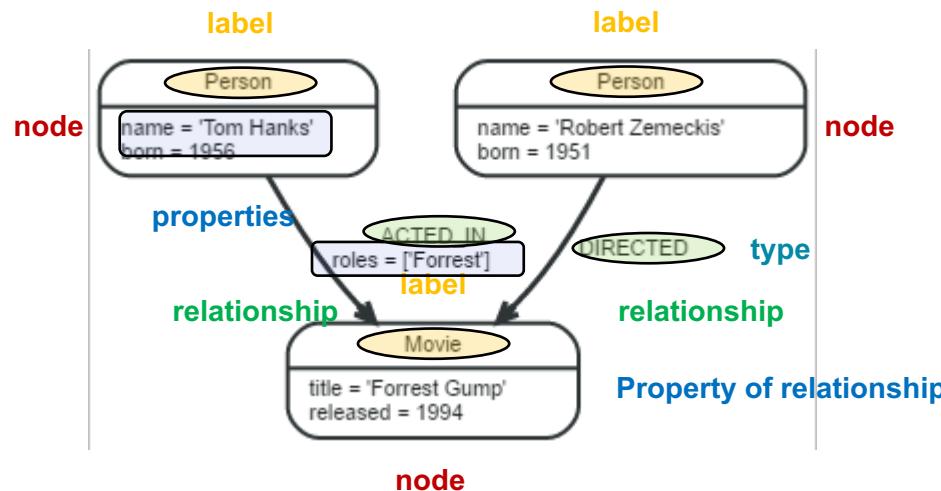


Table concept is not part of the data model

The database is a large graph, could contain independent subgraphs

Community edition only supports one user database; Enterprise edition supports multiple user databases

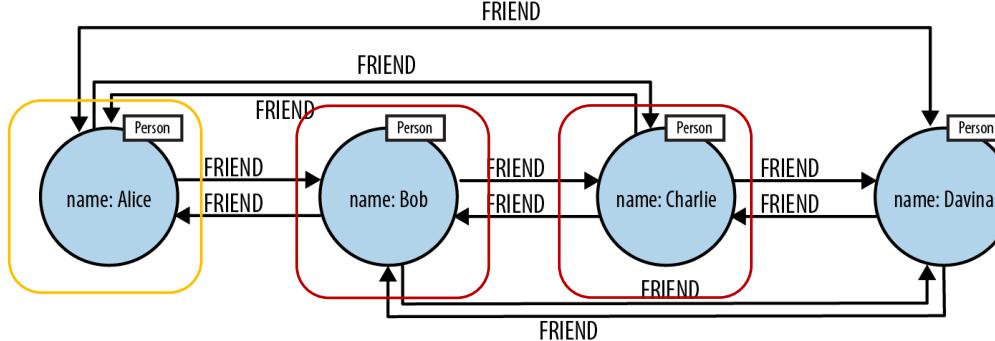
Logic data model and physical storage model could be totally different

It is theoretically possible to construct a property graph model with any storage backend



Index-free Adjacency

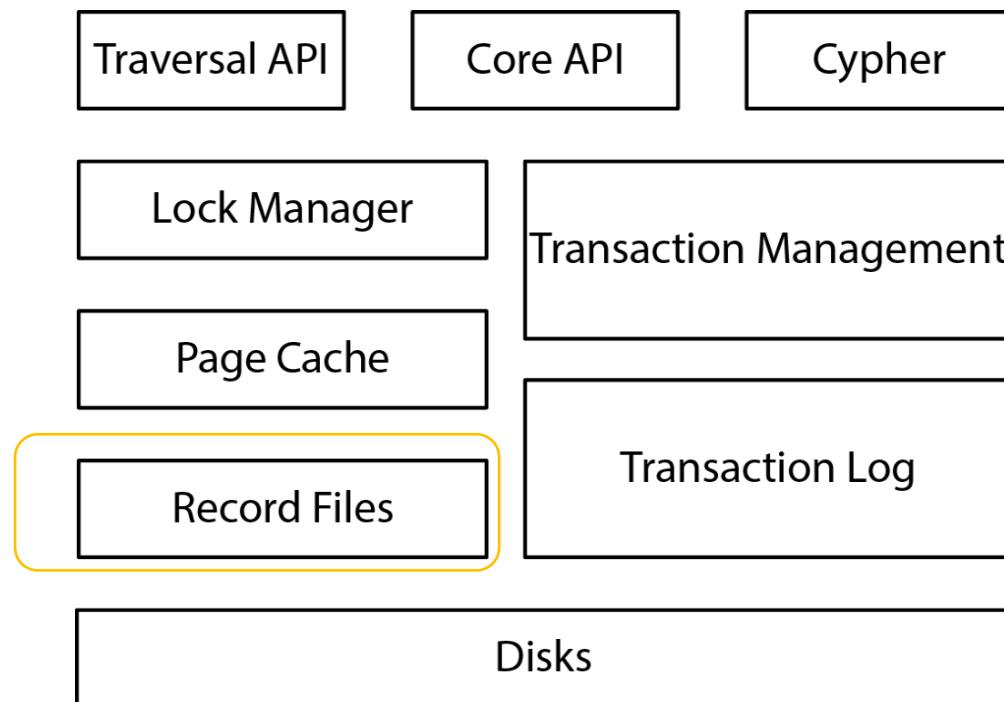
- Native storage of relationships between nodes
 - ▶ Effectively a pre-computed bidirectional join
- Traversal is like pointer dereferencing
 - ▶ Almost as fast as well
- Index-free Adjacency
 - ▶ Each node maintains a direct link to its adjacent nodes
 - ▶ Each node is effectively a micro-index to the adjacent nodes
- Cheaper than global indexes
 - ▶ Query are faster, do not depends on the total size of the graph



Slides 4-11 are based on Graph Database chapter 6.1 and 6.2



Neo4j Architecture



Page 163 of Graph Database



Property Graph and Store files

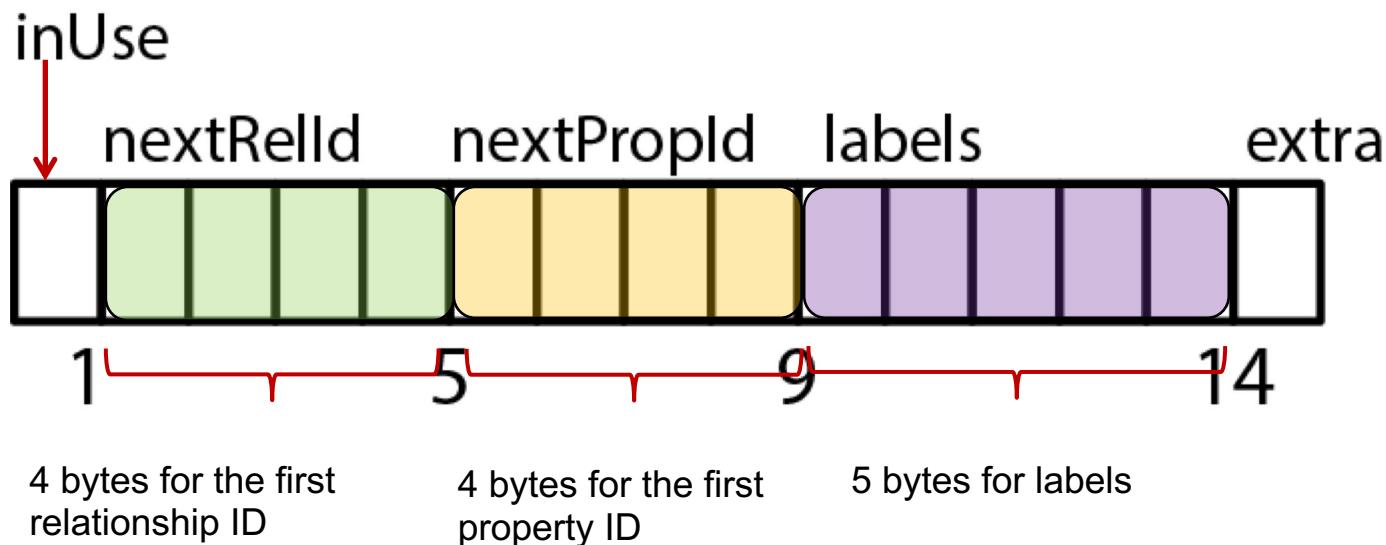
- Graph data is stored in *store files* on disk
 - ▶ Nodes, relationships, properties, labels and types all have their own store files.
 - Check under <neo4j-home>/data/databases/neo4j
 - ▶ Separating graph structure and property data promotes fast traversal
- Node, relationship, property, label and type all have system assigned IDs
- They are stored as fixed length record in respective stores
- user's view of their graph and the actual records on disk are structurally dissimilar

Size	Date	File Name
48K	1 Oct 22:56	neostore.relationshipgroupstore.db.id
1.1M	1 Oct 22:56	neostore.relationshipstore.db
96K	1 Oct 22:56	neostore.relationshipstore.db.id
8.0K	1 Oct 22:17	neostore.relationshiptypestore.db
40K	1 Oct 22:56	neostore.relationshiptypestore.db.id
8.0K	1 Oct 22:17	neostore.relationshiptypestore.db.names
40K	1 Oct 22:56	neostore.relationshiptypestore.db.names
568K	1 Oct 22:56	neostore.nodestore.db
88K	1 Oct 22:56	neostore.nodestore.db.id
8.0K	1 Oct 21:09	neostore.nodestore.db.labels
40K	1 Oct 22:56	neostore.nodestore.db.labels.id
1.3M	1 Oct 22:56	neostore.propertystore.db
1.2M	1 Oct 22:56	neostore.propertystore.db.arrays
48K	1 Oct 22:56	neostore.propertystore.db.arrays.id
112K	1 Oct 22:56	neostore.propertystore.db.id
8.0K	1 Oct 22:17	neostore.propertystore.db.index
40K	1 Oct 22:56	neostore.propertystore.db.index.id
8.0K	1 Oct 22:17	neostore.propertystore.db.index.keys
40K	1 Oct 22:56	neostore.propertystore.db.index.keys.id
8.0K	1 Oct 21:34	neostore.propertystore.db.strings
48K	1 Oct 22:56	neostore.propertystore.db.strings.id



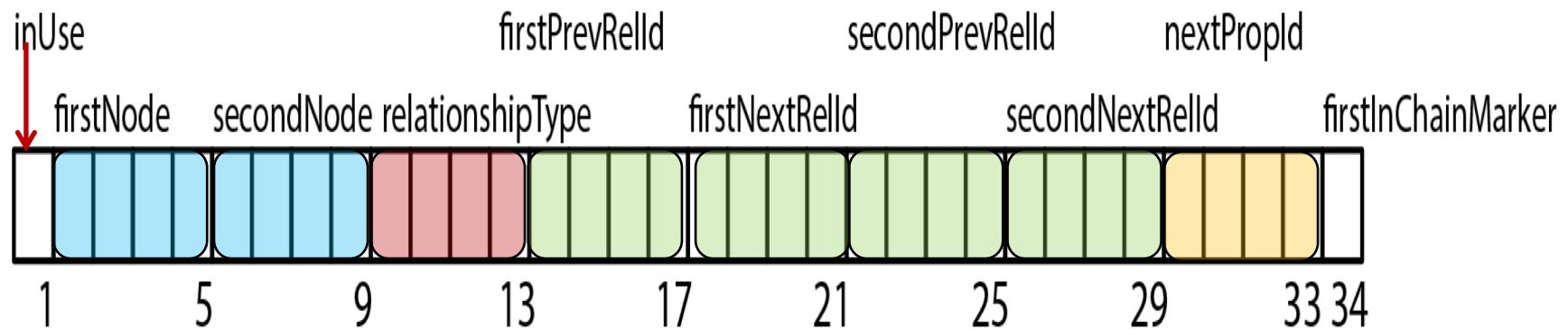
Node store file

- All node data is stored in **one** node store file
- Physically stored in file named *neostore.nodestore.db*
- Each record is of a **fixed size** – 15 bytes (*was 9 bytes in earlier version*)
- Offset of stored node = node id * 15 (node id = 100, offset = 1500)
- Deleted IDs in *.id* file and can be reused



Relationship store file

- All relationship data is stored in **one** relationship store file
- Physically stored in file named *neostore.relationshipstore.db*
- Each record is of a fixed size – 34 bytes
- Offset of stored relationship = relationship id * 34
 - ▶ So, relationship id = 10, offset = 340



Implications

- Both Node ID and Property ID are of 4 bytes
 - ▶ The maximum ID value is $2^{32} - 1$
 - There is a maximum number of nodes/relationships in a database
 - ▶ ID is assigned and managed by the system
 - The corresponding record will be stored in the computed offset
 - ▶ The IDs of deleted nodes/relationships will be reused



Other Files

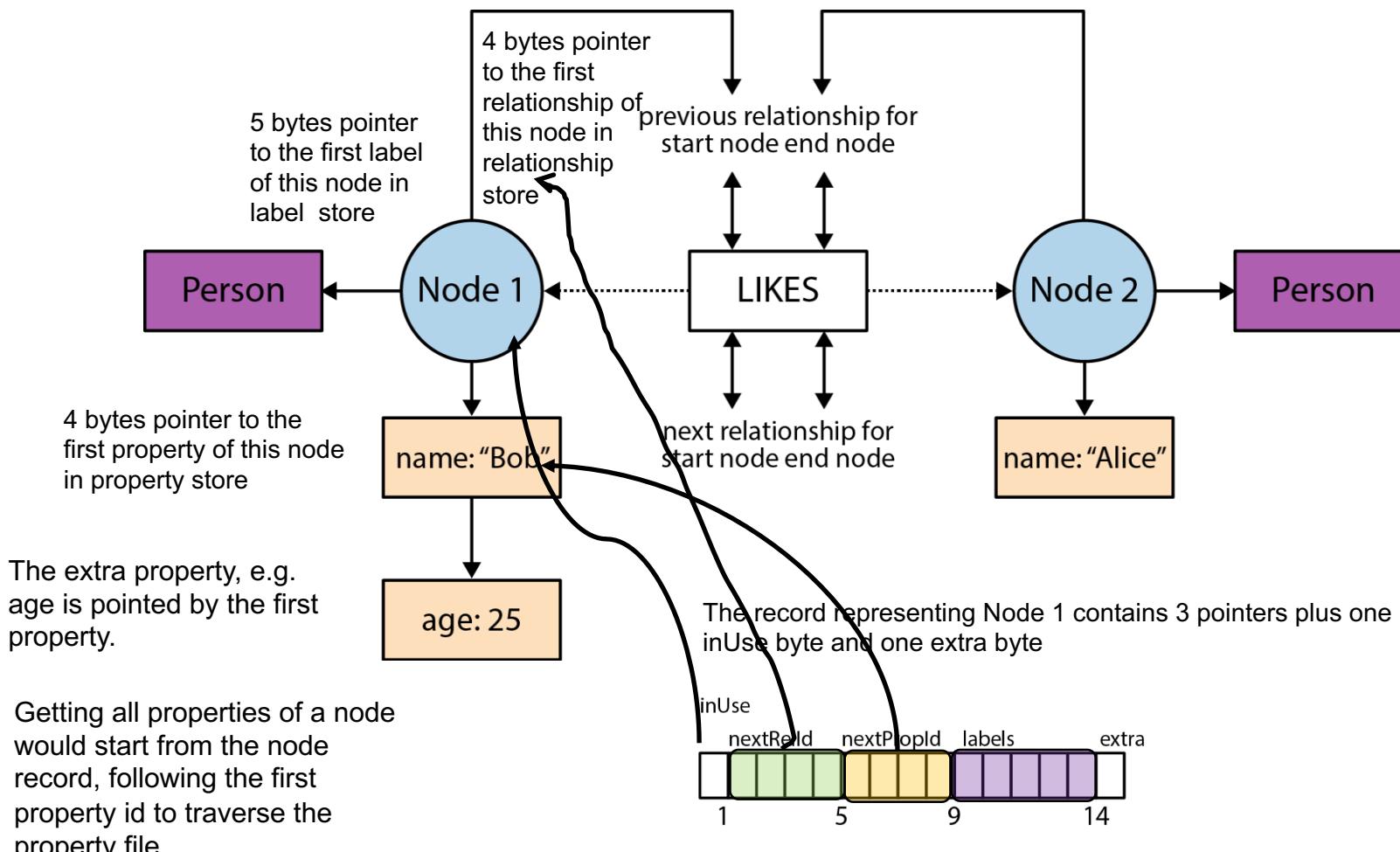
- **Property store** contains fixed size records to store properties for nodes and relationships
 - ▶ Simple properties are stored inline
 - ▶ Complex ones such as long string or array property are stored elsewhere
- Node label in node records references data in **label store**
- Relationship type in relationship record references data in **relationship type store**

10K	1 Oct 22:56	neostore.tabletokenstore.firebaseio.names.id
568K	1 Oct 22:56	neostore.nodestore.db
88K	1 Oct 22:56	neostore.nodestore.db.id
8.0K	1 Oct 21:09	neostore.nodestore.db.labels
40K	1 Oct 22:56	neostore.nodestore.db.labels.id
48K	1 Oct 22:56	neostore.relationshipgroupstore.db.tu ¹
1.1M	1 Oct 22:56	neostore.relationshipstore.db
96K	1 Oct 22:56	neostore.relationshipstore.db.id
8.0K	1 Oct 22:17	neostore.relationshiptypestore.db
40K	1 Oct 22:56	neostore.relationshiptypestore.db.id
8.0K	1 Oct 22:17	neostore.relationshiptypestore.db.names
40K	1 Oct 22:56	neostore.relationshiptypestore.db.names.id
]
		neostore.propertystore.db
		neostore.propertystore.db.arrays
		neostore.propertystore.db.arrays.id
		neostore.propertystore.db.id
		neostore.propertystore.db.index
		neostore.propertystore.db.index.id
		neostore.propertystore.db.index.keys
		neostore.propertystore.db.index.keys.id
		neostore.propertystore.db.strings
		neostore.propertystore.db.strings.id



Node structure

Bob LIKES Alice

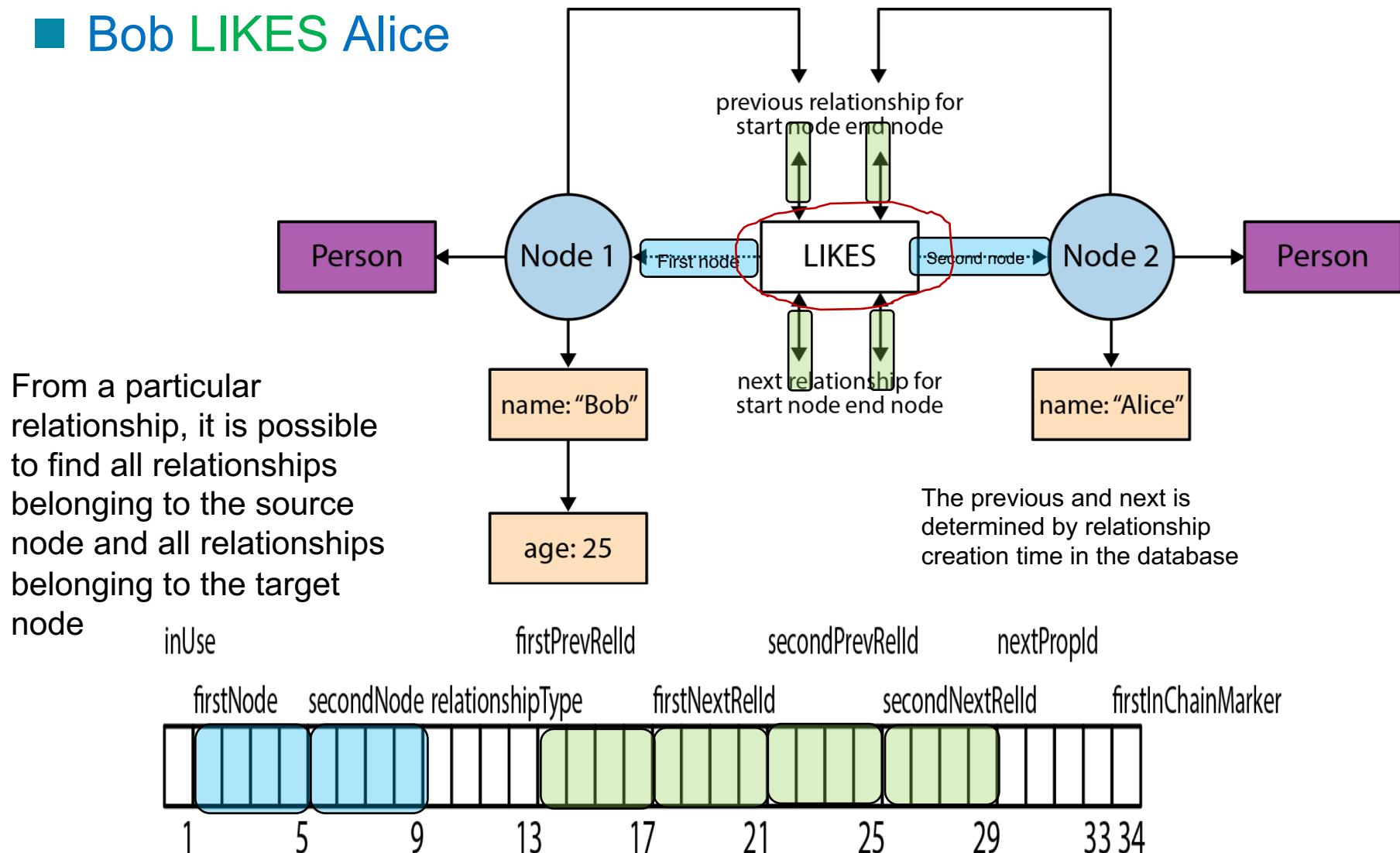


Getting all properties of a node would start from the node record, following the first property id to traverse the property file.

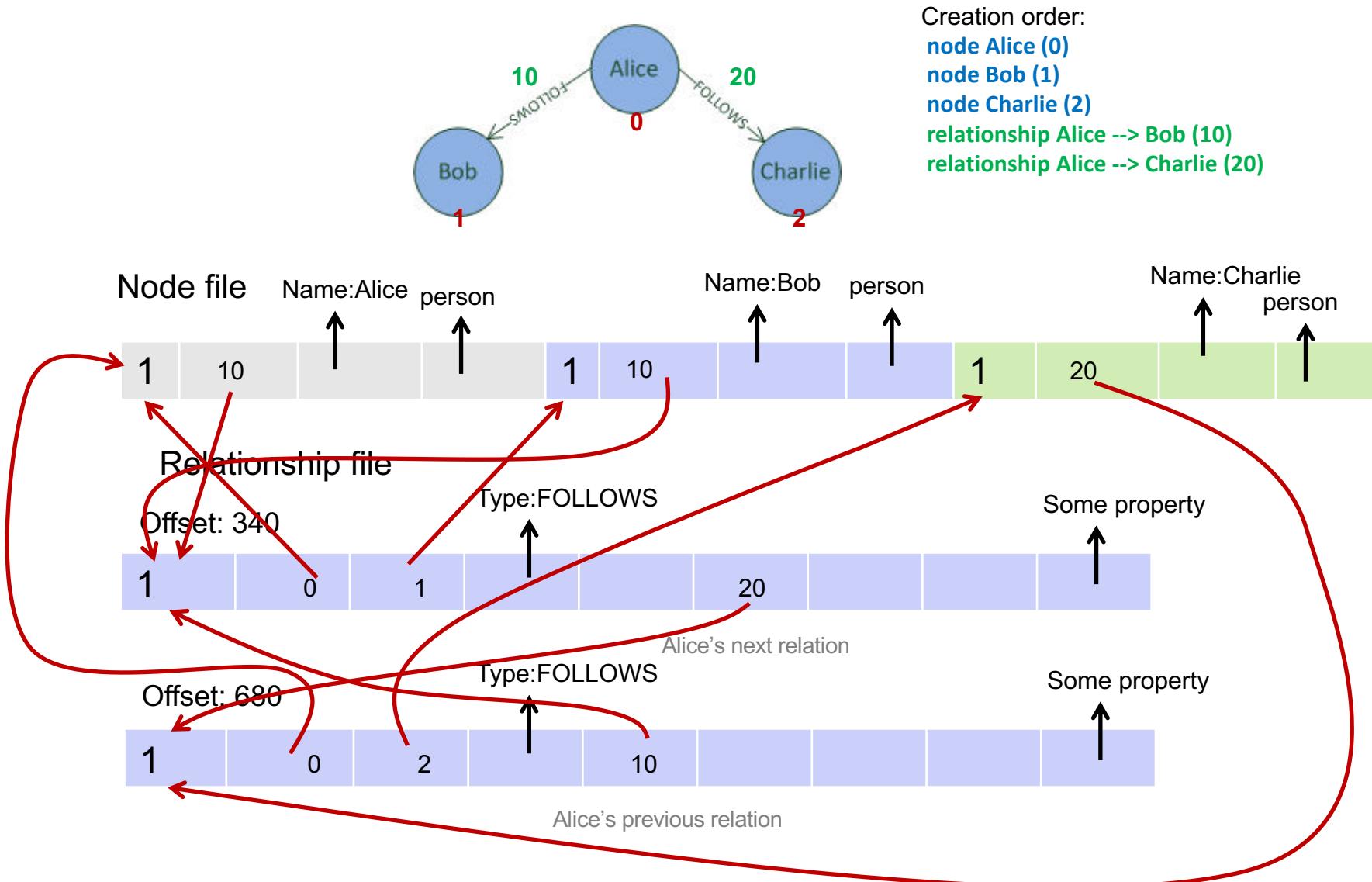


Relationship structure

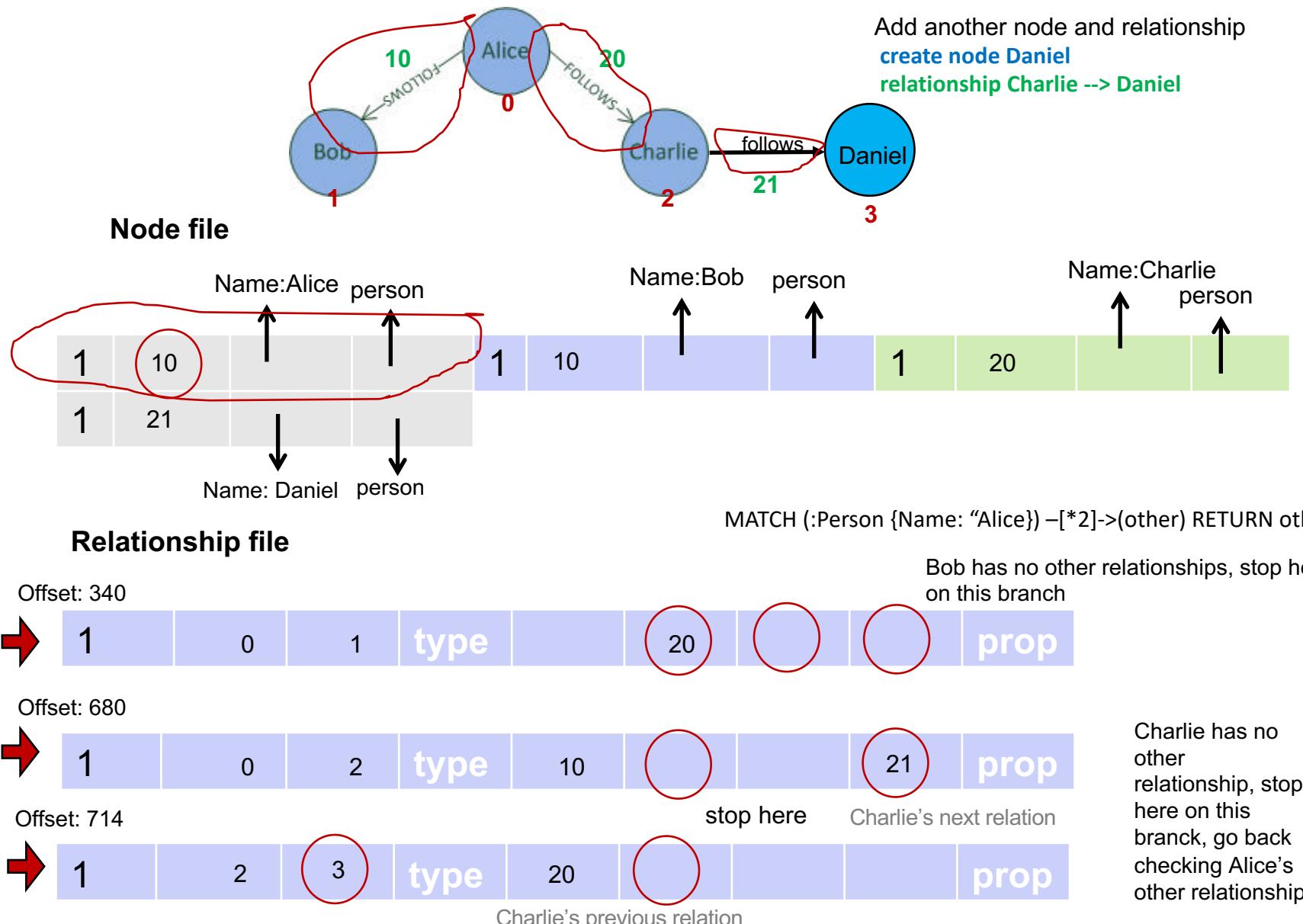
Bob LIKES Alice



Doubly linked list



Doubly linked list (cont'd)



*“The node and relationship stores are concerned **only** with the **structure** of the graph, not its property data. Both stores use fixed-sized records so that any individual record’s location within a store file can be rapidly computed given its ID. These are critical design decisions that underline Neo4j’s commitment to high-performance traversals.”*

-- Chapter 6, Graph Databases



Outline

- Neo4j Storage
- **Neo4j Query Plan and Indexing**

- Neo4j – Data Modeling

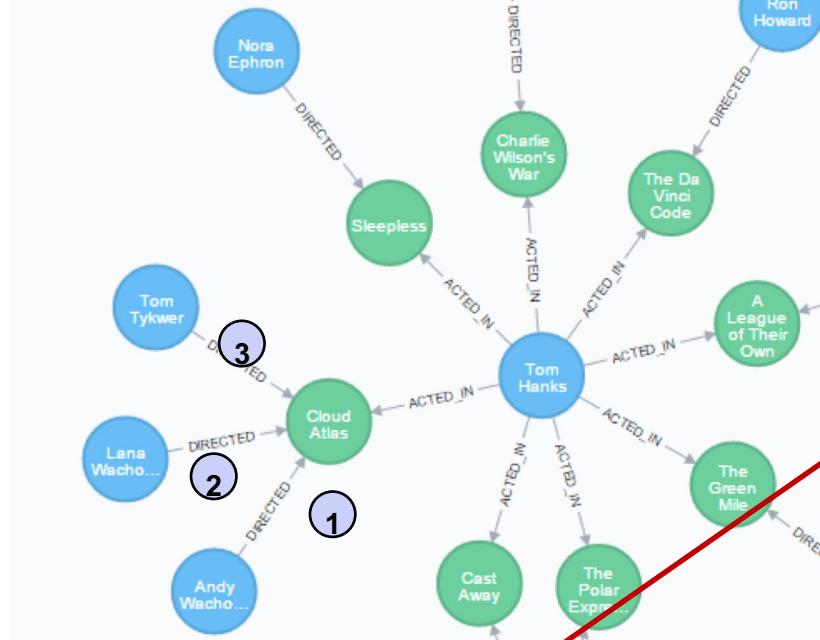


Neo4j Query Execution

- Each Neo4j Query is turned into an execution plan by an **execution planner**
- The **execution plan** is a tree-like structure consists of various **operators**, each implements a specific piece of work
- Query plan stages
 - ▶ Starting point (leaf node)
 - Obtaining data from storage engine
 - ▶ Expansion by matching given pattern in the query statement
 - ▶ Row filtering, skipping, sorting, projection, etc...
 - ▶ Combining operations
 - ▶ Updating
- Execution plans are evaluated based on statistics maintained by database
 - ▶ The number of nodes having a certain label.
 - ▶ The number of relationships by type.
 - ▶ Selectivity per index.
 - ▶ The number of relationships by type, ending with or starting from a node with a specific label.



Query Plan: an example



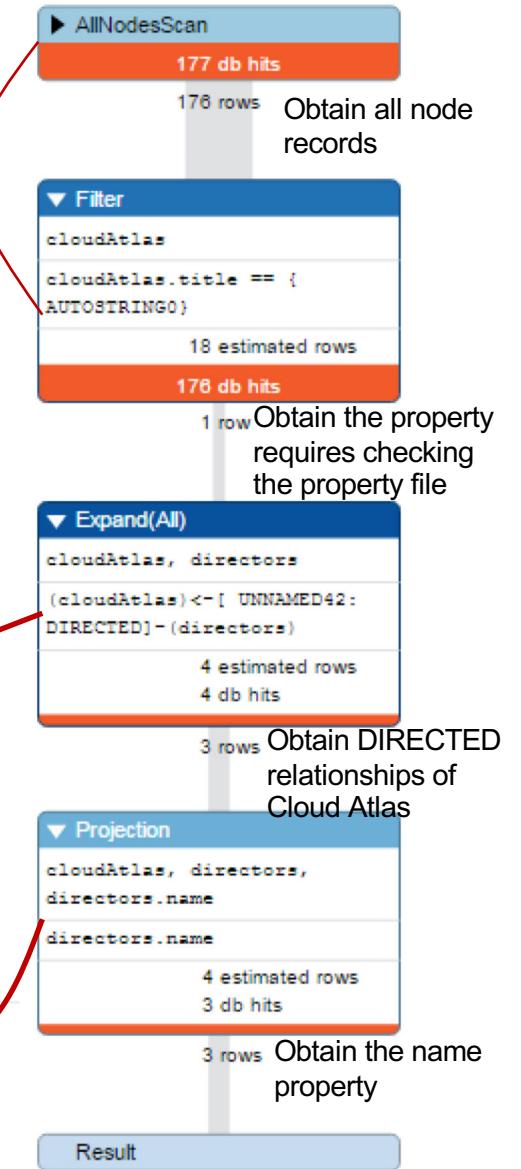
Query:

```
MATCH
  (cloudAtlas {title: "Cloud Atlas"})<-[DIRECTED]-(directors)
RETURN directors.name
```

Each box represents an operator

explain

This is the performance output from Neo4j 3.x



profile



Evaluation Statistics

- Each operator is annotated with some statistics
- **Rows:** The number of rows that the operator produced. This is only available if the query was *profiled*.
- **EstimatedRows:** This is the estimated number of rows that is expected to be produced by the operator.
- **DbHits:** Some operator needs to retrieve data from or update data in the storage. A *database hit* is an abstract unit of this storage engine work
 - ▶ **Creating** a node, a relationship, a label, a type
 - ▶ **Deleting** a node, a relationship,
 - ▶ **Getting** a node, a property of a node, the label,...
 - ▶ **Getting** a relationship, a property of a label, the type,
 - ▶ Updating...



Query Starting Points

- Most queries start with one or a set of **nodes** except if a relationship ID is specified
 - ▶ `MATCH (n1)-[r]->() WHERE id(r)= 0 RETURN r, n1`
 - ▶ This query will start from locating the first record in the relationship file
- Query may start by scanning all nodes
 - ▶ `MATCH(n) RETURN (n)`
 - ▶ `MATCH (cloudAtlas {title: "Cloud Atlas"})<-[DIRECTED]-(directors) RETURN directors.name`
- Query may start by scanning all nodes belonging to a given label
 - ▶ `MATCH (p:Person{name:"Tom Hanks"}) return p`
 - ▶ Labels are implicitly indexed
- Query may start by using index



Query starting from labelled node

```

MATCH (n:Person) -[r]- (something)
WITH n, count(something) as degree
ORDER BY degree DESC
LIMIT 1
RETURN n, degree
  
```



▼ NodeByLabelScan
n
:Person
133 estimated rows
134 db hits
133 rows

Obtain all 133 Person nodes records

▼ Expand(All)
n, r, something
(n)-[r]-(:something)
256 estimated rows
389 db hits
256 rows

Obtain 133 nodes + 256 relationships = 389 db hits

▼ EagerAggregation
degree, n
16 estimated rows
0 db hits
133 rows

Memory processing

► Top1
1 row
► ProduceResults
1 row
Result

The **ProduceResults** operator prepares the result so that it is consumable by the user, such as transforming internal values to user values. It is present in every single query that returns data to the user, and has little bearing on performance optimisation.

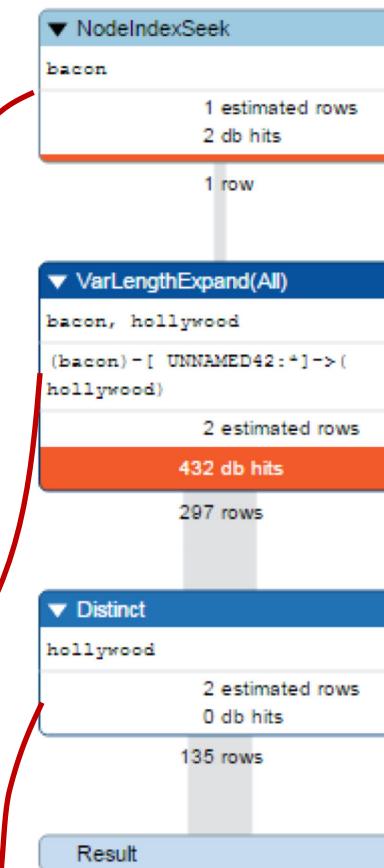


Query Plan With Index

- Neo4j supports index on properties of labelled node
- Index has similar behaviour as those in relational systems
- It can be built on single or composite properties
- Create Index
 - ▶ **CREATE INDEX ON :Person(name)**
- Drop Index
 - ▶ **DROP INDEX ON :Person(name)**

Query:

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)  
RETURN DISTINCT hollywood
```



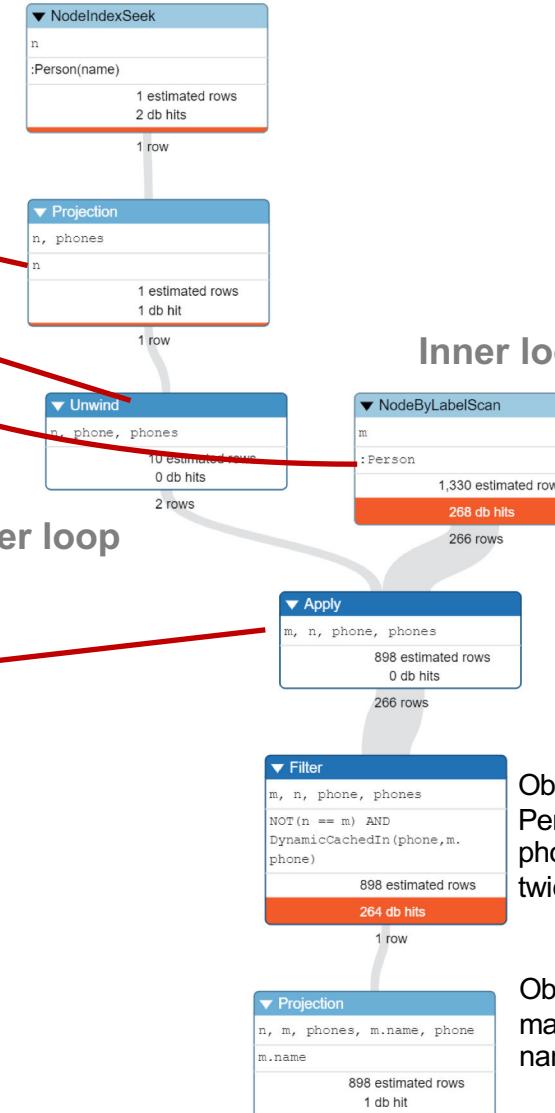
Each row represents a path that may include more than one relationship, so the db hits number is larger than row number



A relatively complex query and plan

```

MATCH (n:Person{name: "Tom Hanks"}) ←
WITH n.phone as phones, n
UNWIND phones as phone
MATCH (m:Person) ←
WHERE phone in m.phone and n>>m
RETURN m.name
    
```



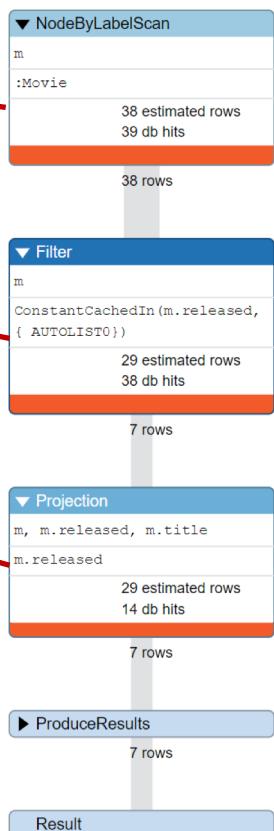
Apply works by performing a nested loop. Every row being produced on the left-hand side of the **Apply** operator will be fed to the leaf operator on the right-hand side, and then **Apply** will yield the combined results



Comparing Execution Plans

```

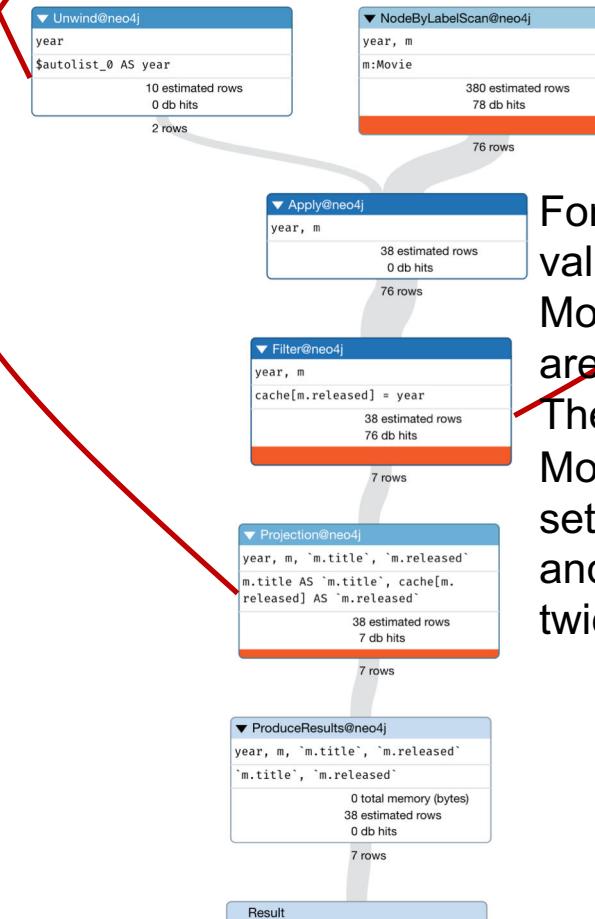
MATCH (m: Movie)
WHERE m.released IN [1999,2003]
RETURN m.title, m.released
  
```



UNWIND [1999,2003] as year

```

MATCH (m: Movie)
WHERE m.released = year
RETURN m.title, m.released
  
```



For each year value, all Movie nodes are examined. The entire Movie nodes set is retrieved and examined twice

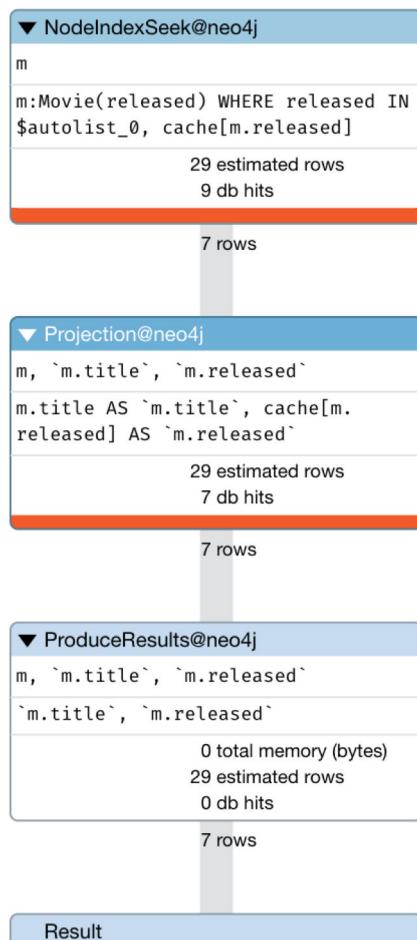


Comparing Execution Plans (with Index)

MATCH (m: Movie)

WHERE m.released IN [1999,2003]

RETURN m.title, m.released



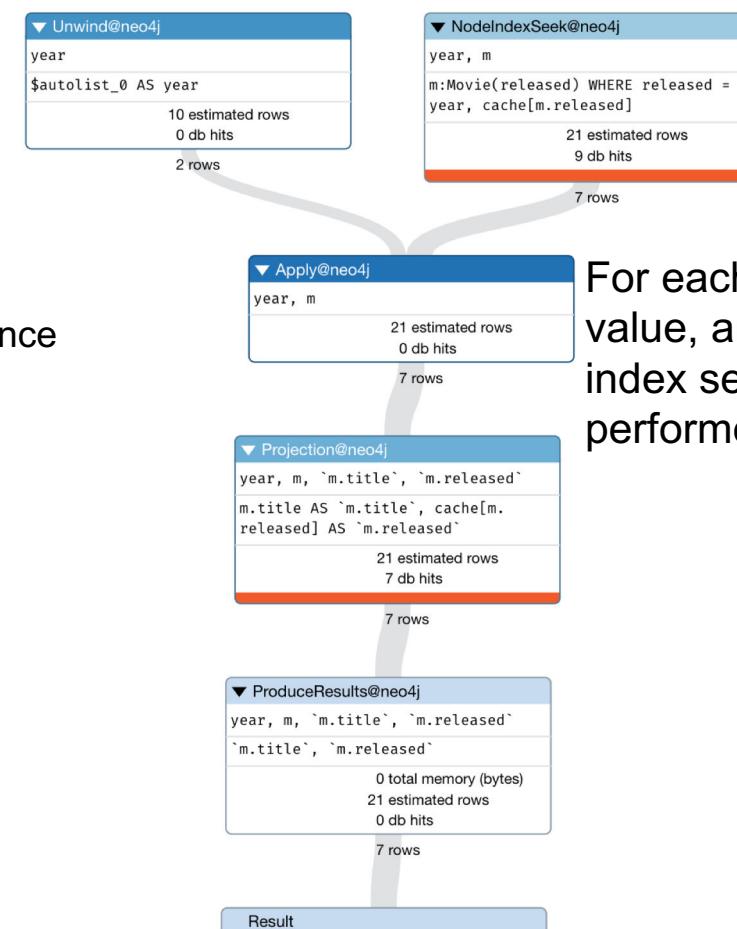
CREATE INDEX ON :Movie(released)

UNWIND [1999,2003] as year

MATCH (m: Movie)

WHERE m.released = year

RETURN m.title, m.released



Similar performance

For each year value, an index search is performed



Another example of comparison

- Question: Find out a list of person who has acted in at least three movies and also directed at least one movie
- Cypher is powerful and flexible
 - ▶ It is possible to write very different queries that produce the same results
 - ▶ The performance could have big difference
 - ▶ The DB engine does not have much knowledge to rewrite the queries as those in SQL
 - Not based on relational algebra



Option 1

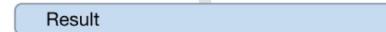
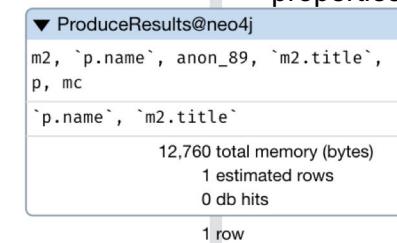
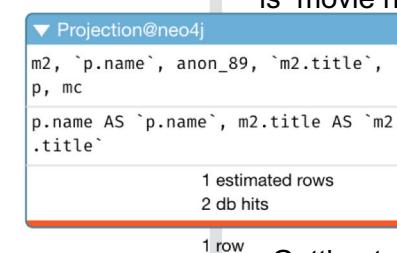
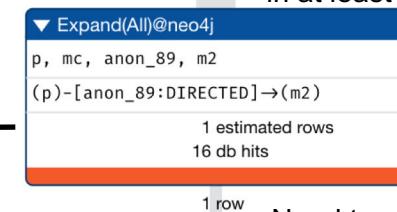
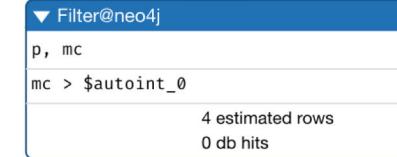
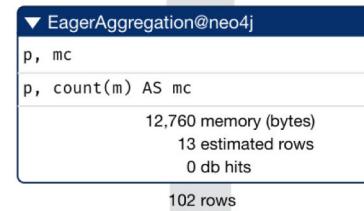
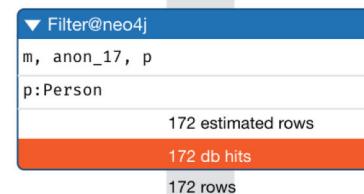
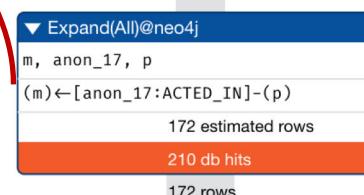
```

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WITH p, count(m) AS mc
WHERE mc > 2
MATCH (p)-[:DIRECTED]->(m2:Movie)
RETURN p.name, m2.title
  
```

Check the other nodes are of **Person** type

Aggregate by p

Start with **Movie** label because there are less **Movie** nodes than **Person** nodes



Option 2

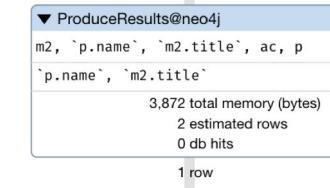
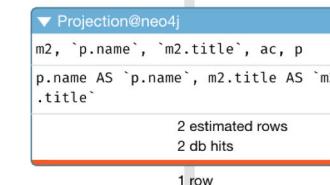
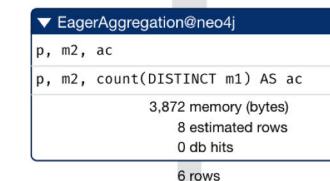
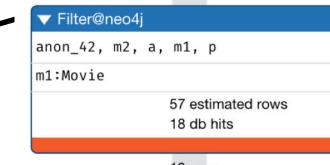
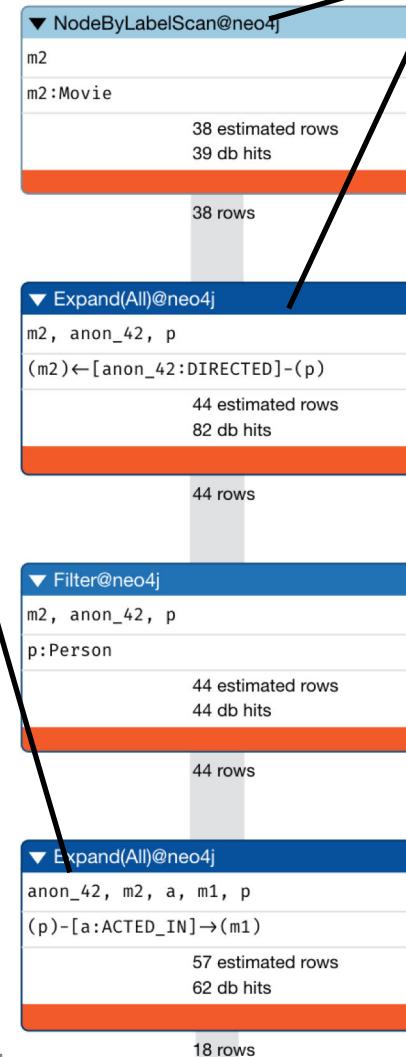
```

MATCH (m1:Movie)<-[a:ACTED_IN]-(p:Person)-[:DIRECTED]->(m2:Movie)
WITH p, count(distinct m1) as ac, m2
WHERE ac > 2
RETURN p.name, m2.title
    
```

38 Movie nodes + 44 relationships

44 Person nodes

Among 62 relationships
only 18 are of ACTED type



Obtain the 18 Movie nodes from the 18 relationships as m1

Aggregation is processed in memory.

Filtering



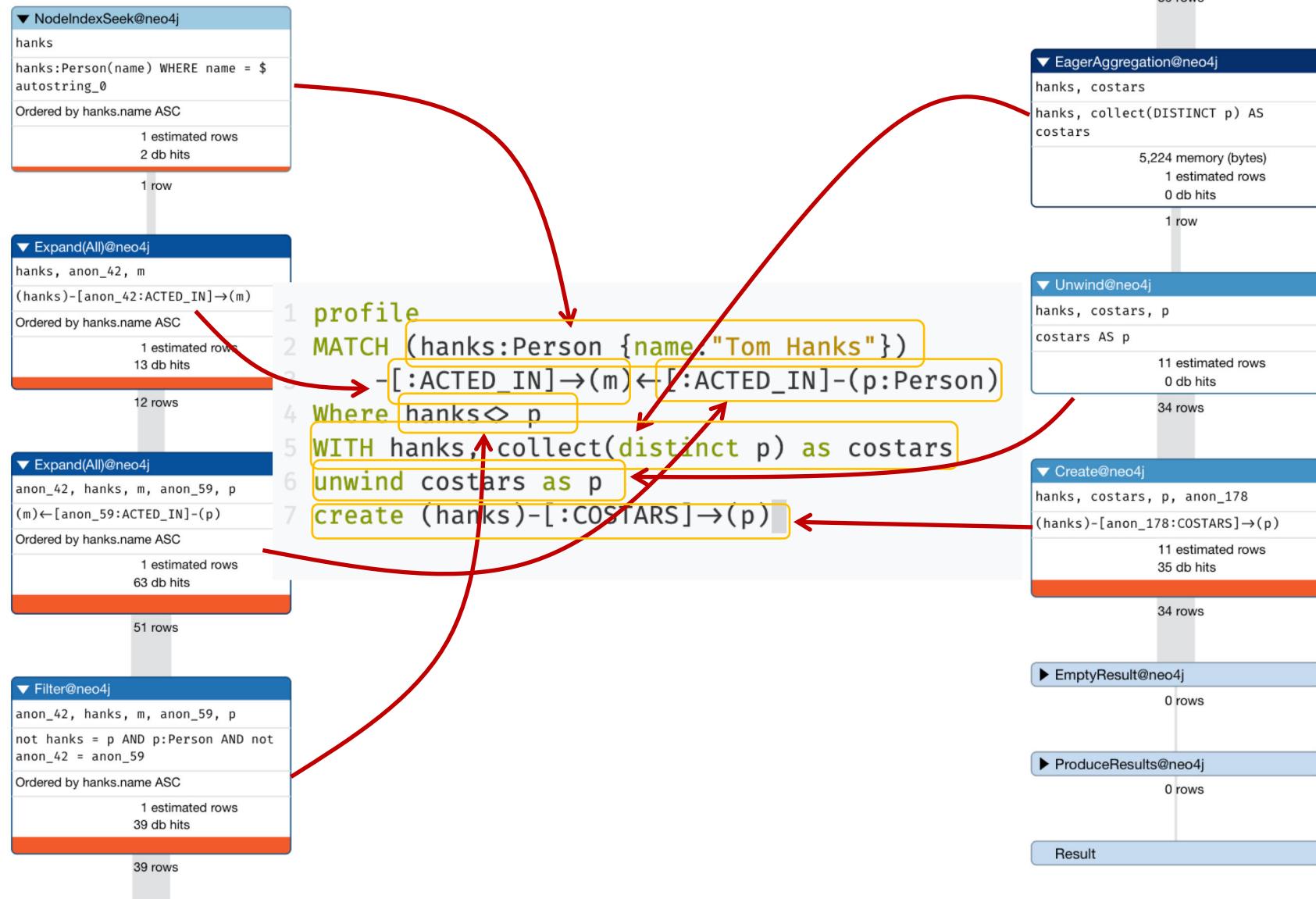
Performance of Creation Operations

- Example: add COSTARTS relationships for Tom Hanks
- Option 1: using CREATE

```
1 profile
2 MATCH (hanks:Person {name:"Tom Hanks"})
3   -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
4 WHERE hanks◊ p
5 WITH hanks, collect(distinct p) AS costars
6 UNWIND costars AS p
7 CREATE (hanks)-[:COSTARS]→(p)
```



Profile Result



Cypher version: CYPHER 4.1, planner: COST, runtime: INTERPRETED. 152 total db hits in 272 ms.



Option 2: Using MERGE

```
1 profile
2 MATCH (hanks:Person {name:"Tom Hanks"})
3   -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
4 WHERE hanks◊ p
5 WITH hanks, p
6 merge (hanks)-[:COSTARS]→(p)
```

Cypher version: CYPHER 4.1, planner: COST, runtime: INTERPRETED. 2482 total db hits in 1498 ms.



Profile Result

For each p , check if there is a COSTARS relationship between node hanks and node p

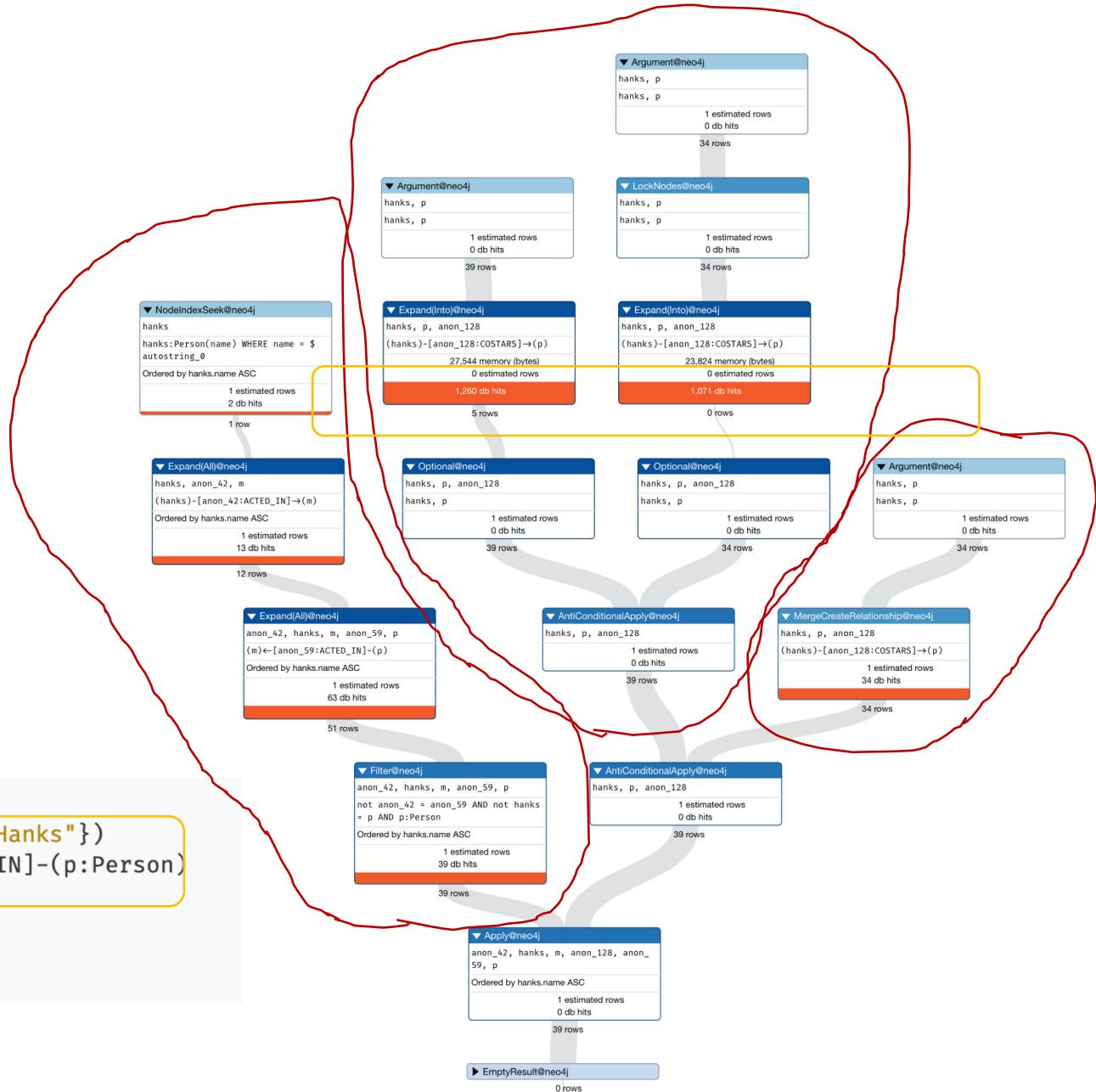
Check relationship existence needs to traverse the entire relationship link.

Create the relationship when it does not exists.
There are 34 create relationship actions.

```

1 profile
2 MATCH (hanks:Person {name:"Tom Hanks"})
3   -[:ACTED_IN]→(m)←[:ACTED_IN]-(p:Person)
4 Where hanks↔ p
5 WITH hanks, p
6 merge (hanks)-[:COSTARS]→(p)

```



Transactions

- Neo4j supports full ACID transactions
 - ▶ Similar to those in RDBMS
- Uses locking to ensure consistency
 - ▶ Lock Manager manages locks held by a transaction
- Logging
 - ▶ Write Ahead Logging (WAL)
- Transaction Commit Protocol
 - ▶ Acquire locks (Atomicity, Consistency, Isolation)
 - ▶ Write Undo and Redo records to the WAL
 - for each node, relationship, property changed is written to the log
 - ▶ Write commit record to the log and flush to disk (Durability)
 - ▶ Release locks
- Recovery – if the database server/machine crashes
 - ▶ Apply log records to replay changes made by the transactions



Outline

- Neo4j Storage
- Neo4j Query Plan and Indexing
- **Neo4j – Data Modeling**



Graph Data Modelling

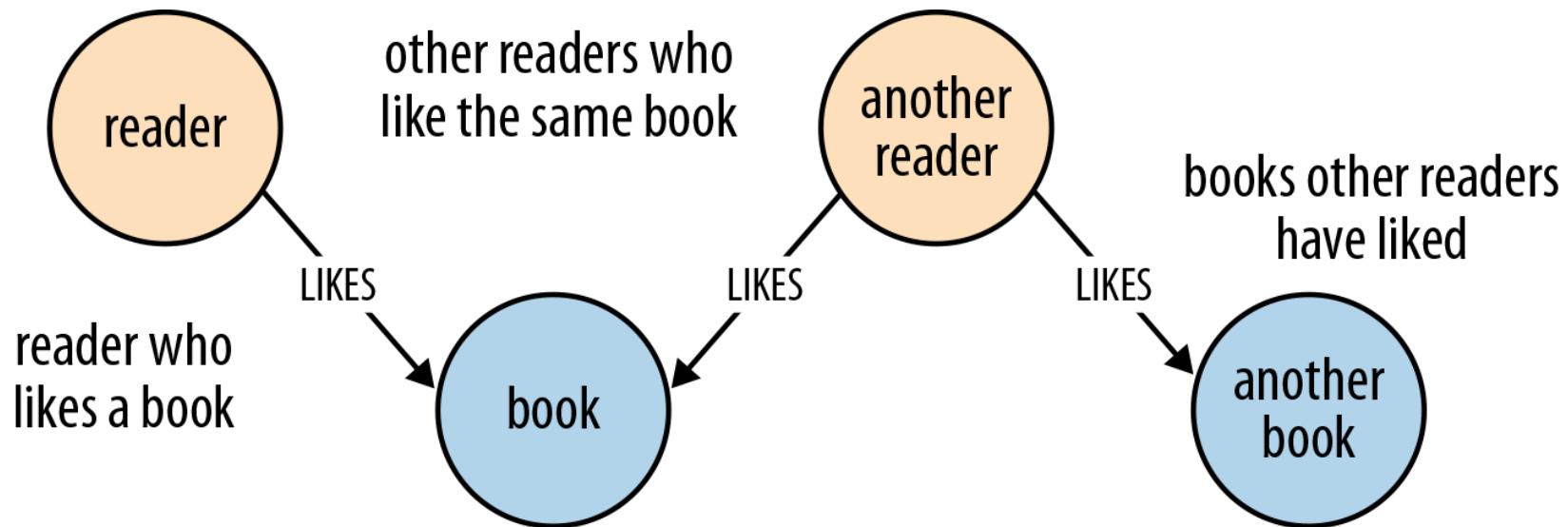
- Graph data modelling is very closely related with domain modelling
- You need to decide
 - ▶ Node or Relationship
 - ▶ Node or Property
 - ▶ Label/Type or Property
- Decisions are based on
 - ▶ Features of entities in application domain
 - ▶ Your typical queries
 - ▶ Features and constraints of the underlying storage system

Slides 35-39 are based on Chapter 4 of Graph Databases book



Node vs. Relationship

- Nodes for Things, Relationship for Structures
 - ▶ AS A reader who likes a book, I **WANT** to know which books other readers who like the same book have liked, **SO THAT** I can find other books to read.

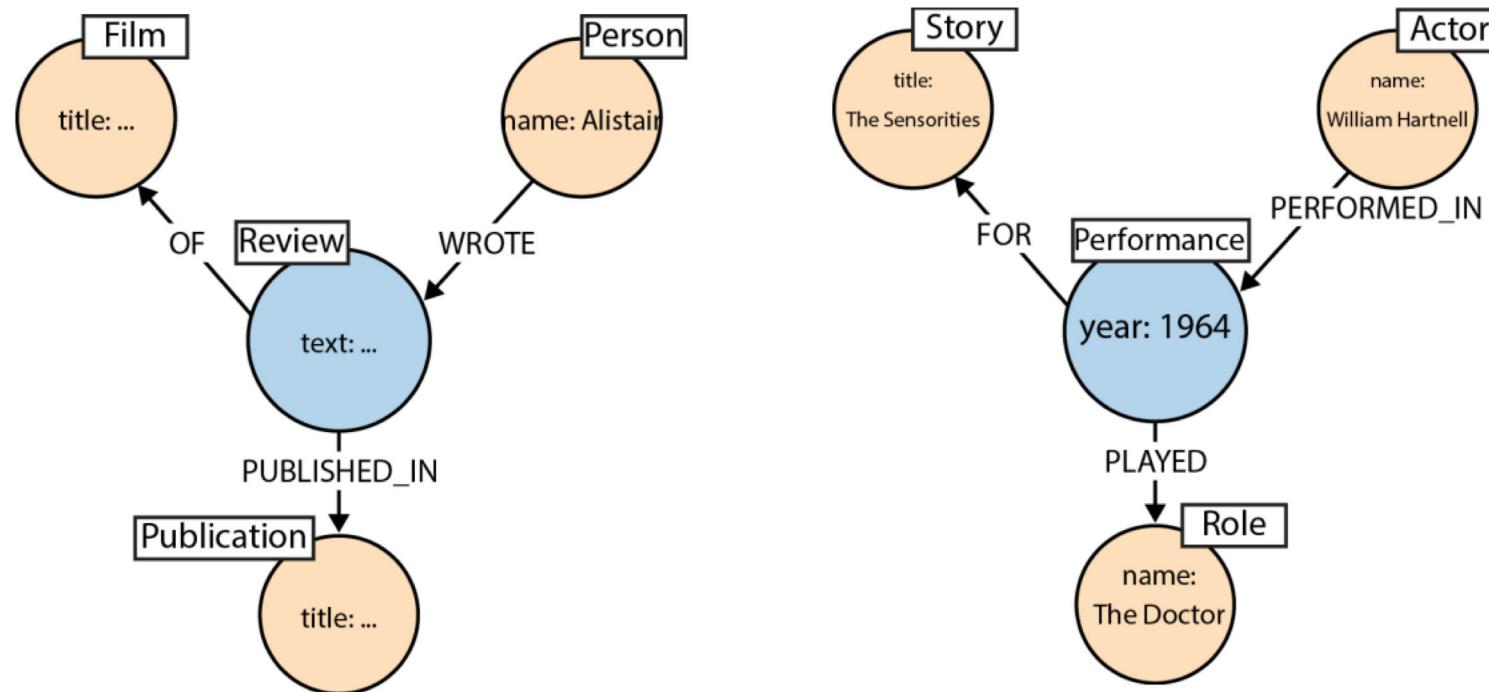


```
MATCH (:Reader {name:'Alice'})-[:LIKES]->(:Book {title:'Dune'})  
<-[:LIKES]-(:Reader)-[:LIKES]->(books:Book)  
RETURN books.title
```



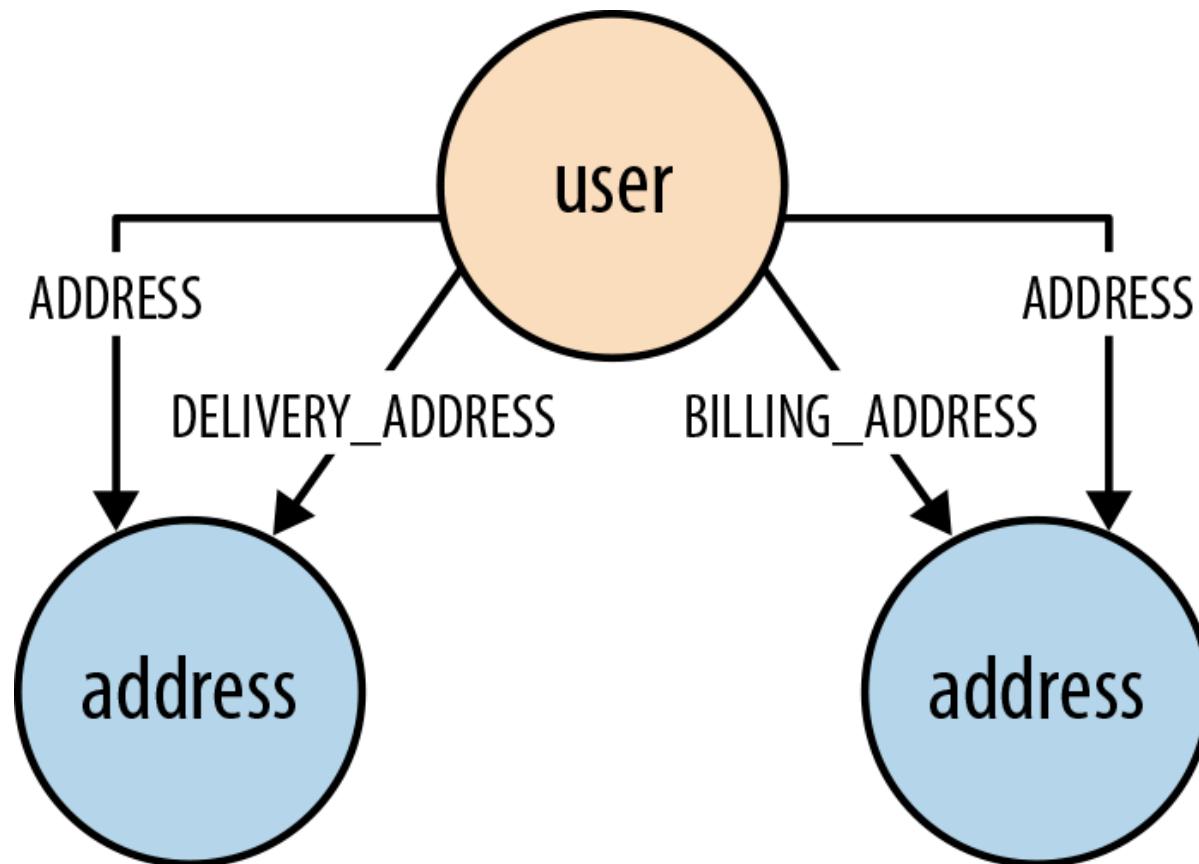
Node vs. Relationship

■ Model Facts as Nodes



Node or Property

- Represent Complex Value Types as Nodes



Relationship Property or Relationship Type

- E.g. The relationship between user node and address node can be:
 - typed as **HOME_ADDRESS**, **BILLING_ADDRESS** or
 - typed as generic **ADDRESS** and differentiated using a type property {type:'home'}, {type:'billing'}
- We use fine-grained relationships whenever we have a closed set of relationship types.
 - ▶ Eg. there are only a finite set of address types
 - ▶ If traversal would like to follow generic type **ADDRESS**, we may have to use redundant relationships
 - MATCH (user)-[:**HOME_ADDRESS**|**WORK_ADDRESS**|**DELIVERY_ADDRESS**]->(address)
 - MATCH (user)-[:**ADDRESS**]->(address)
 - MATCH (user:User)-[r]->(address:Address)



References

- Ian Robinson, Jim Webber and Emil Eifrem, *Graph Databases*, Second Edition, O'Reilly Media Inc.,
 - ▶ You can download this book from the Neo4j site,
<https://neo4j.com/graph-databases-book/?ref=home>
 - Chapter 4, Chapter 6
- Neo4j – Reference Manual
 - ▶ <https://neo4j.com/docs/developer-manual/current/>
- Neo4j reference manual: Execution plan operators in detail
 - ▶ <https://neo4j.com/docs/cypher-manual/current/execution-plans/operators/>



COMP5338 – Advanced Data Models

Week 8: Key Value Storage Systems

Dr. Ying Zhou

School of Computer Sciences



THE UNIVERSITY OF
SYDNEY

Outline

■ Overview

- ▶ K-V store
- ▶ Memcached brief intro

■ Dynamo

- ▶ Overview
- ▶ Partitioning Algorithm
- ▶ Replication and Consistency

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



Key-Value Data Model

■ Simplest form of data model

- ▶ Each data “record” contains a key and a value
 - All queries are key based
- ▶ Similar concept in programming language/data structure
 - Associative array, hash map, hashtable, dictionary
- ▶ Basic API similar to those in hashtable
 - **put** key value
 - value = **get** key

■ There are many such systems

- ▶ Some just provides pure K-V form
 - The system treats **value** as uninterpretable string/byte array
- ▶ Others may add further dimensions in the value part
 - The value can be a json document, e.g HyperDex
 - The value can contain columns and corresponding values, e.g. Bigtable/HBase



From Keys to Hashes

- In a key-value store, or key-value data structure, the key can be of any type
 - ▶ String, Number, Date and Time, Complex object,
- They are usually hashed to get a value of fixed size
 - ▶ Hash function is any function that can be used to map data of arbitrary size to data of a fixed size
 - E.g. any Wikipedia page's content can be hashed by SHA-1 algorithm to produce a message-digest of 160-bit value
 - ▶ The result is called a hash value, hash code, hash sum or hash
- Hash allows for efficient storage and lookup

key	Hash
Alice	1
Tom	3
Bob	4

Hash Entry	Data
1	(Alice, 90)
3	(Tom, 85)
4	(Bob, 87)



Memcached: motivation

- Memcached is a very early in-memory key-value store
- The main use case is to provide caching for web application, in particular, caching between the database and application layer
- Motivations:
 - ▶ Database queries are expensive, cache is necessary
 - ▶ Cache on DB nodes would make it easy to share query results among various requests
 - But the raw memory size is limited
 - ▶ Caching more on Web nodes would be a natural extension but requests by default have their own memory spaces and do not share with each other
- Simple solution
 - ▶ “Have a global hash table that all Web processes on all machines could access simultaneously, instantly seeing one another’s changes”

<https://www.linuxjournal.com/article/7451>



Memcached: features

■ As a cache system for query results

- ▶ Durability is not part of the consideration
 - Data is persisted in the database
 - No replication is implemented
- ▶ Mechanisms for freshness guarantee should be included
 - Expiration mechanism
- ▶ Cache miss is a norm
 - But want to minimize that

■ Distributed cache store

- ▶ Utilizing free memories on all machines
- ▶ Each machine may run one or many Memcached server instances
- ▶ It is beneficial to run multiple Memcached instances on single machine if the physical memory is larger than the address space supported by the OS



Memcached: the distributed caching

- The keys of the global hash table are distributed among a number of **Memcached** server instances
 - ▶ How do we know the location of a particular key
- Simple solution
 - ▶ Each `memcached` instance is totally independent
 - ▶ A given key is kept in the same server
 - ▶ Client library knows the list of servers and can use a predetermined partition algorithm to work out the designated server of a key
- Client library runs on web node
- There are many implementations
- May have different ways to partition keys
 - ▶ Simple hash partition or consistent hashing

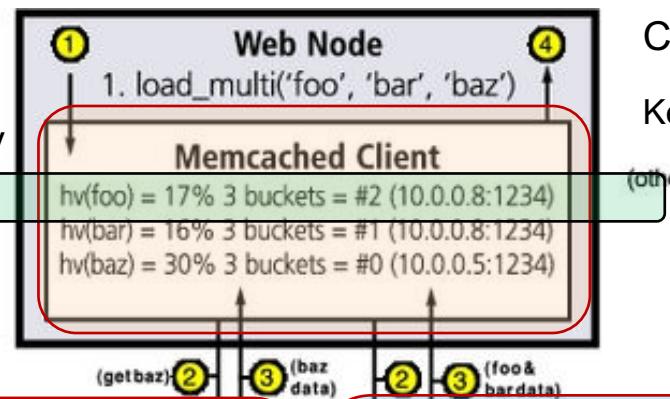


Memcached: basic hash partition

- Treat Memcached instance as buckets
 - ▶ Instance with larger memory may represent more buckets
- Use modulo function to determine the location of each key
 - ▶ Hash(key) mod #buckets

Client library knows there are 3 buckets, so would determine the location of a key by mod 3 of any key's hash value

(other Web nodes)



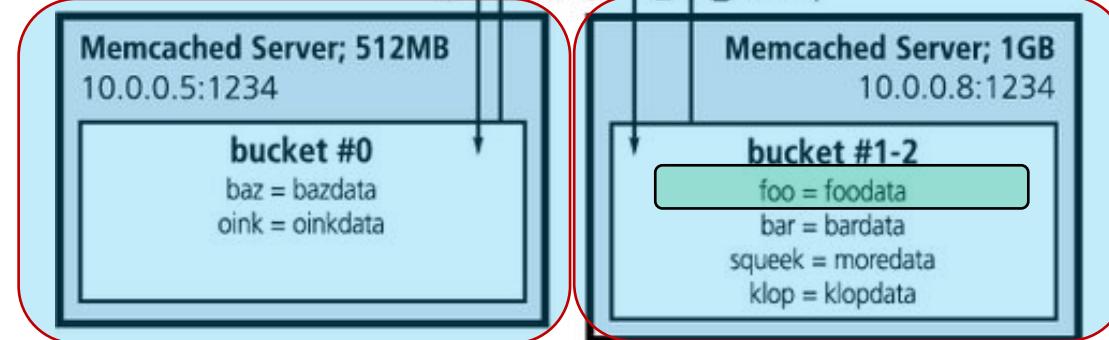
Client library runs on web node

Key 'foo' has a hash value of 17, mod 3 is 2

(other Web nodes)

Two memcached servers

One server has 512M memory for Memcached, and is considered as one bucket with id 0



Another server has 1G memory for Memcached, and is considered as two bucket with id 1 and 2

This key should be stored on the server managed bucket id 2



Outline

■ Overview

- ▶ K-V store
- ▶ Memcached brief intro

■ Dynamo

- ▶ Overview
- ▶ Partitioning Algorithm
- ▶ Replication and Consistency



Dynamo

■ Motivation

- ▶ Many services in Amazon only need primary key access to data store
 - E.g. shopping cart
- ▶ Both scalability and availability are essential terms in the service level agreement
 - Always writable (write never fails)
 - Guaranteed latency
 - Highly available

■ Design consideration

- ▶ Simple key value model
- ▶ Sacrifice strong consistency for availability
- ▶ Conflict resolution is executed during **read** instead of write
- ▶ Incremental scalability



Dynamo Techniques Summary

■ Dynamo is a decentralized peer-to-peer system

- ▶ All nodes taking the same role
- ▶ There is no master node

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.



Outline

■ Overview

- ▶ K-V store
- ▶ Memcached brief intro

■ Dynamo

- ▶ Overview
- ▶ Partitioning Algorithm
- ▶ Replication and Consistency



Partitioning Algorithm

■ Partition or shard a data set

- ▶ There is a partition or shard key
- ▶ There is an algorithm to decide which data goes to which partition
 - Range partition vs. Random(Hash) partition

■ Range partition requires keys to be of same type

- ▶ So we can define a range of keys for each partition

■ Hash partition is more flexible

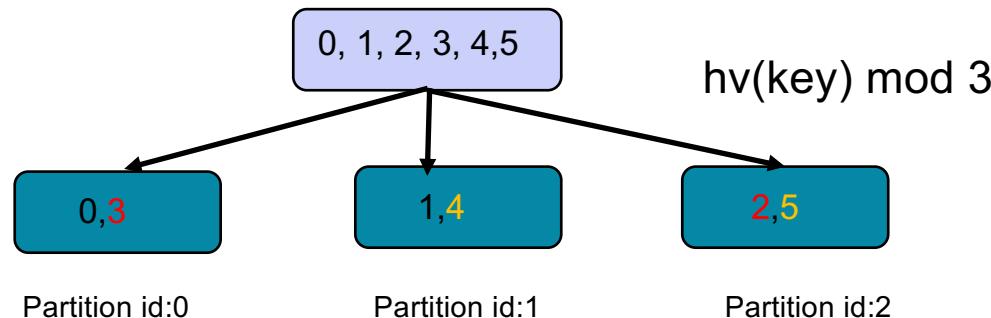
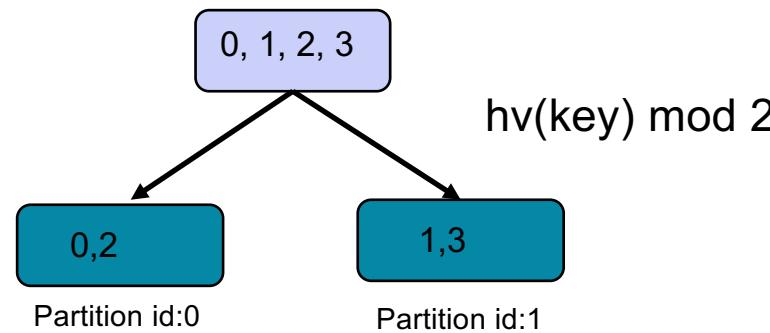
- ▶ Key distribution is based on its hash value instead of raw value
- ▶ There are many options to determine the key placement
 - Modulo function as in previous section
 - Predefined ranges on hash value as in MongoDB hashed sharding
 - Consistent hashing in Dynamo



Modulo Based Hash Partition- Repartition

■ Repartition may involve a lot of data movement

- ▶ The modulo function of key's hash value will redistribute large number of keys to different partitions



Consistent Hashing

■ Consistent hashing

- ▶ “a special kind of hashing such that when a hash table is resized, only K/n keys need to be remapped on average, where K is the number of keys, and n is the number of slots.”

[Wikipedia: Consistent Hashing]

- ▶ It does not identify each partition as a number in $[0, n-1]$
- ▶ The output range of a hash function is treated as a fixed circular space or “ring” (i.e. the largest hash value wraps around to the smallest hash value).
- ▶ Each partition represents a range in the ring space, identified by its position value (token)
- ▶ The hash of a data record’s key will uniquely locate in a range
- ▶ In a distributed system, each node represents one partition or a number of partitions if “virtual node” is used.

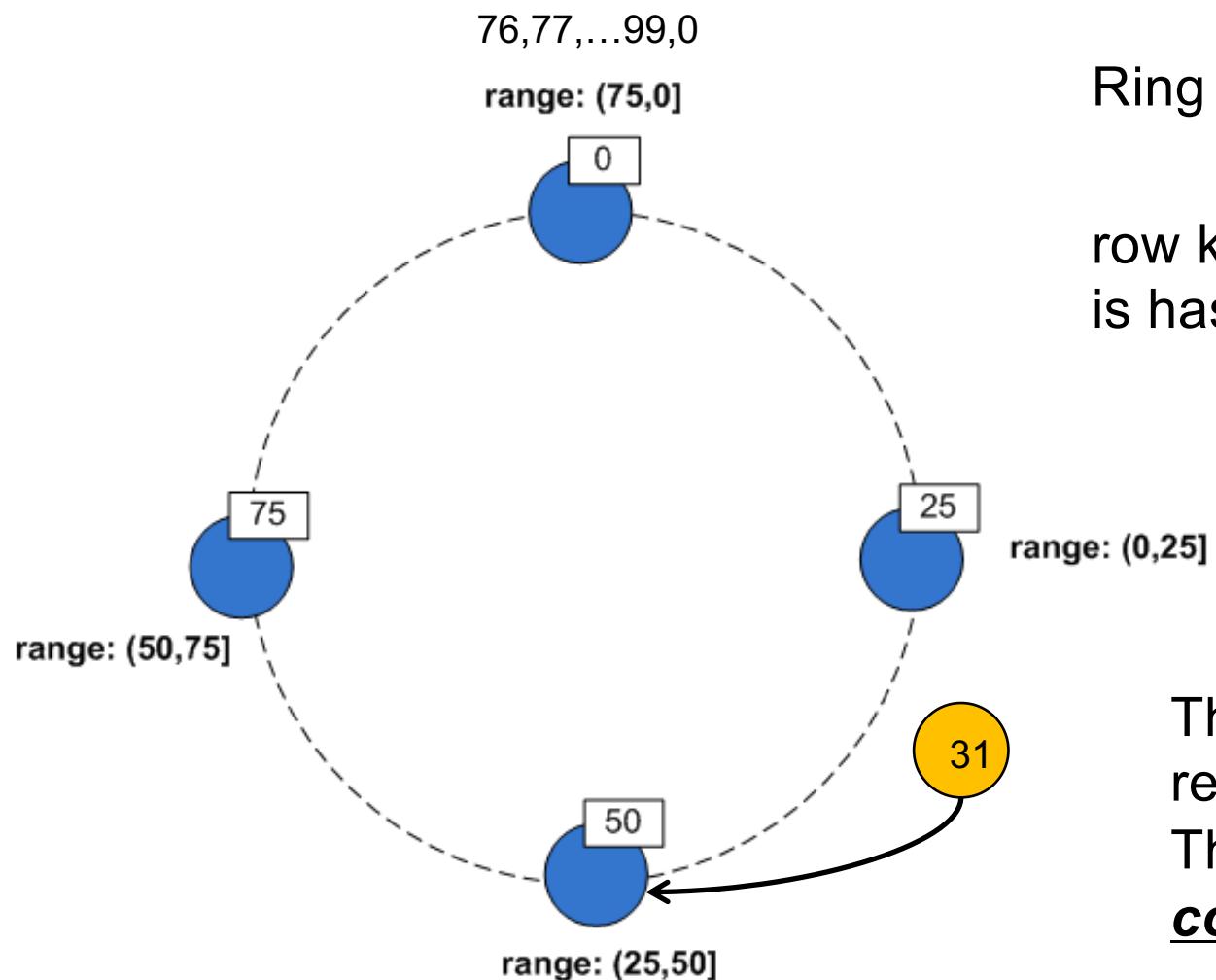


Consistent Hashing in Dynamo

- Each node in the Dynamo cluster is assigned a “token” representing its position in the “ring”
- Each node is responsible for the region in the ring between it and its predecessor node
- The ring space is the MD5 Hash value space (128 bit)
 - ▶ 0 to $2^{127} - 1$
- The MD5 Hash of the key of any data record is used to determine which node is the coordinator of this key. The coordinator is responsible for
 - ▶ Storing the row data locally
 - ▶ Replicating the row data in $N-1$ other nodes, where N is the replication factor



Consistent hashing example



Ring space: 0~99

row key: “bsanderson”
is hashed to a number **31**

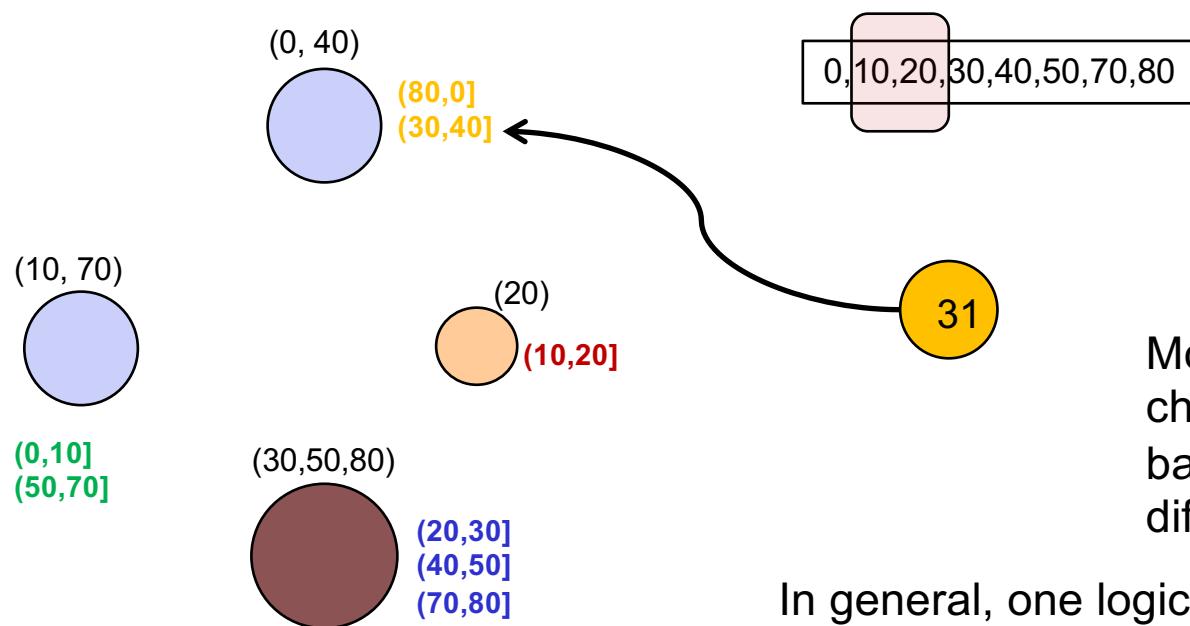
The node with token 50 is
responsible for this key
This is called the
coordinator of this key

Virtual Nodes

■ Possible issue of the basic consistent hashing algorithm

- ▶ Position is randomly assigned, cannot guarantee balanced distribution of data on node
- ▶ Assume nodes have similar capacity, each is handling one partition

■ Virtual nodes and multiple partitions per node

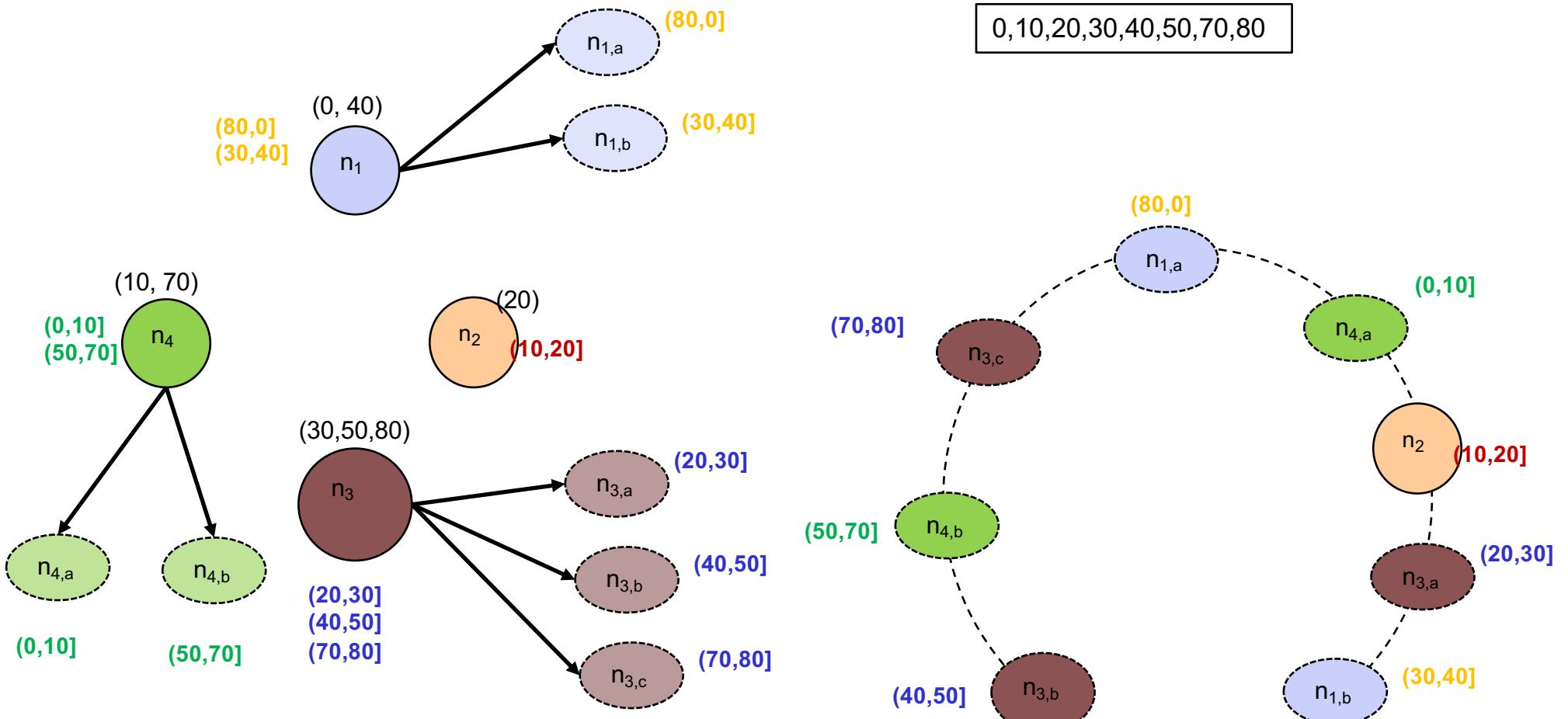


Memcached has similar design consideration: an instance with a lot of available memories can be allocated more than one buckets

MongoDB uses smaller sized chunk to achieve shard load balance. Shards may contain different number of chunks

In general, one logical partition of the data does not map to one physical machine

Virtual Nodes Ring



Outline

■ Overview

- ▶ K-V store
- ▶ Memcached brief intro

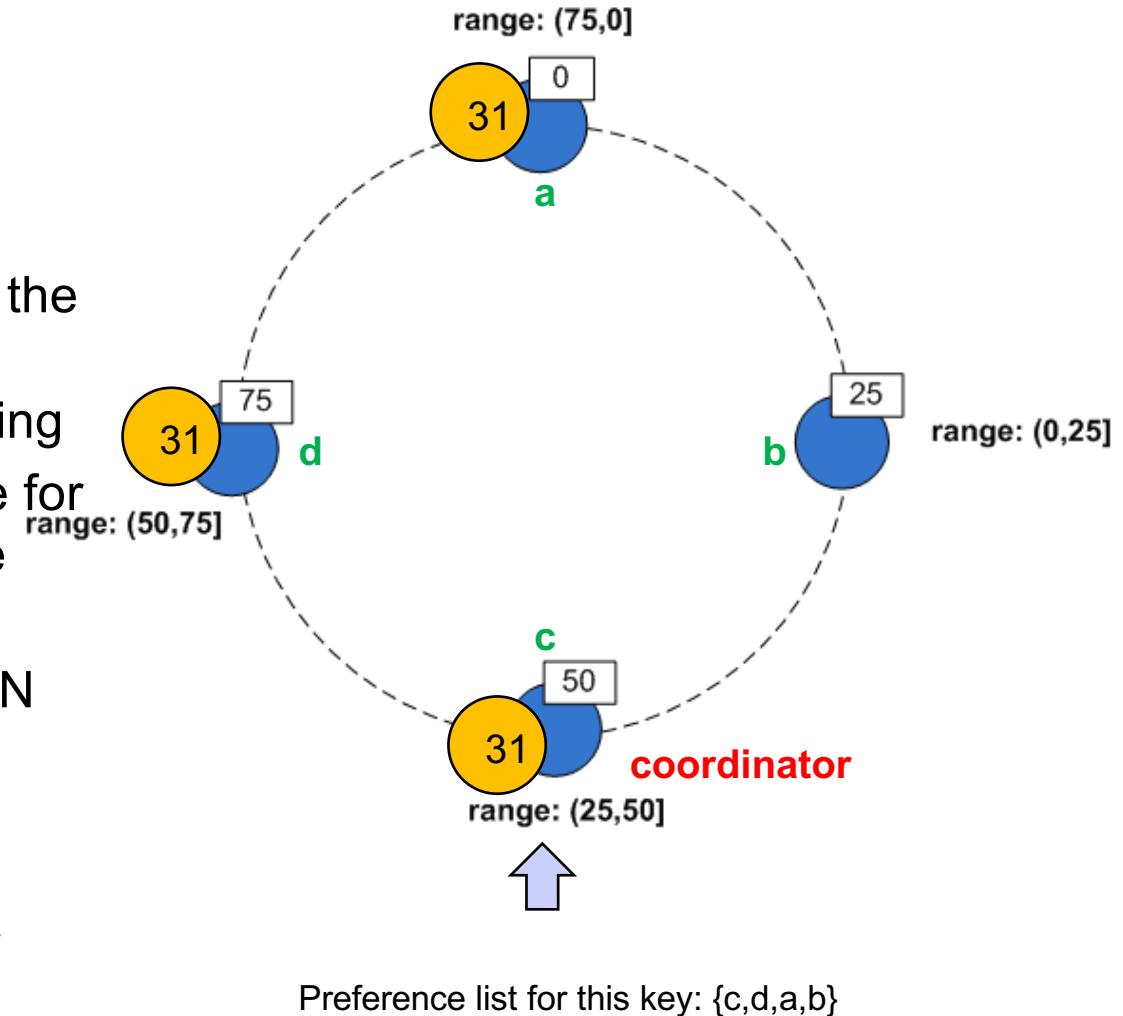
■ Dynamo

- ▶ Overview
- ▶ Partitioning Algorithm
- ▶ Replication and Consistency



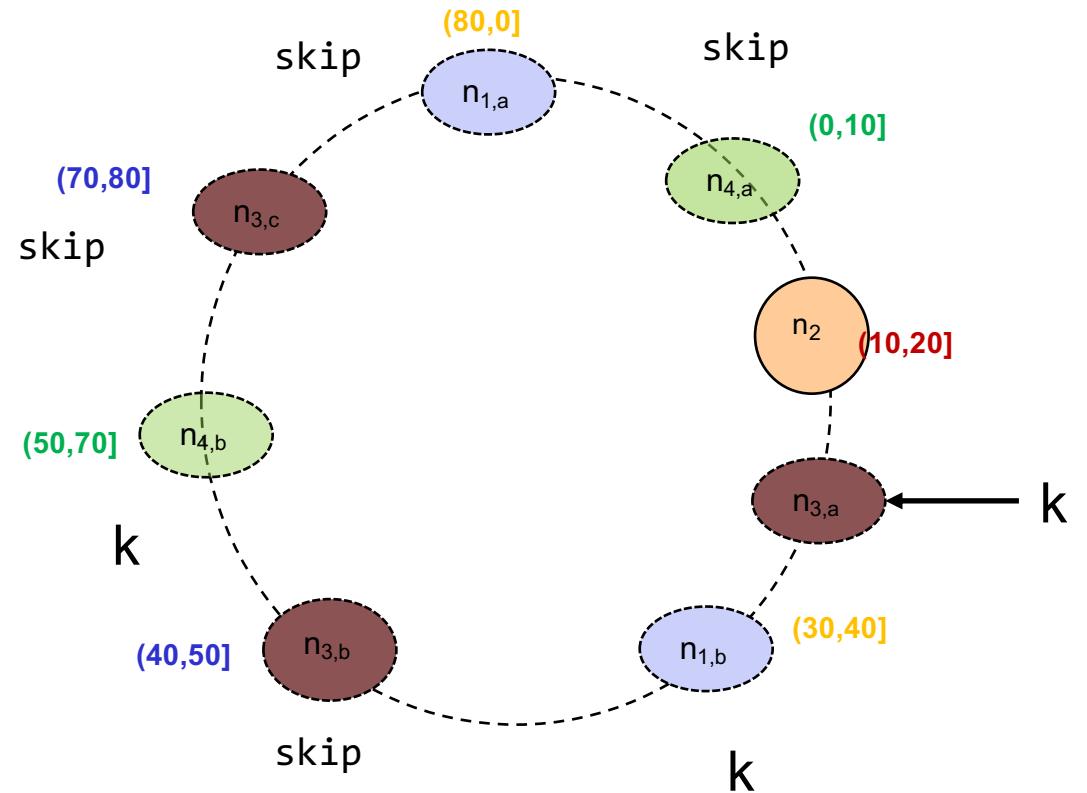
Replication

- Replication is essential for high availability and durability
 - ▶ Replication factor (N)
 - ▶ Coordinator
 - ▶ Preference list
- Each key (and its data) is stored in the coordinator node as well as N-1 clockwise successor nodes in the ring
- The list of nodes that is responsible for storing a particular key is called the *preference list*
- Preference list contains more than N nodes to allow for node failures
 - ▶ Some node are used as temporary storage.
 - ▶ Can be computed on the fly by any node in the system



Replication with Virtual Nodes

- “Note that with the use of virtual nodes, it is possible that the first N successor positions for a particular key may be owned by less than N distinct physical nodes (i.e. a node may hold more than one of the first N positions). To address this, the preference list for a key is constructed by skipping positions in the ring to ensure that the list contains only distinct physical nodes.”



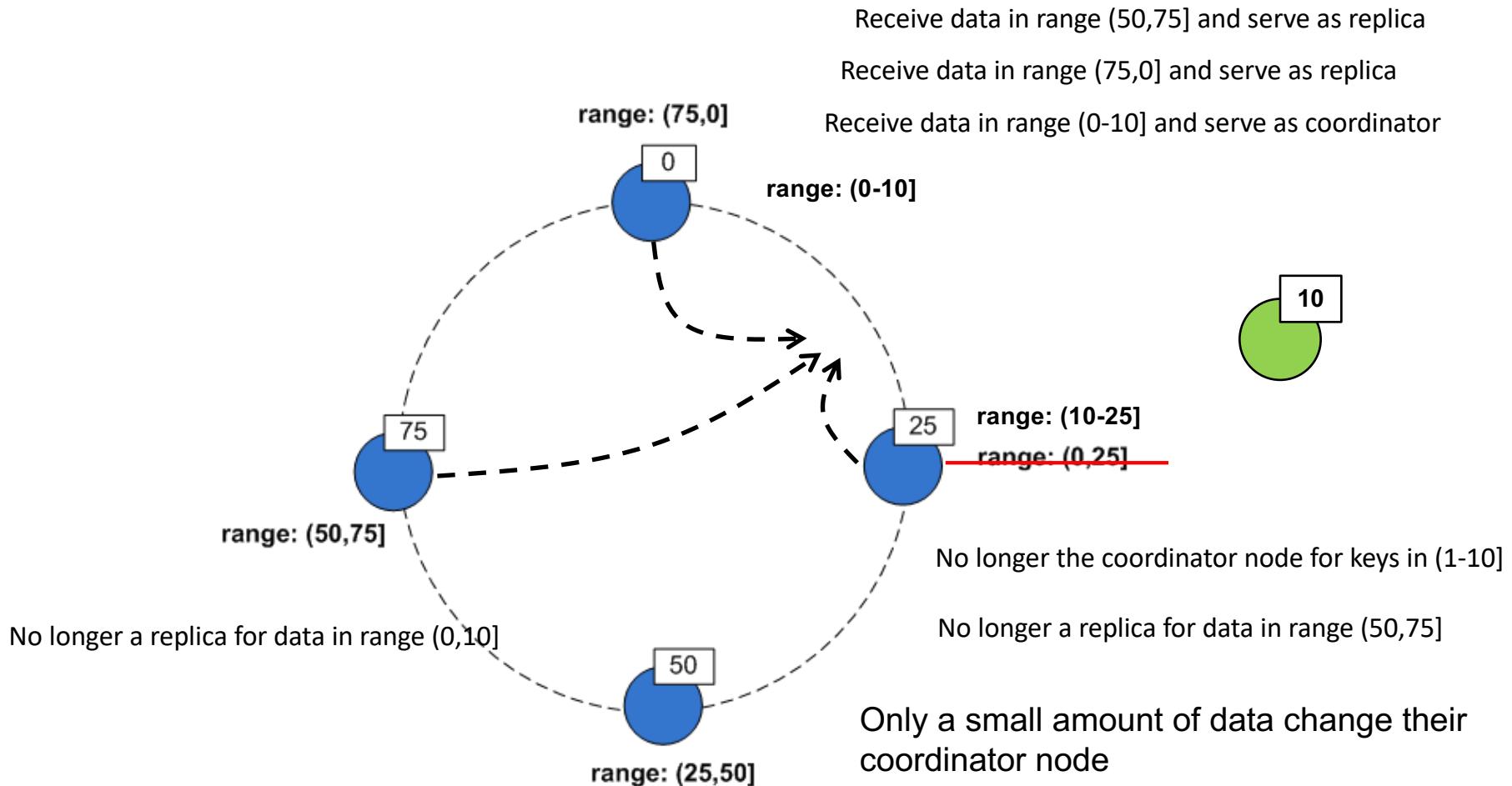
Preference list of K is: {n3, n1,n4, n2}

Membership and Failure Detection

- Each node in the cluster is aware of the token range handled by its peers
 - ▶ This is done using a gossip protocol
 - ▶ New node joining the cluster will randomly pick a token
 - ▶ The information is gossiped around the cluster
- Failure detection is also achieved through gossip protocol
 - ▶ Local knowledge
 - ▶ Nodes do not have to agree on whether or not a node is “really dead”.
 - ▶ Used to handle temporary node failure to avoid communication cost during read/write
 - ▶ Permanent node departure is handled externally



Adding Storage Node



<http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2>



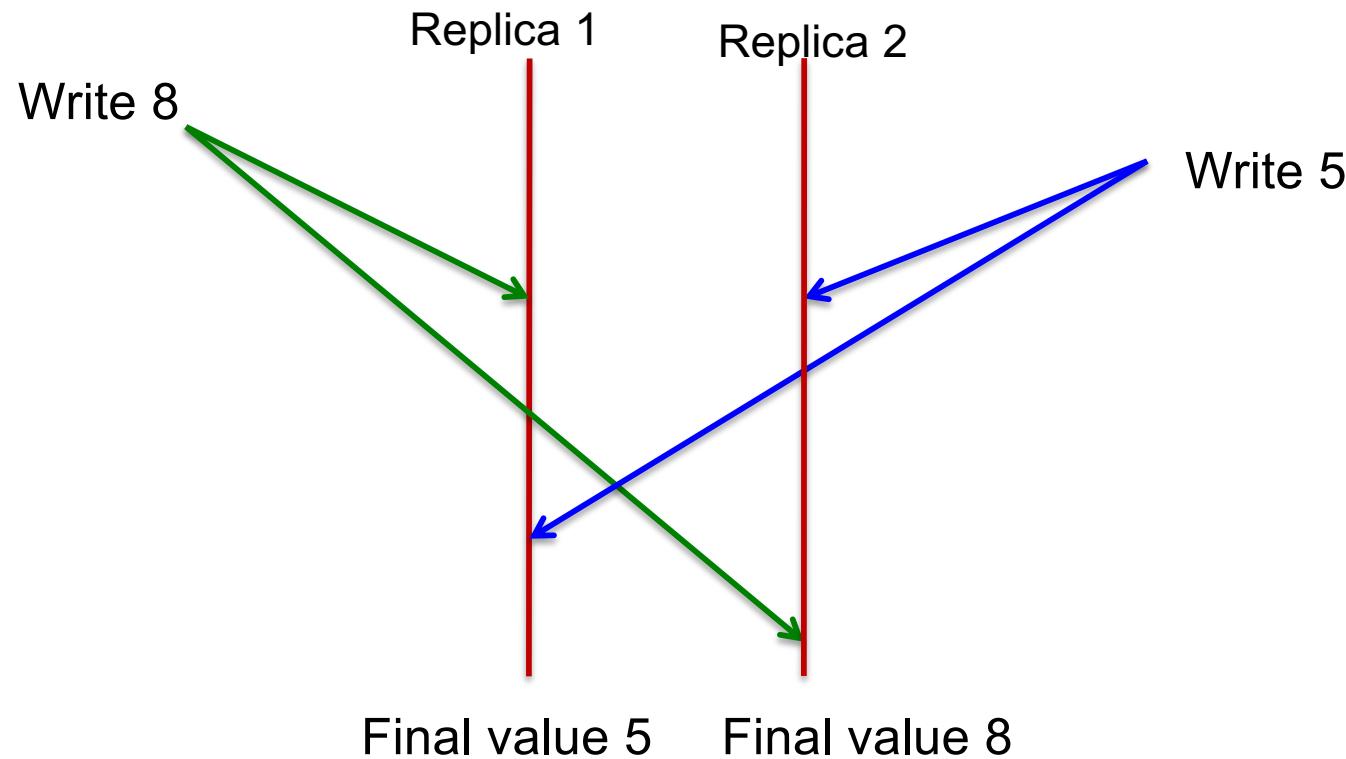
Read and Write with Replication

- When there are replicas, there are many options for read/write
- In a typical Master/Slave replication environment
 - ▶ Write happens on the master and may propagate to the replica immediately and wait for all to ack before declaring success, or lazily and declare success after the master finishes write
 - ▶ Read may happen on the master (strong consistency) or at one replica (may get stale data)
- In an environment where there is no designated master/coordinator, other mechanisms need to be used to ensure certain level of consistency
 - ▶ Order of concurrent writes
 - ▶ How many replica to contact before answering/acknowledging



Concurrent Write

- Different clients may try to update an item simultaneously
- If nothing special is done, system could end with “split-brain”



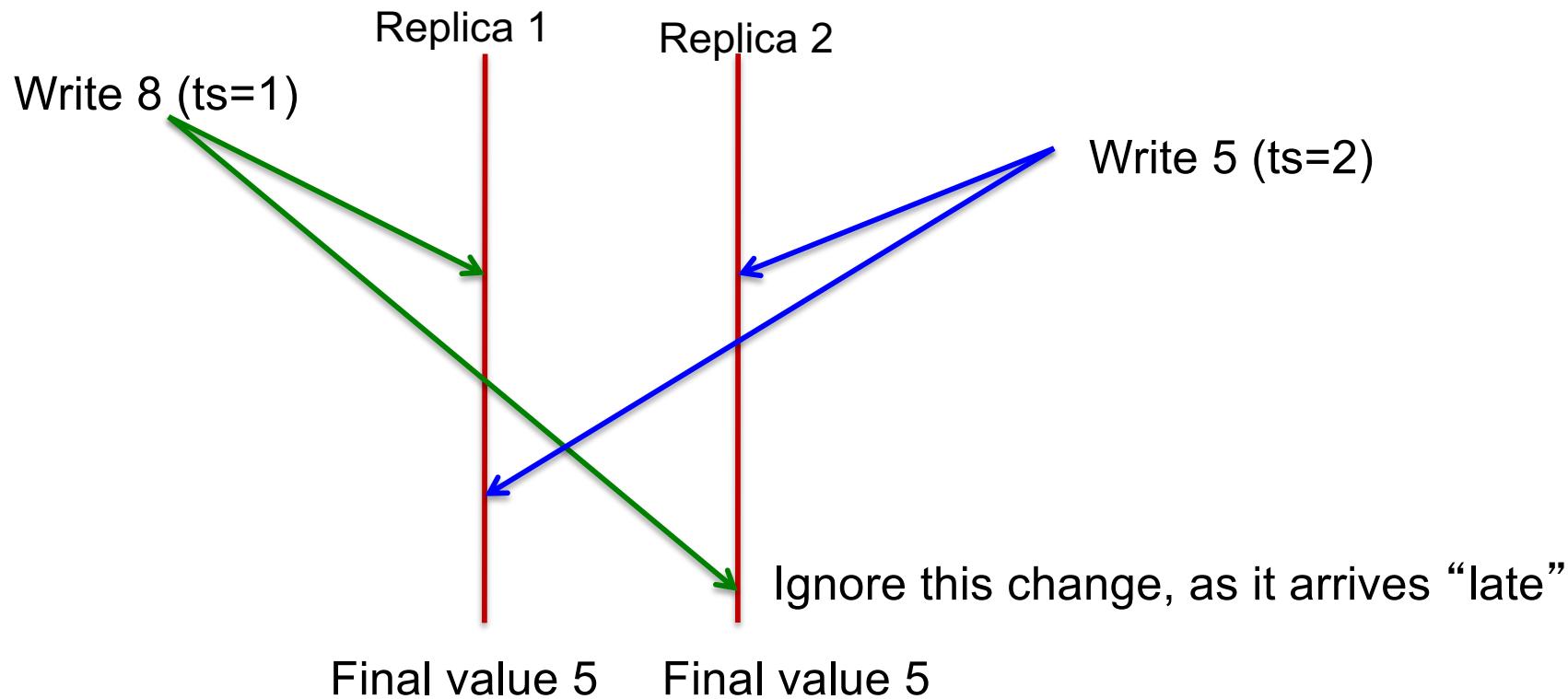
Timestamp is a typical way to order concurrent writes

Slide based on material by Alan Fekete



Ordering concurrent writes

- Associate timestamps on the writes, and let higher timestamp win
 - ▶ Node ignores a write whose timestamp is lower than the value already there (Thomas Write Rule)



Slide based on material by Alan Fekete



Quorums

- Suppose each write is *initially* done to W replicas (out of N)
 - ▶ Other replicas will be updated later, after write has been acked to the client
- How can we find the current value of the item when reading?
- Traditional “quorum” approach is to look at R replicas
 - ▶ Consider the timestamp of value in each of these
 - ▶ Choose value with highest timestamp as result of the read
- If $W > N/2$ and $R + W > N$, this works properly
 - ▶ any write and read will have at least one site in common (quorums intersect)
- Any read will see the most recent completed write,
 - ▶ There will be at least one replica that is BOTH among the W written and among the R read

Slide based on material by Alan Fekete



Dynamo Mechanisms

■ Vector Clock

- ▶ A way to implement the “timestamp” concept in distributed system
- ▶ Aid for resolving version conflict
 - E.g. when read receives R copies, how do we decide if there is causal order among the copies or if they are on different branches?

■ Sloppy Quorum + hinted hand off

- ▶ Achieve the “always writable” feature
- ▶ Eventual consistency



Dynamo Read/Write Route

- Any node is eligible to receive client read/write request
 - ▶ get (key) or put (key, value)
- The node receives the request can direct the request to the node that has the data and is available
 - ▶ Any node knows the token of other nodes
 - ▶ Any node can compute the hash value of the key in the request and the preference list
 - ▶ A node has local knowledge of node failure
- In Dynamo, “A node handling a read or write operation is known as the coordinator. Typically, this is the first among the top N nodes in the preference list.”, which is usually the coordinator of that key unless that node is not available.
 - ▶ Every node can be the coordinator of some operation
 - ▶ For a given key, the read/write is usually handled by its coordinator or one of the other top N nodes in the preference list



Data Versioning

- A **put()** call may return to its caller before the update has been applied at all the replicas
 - ▶ $W < N$
- A **get()** call may return many versions of the same object/value.
 - ▶ $R > 1$ and $R < N$
- Typically when $N = 3$, we may have $W = 2$ and $R = 2$
- Challenge: an object may have distinct version sub-histories, which the system will need to reconcile in the future.
- Solution: uses vector clocks in order to capture causality between different versions of the same object.



Vector Clock: definition

- “A vector clock is effectively a list of (node, counter) pairs. One vector clock is associated with every version of every object.”
 - ▶ E.g. $[(n_0, 1), (n_1, 1)]$, $[(n_1, 2), (n_2, 1), (n_3, 2)]$
 - ▶ Node refers to the node that coordinate the write
 - ▶ Counter remembers how many times this node has coordinated the write
- $[(n_0, 1), (n_1, 1)]$ means the associated version of the object has been updated twice: once coordinated by n_0 and once coordinated by n_1 ;
- $[(n_1, 2), (n_2, 1), (n_3, 2)]$ means the associated version of the object has been updated 5 times coordinated by three different nodes;
- Coordinator variation is caused by temporary node failure
- Vector clock is not designed for deriving the coordination history of a single version.
- It is designed to determine causal ordering between a pair of clocks belonging to two versions of the same object.



Vector Clock: causal ordering

■ “One can determine whether two versions of an object are on parallel branches or have a causal ordering, by examine their vector clocks. If the counters on the first object’s clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten. Otherwise, the two changes are considered to be in conflict and require reconciliation.”

- ▶ $[(n_0, 1), (n_1, 1)] < [(n_0, 2), (n_1, 1), (n_3, 1)]$
 - The second version is 3 2 updates ahead of the first one
- ▶ $[(n_0, 1), (n_1, 1)] ?? [(n_0, 2), (n_2, 1)]$
 - The two versions branch off after the initial insert with $[(n_0, 1)]$,
 - the left version is then updated once, and this is coordinated by n_1
 - the right version is updated twice: once by n_2 and once by n_0 ; we don’t know which happens first

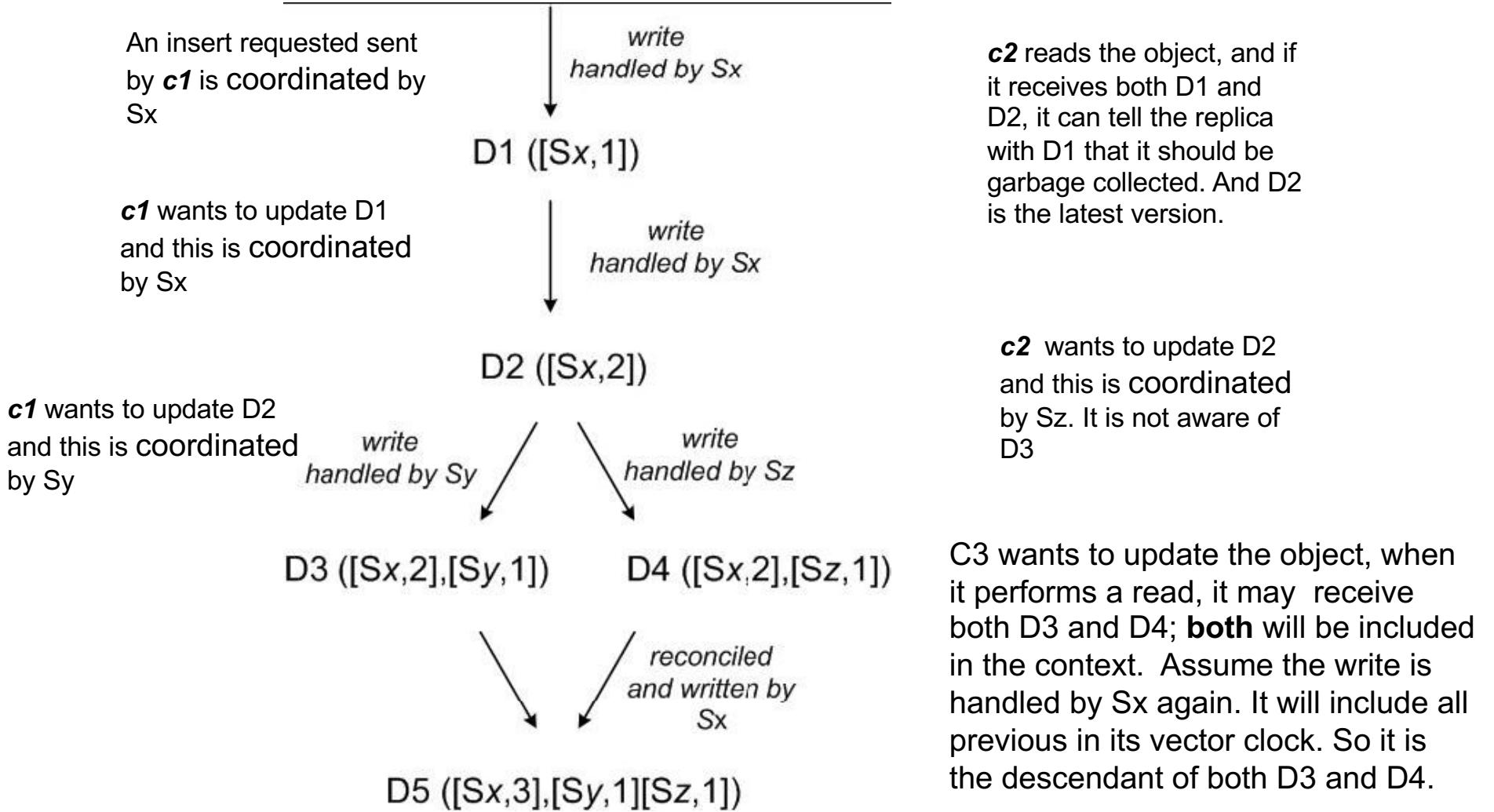


Update with vector clock

- Data replication revisits
 - ▶ Each replica stores a copy of an object
 - ▶ The copy stored on different replica may be of different version, as indicated by its vector clock
 - ▶ This happens because any machine hosting the replica may be temporarily unavailable and miss some update requests
 - ▶ Both read/write may contact multiple replica and obtain multiple versions
- Update with vector clock mechanism
 - ▶ “In Dynamo, when a client wishes to update an object, it must specify which version it is updating. This is done by passing the context it obtained from an earlier read operation, which contains the vector clock information. Upon processing a read request, if Dynamo has access to multiple branches that cannot be syntactically reconciled, it will return all the objects at the leaves, with the corresponding version information in the context. An update using this context is considered to have reconciled the divergent versions and the branches are collapsed into a single new version.”



Vector Clock Example



Vector Clock Size

■ The size of vector clock is relatively small

- ▶ Only the node coordinates the write request has an entry in the vector clock
- ▶ If the preference list contains 4 nodes, most vector clocks contain up to 4 entries
- ▶ In rare situation when all nodes in the preference list is not available, another node will coordinate the write and add an entry in the vector clock
- ▶ A vector clock truncate mechanism is used to remove the oldest entry
 - Additional timestamp required to determine the 'oldest entry'
- ▶ Truncating may remove histories that needed in reconciliation
 - It has never occurred in real life so is not explored extensively



Vector Clock More Examples

- In a system with 6 nodes: n0~n5, for a key with preference list {n1~n4} which of the following vector clock pair(s) has(have) causal order:
 - ▶ [(n1, 1), (n2,2)] and [(n1,1)]
 - ▶ [(n1, 3), (n3,1)] and [(n1,2),(n2,1)]
 - ▶ [(n1,2),(n3,1)] and [(n1,4),(n2,1),(n3,1)]



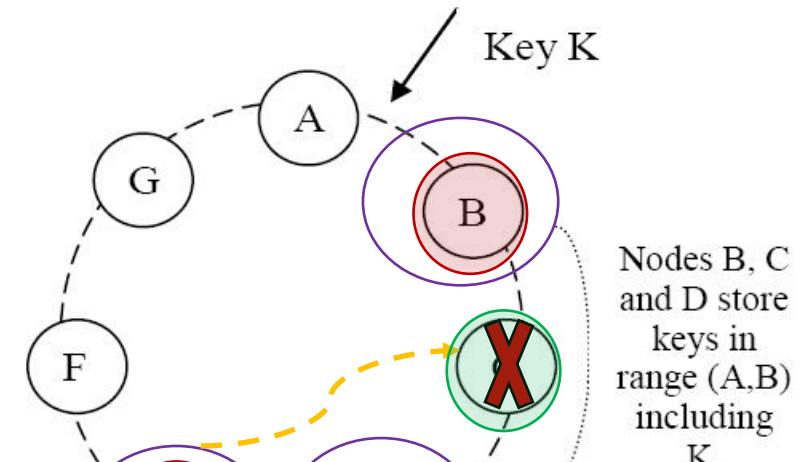
Sloppy Quorum

- Quorum members may include nodes that do not store a replica
 - ▶ Preference list is larger than N
 - ▶ Read/Write may have quorum members that do not overlap
- Both read and write will contact the first N *healthy* nodes from the preference list and wait for **R** or **W** responses
- Write operation will use hinted handoff mechanism if the node contacted does not store a replica of the data



Hinted Handoff

- Assume $N = 3$. When C is temporarily down or unreachable during a write, send replica to E.
- E is hinted that the replica belongs to C and it should deliver to C when C is recovered.
- Sloppy quorum does not guarantee that read can always return the latest value
 - ▶ Write set: (B,D,E)
 - ▶ Read set: (C,D, E)



Write contacts nodes B,D,E and acks to client after B and E reply

Read contacts nodes C,D,E and replies the client a merged value based on replies from C and D

References

- Brad Fitzpatrick, “Distributed Caching with Memcached” Linux Journal” [Online]. August 1, 2004. Available: <https://www.linuxjournal.com/article/7451>.
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. **Dynamo: amazon's highly available key-value store**. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (SOSP '07). 205-220.



COMP5338 – Advanced Data Models

Week 9: Spatial Data Model and Query

Dr. Ying Zhou
School of Computer Science



Outline

- Motivation
- Spatial Data Model
- Spatial Data Queries
- Spatial Query in MongoDB



Motivation

- Many entities represent physical objects, and some physical features also matter for business purposes
 - ▶ Eg. A store has a name, business category, contact number and is located at a particular place (geo-spatial)
 - The geo-spatial feature can help find a store that is close to a customer
 - ▶ Eg. A toy has name, category, material, and shape
 - The shape feature helps to find a box that can fit the toy
- There are large amount of geospatial data:
 - ▶ Businesses and homes have addresses
 - Both the logic aspect and physical aspect
 - ▶ Google Maps, Google Earth
 - ▶ Weather and Climate Data



Spatial Data and Object Concept

- Spatial feature could refer to a
 - ▶ Point, a 2d shape, a 3d shape, or shape in higher dimension
- Spatial features of an entity cannot be represented as simple value type
 - ▶ A point in 2d space has two coordinate values
- It is natural to use object to represent spatial features (spatial data)
- There are also spatial related operations we need to perform on spatial data
 - ▶ Compute distance between points
 - ▶ Compute area of 2d shape or volume of 3d shapes
 - ▶ Compute various relationships among spatial objects

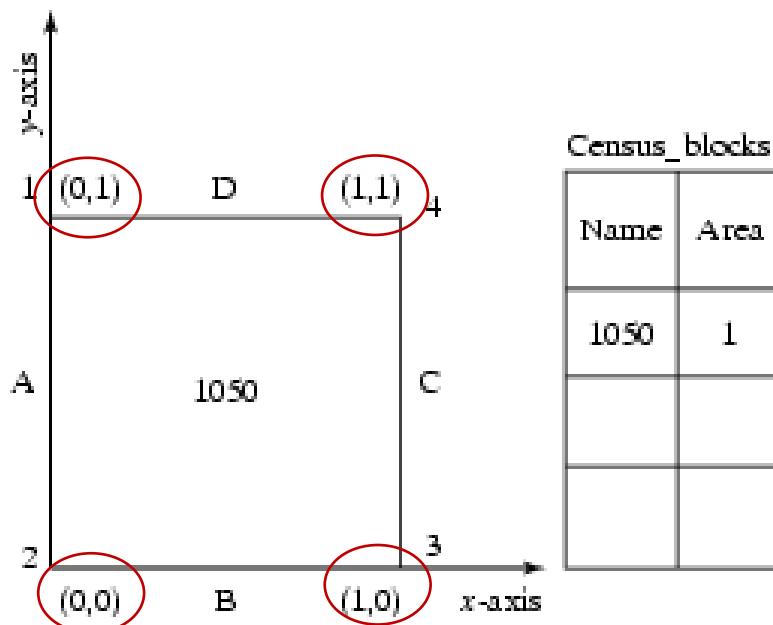


Spatial Data as Object Example

- Consider a spatial data representing census block:

```
CREATE TABLE census_blocks (
    name      string,
    area      float,
    population   number,
    boundary    Polyline );
```

A User
Defined Type



Census_blocks			
Name	Area	Population	Boundary
1050	1	1839	Polyline ((0,0),(0,1),(1,1),(1,0))

From Shekhar-Chawla, Fig 1.4



Spatial Data in Purely Relational Form

Census_blocks

Name	Area	Population	boundary-ID
340	1	1 839	1050

Polygon

boundary-ID	edge-name
1050	A
1050	B
1050	C
1050	D

Edge

edge-name	endpoint
A	1
A	2
B	2
B	3
C	3
C	4
D	4
D	1

Point

endpoint	x-coor	y-coor
1	0	1
2	0	0
3	1	0
4	1	1

From Shekhar-Chawla, Fig 1.4



Spatial Database Management System

- A SDBMS is a software module that
 - ▶ can work with an underlying DBMS
 - ▶ supports spatial data models, spatial abstract data types (ADTs) and a query language from which these ADTs are callable
 - ▶ supports spatial indexing, efficient algorithms for processing spatial operations, and domain specific rules for query optimization
 - ▶ Many RDBMS and NoSQL storage systems have support for spatial data
 - Oracle, SQL Server, MongoDB, Neo4j, ...
- SDBMS components
 - ▶ spatial data model
 - ▶ query language
 - ▶ query processing
 - ▶ file organization and indices
 - ▶ query optimization
 - ▶ etc.



Outline

■ Motivation

■ Spatial Data Model

- ▶ Field vs. Object Models
- ▶ Coordinate System
- ▶ Topological Operations

■ Spatial Data Queries

■ Spatial Query in MongoDB



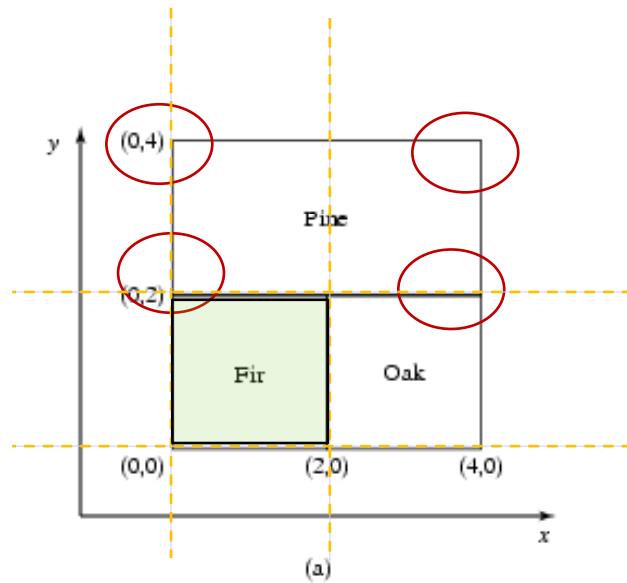
Models of Spatial Information

■ Two common models

- ▶ Field based (*also: space-based*)
 - Model properties of underlying space
 - Good for expressing values vary continuously over space (e.g. temperature, rainfall, elevation, depth, etc.)
 - Fields are actually **functions** that map spatial locations to values
- ▶ Object based (*also: entity-based*)
 - Model boundaries of spatial feature (e.g. the census block is modelled by a polygon)
 - Good for expressing discrete spatial entities



Examples of Field and Object Models



- (a) forest stand map
(b) Object model has 3 polygons defining the boundaries
(c) Field model uses a function to calculate the property "tree species"

Area-ID	Dominant Tree Species	Area/Boundary
FS1	Pine	$[(0,2), (4,2), (4,4), (0,4)]$
FS2	Fir	$[(0,0), (2,0), (2,2), (0,2)]$
FS3	Oak	$[(2,0), (4,0), (4,2), (2,2)]$

(b)

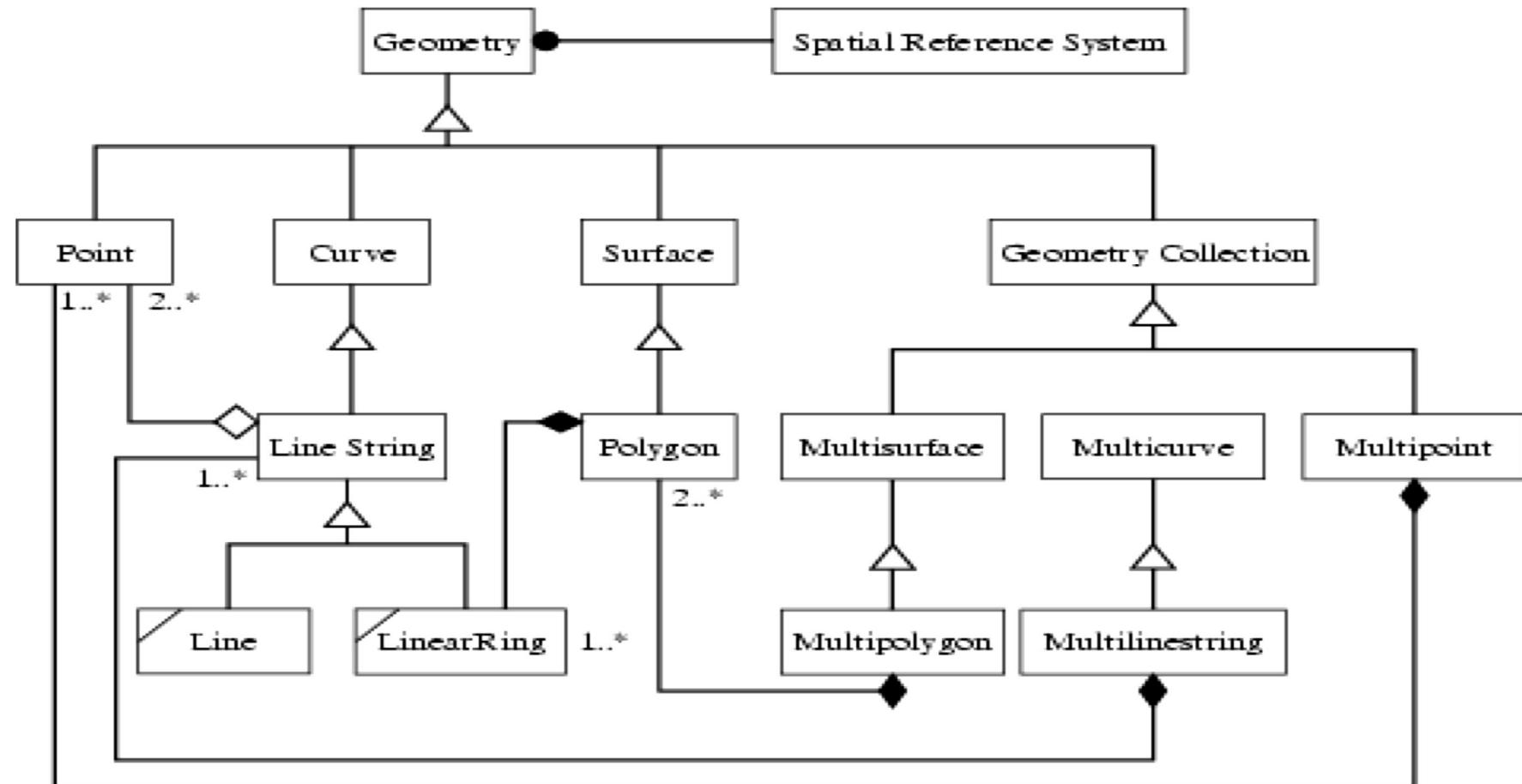
$$f(x,y) = \begin{cases} \text{"Pine," } 2 \leq x \leq 4 ; 2 < y \leq 4 \\ \text{"Fir," } 0 \leq x \leq 2; 0 \leq y \leq 2 \\ \text{"Oak," } 2 < x \leq 4; 0 \leq y \leq 2 \end{cases}$$

(c)

From Shekhar-Chawla, Fig 2.1



OpenGIS Geometry Model



From Shekhar-Chawla, Fig 2.2



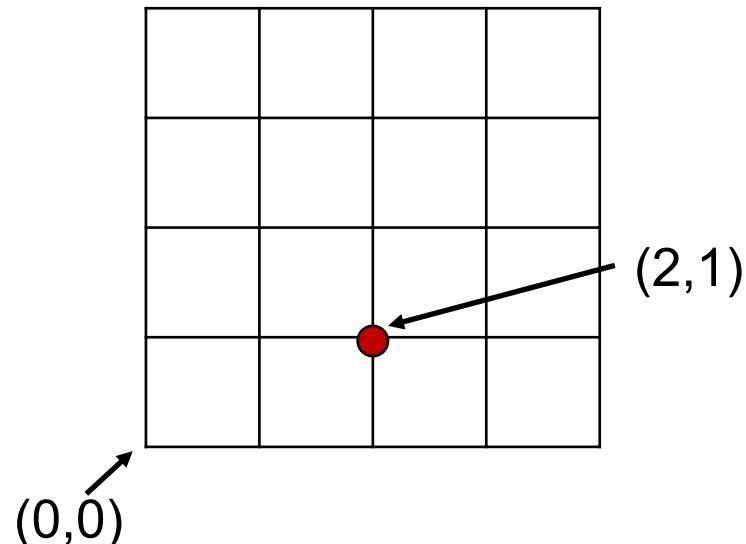
Example Spatial Objects

- **Point:** represented by its coordinates eg (-10, 30)
- Collection of several **points**
- **Line String**
 - ▶ Simplest form is piece-wise linear, given by points (and implying the straight segments between them)
 - ▶ Eg (0,1), (1,1), (2,2)
- **Polygon**
 - ▶ A 2-d region whose boundary is given
 - ▶ Simplest form: boundary is a **line string** that returns to its start
 - ▶ More complicated: region with holes
- Collection of **polygons**



Coordinate Systems

- Points from a 2-d space are represented by pairs of numbers
 - ▶ The numbers could refer to a dot on a drawing area, a piece of land in a game setting, or a location on earth
- There are many ways to associate numbers with points
- Simple 2d Cartesian coordinate
 - ▶ Choose a point as origin $(0,0)$
 - ▶ Choose a direction for the x-axis, and a scale (how far is $(1,0)$) from $(0,0)$?)



A Round World

- The surface of the earth is 2-dimensional, but curved
 - ▶ Cartesian systems work reasonably in small regions
- Traditional geographic coordinate system
 - ▶ 2d: longitude and latitude
 - ▶ 3d: longitude, latitude, elevation
 - ▶ The surface is a sphere
 - (-179,10) is very close to (179, 10)
 - A linestring might cross the (long = 180) line;
- Many SDBMS supports both flat space and sphere



Operations on Spatial Objects in the Object Model

■ Classifying operations

- ▶ **Set based:**
 - a set operation (e.g. intersection) of 2 polygons produce another polygon
- ▶ **Topological operations:** Boundary of USA touches boundary of Canada
- ▶ **Directional:** New York city is to east of Chicago
- ▶ **Metric:** Chicago is about 700 miles from New York city.

Set theory based	Union, Intersection, Containment
Topological	Touches, Disjoint, Overlap, etc.
Directional	East, North-West, etc.
Metric	Distance



Topological Relationships

■ Topological Relationships

- ▶ invariant under elastic deformation (without tear, merge).
- ▶ Two countries which touch each other in a planar paper map will continue to do so in spherical globe maps.

■ Example queries with topological operations

- ▶ What is the topological relationship between two objects A and B ?
- ▶ Find all objects which have a given topological relationship to object A?
 - E.g. find all rivers that cross a city

■ Can we express topological relationship mathematically?

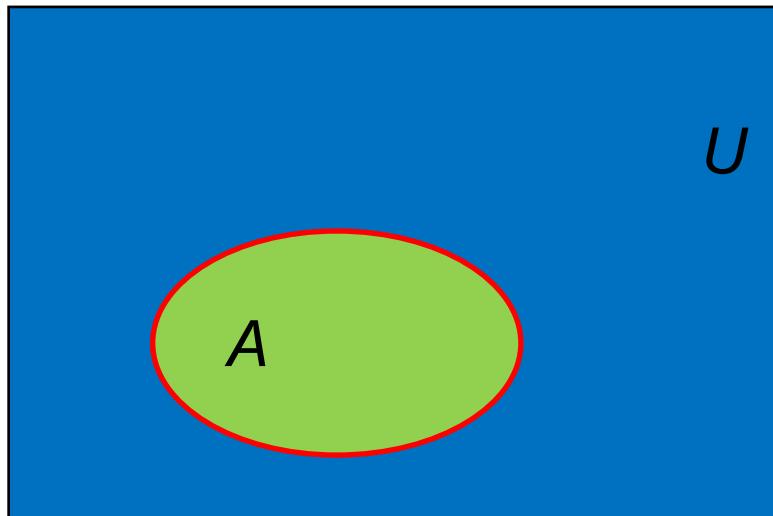
- ▶ Metric operations may be expressed using various functions, e.g. distance function
- ▶ Set operations can be express mathematically
- ▶ The mathematical form helps to calculate such relationships



Topological Concepts

■ Interior, boundary, exterior

- ▶ Let A be an object in a “Universe” U .



Green is A interior (A^o)

Red is boundary of A (∂A)

Blue – (Green + Red) is
 A exterior (A^-)

- ▶ Exterior is also referred to as the *complement* of an object

Nine-Intersection Model of Topological Relationships

- Many topological Relationship between A and B can be specified using 9 intersection model
 - ▶ Eight possible 2D topological relationships for objects without holes;
- Nine intersections
 - ▶ intersections between interior, boundary, exterior of A, B
 - ▶ A and B are spatial objects in a two dimensional plane.
 - ▶ Can be arranged as a 3 by 3 matrix
 - ▶ Matrix element take a value of 0 (false) or 1 (true)

$$\Gamma_9(A, B) = \begin{pmatrix} A^0 \cap B^0 & A^0 \cap \partial B & A^0 \cap B^- \\ \partial A \cap B^0 & \partial A \cap \partial B & \partial A \cap B^- \\ A^- \cap B^0 & A^- \cap \partial B & A^- \cap B^- \end{pmatrix}$$

From Shekhar-Chawla, p 28



Specifying Topological Operations using the 9-Intersection Model

	I	B	E
I			
B			
E			

For **disjoint** relation:
 A's exterior intersects with
 B's everything and vice
 versa

If A **contains** B:
 A's interior intersects with
 B's everything and B's
 exterior intersects with A's
 everything

$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ <p>disjoint</p>	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ <p>contains</p>	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ <p>inside</p>	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ <p>equal</p>
$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ <p>meet</p>	$\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$ <p>covers</p>	$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ <p>coveredBy</p>	$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$ <p>overlap</p>

From Shekhar-Chawla, Fig 2.3



Outline

■ Motivation

■ Spatial Data Model concepts

■ Spatial Data Queries

- ▶ Query type
- ▶ General processing steps

■ Spatial Query in MongoDB



Spatial Processing

- Find one or more entities, based on non-spatial aspects, then use spatial operations to get interesting data associated with these items
 - ▶ Eg find the length of the river called ‘Mississippi’
 - Find the river based on name (non-spatial), use spatial operation to compute the length (assuming it is not stored as a numeric value)
 - ▶ Eg find the total area of all counties whose population exceeds 1,000,000
 - Find the counties with population exceeds 1,000,000 (non-spatial), compute each county’s area(spatial operation) and sum all up.
- To answer these: use conventional index or table scan to find the appropriate rows, then call spatial functions on the spatial attribute of each



Spatial selection queries

- Find items whose spatial attribute has certain properties
 - ▶ Eg find rivers whose length is at least 10000
- **WHERE** clause will involve spatial operations
- Typical processing: scan all rows, apply appropriate spatial operation to the spatial attribute of each



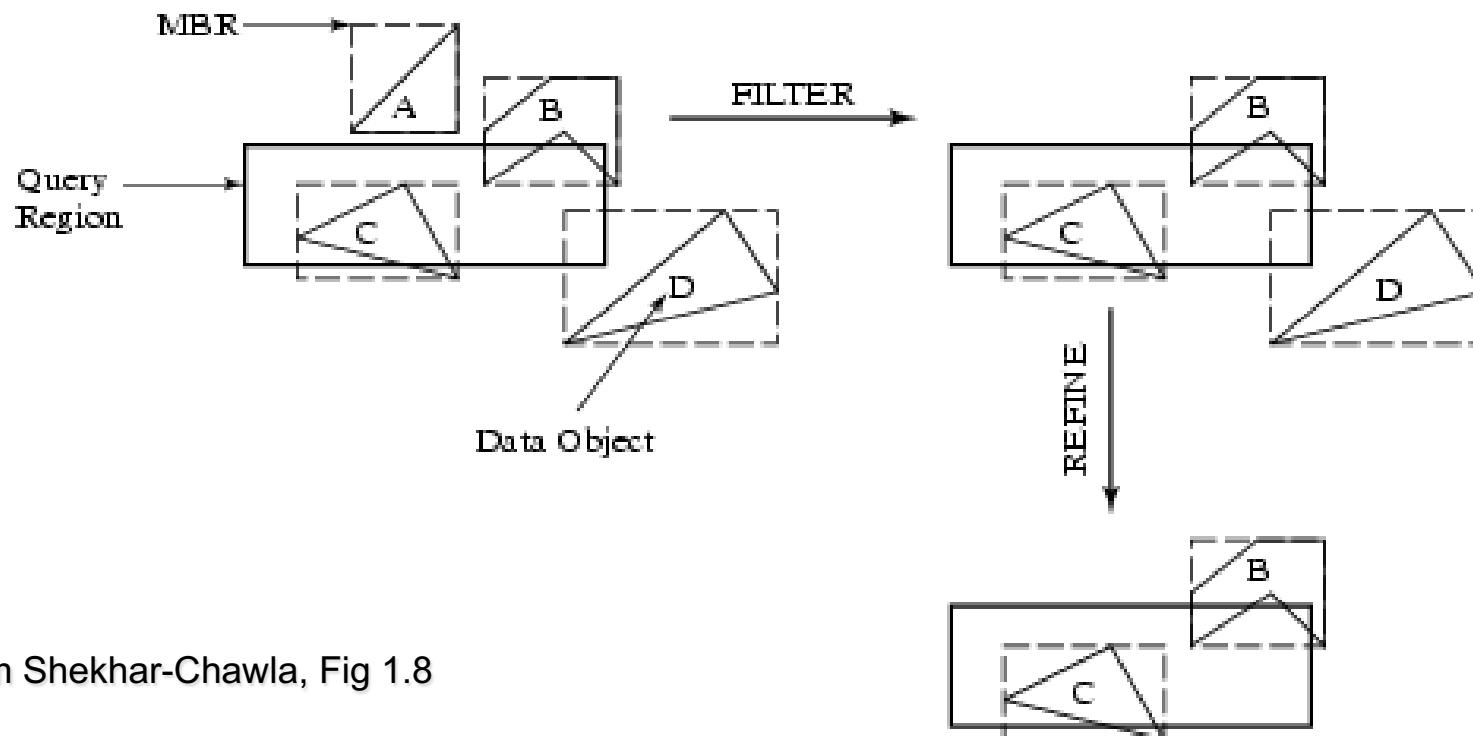
Spatial Range Query

- A particular kind of spatial selection, in which the condition involves a topological or metric relationship to a given object
 - ▶ Eg find all bookshops whose location is inside a given region
 - ▶ Eg find all farms that contain (part of) a given curve
 - ▶ Eg find all rivers that flow through a given region
 - ▶ Eg find all bookshops within 100 km of a given point
 - Equivalent to: find all whose location is inside a circular region of radius 100 km!
- Simple processing: scan the appropriate table, apply appropriate spatial operation to each item's spatial attribute
- But often one can do better: first filter to find a small set where the condition might be feasible; then refine the list by checking in detail each that pass the filter



Spatial Query Processing: Filter-Refine Strategy

- Eg find objects that intersect a query region
- **Filter Step:** made easy if each object has associated to it a simple shape that surrounds it (eg Minimum Bounding Rectangle)
 - ▶ If object's MBR doesn't intersect query {or MBR of query}, there is no possibility that the object itself will intersect the query
- **Refine Step:** Actually perform intersection method for those objects that get through the filter



From Shekhar-Chawla, Fig 1.8



Nearest neighbours

- Find entities that are as close as possible to given location,
Always give a bound on how many to find (K Nearest
Neighbours)
 - ▶ Eg find 5 closest restaurants to (100, 350) and return them ranked by
closeness
- Simple processing: scan all, compute distance; keep track
of the ones with lowest distances seen so far
- Many index based algorithms, see next week



Spatial Join Query

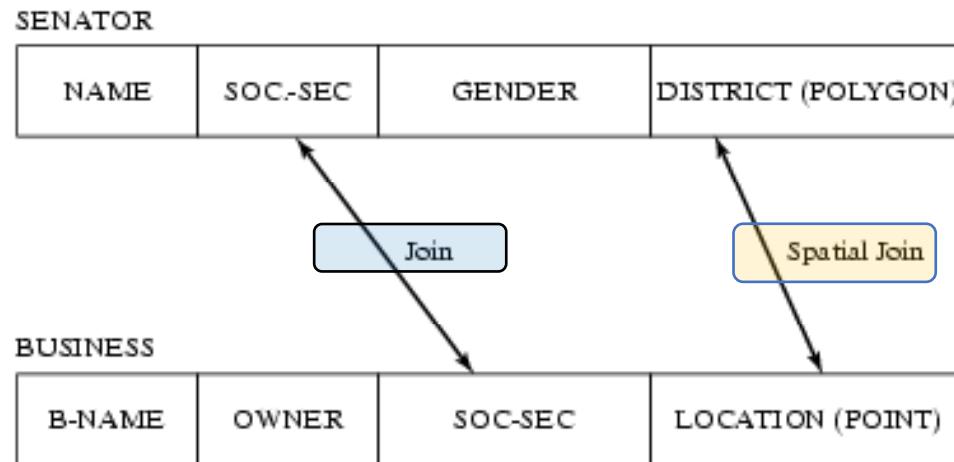
■ Spatial join example

```
SELECT      S.name  
FROM        Senator S, Business B  
WHERE       S.district.Area() > 300  
            AND Within(B.location, S.district)
```

■ Non-Spatial Join example

```
SELECT S.name  
FROM   Senator S, Business B  
WHERE  S.soc-sec = B.soc-sec AND S.gender = 'Female'
```

■ Similar to non-spatial join, spatial join are usually very expensive to process



From Shekhar-Chawla, Fig 1.7



Outline

- Motivation
- Spatial Data Model concepts
- Spatial Data Queries
- Spatial Query in MongoDB



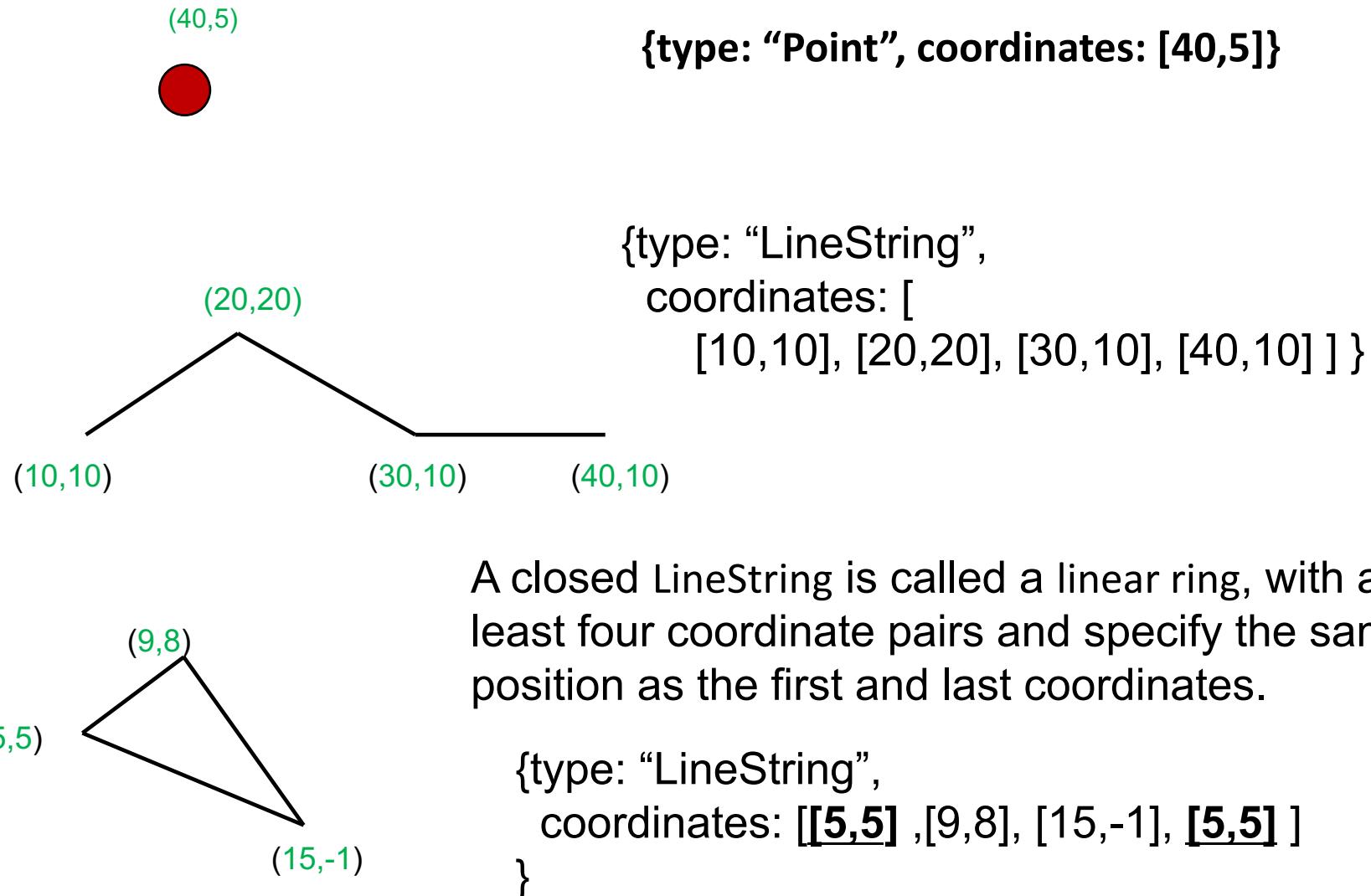
Spatial data: GeoJSON

- Spatial data in MongoDB can be stored as **GeoJSON** object or as legacy coordinate pairs
 - ▶ **GeoJSON** data assumes earth-like sphere
 - ▶ Legacy coordinate pairs assumes flat plane
- **GeoJSON** object uses JSON format to represent spatial objects in OpenGIS
 - ▶ Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, Geometry Collection
 - ▶ General format

```
{ type: "<GeoJSON type>" , coordinates: <coordinates>}
```
 - ▶ The coordinate reference system for all GeoJSON coordinates is a geographic coordinate reference system, using the World Geodetic System 1984 (WGS 84) [WGS84] datum, with longitude and latitude units of decimal degrees.



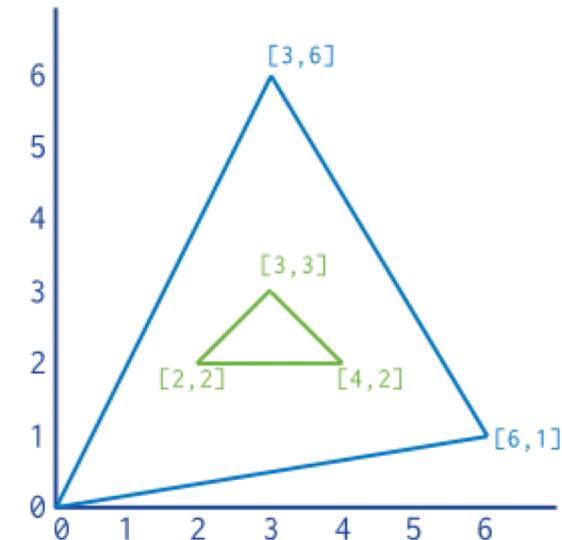
Point and LineString



Polygon

- Polygon is used to model two dimensional surface
 - ▶ Triangle, Rectangle, Pentagon, ...
 - ▶ A polygon is represented as one or many linear rings, the first is the exterior ring bounds the surface, the others are interior rings bound holes within the surface

```
{  
    type : "Polygon",  
    coordinates : [  
        [[0 ,0 ],[ 3 ,6 ],[ 6 ,1 ],[ 0 ,0 ]],  
        [[2 ,2 ],[ 3 ,3 ],[ 4 ,2 ],[ 2 ,2 ]]  
    ]  
}
```



Model multiple disjoint objects

■ MultiPoints, MultiPolygon

```
{ type: "MultiPoint",
  coordinates: [ [ -73.9580, 40.8003 ],
    [ -73.9498, 40.7968 ],
    [ -73.9737, 40.7648 ],
    [ -73.9814, 40.7681 ] ] }

{ type: "MultiPolygon",
  coordinates: [ [ [ [ -73.95, 40.80 ], [ -73.9498, 40.79 ], [ -73.97, 40.76 ], [ -73.95, 40.80 ] ] ],
    [ [ [ -73.95, 40.80 ], [ -73.94, 40.79 ], [ -73.97, 40.76 ], [ -73.95, 40.80 ] ] ] ] }
```

■ Geometry collection

```
{ type: "GeometryCollection",
  geometries: [ { type: "MultiPoint",
    coordinates: [ [ -73.95, 40.80 ], [ -73.94, 40.79 ] ] },
    { type: "MultiLineString",
      coordinates: [ ... ] }
  ] }
```



Spatial Queries

■ **\$near** and **\$nearSphere**

- ▶ Specifies a point for which a **geospatial** query returns the documents from nearest to farthest
- ▶ Can be used to run queries like find all car parks/restaurants within certain distance

■ **\$geoWithin**

- ▶ Find all geo objects contained in a query shape

■ **\$geoIntersects**

- ▶ Find all geo objects intersects with a query shape. Here intersect includes relationships such as cover, equal, overlap, touch and so on.

■ Others



Spatial Index

- **2dsphere** indexes supports all MongoDB geospatial queries

- Eg.

```
db.places.insert(  
  {  
    loc : { type: "Point", coordinates: [ -73.97, 40.77 ] },  
    name: "Central Park",  
    category : "Parks"  
  }  
)
```

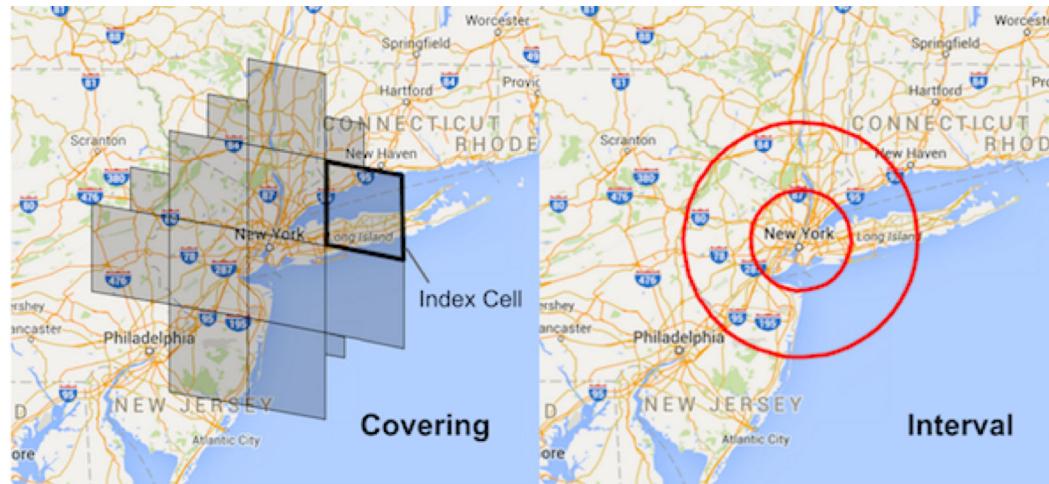
- **db.places.createIndex({ loc : "2dsphere" })**



MongoDB Spatial Index

- “ MongoDB’s 2dsphere index actually combines the strength of discrete global grids and B+ -tree structures, which first partitions the Earth surface into cells at multiple resolution levels and then applies a B+ -tree to index geographical features approximated as one or multiple cells.”

L. Xiang, J. Huang , X. Shao and D. Wang: MongoDB-Based Management of Planar Spatial Data with a Flattened R-Tree
<https://pdfs.semanticscholar.org/860f/0cfe3e4b4cb66b2b2895016a60e824e9e9f8.pdf>

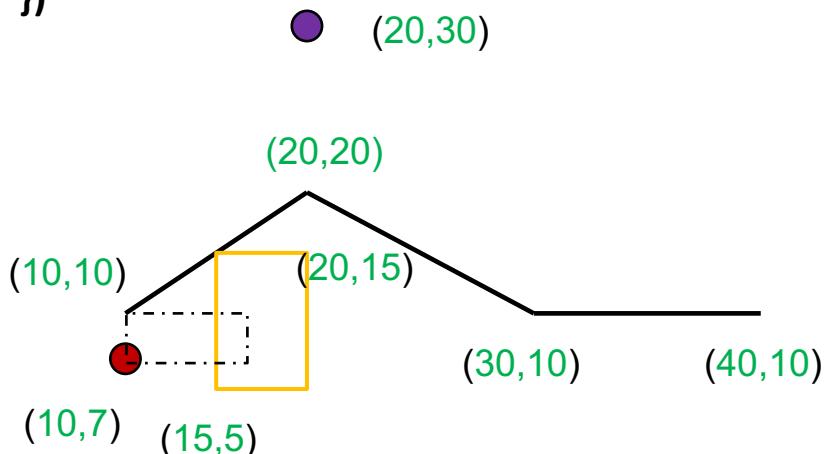


MongoDB Blog: Geospatial Performance Improvements in MongoDB 3.2
<https://www.mongodb.com/blog/post/geospatial-performance-improvements-in-mongodb-3-2>



Spatial Query -- \$geoIntersects

```
db.places.find({  
    loc :{$geoIntersects:  
        {$geometry:{  
            type: "Polygon",  
            coordinates: [[[10,7], [10,10], [17,10],[17,7],[10,7]]]}  
        }  
    }  
})
```



References

- S. Shekhar and S.Chawla: *Spatial Databases: A Tour*. Prentice Hall, 2002. [<http://www.spatial.cs.umn.edu/Book/>]
 - ▶ Chapter 1-3
- MongoDB document on geospatial query
 - ▶ <https://docs.mongodb.com/manual/geospatial-queries/>



COMP5338 – Advanced Data Models

Week 10: Spatial Index

Dr. Ying Zhou
School of Computer Science



Outline

- Index Motivation
- Hash Structure
- Tree Structure
- Space Filling Curve Techniques

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

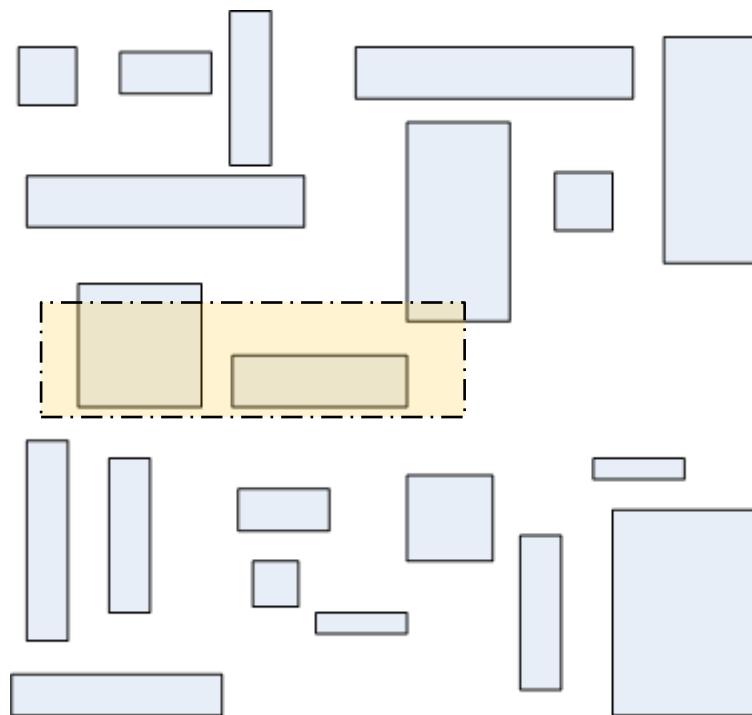


Revisit: File Organization and Index Basics

- Any disk based storage systems store data in files
- Part of the file is read into memory during read/write operation
- Files are organized into fixed sized disk blocks (e.g. 4K in NTFS)
 - ▶ Each block may contain a few data records
 - ▶ It is the basic IO unit
- Index is also stored as file
 - ▶ Also consists of blocks
 - ▶ May be loaded entirely in the memory
 - ▶ Is used to decide which block(s) of the data file need to be loaded
 - ▶ It reduces disk I/O cost as well as record inspection cost



Typical Spatial Query



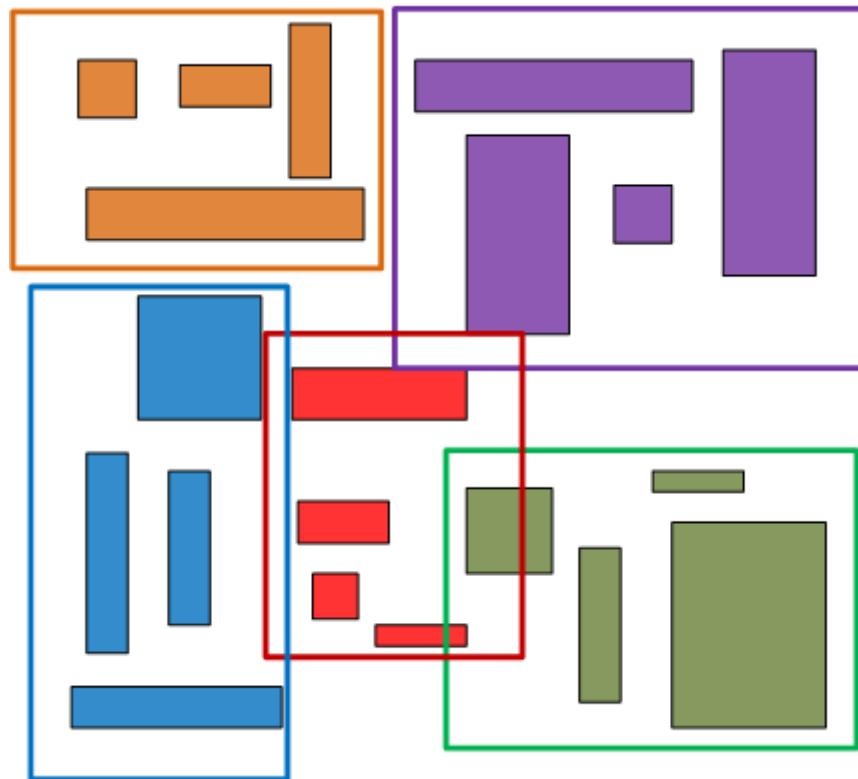
Find all rectangles that intersect a rectangular query range



Organizing Storage Block

The assumption is that the database hold mostly spatial objects and most query workload consists of spatial queries

We store objects spatially close to each other in the same disk block

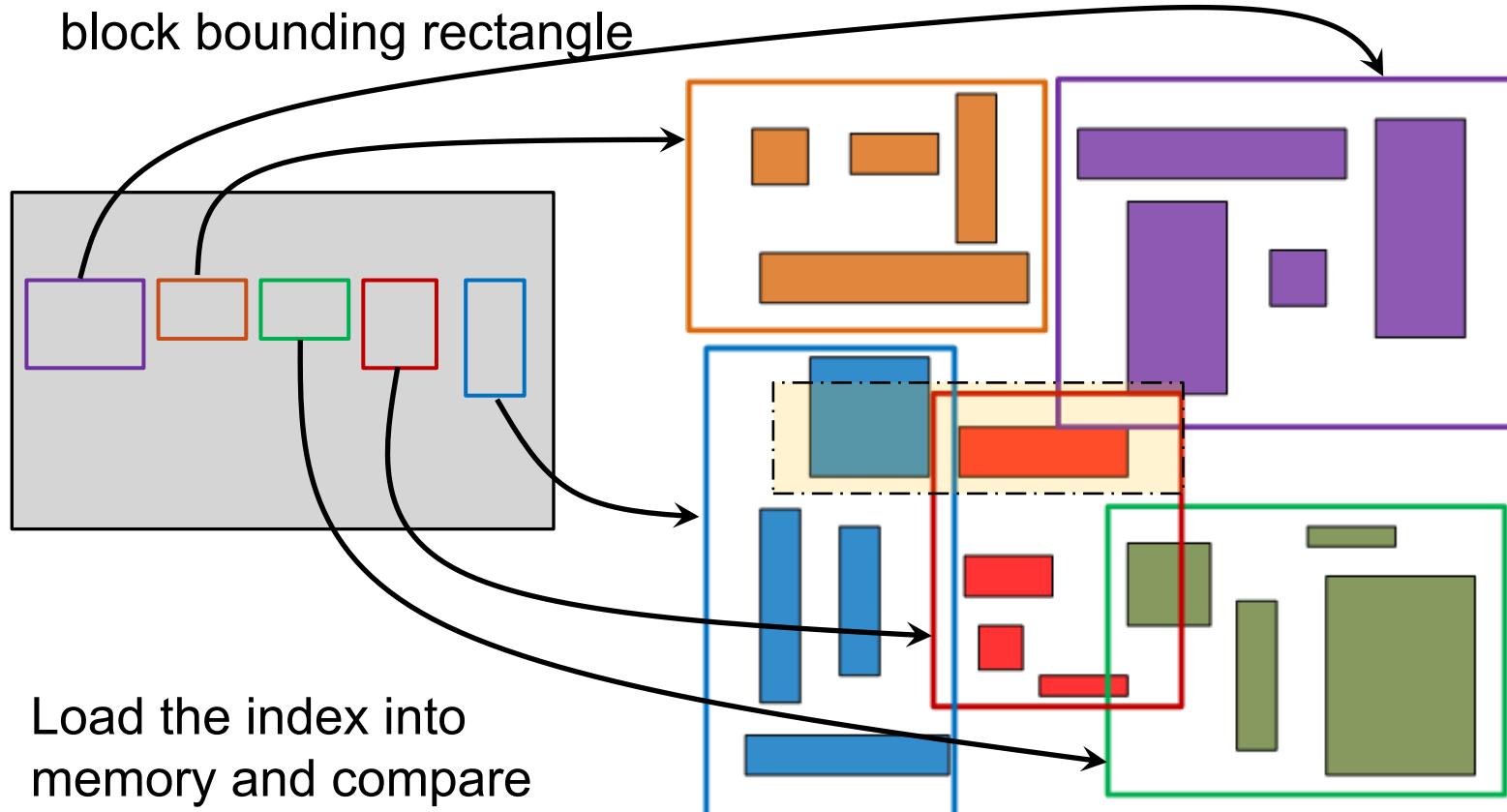


Suppose each storage block can hold up to 4 objects, we want to minimize the number of blocks retrieved



Indexing Storage Block

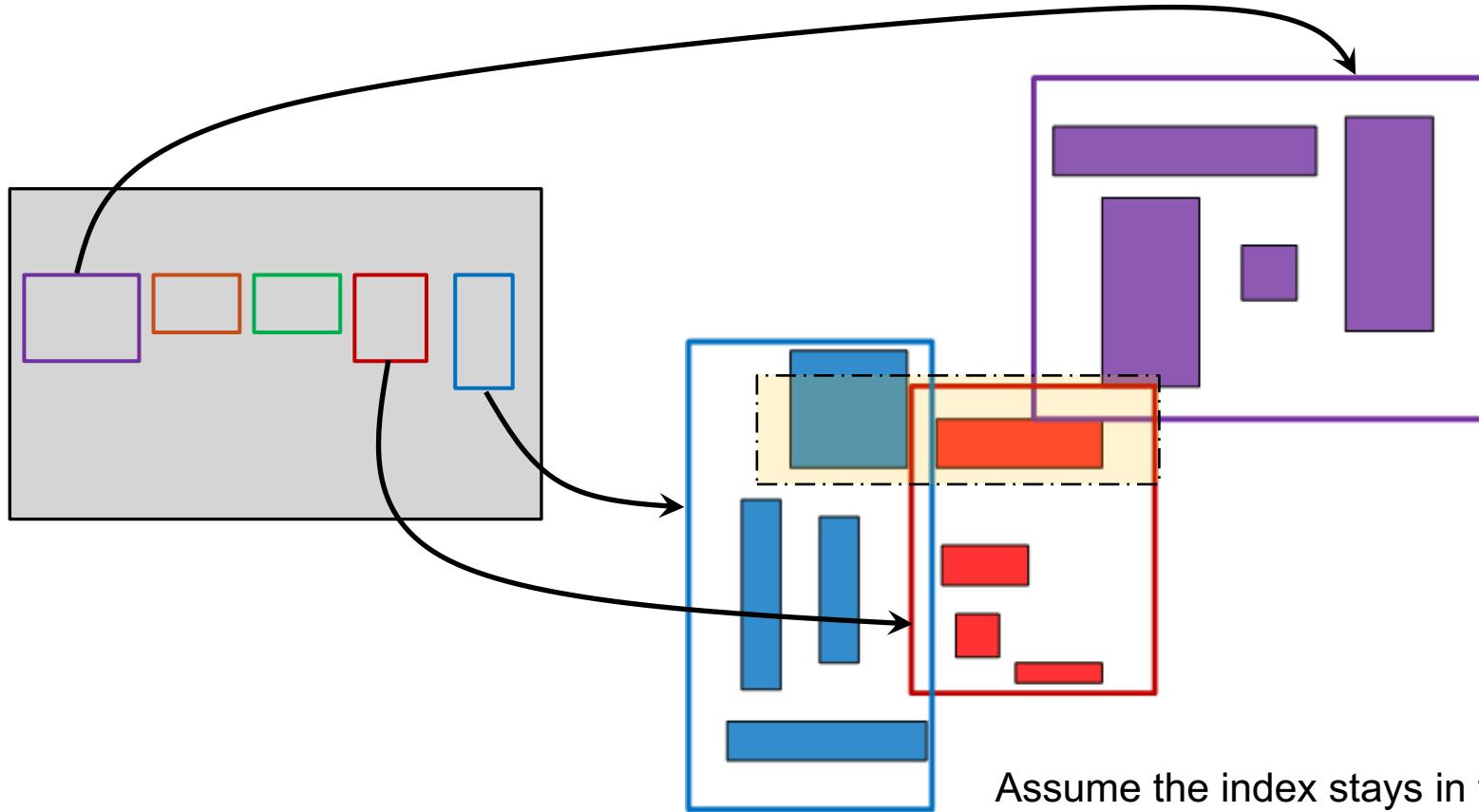
Create indexes using
block bounding rectangle



Load the index into
memory and compare
each bounding
rectangle with the
query rectangle



Read 3 blocks



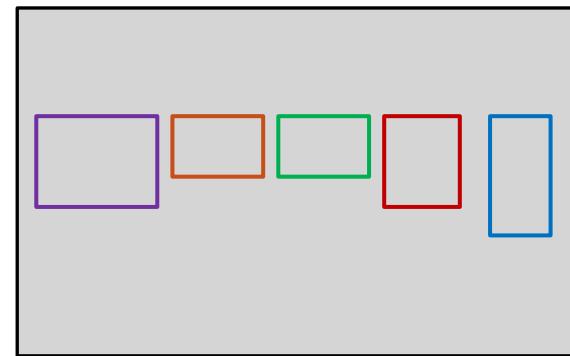
Two issues:

- Space segmenting
- Index organization structure

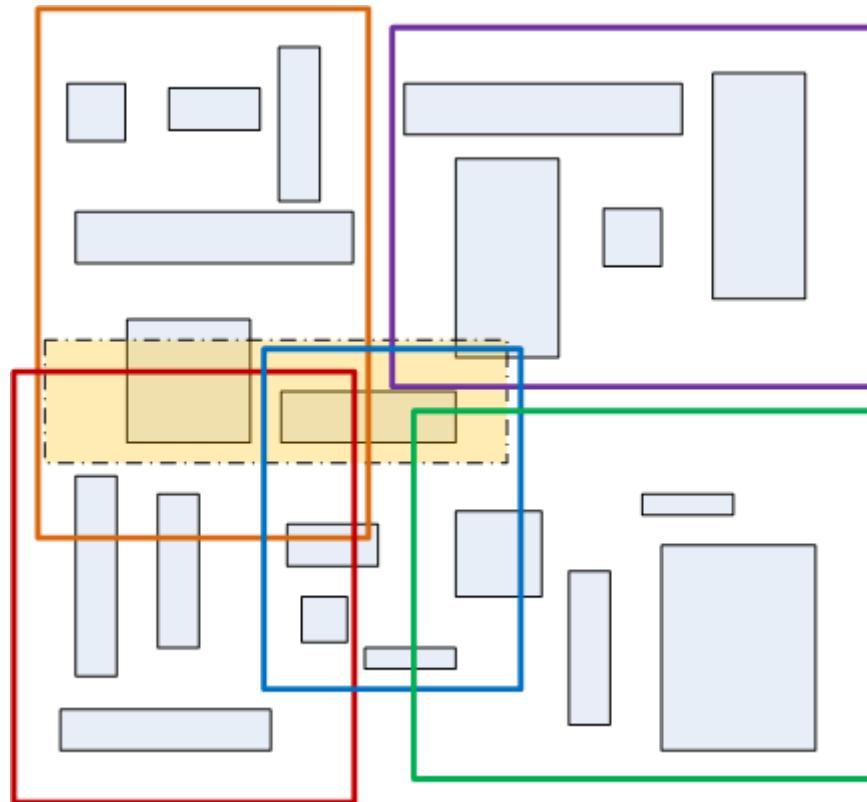
Assume the index stays in the memory.
To answer this query, **3** data blocks need
to be read in memory and **5** rectangle
comparison operations need to be
carried out to identify a subset of objects



Issues in spatial index design



If the space is segmented in this way. We need to load **5** blocks and carry out **5** rectangle comparison operations to identify a “subset” of objects



Space segmenting method (minimize the block I/O)
Index organization structure (minimize index operation)



Outline

- Indexing Motivation
- Hash Structure
 - ▶ Grid Files
- Tree Structure
- Space Filling Curve Techniques



General Hash Index Structure

A table with record, keys are letters **a** through **f**

A hash function that maps

[**a,f**] to [0,3]

$h(d) = 0$

$h(c) = h(e) = 1$

$h(b) = 2$

$h(a) = h(f) = 3$

Constant cost for point query regardless of the bucket size
No use for range query.

Bucket array of size **B**, each bucket is a block that can store records

	Block
0	d
1	c
	e
2	b
3	a
	f



Grid file

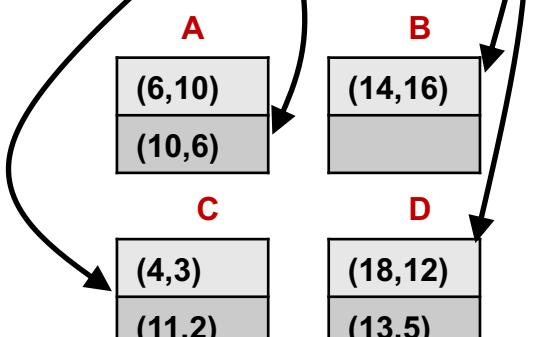
- Space segmenting method
 - ▶ Each dimension of the space is partitioned into stripes using grid lines
 - ▶ The number of grid lines in different dimensions may vary
 - ▶ The spacing between adjacent grid lines in the same dimension can be different
- Index is organized using “hash like” structure
 - ▶ Space region is like bucket in a hash table



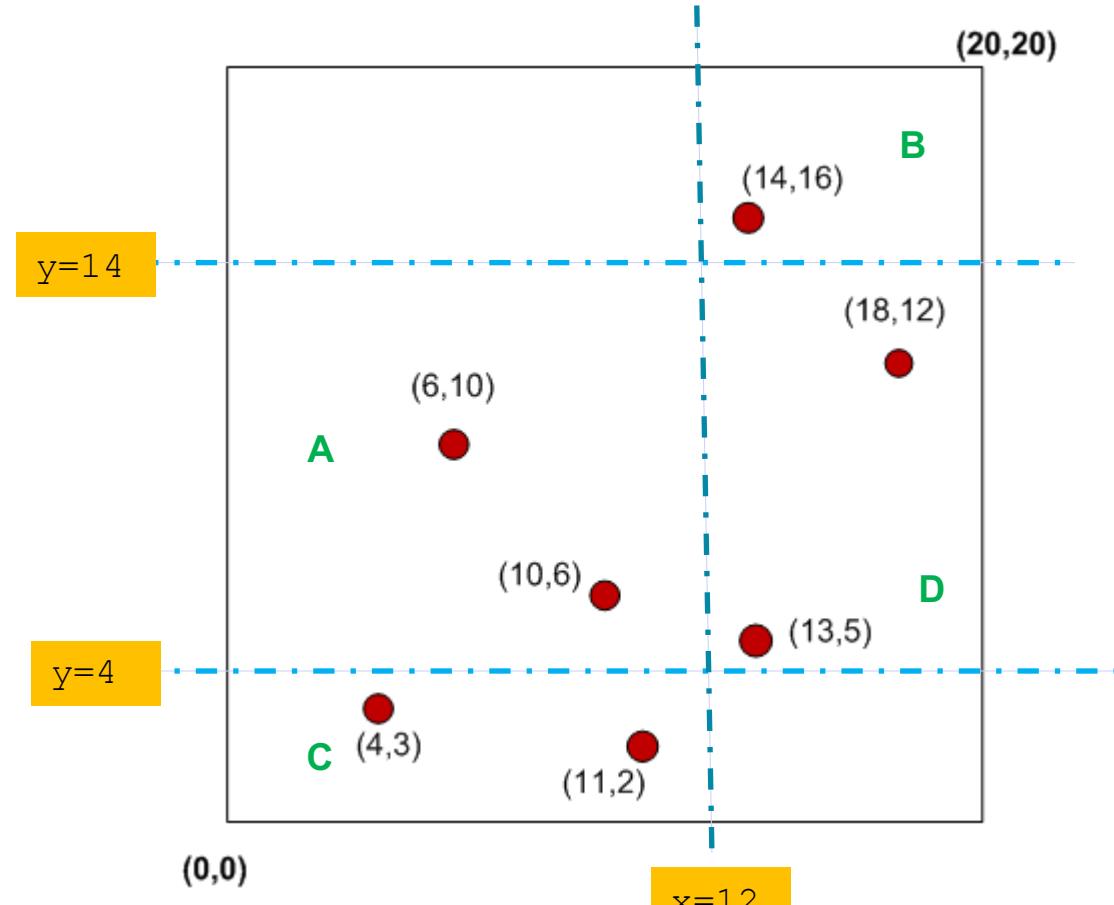
Grid File in Two Dimensional Space

Bucket array

	0-12	12-20
14-20		B
4-14	A	D
0-4	C	



disk block



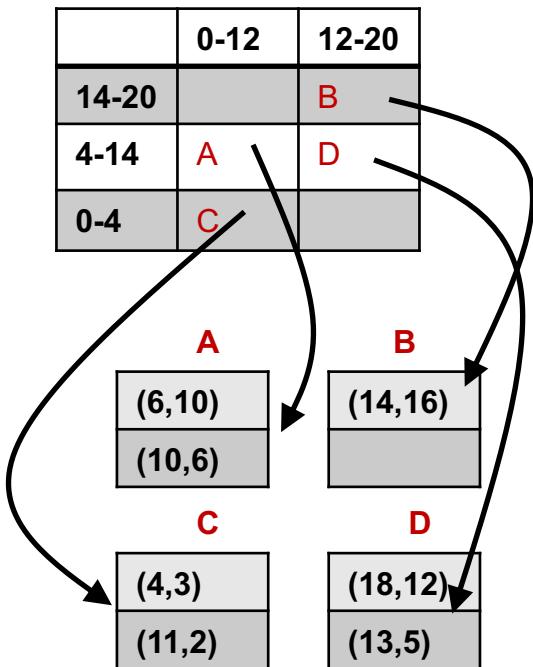
Assuming each disk block can hold up to 2 records



Index Structure

■ Grid Directory

- ▶ One dynamic d -dimensional array to store bucket location
- ▶ A set of d one-dimensional array called **linear scales** to store grid line locations



The d -dimension array for buck location :

```
[  
  [ ,B],  
  [A, D],  
  [C]  
 ]
```

The linear scales are:

X: [0,12,20]

Y: [0,4,14,20]



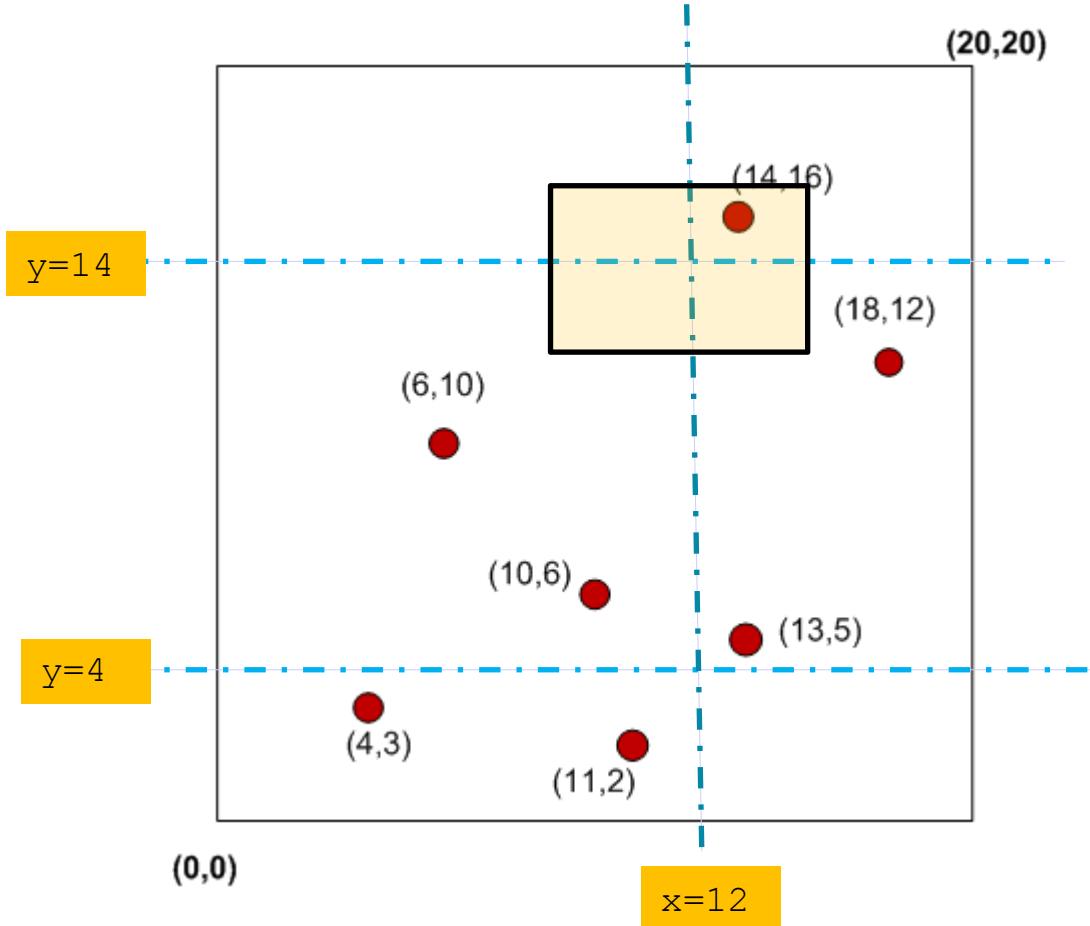
Common Queries – Point Query

- Lookup of specific point
 - ▶ Load grid directory
 - ▶ Locate the proper bucket the point might be in
 - ▶ Load the bucket



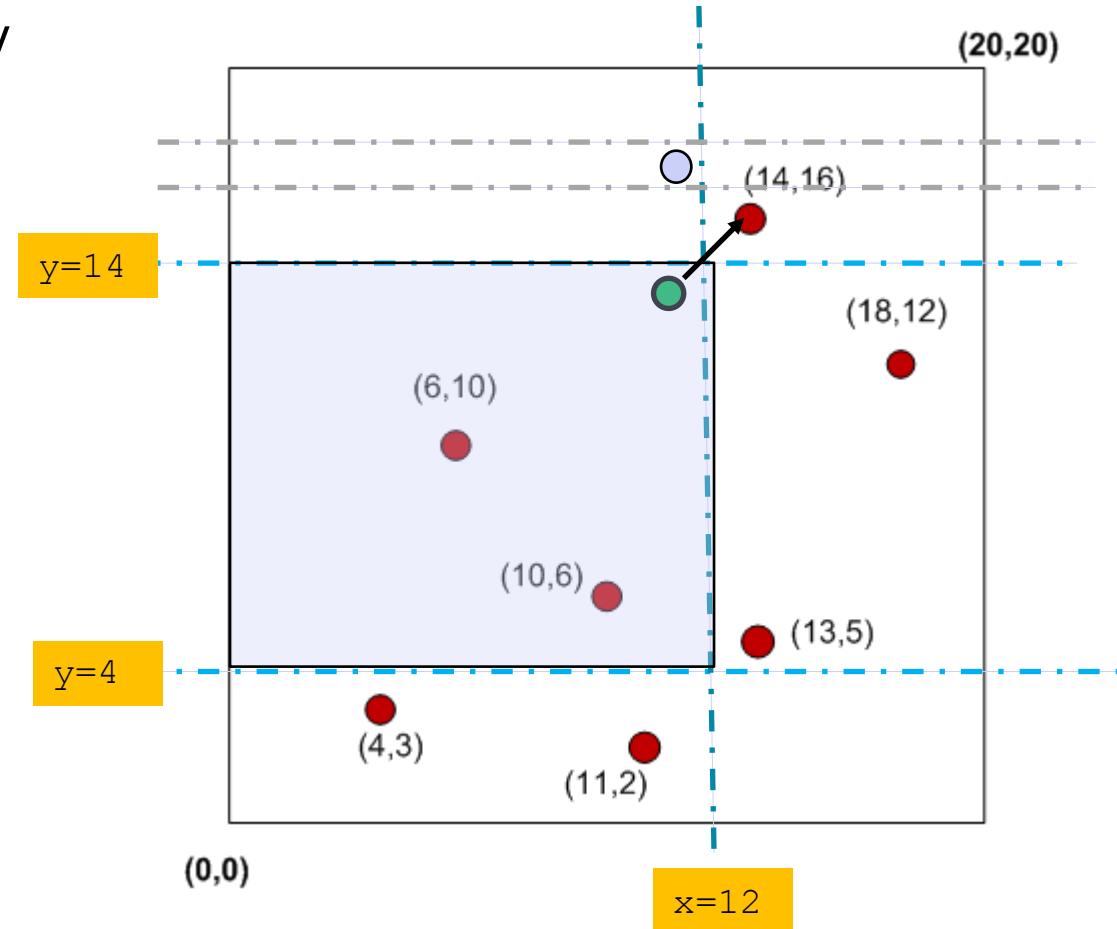
Common Queries – Range Query

- Load grid directory
- Find all buckets that interact with the query region
- Load those buckets
- May have to evaluate many candidates



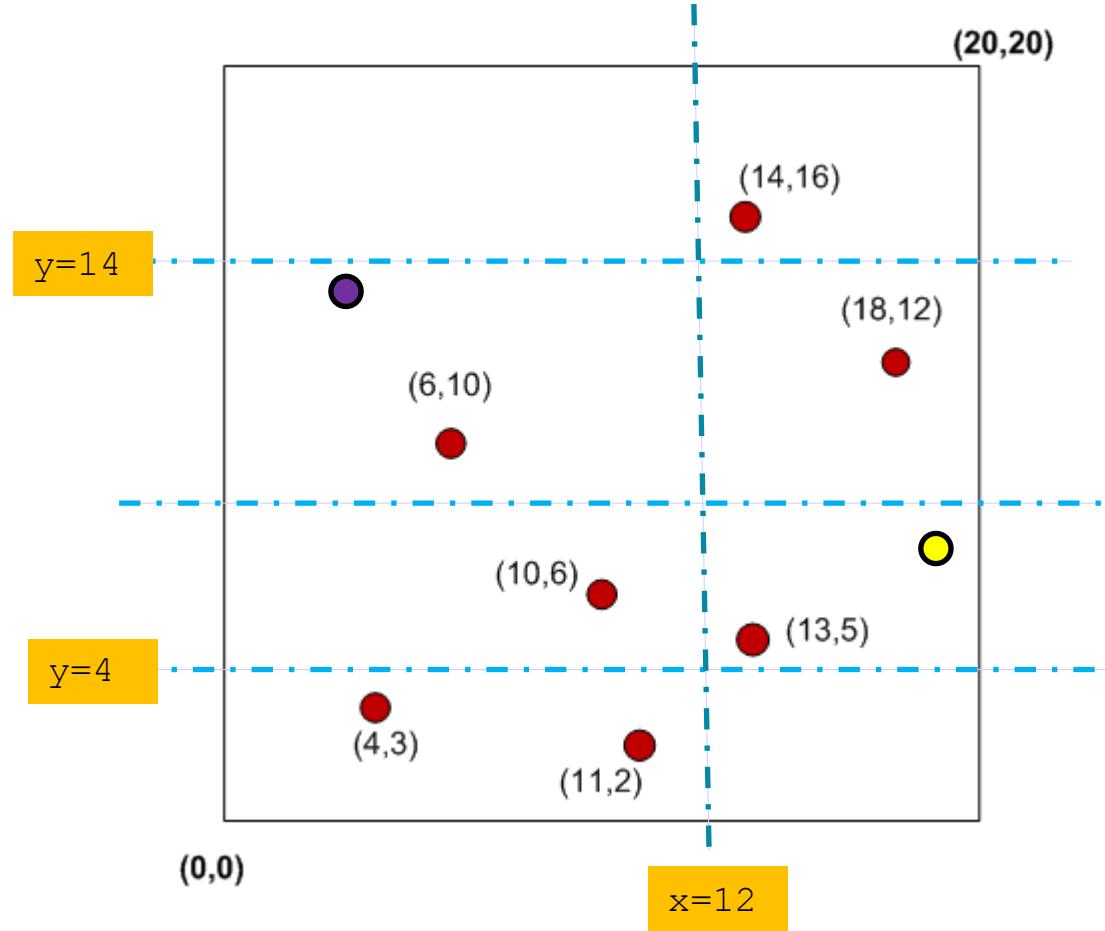
Common Queries – Nearest Neighbour

- Find the bucket the query point belongs to
- All points there would be candidate
- May have to load adjacent buckets to look for other candidates
- If the shape is much longer in one dimension than the other, it may be necessary to load buckets that are not adjacent for other candidates



Insertion into Grid Files

- Find the bucket the new point belongs to
- Add it in the bucket if there is room, otherwise
 - ▶ Add overflow block to the bucket
 - ▶ Reorganize the structure by adding or moving the grid lines



Outline

■ Indexing Motivation

■ Hash Structure

■ Tree Structure

- ▶ Kd-Tree
- ▶ Quad-Tree
- ▶ R-Tree

■ Space Filling Curve Techniques



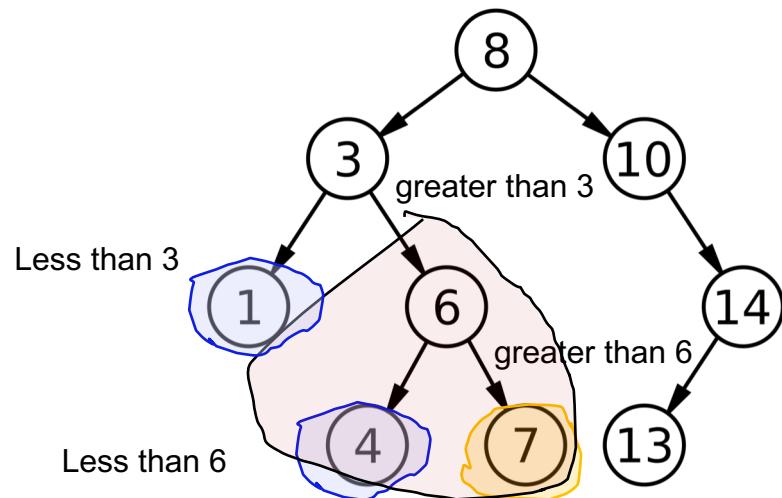
kd- Tree (K-dimensional Search Tree)

- A generalization of binary search tree to handle multidimensional data

- ▶ Main memory data structure
 - ▶ Storage based index structure

■ Binary Search Tree

- ▶ The internal node each stores a key greater than all the keys in the node's left subtree and less than those in its right subtree.



https://en.wikipedia.org/wiki/Binary_search_tree



Kd-tree

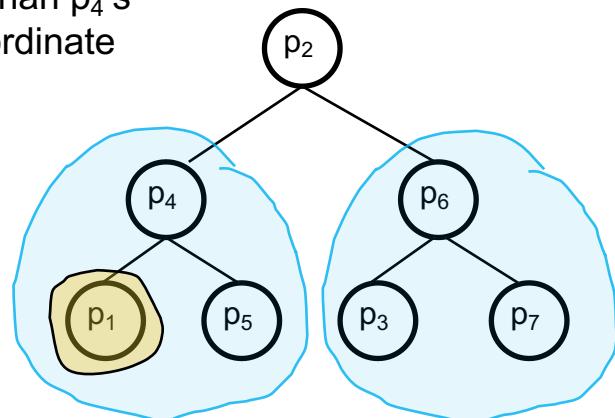
- Search key used at different level belongs to a different dimension
- In two dimension space, the x and y dimension alternates at levels
- Split the point set alternatively by x-coordinate and by y-coordinate
 - ▶ split by **x-coordinate**: split by a vertical line that has half the points left or on, and half right
 - ▶ split by **y-coordinate**: split by a horizontal line that has half the points below or on, and half above



Classic kd-tree example

- The tree nodes are the points in the data set, there is a predefined order of which dimension to use

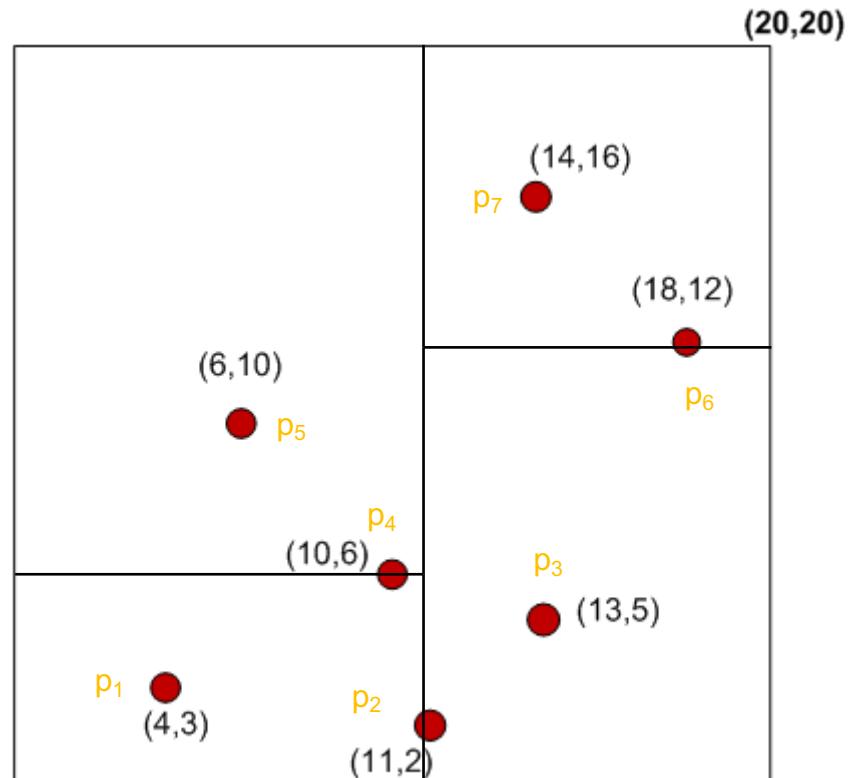
p_1 's y-coordinate is less than p_4 's y-coordinate value



All nodes have x-coordinate values less than P_2 's x-coordinate value

All nodes have x-coordinate values greater than P_2 's x-coordinate value

Used as in memory tree



(0,0)

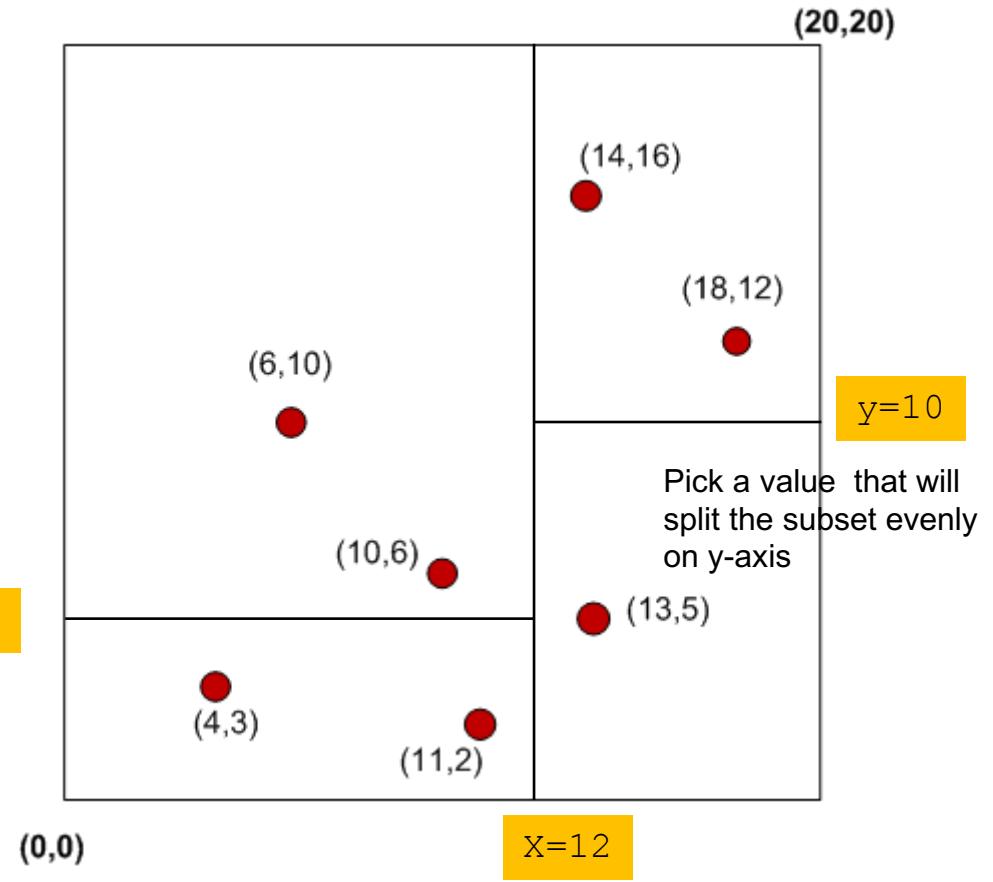
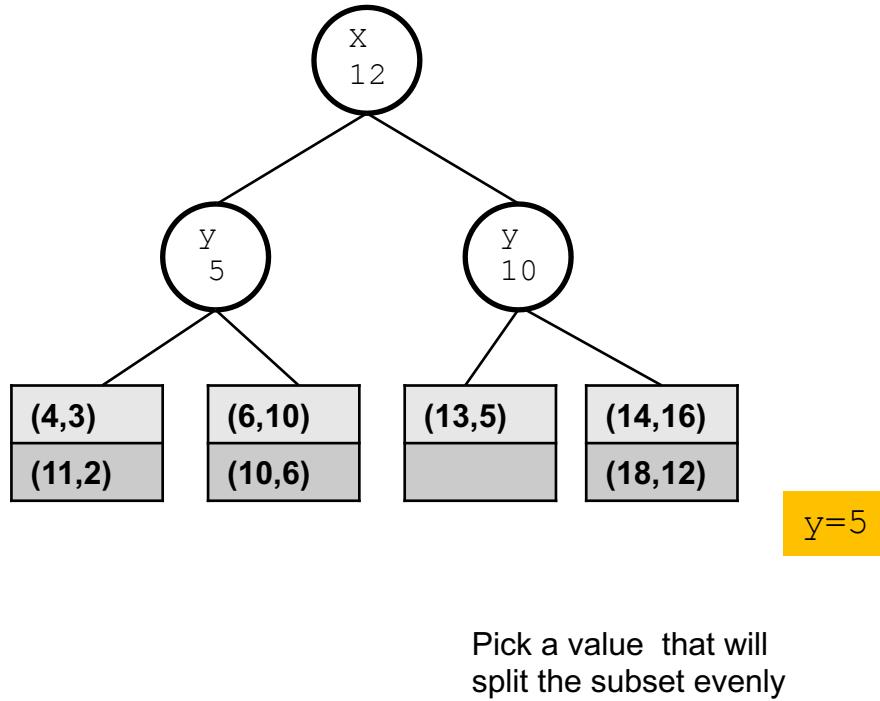
Pick a node that will split the set evenly on x-axis



Kd-tree index example

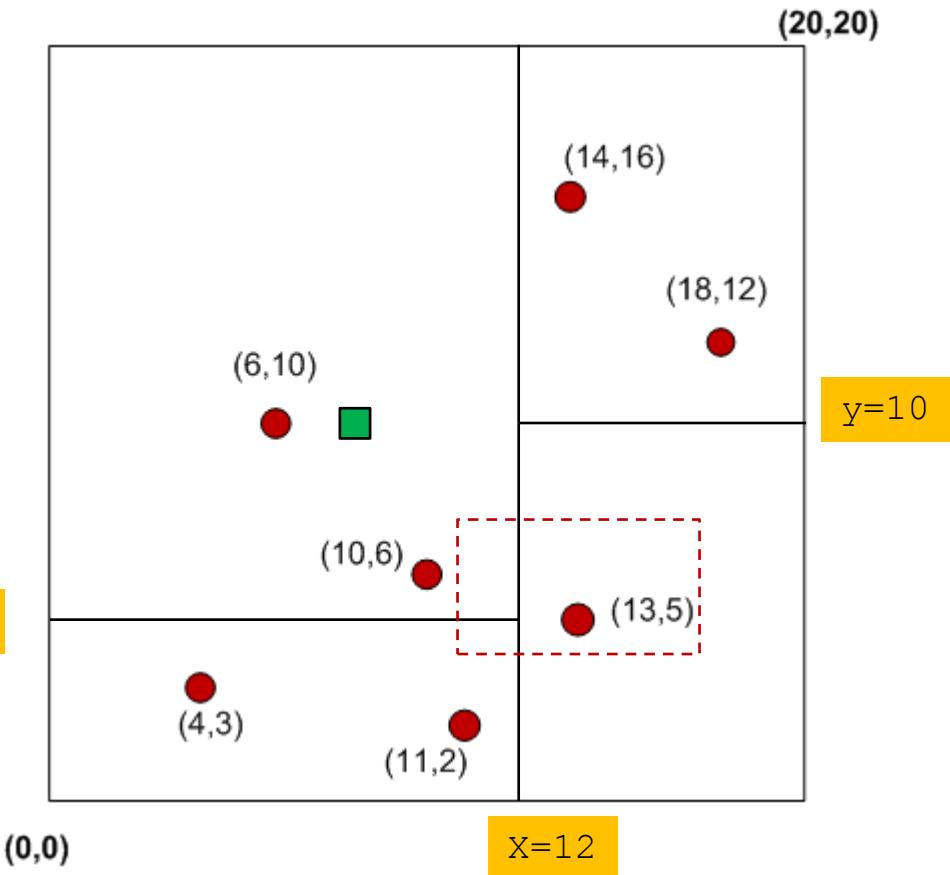
■ Modification of classic kd-tree to use as **index**

- ▶ Interior node will have only **one attribute**, a **dividing value** for that attribute, and pointers to left and right children
- ▶ Leaves will be blocks



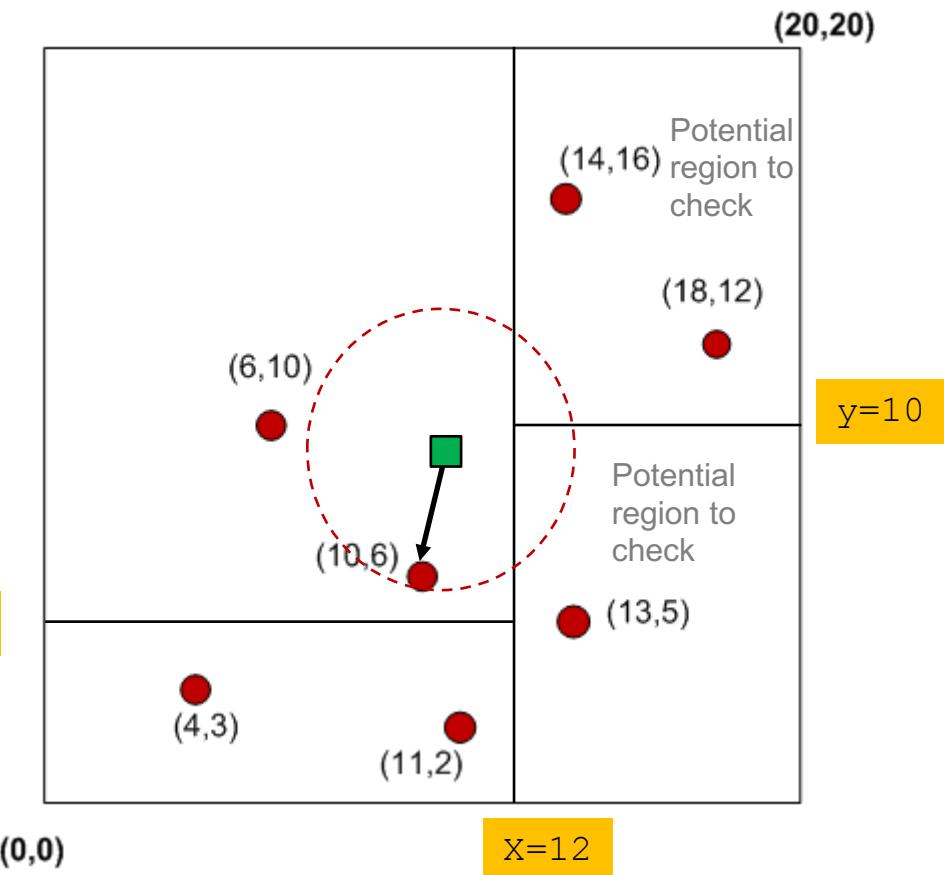
Common Queries

- Kd tree is used to store point data
- Point Query
 - ▶ Start from root, move to either the left or the right child depending on whether the query point is on the "left" or "right" side of the splitting line. Load the leaf block to check if the query point exists in the database
 - ▶ Similar process for inserting and updating points
- Range query
 - ▶ Need to check all regions intersect with the query region



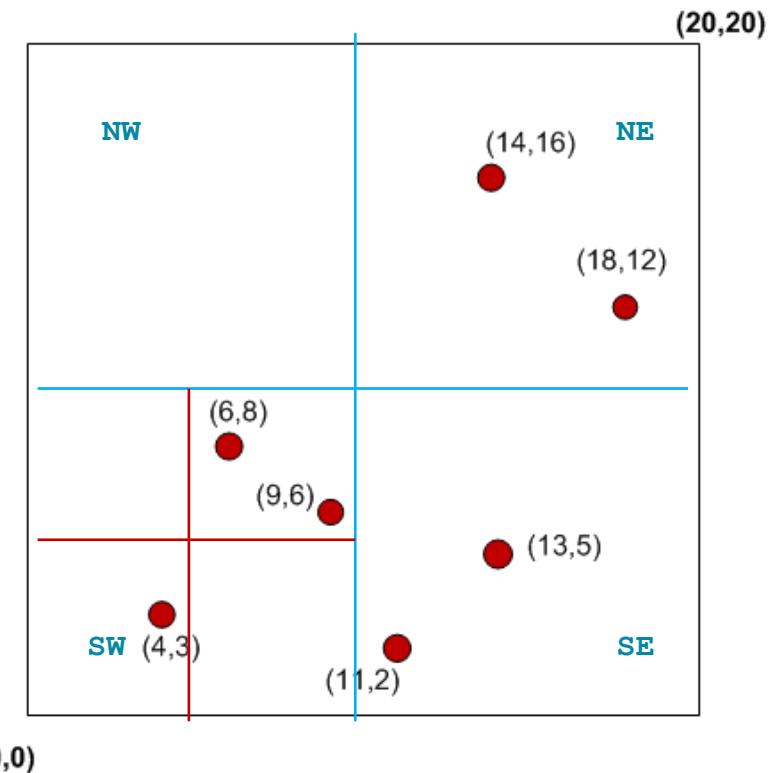
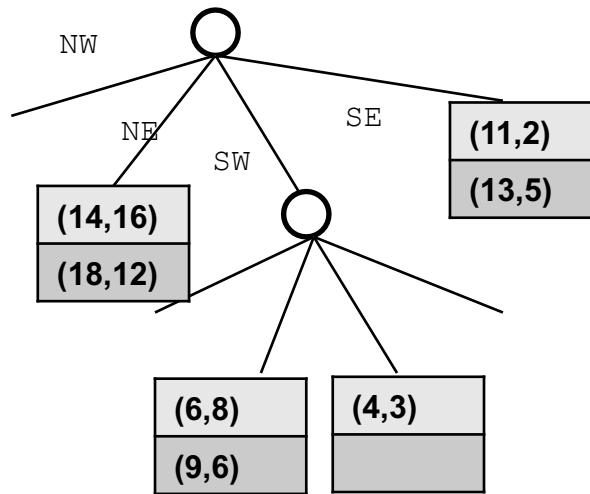
Nearest Neighbor Query

1. Use the point query process to locate the spatial segment the query node is supposed to be in.
2. Find the current nearest neighbor in this segment and the current shortest distance
3. Identify other regions that needs to check based on the current shortest distance
4. Repeat 2-3 until no further region need to be checked



Quadtree

- In a quad tree, each interior node has exactly four children representing the four quadrants (normally square or rectangle shape) of the underlying space
- The query processing is similar to kd tree



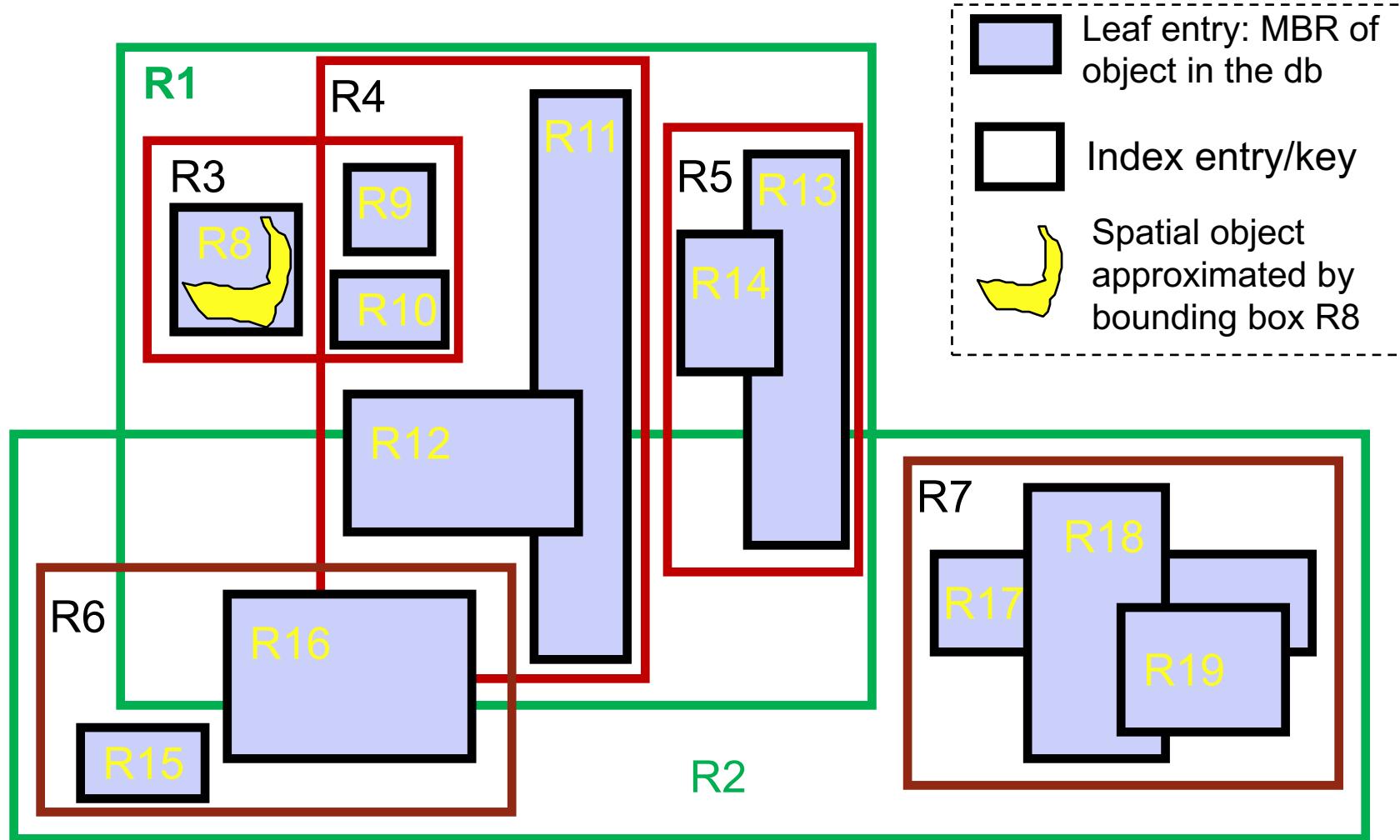
R-Tree

- R-Trees are hierarchical data structure based on B+ trees, except that it represents data in 2-dimensional regions
- Difference to B+ tree
 - ▶ Keys are minimum bounding rectangle (MBR) regions, instead of single value
 - No strict order of all keys
 - ▶ Interior node represents MBR's of its children
 - There is order of keys along a tree branch
 - ▶ Leaf node represents a number of MBRs of the spatial objects in the database
 - ▶ MBRs for different nodes may overlap
 - ▶ One MBR maybe covered by many upper level nodes' MBRs, but it can be associated with only one node

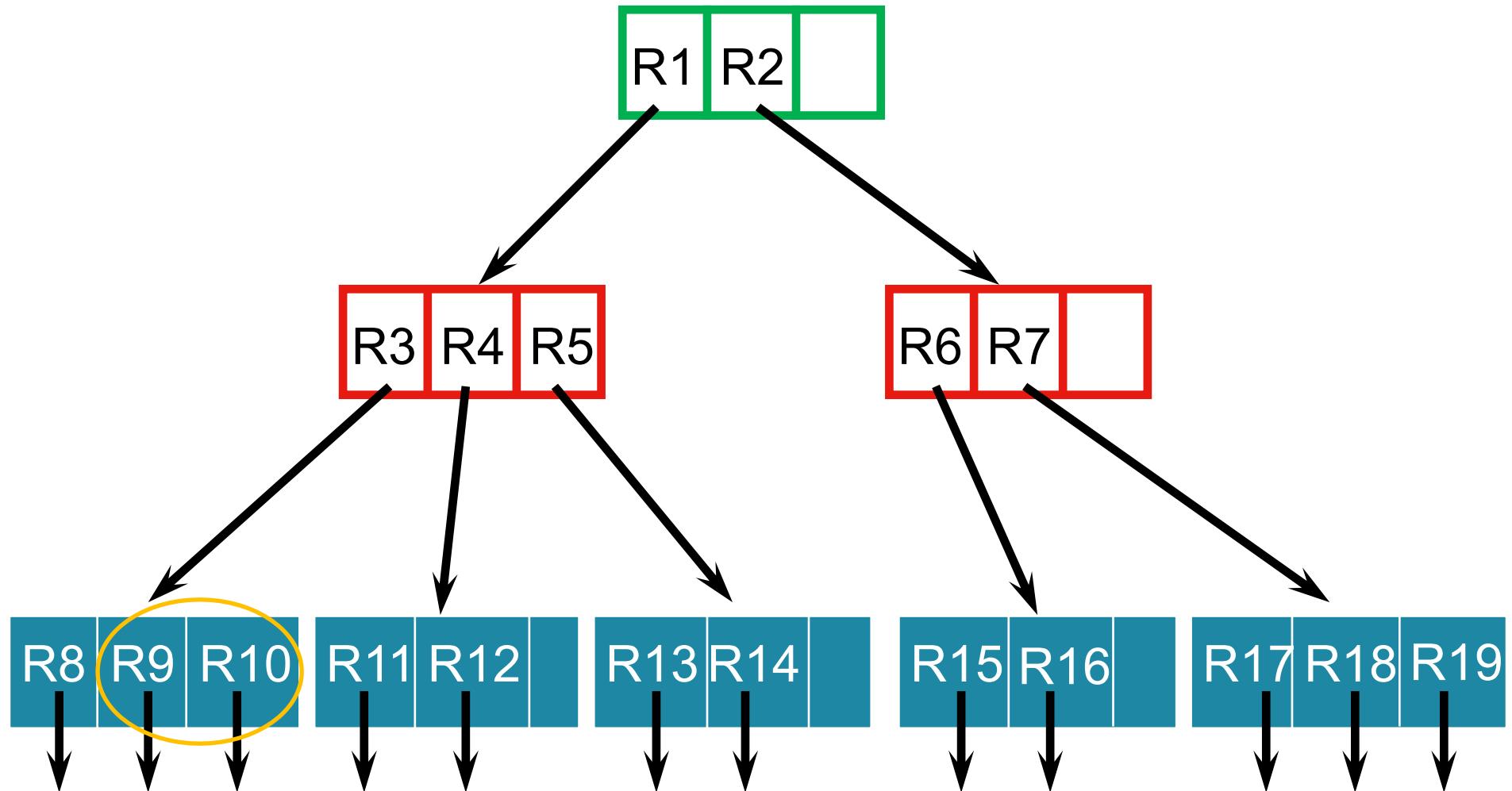


R-Tree Example -- Space

Each internal node can have maximum 3 and minimum 2 children



R-Tree Example – Index

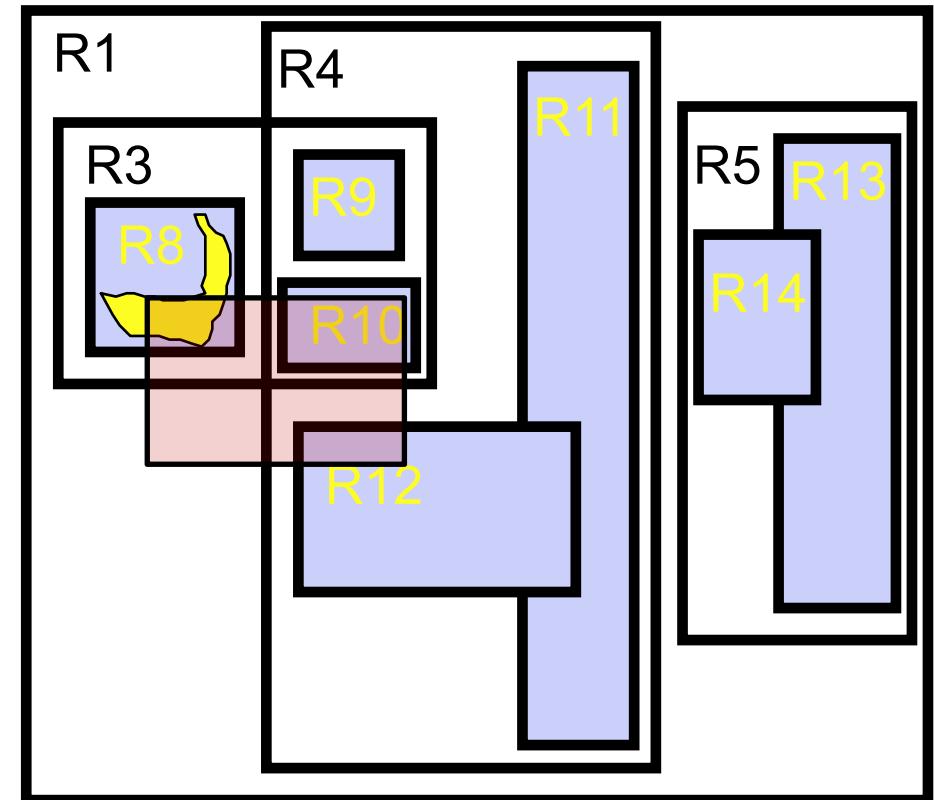
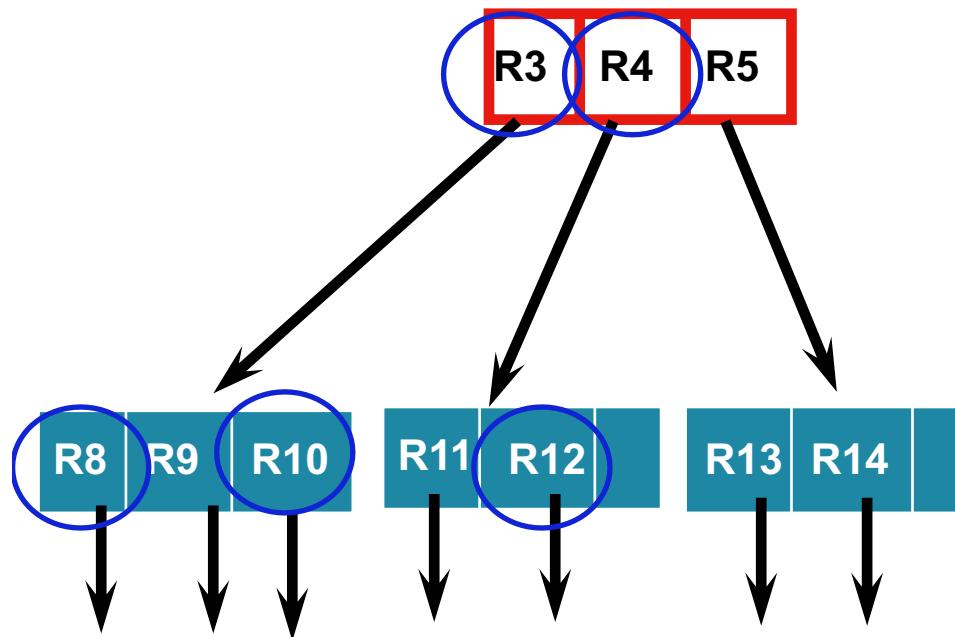


Operations on R-Tree

- Most are similar to B+ tree operation
 - ▶ A search may have to examine several siblings
- Where am I query: given a location (as a point P), find the data region or regions the point belongs to
 - ▶ Start with root
 - ▶ Find **subregions** S containing P :
 - if S is a data region: return S
 - else: recursively search S
 - ▶ If no subregion containing P
 - Stop and return that P is not in any data region



Examining multiple siblings

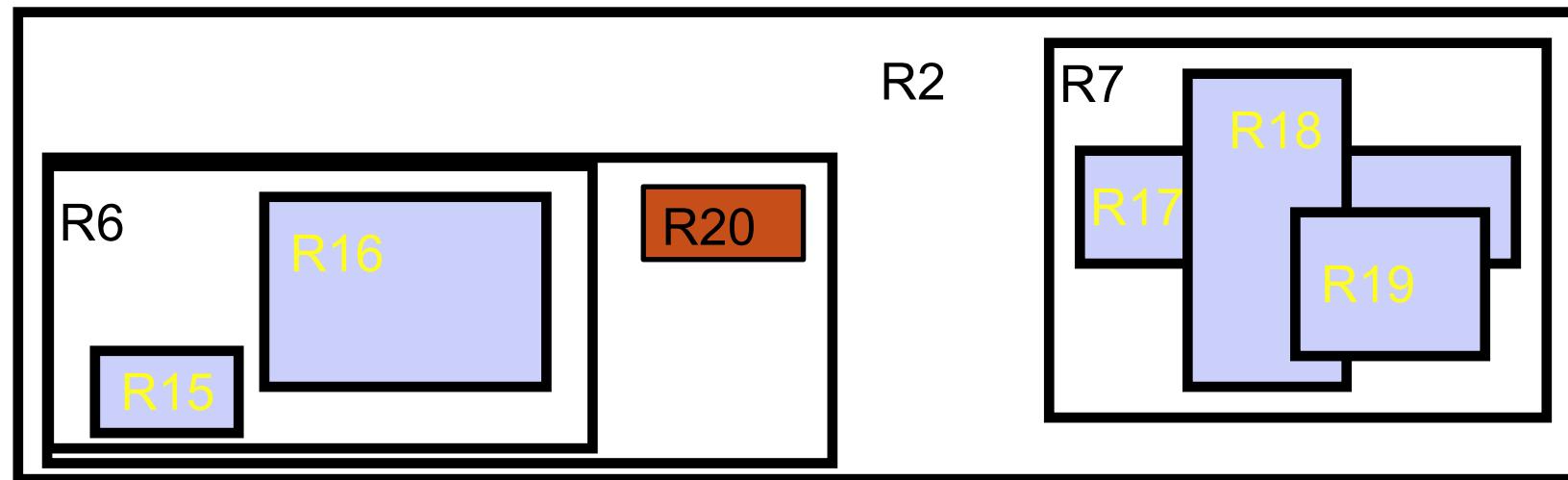


One internal node's region

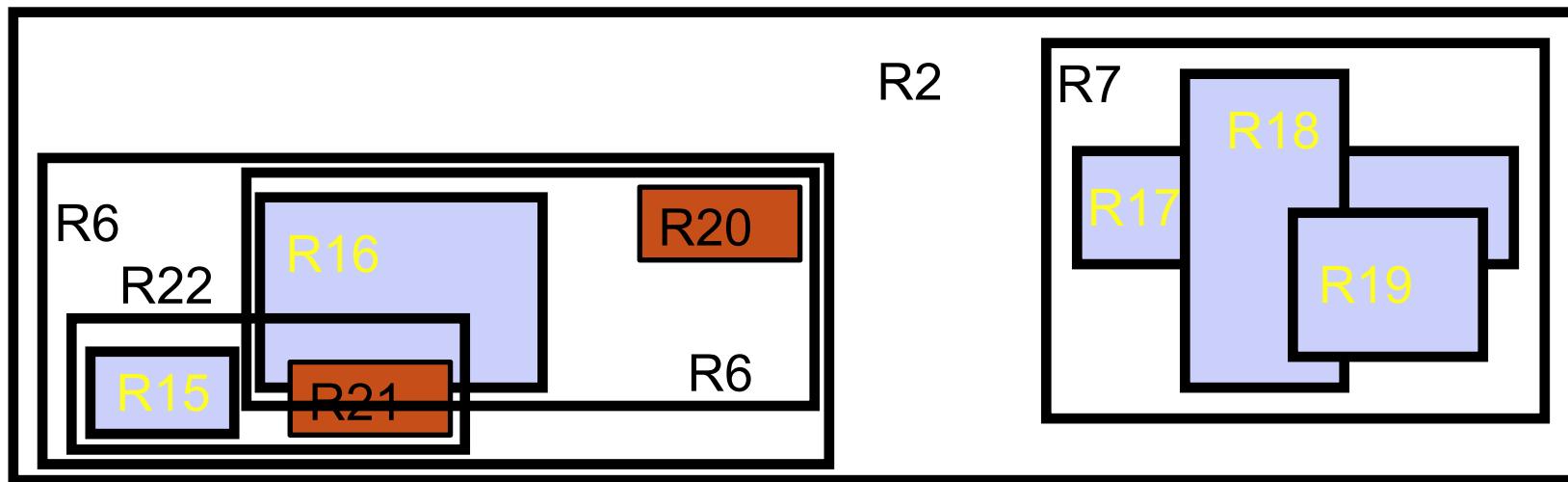
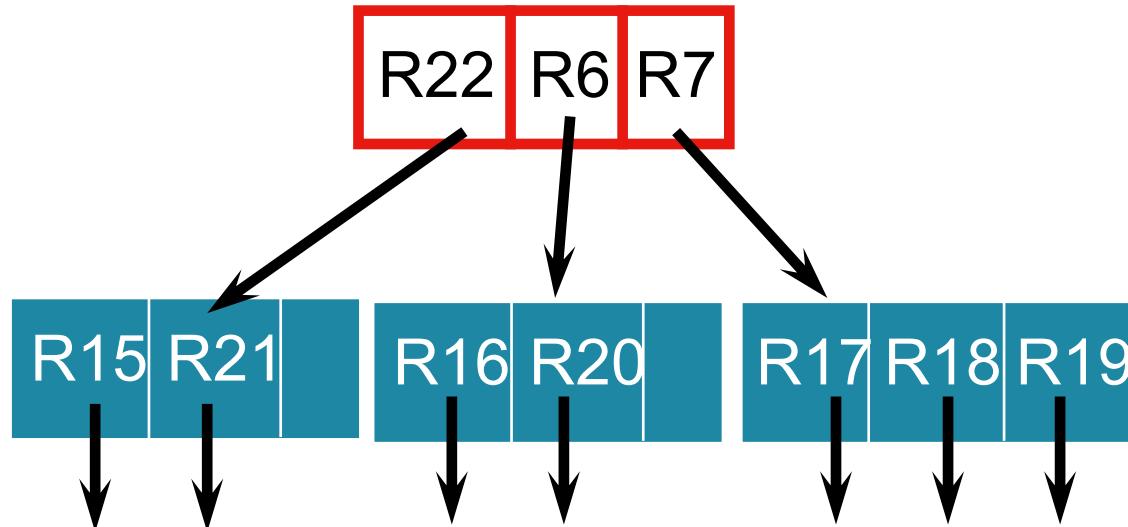


Expansion and Split Region

- Regions might be expanded or split during data insertion
- Objectives
 - ▶ minimize covered area of the containing region



Expansion and Split Region



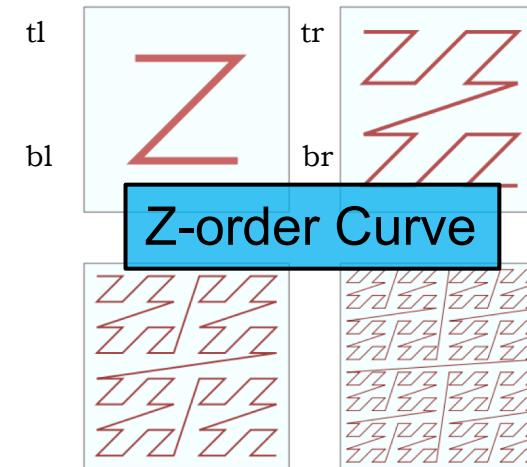
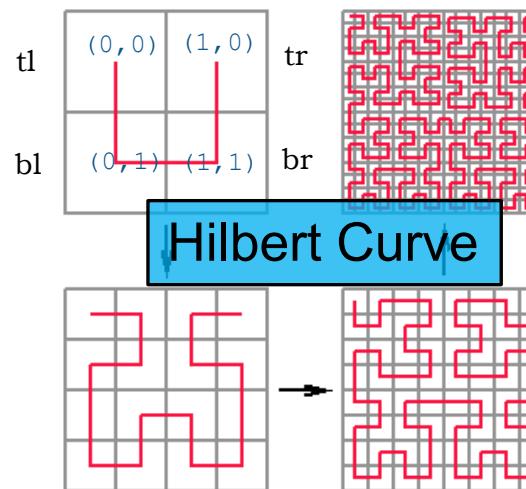
Outline

- Indexing Motivation
- Hash Structure
- Tree Structure
- Space Filling Curve Techniques
 - ▶ Z-order Curve



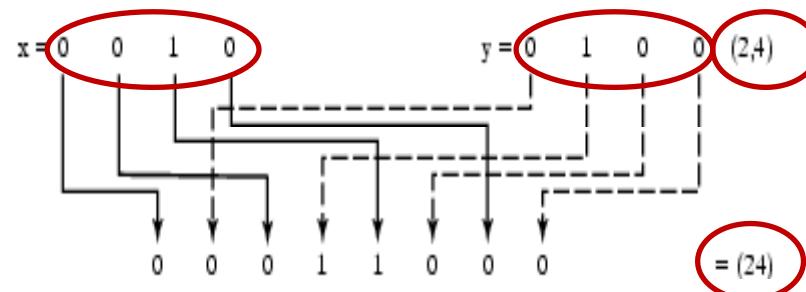
Space Filling Curve Techniques

- Techniques of encoding multidimensional data as 1 dimensional value
- Relative locality of multidimensional data are preserved in most cases
- We can think of placing all the points (regions) in space in some order, on a curve
- There are many ways of ordering points



Z-order Curve

- Also called **Morton order** or **Morton code**
- Computing the z-value or order of a point is simple if coordinate can only take integer values
 - ▶ Interleaving the binary representation of the point's coordinate values
- Not all neighbours have close z-value



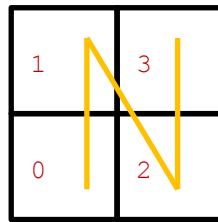
Shekhar Fig 4.6



Computing level 1 Z-value

(0, 1)	(1, 1)
(0, 0)	(1, 0)

4 points in 2 dimensional space



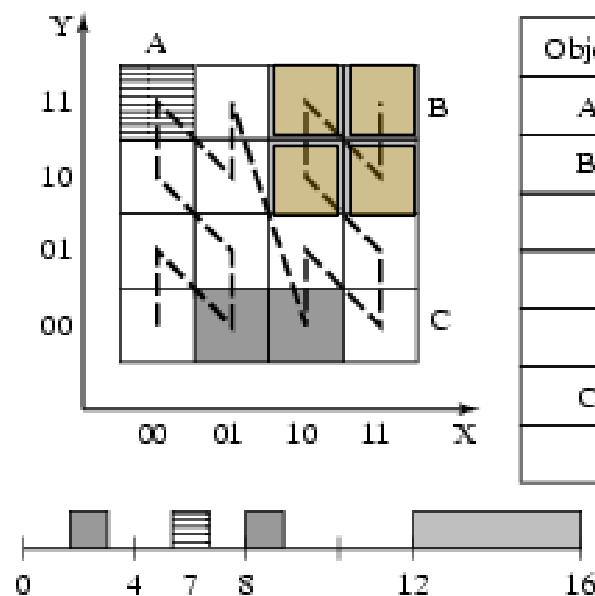
Corresponding Z-value of the 4 points

Point coordinate	Binary representation	Z-value
(0,0)	(00,00)	0000
(0,1)	(00,01)	0001
(1,0)	(01,00)	0010
(1,0)	(01,01)	0011



Example of Z-Values

- Left part shows a map with spatial object A, B, C
- Right part and Left bottom part Z-values within A, B and C
- Note C gets z-values of 2 and 8, which are not close
- Exercise: Compute z-values for B.



Object	Points	x	y	interleave	z-value
A	1	00	11	0101	5
B	1	10	10	1100	12
	2	10	11	1101	13
	3	11	10	1110	14
	4	11	11	1111	15
C	1	01	00	0010	2
	2	10	00	1000	8

Shekhar Fig 4.7



References

- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom, Database systems : the complete book (2nd edition)
 - ▶ Chapter 14
- A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. Boston, Massachusetts: ACM, 1984.
- S. Shekhar and S.Chawla: *Spatial Databases: A Tour*. Prentice Hall, 2002. [<http://www.spatial.cs.umn.edu/Book/>]
 - ▶ Chapter 4



COMP5338 – Advanced Data Models

Week 11: LSM and Google Bigtable

Dr. Ying Zhou
School of Computer Science



Outline

- Log Structured Merge Tree
- Bigtable Data model
- Bigtable Architecture

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



DB Queries

- Reading and writing data are the most fundamental functions a database should provide
- Different application domains have different requirements on read and write queries
 - ▶ The ratio of read and write queries
 - Read heavy/write heavy/balanced
 - ▶ The ratio of different types of read and write
 - Write query: ratio of insert/update/delete query
 - Read query: ratio of random point query and range query
- Different queries have different memory and disk access patterns
- There is no implementation/technique that can simultaneously optimize the performance of all types of queries



DB Query and Performance

- Read vs Write Query
 - ▶ E.g Index speeds up read query but slows down write query, so we only build index selectively
- Different Read Queries
 - ▶ For point query, hash index provides better performance than tree based index
 - ▶ For range query, hash index is useless
- Early database systems try to provide a general solution to relatively balanced query workload
- There are many new systems specialized for a particular workload type



Write Heavy Workload

- Many systems do not maintain traditional business data and do not need transactional processing
 - ▶ Business data domains: bank transaction, airline reservation, course enrolment, library record, etc
 - ▶ Other domains: system monitoring data, scientific data collected by sensors, user activity data collected by system, etc
- Many application domains require support for large data size, very high write throughput and relatively simple read requests
 - ▶ Data are collected and inserted by some application in contrast to initiated by end user transactions
 - Mostly append(insert) type of write
 - ▶ Data are analyzed in batch to discover patterns or to make predictions
 - Mostly sequential read



Google Search Engine Data

- A whole copy of the web collected using crawler and done periodically
- Typical features of the data
 - ▶ Large data size
 - ▶ Frequently inserted into the system
 - ▶ Rarely deleted
 - ▶ Scanned to build inverted index (word -> page)
 - ▶ Page meta data such as links between pages need to be used frequently to compute PageRank score as part of the ranking indicator
- Solution:
 - ▶ Build **Bigtable** cluster to store web data



Facebook System Monitoring data

- System measuring data points like *CPU load, error rate, latency*, etc generated by “thousands of individual systems running on many thousands of machines, often across multiple geo-replicated data centers”
- Run real time queries to identify and diagnose problems as they arise.
- Very high write throughput
- Relatively simple read query that usually scan a range of recent data points
- Solution:
 - ▶ Previous: A time Series Database(TSDB) build on top of HBase (the open source version of **Bigtable**)
 - ▶ Now: An in-memory TSDB called Gorilla for recent data (26 hours)



Log Structured Merge Tree: Motivation

- Many disk based database systems designed for such write heavy workload (most of them are TSDB) use storage engine based on LSM tree
- LSM was initially proposed in 1996 by Patrick O'Neil et.al.
- The motivation is to provide efficient query on logs of long running transactions
 - ▶ Logs are append-only files in time order
 - ▶ Querying logs for early transaction events on certain attribute is not efficient
 - ▶ There is a need for indexed logs
 - ▶ Tree structure (B-tree structure) is the most common index structure



LSM Tree: general solution

■ Design principles

- ▶ Memory access is much faster than disk access
- ▶ Memory space is much smaller than disk space
- ▶ At disk level, appending to a file is faster than randomly updating a file

■ General solution path

- ▶ Maintain several levels of indexed data (tree) at different storage levels
- ▶ Periodically merge and sort data at different levels
- ▶ Files are only appended (similar to log writing)

■ A typical two-level solution

- ▶ Maintain latest entries in memory (C_0), organized as tree structure, for easy query
- ▶ When memory threshold is reached, migrate the in memory data structure to disk as a new file, maintaining the tree structure in the file (C_1)
 - The file is indexed
- ▶ Periodically sort merge the files to create large files while still maintaining the indexed structure (C_1)



LSM Tree: standard implementation

- The 1996 paper gave general solution path but not textbook implementation
- Many terminologies later associated with LSM are proposed in Google's Bigtable paper
 - ▶ Memtable, SSTable, Compaction, etc
- Bigtable paper also proposed a detailed enough implementation that are used in many other systems
 - ▶ HBase, Cassandra, LevelDB, RockDB, InfluxDB, etc.
 - ▶ Most of the systems support key based query and may operate in pure key-value model, wide column model or time series model.
 - ▶ Tree structure in memory and disk is just one type of index structure; Bigtable use sorted map instead.



Outline

- Log Structured Merge Tree
- Bigtable Data model
- Bigtable Architecture



Data Model

- Bigtable is generally classified as wide-column data model
- “A Bigtable is a sparse, distributed, persistent multidimensional sorted map”
- Basic concepts: table, row, **column family**, column, timestamp
 - ▶ (rowkey:**string**, columnKey:**string**, timestamp:**int64**) -> value: **string**
- Example Bigtable to store web pages
 - ▶ Stores the data about home page of *cnn* website
 - The URL is “www.cnn.com”
 - The language is “EN”
 - The content is “<html> ...</html>”
 - It is referenced by two other pages
 - Sports Illustrated (cnnsi.com) , using an anchor text “CNN”
 - My-Look (my.look.ca), using an anchor text “CNN.com”



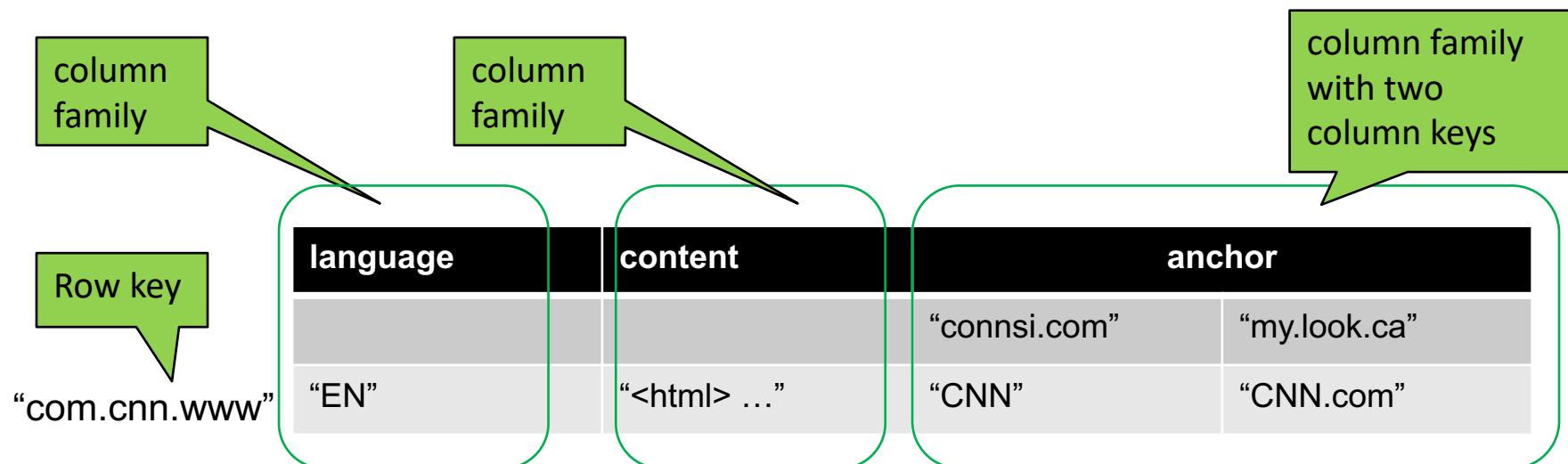
Relational Data Model vs Bigtable Model

web table

url	language	content
“www.cnn.com”	“EN”	“<html> ... </html>”

link table

url	referencingUrl	anchorText
“www.cnn.com”	“connsi.com”	“CNN”
“www.cnn.com”	“my.look.ca”	“CNN.com”



Rows

sorted

Using reversed URL to ensure similar web page would be put in neighboring rows

“com.cnn.www”
“com.cnn.www/WORLD”
“com.cnn.weather”
“com.cts.www”

language	content	anchor

- Row keys are arbitrary strings
- Row keys are sorted in lexicographic order
- Large table is dynamically partitioned by row key ranges
 - ▶ Row key is partition (sharding) key
 - ▶ Each partition is called a **tablet**, and is served by a server
 - ▶ Nearby rows will usually be served by the same server
 - ▶ Accessing nearby rows requires communication with a small number of machines



Table Splitting

- A table starts as one tablet
- As it grows it splits into multiple tablets
 - ▶ Approximate size: 100-200 MB per tablet by default

language	content	anchor
“com.cnn.www”		
“com.cnn.www/WORLD”		
“com.cnn.weather”		
“com.cts.www”		

One tablet



Table Splitting (cont'd)

sorted

“com.cnn.www”
“com.cnn.www/WORLD”
“com.cnn.weather”
“com.cts.www”

last key

language	content	anchor

“com.nytimes.www”
“com.seattletimes.www”
“com.washingtonpost.www”
“com.zdnet.www”

last key

language	content	anchor



Columns and Column Families

- Relational model only has “row” and “column” concepts
- Bigtable has “row”, “column” and “column family” concepts
- Column family
 - ▶ Just a group of columns with a **printable name**
 - ▶ Each **column** inside a **column family** has a **column key**
 - Column key is named as **family:qualifier**
- Column family can be viewed is a convenient way to store “collection” type data at design level
- Data stored in a column family is usually of the same type



Columns and Column Families (cont'd)

■ Column Family is part of the **schema definition**

- ▶ When we create a table, we also create a few column families by specifying their names
- ▶ The number of column families in a table is typically small and relatively stable
 - Less than hundred
- ▶ A column family theoretically can have unlimited number of columns
 - The row could be very “wide” with many columns
 - Wide-column store
 - E.g. a popular web page in the web table may be referenced by thousands, or even millions of other pages
 - *Implications:* we may have some tablet storing only one row!



Column Family Examples

- The web table example has three column families
 - ▶ “language” -- with only one column to store a web page’s language
 - Each web page can only have one language
 - Just like a normal column in relational table
 - Column key is “language:”
 - ▶ “content” -- again with only one column to store the actual HTML text
 - Column key is “content:”
 - ▶ “anchor” -- with dynamic number of columns
 - Each web page may be referenced by different number of other pages
 - E.g. www.cnn.com page has two referencing sites
 - Column key is “anchor:<referencing site url>”



Timestamps

- Classic relational model can only store the “current” value of a particular row and its columns
- Bigtable stores multiple versions of a column by design
- Version is indexed by a 64-bit timestamp
 - ▶ System time or assigned by client
 - ▶ If system time is used, this is equivalent to transaction time
 - ▶ Client assigned time can have various meanings
- Per-column-family settings for garbage collection
 - ▶ Keep only latest **n** versions
 - ▶ Or keep only versions written since time **t**
- Retrieve most recent version if no version specified
 - ▶ If specified, return version where timestamp \leq requested time



Web Table with Timestamp

language	content	anchor	
		“connsi.com”	“my.look.ca”
“com.cnn.www”	<div style="display: flex; align-items: center;"><div style="border: 1px solid black; padding: 2px;">“EN”</div><div style="margin: 0 10px;">← t1</div><div style="border: 1px solid black; padding: 2px; background-color: #f0f0f0; margin-right: 10px;">“<html>...”</div><div style="border: 1px solid black; padding: 2px; background-color: #f0f0f0; margin-right: 10px;">“<html>...”</div><div style="border: 1px solid black; padding: 2px; background-color: #f0f0f0; margin-right: 10px;">“<html>...”</div><div style="margin-left: auto;"><div style="border: 1px solid black; padding: 2px;">“CNN”</div><div style="margin: 0 10px;">← t9</div><div style="border: 1px solid black; padding: 2px;">“CNN.com”</div><div style="margin-left: 10px;">← t7</div></div></div>	“connsi.com”	“my.look.ca”

■ The multidimensional sorted map concept

- ▶ **(rowkey:string, columnKey:string, timestamp:int64) -> value: string**
- ▶ Examples:
 - (“com.cnn.www”, “anchor:consi.com”, t9) -> “CNN”
 - (“com.cnn.www”, “language:”, t1) -> “EN”



Typical APIs

■ Data definition API

- ▶ Create/delete table and column families
- ▶ Update table/column family metadata

■ Data Manipulation API

- ▶ Write or delete value as specified by rowkey and some column qualifier
- ▶ Look up specific row by row key
- ▶ Scan a short range of rows
- ▶ Support single row transaction



Outline

- Log Structured Merge Tree
- Bigtable Data model
- **Bigtable Architecture**
 - ▶ Immutable SSTable file
 - ▶ Master-Tablet Server Architecture
 - ▶ Chubby Services
 - ▶ Tablet Representation and Write/Read Path

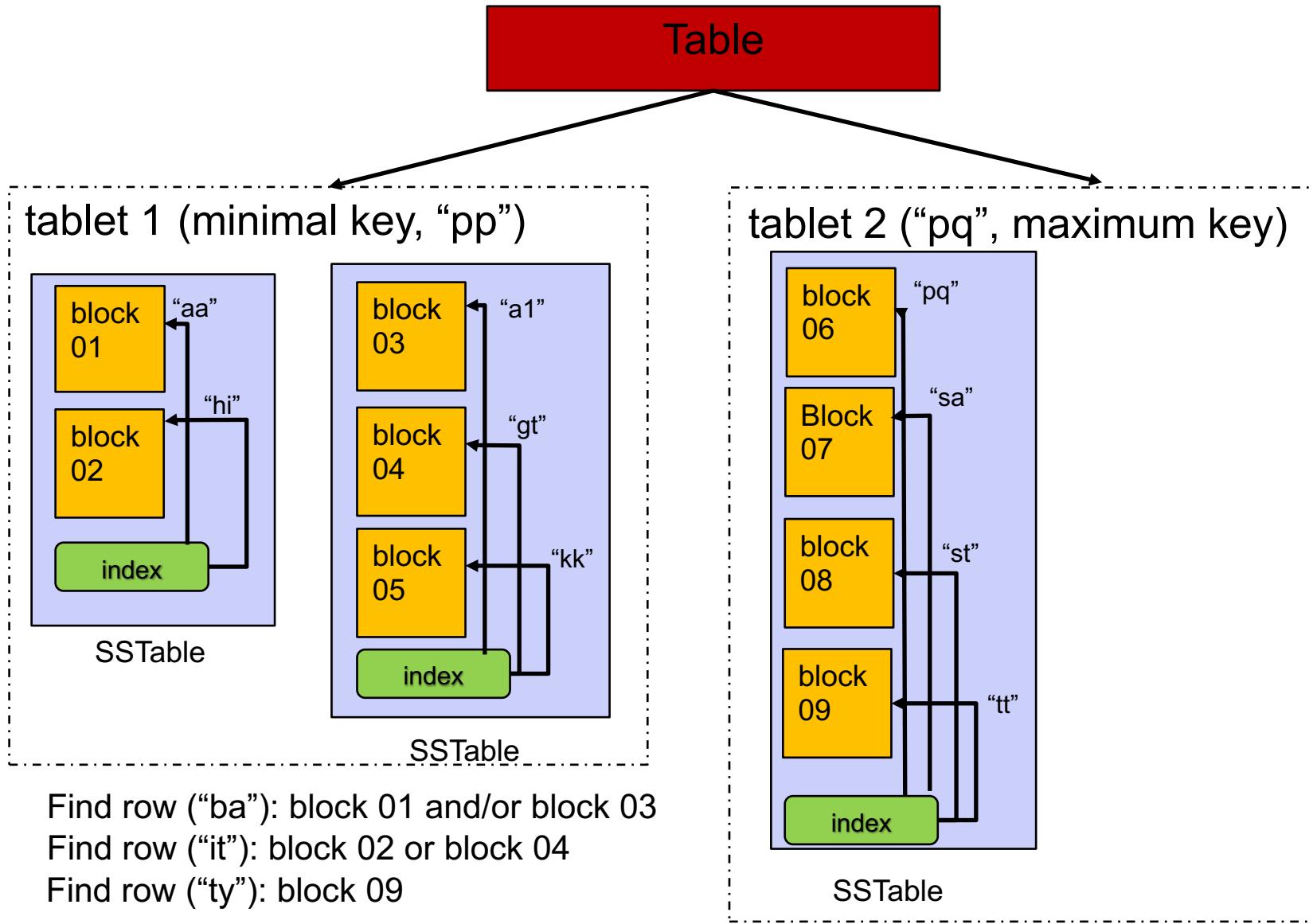


Data Storage

- Google File System (GFS)
 - ▶ Is used to store actual Bigtable data (log and data files)
 - ▶ It provides replication/fault tolerance and other useful features in a cluster environment
- Google SSTable file format
 - ▶ Bigtable data are stored internally as SSTable format
 - Sorted String Table
 - ▶ Each SSTable consists of
 - Blocks (default 64KB size) to store **ordered immutable** map of key value pairs
 - Block index
- The SSTable is stored as GFS files and are replicated



Table-Tablet-SSTable



Architecture

■ Many *tablet servers*

- ▶ Can be added or removed dynamically
- ▶ Each manages a set of tablets (typically 10-1,000 tablets/server)
- ▶ Handles **read/write** requests to tablets
- ▶ Splits tablets when too large

■ One *master server*

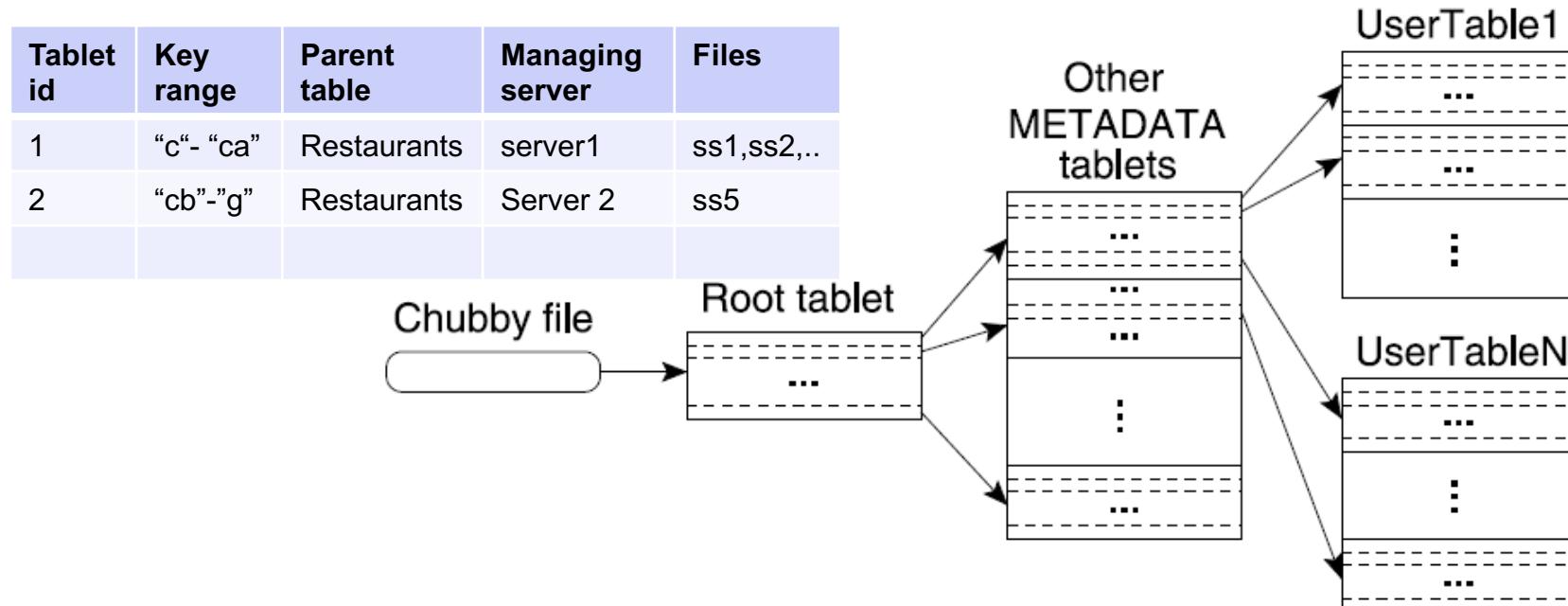
- ▶ Assigns tablets to tablet server
- ▶ Balances tablet server load
- ▶ Garbage collection of unneeded files
- ▶ Schema changes (table & column family creation)
- ▶ It is NOT in the read/write path

■ Client library



Tablet Location

- METADATA table contains the location of all tablets in the cluster
 - ▶ It might be very big and split into many tablets
- The location of METADATA tablets is kept in a root tablet
 - ▶ This can never be split and its location is stored in Chubby
- Each tablet is assigned to be managed by **ONE** tablet server at a time.
- Both ROOT and METADATA tablets are managed by tablet servers as well



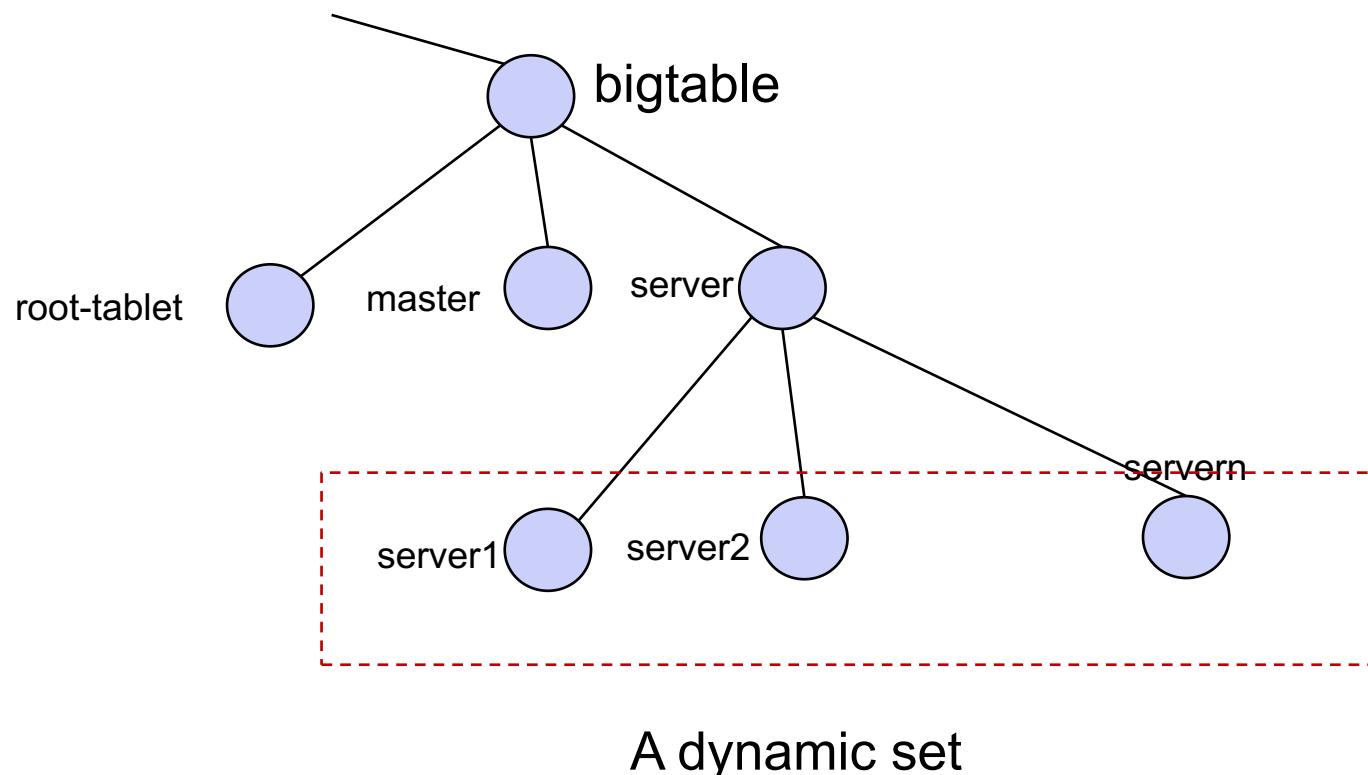
Chubby Services

- Chubby is distributed lock service consists of a small number of nodes (~5)
 - ▶ Each is a replica of one another
 - ▶ One is acting as the master
 - ▶ Paxos is used to ensure majority of the nodes have the latest data
- Usage in Bigtable
 - ▶ Ensure there is only one master
 - ▶ Keep track of all tablet servers
 - ▶ Stores the root table location
 - ▶ If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.



Chubby Bigtable File Hierarchy Example

- Chubby exports UNIX file system like APIs.
- It allows clients to create directory/file (node) and locks on them
 - ▶ Lock has short lease time and needs to be renewed periodically



Chubby and Tablet Servers

- Tablet servers are able to join or leave a running cluster without interfering the normal cluster operation
- Chubby is used to keep track of tablet servers
- Normal handling
 - ▶ Each server creates & locks a unique file in Server Directory when it starts
 - ▶ The lock has short lease and needs to be renewed periodically
 - ▶ If a tablet server is scheduled to leave the cluster, it will release its lock
- Error handling
 - ▶ A tablet server may lose the lock (e.g. expires)
 - It will stop serving the tablets
 - It will report to master that the lock is lost
 - It will attempt to reclaim the lock if the file still exists, otherwise it kills itself
 - ▶ A tablet server may crash and its file become orphaned
 - Master will come to the rescue



Chubby and Master Operation

- Master also obtains an *exclusive master lock* from chubby to ensure there is only one master server
- Master monitors Chubby's server directory to find the current list of tablet servers in the cluster
- Master detects the status of tablet servers by periodically asking each server for the status of its lock
- Error handling
 - ▶ If tablet server is alive but has no lock or if the tablet server is unreachable
 - The master will contact Chubby to acquire a lock on the orphaned server file and delete it
 - The master also assigns all tablets to other servers
 - ▶ If a master cannot contact Chubby to renew its lock, it kills itself



Master Start Up

■ When a master is started

1. It grabs a unique master lock in Chubby
2. Find out all live servers
3. Communicate with all servers to find out what tablets they serve
4. Scan the METADATA table to find the total set of tablets in the cluster
 - May discover tablets that are not assigned
5. Assign tablets without a server to a new tablet server

■ Any cluster has a **root** tablet, in step 3, the master may

- ▶ Find the server that manages the **root** tablet and proceed with step 4
- ▶ Find that the **root** tablet is not assigned to any server, the master will assign it to a server and proceed with step 4



Assigning Tablet to Tablet Server

- Scenarios that will trigger tablet assignment
- During start up
 - ▶ Master assign tablets to servers to balance the load
- When data changes
 - ▶ Tables are created or deleted (master initiates)
 - ▶ Two tablets are merged to form one (master initiates)
 - ▶ An existing tablet is split into two smaller ones (tablet server initiates)
- When a tablet server is down
 - ▶ The tablets it manages need to be assigned to other tablet servers
- When a new tablet server joins
 - ▶ The master needs to allocate tablets to it.



Assigning Tablet to Tablet Server (cont'd)

- The assignment is initiated by master sending a **load tablet** request to a tablet server.
- Upon receiving such request, a tablet server performs the following:
 - ▶ Scan the METADATA table to find information about this tablet
 - List of SSTable files
 - Log file
 - ▶ Read the block indexes in memory
 - ▶ Play the log file to reconstruct the memory with all updates are not yet persisted in SSTables



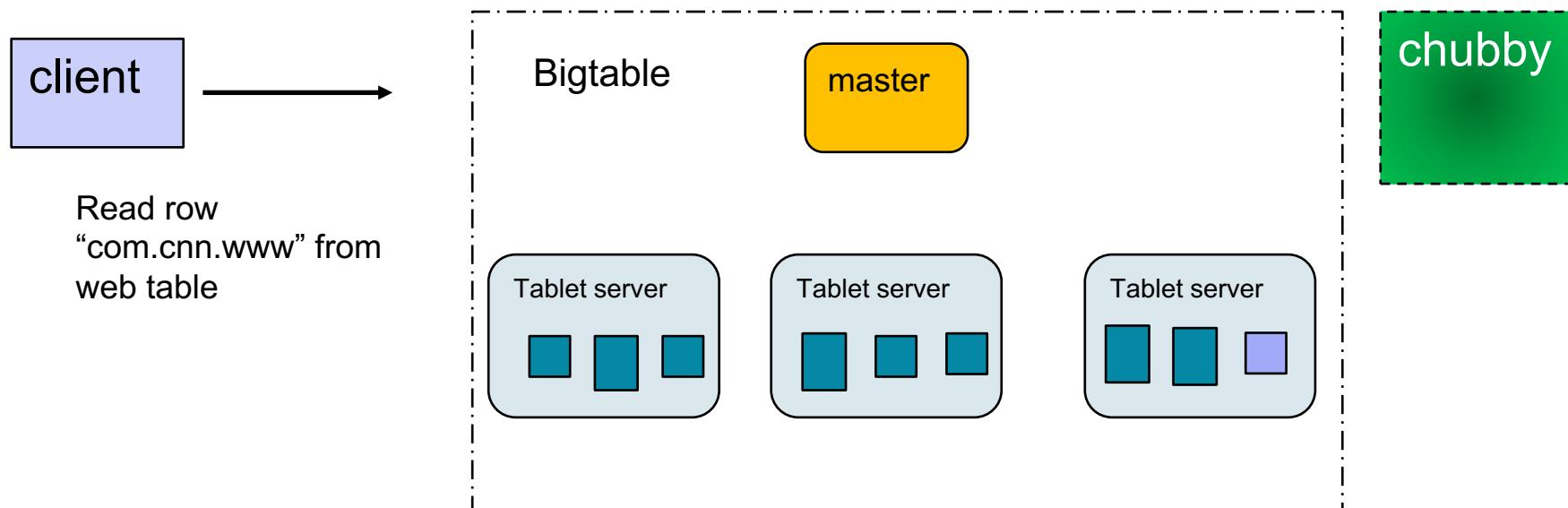
Tablet Serving

■ Client read/write request

- ▶ E.g. client wants to read the row corresponding to “com cnn www” from the web table

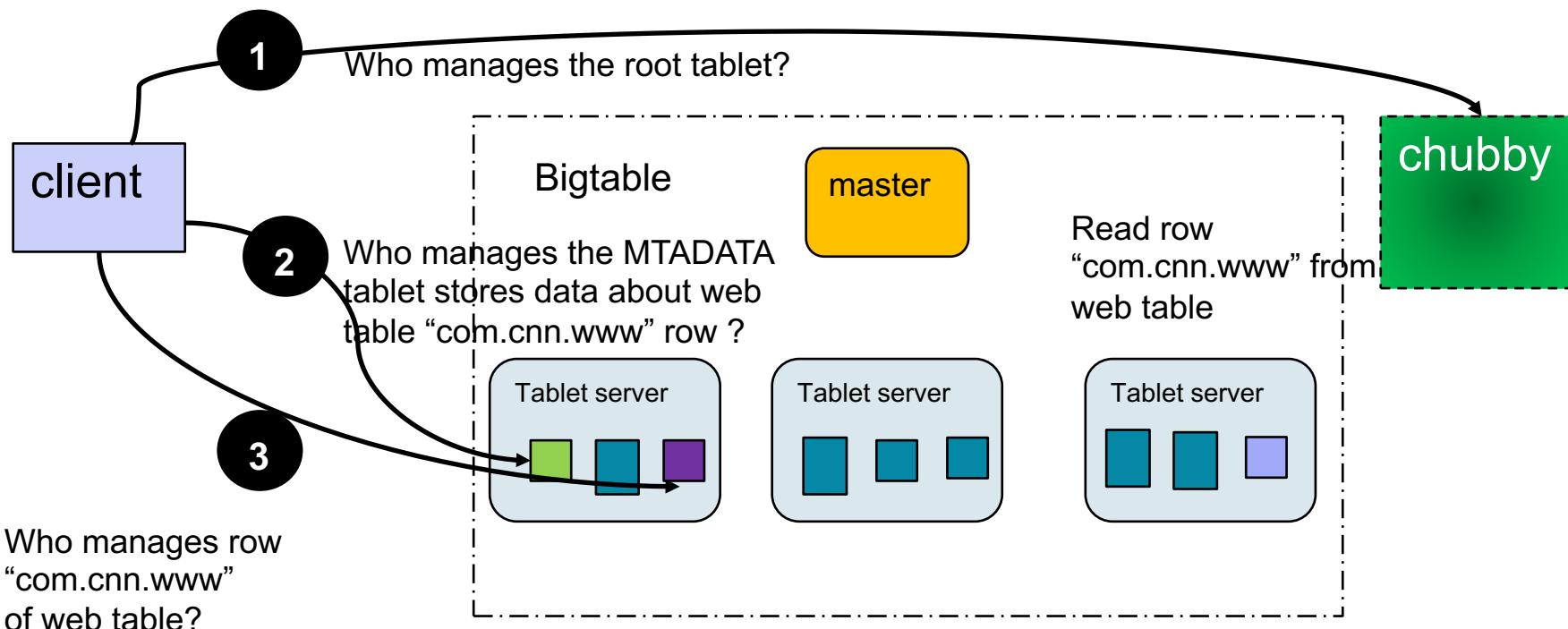
■ Steps

- ▶ Find the *tablet location*, the table server that serves the tablet
- ▶ Contact the tablet server to perform the read/write request



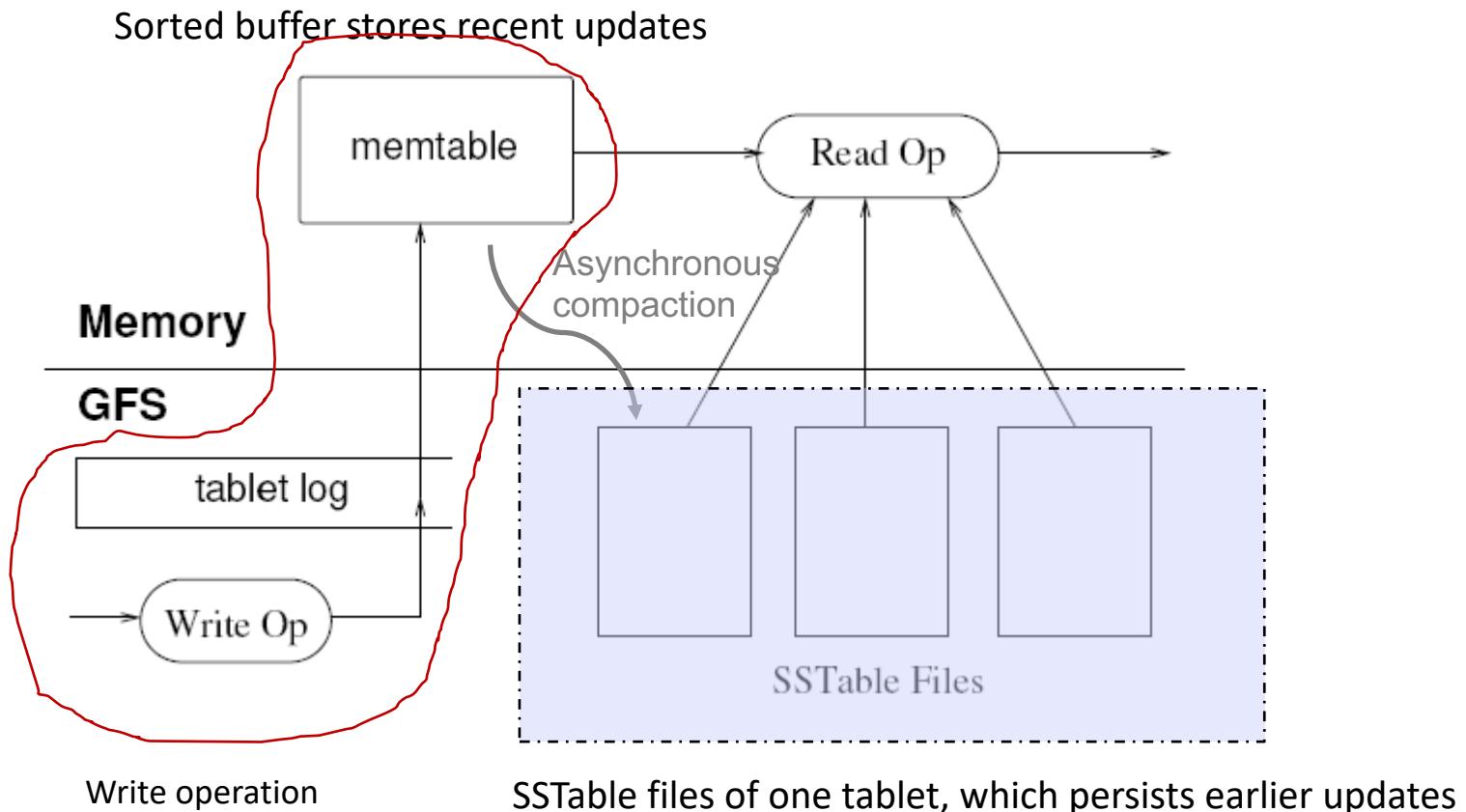
Find the tablet server

- If the client is requesting the data for first time
 - ▶ One round trip from chubby to find the root tablet's location
 - ▶ One round trip to the tablet server manages the root tablet
 - ▶ One round trip to the tablet server manages the METADATA tablet
- The client caches the tablet location for later use



Tablet Representation

Like Two level LSM Tree



Tablet Representation Implications

- A tablet server manages many tablets
 - ▶ Its memory contains latest updates of those tablets
 - ▶ BUT, the actual persisted data of those tablets might not be stored in this tablet server
 - Logs and SSTable Files are managed by the underlying file system GFS
 - GFS might replicate the files in any server
- Bigtable system is not responsible for actual file replication and placement
- The separation of concern simplifies the design



Write Path

- A write operation may insert new data, update or delete existing data
- The client sends write operation directly to the tablet server
 - ▶ The operation is checked for syntax and authorization
 - ▶ The operation is written to the **commit log**
 - ▶ The actual mutation content is inserted in the **memtable**
 - Deleted data will have a special tombstone entry/marker
- The only disk operation involved in write path is to append update to commit log



Compactions

- After many write operations, the size of memtable increases
- When memtable size reaches a threshold
 - ▶ The current one is frozen and converted to an SSTable and written to GFS
 - ▶ A new memtable is created to accept new updates
 - ▶ This is called **minor compaction**
- Why minor compaction
 - ▶ Memory management of tablet server
 - ▶ Reduce the size of active log entries
 - Minor compaction persists the updates on disk
 - Log entries reflecting those updates are no longer required



Compactions (cont'd)

- Every **minor compaction** creates a new SSTable
 - ▶ A tablet may contain many SSTable with overlapping key ranges
- **Merging compaction** happens periodically to merge a few SSTables and the current memtable content into a new SSTable
- **Major compaction** write all SSTable contents into a single SSTable. It will permanently remove the deleted data.



Compaction Process (t1-t5)

	language	content	anchor	
"cnn"	"language:" -> "EN" ← t1	"content:" -> "<html>..." "content:" -> "<html>..." ← t20	"anchor:cnnsi.com" -> "CNN" ← t6	"anchor:my.look.ca" -> "CNN.com" ← t13
"zdnet"	"language:" -> "EN" ← t4	"content:" -> "<html>..." ← t4	"anchor:slashdot.com" -> "zdnet" ← t8	

- Suppose a minor compaction happens at **t5**

Memstore

```
(“cnn”, “content:”, t1) -> “<html..”  

(“cnn”, “language:”, t1) -> “EN”  

(“zdnet”, “content:”, t4) -> “<html..”  

(“zdnet”, “language:”, t4) -> “EN”
```

SSTable File 1



Compaction Process (t6-t14)

	language	content	anchor	
"cnn"	"language:" -> "EN" $\leftarrow t_1$	"content:" -> "<html>..." "content:" -> "<html>..." $\leftarrow t_{20}$	"anchor:cnnsi.com" -> "CNN" $\leftarrow t_6$	"anchor:my.look.ca" -> "CNN.com" $\leftarrow t_{13}$
"zdnet"	"language:" -> "EN" $\leftarrow t_4$	"content:" -> "<html>..." $\leftarrow t_4$	"anchor:slashdot.com" -> "zdnet" $\leftarrow t_8$	

Memstore

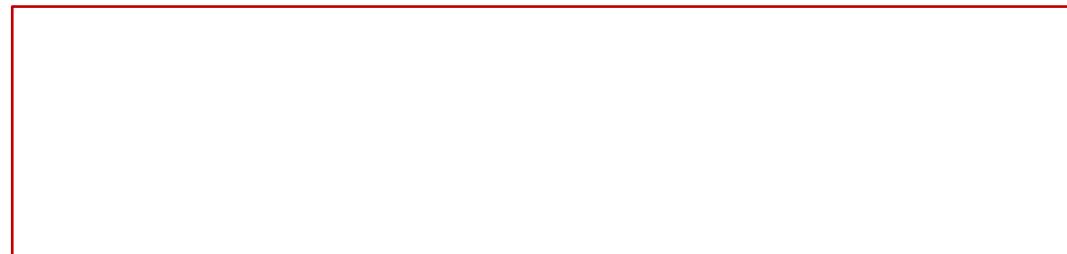
- (“cnn”, “anchor:cnnsi.com”, t6) >“CNN”
- (“cnn”, “anchor:my.look.ca:”, t13) ->“CNN.com”
- (“zdnet”, “anchor:Slashdot.com”, t8) ->“zdnet”

- Suppose another minor compaction happens at t14

SSTable File 1

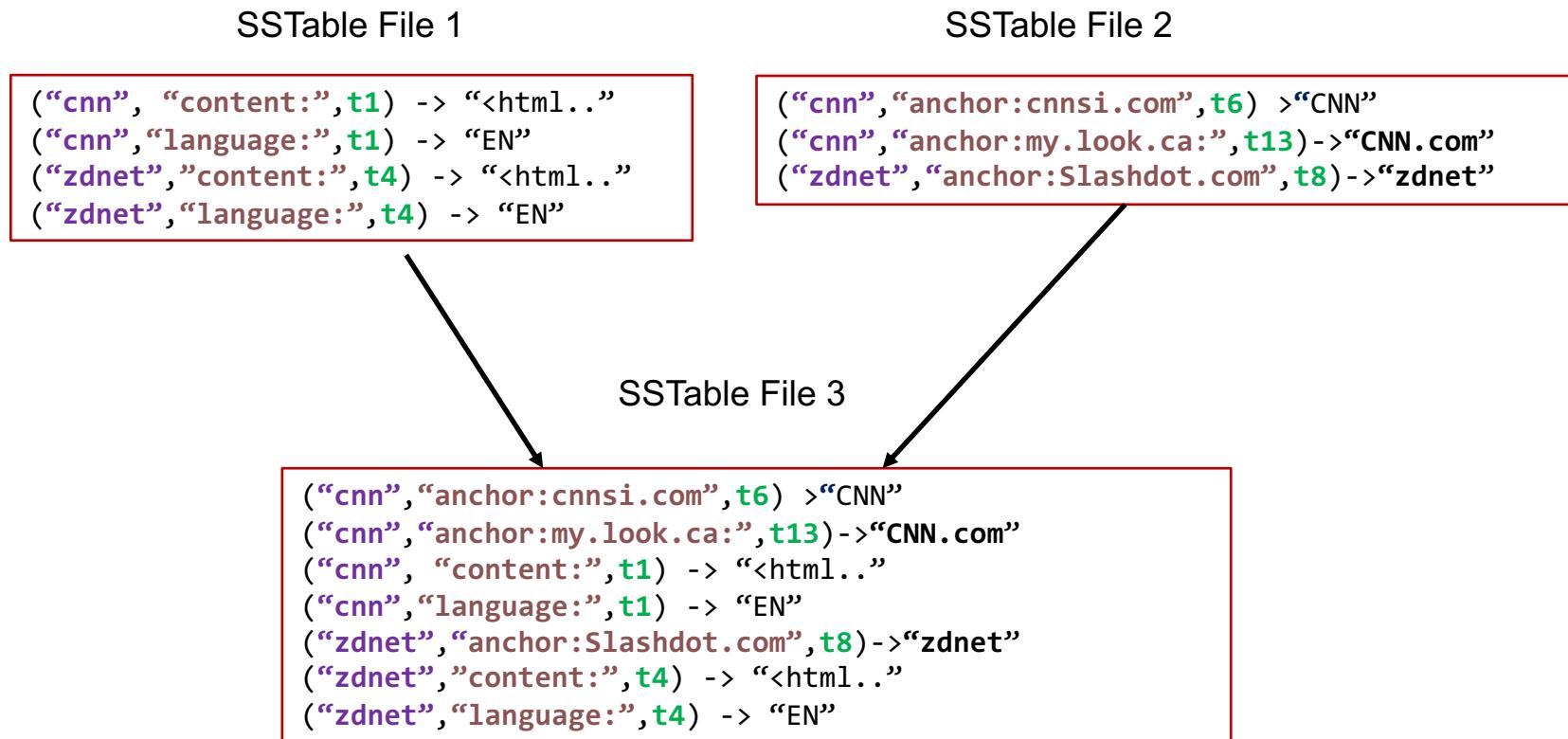
```
(“cnn”, “content:”, t1) -> “<html..”
(“cnn”, “language:”, t1) -> “EN”
(“zdnet”, “content:”, t4) -> “<html..”
(“zdnet”, “language:”, t4) -> “EN”
```

SSTable File 2

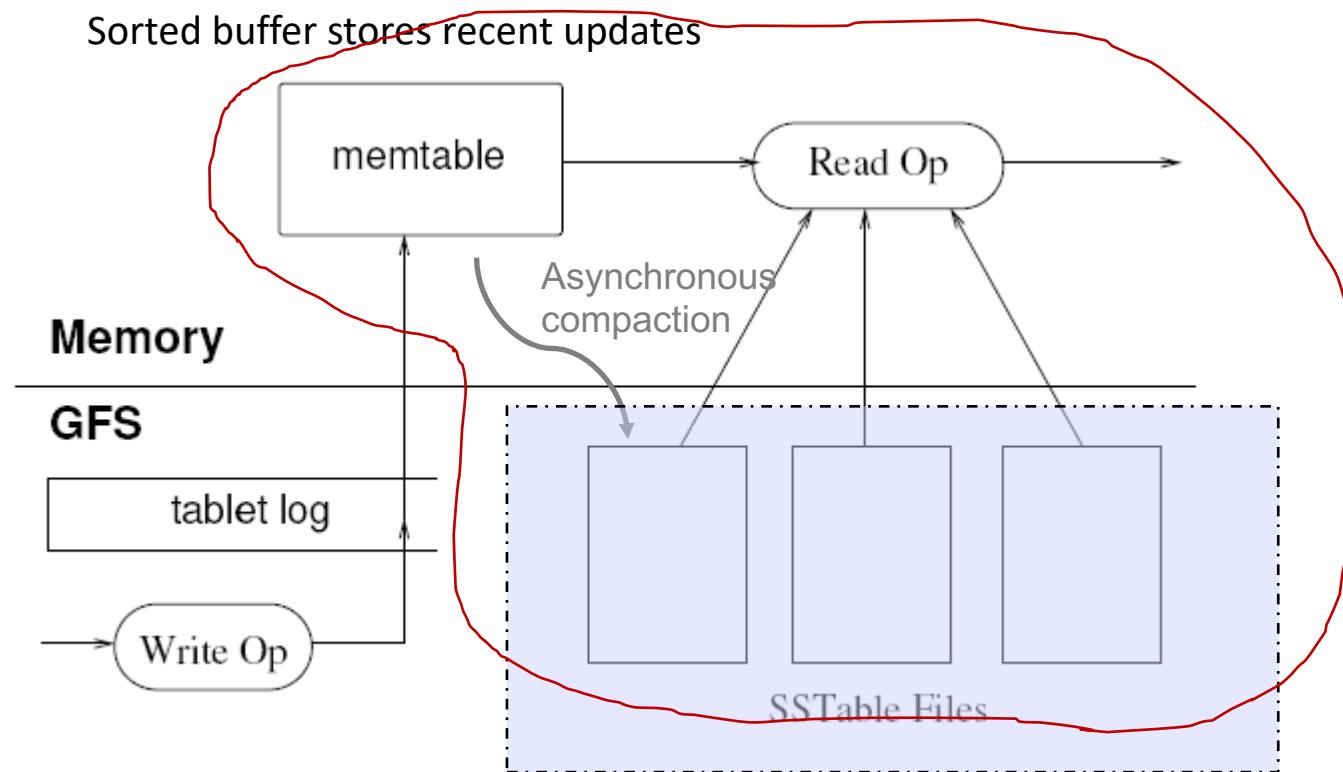


Compaction Process (t15)

- Assume a merging compact happens at t15



Read Path



Read Path

- The client sends read operation directly to the tablet server
 - ▶ The operation is check for syntax and authorization
 - ▶ Both memory and disk maybe involved to obtain the data
- What are kept in memory
 - ▶ Most recent updates in memtable (sorted by key)
 - ▶ Block indexes of SSTable files
- What are kept in disk
 - ▶ Earlier updates persisted in one or many SSTable files
- How does tablet server find the data
 - ▶ Check if the memtable contains partial data, or special mark indicating certain data is deleted
 - ▶ Check the index to find the block(s) that may contain partial data
 - ▶ Load the block and extract the data if there is any
 - ▶ Combine the data from memtable and disk block to obtain the final result



References

- O'Neil, P., et al. (1996). "The log-structured merge-tree (LSM-tree)." Acta Informatica **33**(4): 351-385.
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, Bigtable: A Distributed Storage System for Structured Data, OSDI'06: In Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI'06), Seattle, WA, 2006



COMP5338 – Advanced Data Models

Week 12: Time Series Database

Dr. Ying Zhou

School of Computer Science



THE UNIVERSITY OF
SYDNEY

Outline

- Time Series Data
- Facebook's Gorilla In-Memory TSDB
- InfluxDB Data Model and Storage

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice



Time Series Data

- “A **time series** is a series of *data points* indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time.”
- Sample time series data
 - ▶ Financial data
 - ▶ Scientific data
 - ▶ Health and personal data
 - ▶ Engineering data
 - Computer Hardware, software monitoring data
 - Other devices monitoring data
 - ▶ Traffic data
 - ▶ Business data



Time Series Analysis

- Time series analysis has a long history in various disciplines
- There are established ways of running such analysis developed in those disciplines
- The purpose is to use past behaviors (observations) to predict future
- Two main approaches
 - ▶ Statistical time series analysis
 - ▶ Machine learning time series analysis



Time Series Database: Motivations

■ Scalability

- ▶ Data are collected and accumulated from a large variety of devices and in high frequency sometimes
- ▶ Traditional database are not designed to handle data at such scale

■ Relative standards set of functions and operations

- ▶ Data retention policies
- ▶ Continuous queries
- ▶ Flexible Time aggregation



Outline

■ Time Series Data

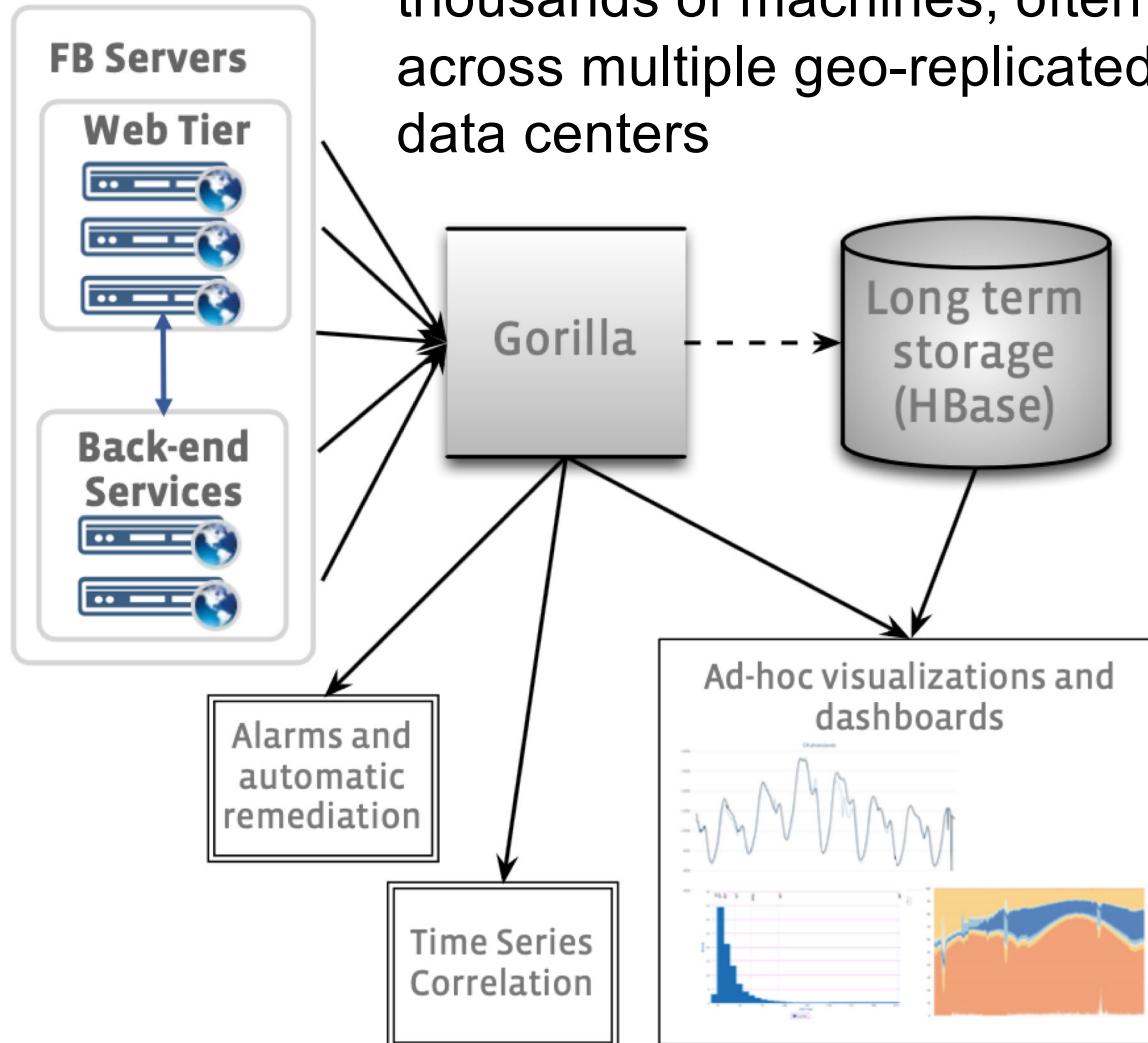
■ Facebook's Gorilla In-Memory TSDB

- ▶ Motivation
- ▶ Data Model
- ▶ Compression Algorithms

■ InfluxDB Data Model and Storage



Facebook's time series data and management



thousands of individual systems running on many thousands of machines, often across multiple geo-replicated data centers

As of Spring 2015, Facebook's monitoring systems generate more than **2 billion unique time series** of counters, with about **12 million data points added per second**. This represents over **1 trillion points per day**.

Requirements of TSDB in Facebook

■ Writes dominate

- ▶ It should always be available to take writes. The write rate might easily exceed tens of millions of data points each second.
- ▶ In contrast, the read rate is usually a couple orders of magnitude lower

■ State transitions

- ▶ Quickly identify the effect of new software, configuration, etc through fine-grained aggregation over short-time windows

■ Highly available

- ▶ Fail over to local storage when network partition happens

■ Fault Tolerance/durability

- ▶ Data Replication at multiple regions



What can be given up

- Users of monitoring systems do not place much emphasis on individual data points but rather on aggregate analysis.
- Traditional ACID guarantees are not a core requirement for TSDB
- Recent data points are of higher value than older points
 - ▶ Intuition: knowing if a particular system or service is broken right now is more valuable to an operations engineer than knowing if it was broken an hour ago.



Issue with HBase based solution

- HBase is an open source implementation of Bigtable
 - ▶ Optimize write performance at the cost of read performance
- With the increase of data size, the old monitoring system cannot scale to meet the read performance
- Issues observed
 - ▶ While the average read latency was acceptable for interactive charts, the 90th percentile query time had increased to multiple seconds
 - ▶ Queries of a few thousand time series took tens of seconds to execute
 - ▶ Larger queries executing over sparse datasets would time out
- Solution:
 - ▶ Design a memory based TSDB (Gorilla) to handle recent data
 - ▶ Older data are still persisted in HBase



Gorilla's Requirements

- 2 billion unique time series identified by a string key.
- 700 million data points (time stamp and value) added per minute.
- Store data for 26 hours.
- More than 40,000 queries per second at peak
- Reads succeed in under one millisecond.
- Support time series with 15 second granularity (4 points per minute per time series).
- Two in-memory, not co-located replicas (for disaster recovery capacity).
- Always serve reads even when a single server crashes.
- Ability to quickly scan over all in memory data.
- Support at least 2x growth per year.



Gorilla Data Model

- The monitoring data is a 3-tuple
 - ▶ (**String key, 64 bit timestamp, double precision value**)
- The **key** uniquely identifies a time series, e.g. the cpu usage of a web server w_1 in data center c_1
- The key is used to shard the monitoring data,
 - ▶ Each time series dataset can be mapped to a single Gorilla host.
 - ▶ Very easy to scale
- At storage level each data point consists of only the tuple **(timestamp, value)**
 - ▶ Raw size is 16 bytes



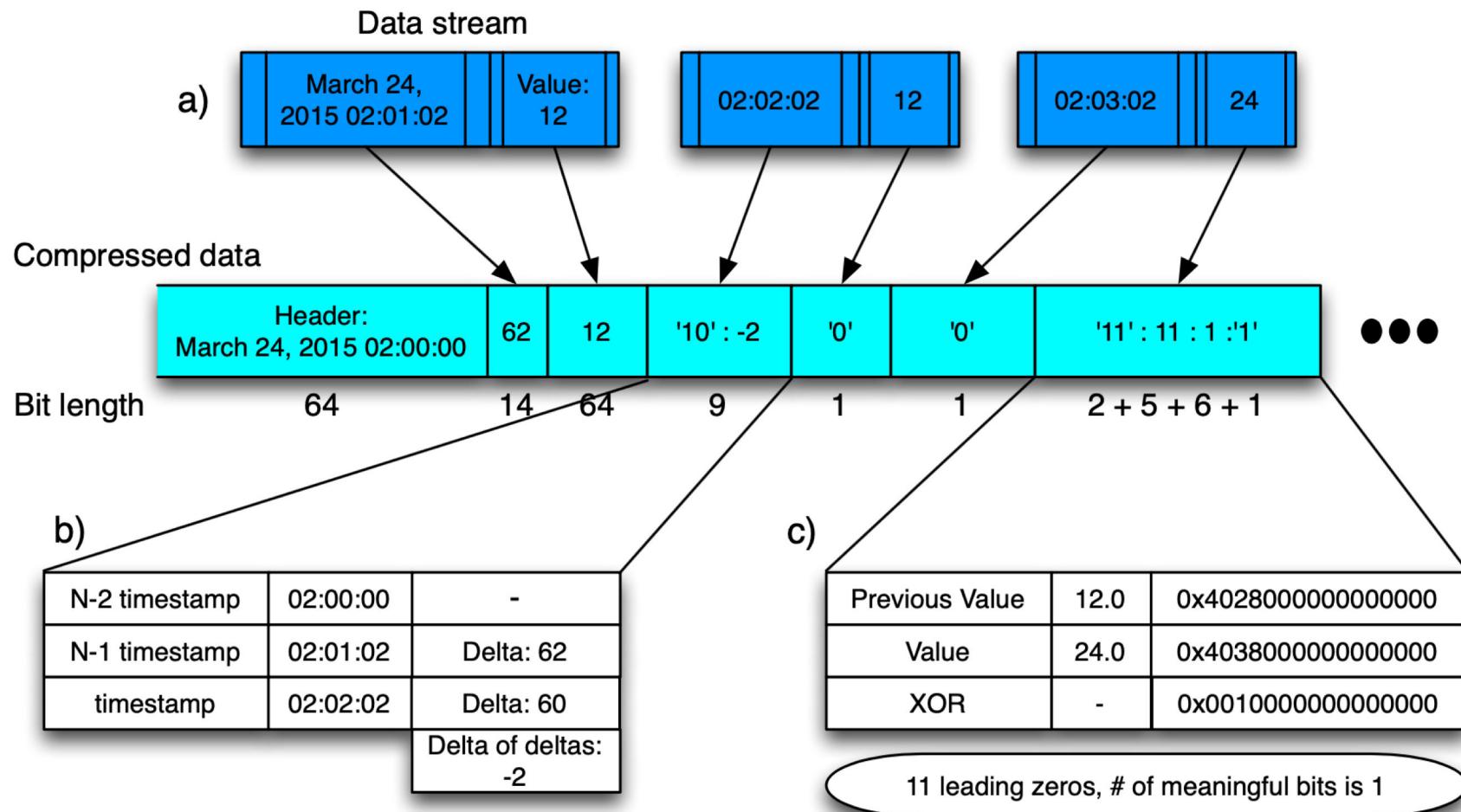
Main Contribution of Gorilla

- Very efficient compression algorithm
 - ▶ Compress each data point down from 16 bytes to an average of 1.37 bytes, a 12x reduction in size.
 - ▶ Used in many other systems
- Efficient memory data structures to allow fast query
 - ▶ Fast and efficient scan of all data
 - ▶ Constant time lookup of individual time series



Compression Algorithm

- Compress integer timestamp and double precision value separately
- Compress time stream into blocks partitioned by time, e.g. 2 hours data is compressed in a block



Timestamp Compression

- Vast majority of monitoring data arrived at a fixed interval, with occasional slightly early or late data points
 - ▶ E.g. a time series to log a single point every 60 seconds. Occasionally, the point may have a time stamp that is 1 second early or late, but the window is usually constrained.
- Solution
 - ▶ Do not store timestamps in their entirety (64 bit)
 - ▶ Store must smaller ***delta of delta***
 - Difference of difference between adjacent timestamps
 - ▶ The block header stores the starting time stamp, t_1 , which is aligned to a two hour window; the first timestamp, t_0 , in 14 bit, the block is stored as a delta from t_1 , the rest are stored as delta of delta
- Result: 96% of all timestamps can be compressed to a single bit.



Value Compression

- Using compression scheme similar to existing floating point compression algorithms
- Simplify the implementation based on data feature observed
 - ▶ Neighbouring data points have similar values
 - ▶ Many data point are of integer type
- Simple property of binary representation of double precision value
 - ▶ If values are close together, the sign, exponent, and first few bits of the mantissa will be identical.
- Solution
 - ▶ Compute XOR of current and previous value
 - ▶ Only stores information about the non-zero bits in the XOR value



Property of XOR

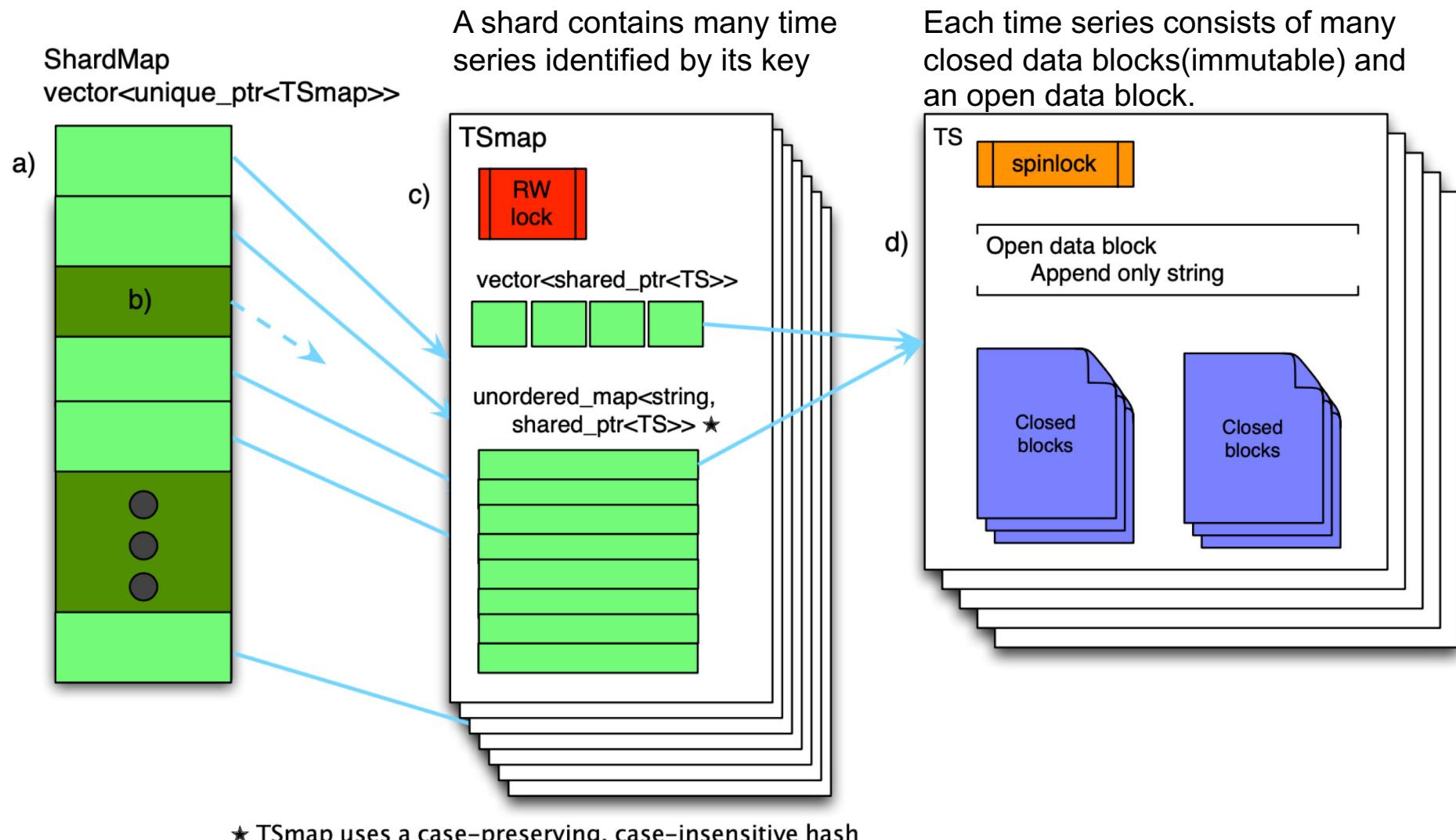
Decimal	Double Representation	XOR with previous
12	0x4028000000000000	
24	0x4038000000000000	0x0010000000000000
15	0x402e000000000000	0x0016000000000000
12	0x4028000000000000	0x0006000000000000
35	0x4041800000000000	0x0069800000000000

Decimal	Double Representation	XOR with previous
15.5	0x402f000000000000	
14.0625	0x402c200000000000	0x0003200000000000
3.25	0x400a000000000000	0x0026200000000000
8.625	0x4021400000000000	0x002b400000000000
13.1	0x402a333333333333	0x000b733333333333

Figure 4: Visualizing how XOR with the previous value often has leading and trailing zeros, and for many series, non-zero elements are clustered.



In memory data structure



At top level, there is mapping from time series key to shard (TSmap)

Then there is mapping between key to the actual time series



Outline

- Time Series Data
- Facebook's Gorilla In-Memory TSDB
- InfluxDB Data Model and Storage



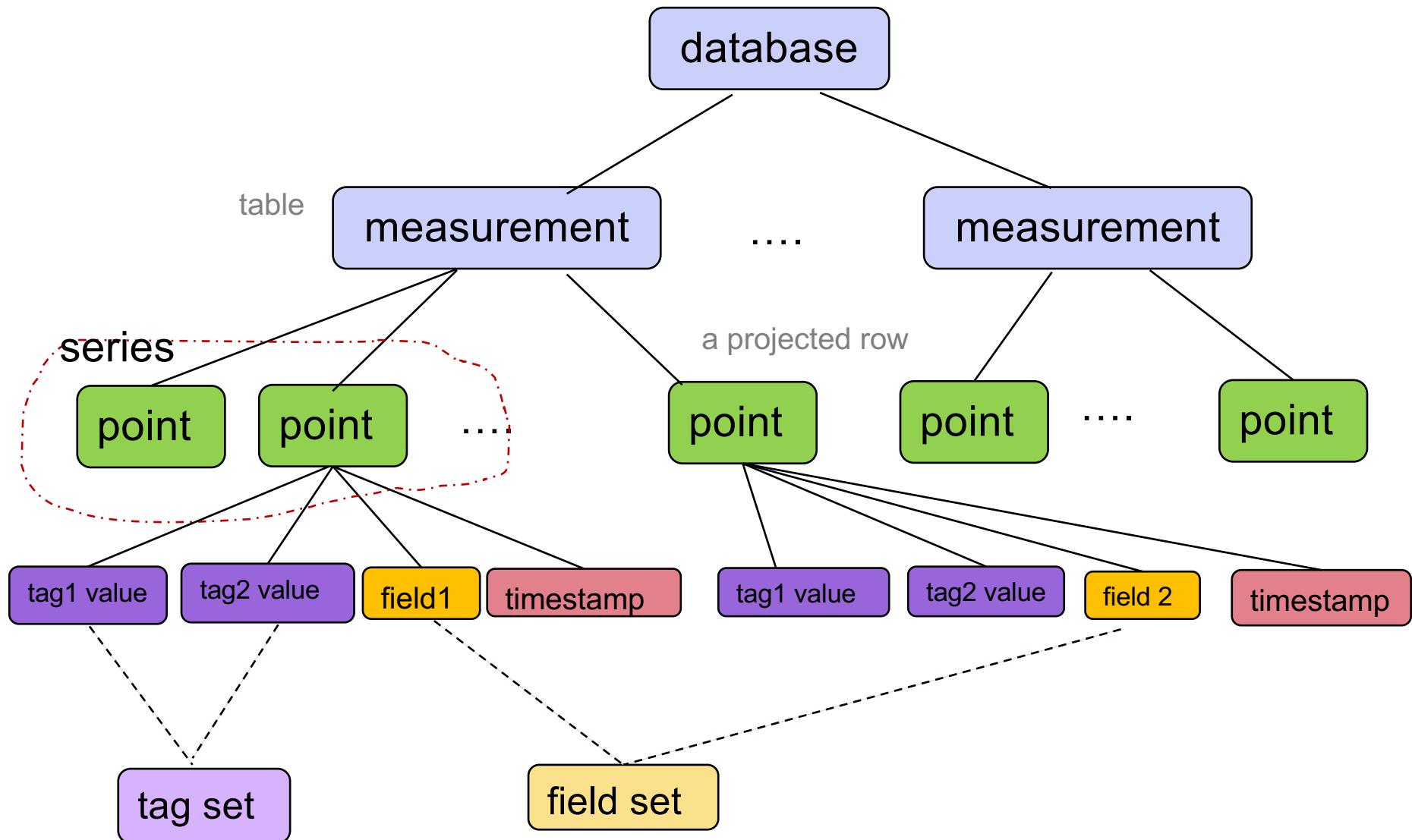
InfluxDB data model

- A single time series is just a set of (*timestamp*, *value*) pairs
- A TSDB manages many time series, it needs a key to identify unique time series
- Gorilla uses a flat string key for that purpose
 - ▶ (String **key**, 64 bit **timestamp**, double precision **value**)
- At the highest level, InfluxDB data point is also a 3-tuple
 - ▶ (**Series Key**, **timestamp**, **value**)
- **Series Key** has *multiple* dimensions defined by
 - ▶ **Field** (required) to represent the actual measurement
 - ▶ **tag** (optional) to represent meta data
- Data points with the same series key form a **time series**
- Related **series** can be grouped as a **measurement**

<https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/#point>



InfluxDB Key Concepts (1.x)



https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/



Example Data

- System load (cpu and memory) monitoring data of two virtual machines
- Four series represented by four keys in **Gorilla** model
 - ▶ “vm1.cpu”, “vm1.mem”, “vm2.cpu”, “vm2.mem”
 - ▶ A particular data point look like (“vm1.cpu”, 1605181049, 0.65)
- One **measurement** with one **tag** and two **fields** in InfluxDB model
 - ▶ **Measurement:** load
 - ▶ **Tag:** vm with two values: “vm1” and “vm2”
 - ▶ **Field:** cpu and mem
 - ▶ A particular data point would look like
2020-11-12T22:00:00Z, load, vm=“vm1”, cpu=0.65
- The rich data model allows flexible query and aggregation



Gorilla vs. InfluxDB model

vm1.cup:

Timestamp	Value
1605181049	0.65
1605182049	0.63
1605183049	0.64
...	...

vm1.mem:

Timestamp	Value
1605181049	0.40
1605182049	0.49
1605183049	0.45
...	...

load:

time	VM	cpu	mem
1605181049	"vm1"	0.65	0.40
1605181049	"vm2"	0.71	0.58
1605182049	"vm1"	0.63	0.49
1605183049	"vm2"	0.64	0.45
...

Meta data is modelled as tag

Actual measurements are modelled as fields, multiple fields form a field set



A More Complex Example

- Assume we want fine grained monitoring data at service level, instead of VM level
- In Gorilla model, we may need to come up with different key strings: “vm1.hdfs.dn.cpu”, “vm1.hbase.rs.cpu”, ...
- If we are monitoring the **cpu** and **memory** usage of 2 services on 2 virtual machines, there will be in total 8 time series
- In **InfluxDB** model, we need another tag **service** to indicate the services



The InfluxDB example model

time	VM	Service	cpu	mem
1605181049	“vm1”	“hdfs.dn”	0.35	0.10
1605181049	“vm1”	“hbase.rs”	0.30	0.40
1605181049	“vm2”	“hdfs.dn”	0.33	0.12
1605181049	“vm2”	“hbase.rs”	0.29	0.10
1605182049	“vm1”	“hdfs.dn”	0.34	0.20
1605182049	“vm1”	“hbase.rs”	0.25	0.13
...

Meta data are modelled as tags and multiple tags form a tag set

Actual measurements are modelled as field set



Another example

“the number of **butterflies** and **honeybees** counted by **two scientists** (langstroth and perpetua) in **two locations** (location 1 and location 2) over the time period from August 18, 2015 at midnight through August 18, 2015 at 6:12 AM”

Name: census

time	butterflies	honeybees	location	scientist
2015-08-18T00:00:00Z	12	23	1	langstroth
2015-08-18T00:00:00Z	1	30	1	perpetua
2015-08-18T00:06:00Z	11	28	1	langstroth
2015-08-18T00:06:00Z	3	28	1	perpetua
2015-08-18T05:54:00Z	2	11	2	langstroth
2015-08-18T06:00:00Z	1	10	2	langstroth
2015-08-18T06:06:00Z	8	23	2	perpetua
2015-08-18T06:12:00Z	7	22	2	perpetua

timestamp

field set

tag set

https://docs.influxdata.com/influxdb/v1.8/concepts/key_concepts/



Series

- In InfluxDB, a ***series*** is a collection of points that share a measurement, tag set, and field key.
 - ▶ E.g. time based data points belonging to the same feature of the same “entity”
 - cpu usage of hdfs data node process on vm 1
 - Stock price of a particular stock
 - Location of a particular car
 - ▶ In the insect counting data set, a particular insect's count at a particular location made by a particular scientist form a series
 - ▶ There are in total 8 series: 2 (insects) x 2 (locations) x 2 (scientists)
 - ▶ The following is a data series representing butterfly count made by Langstroth at location 1

time	butterflies	location	scientist
2015-08-18T00:00:00Z	12	1	langstroth
2015-08-18T00:06:00Z	11	1	langstroth



Fields and Tags

- Fields contain the actual data, they are required pieces of the **InfluxDB** data model
- Tags contain auxiliary data to differentiate series, they are optional
- Tags are indexed but fields are not
 - ▶ Find out the time instances when butterfly count is greater than 10 will need full table scan
 - ▶ But such query is not meaningful and considered very rare
 - ▶ More meaningful one that involves field value would be: find out the time instances when butterfly number in location 1 as observed by perpetua is greater than 10



InfluxDB Storage

■ InfluxDB's storage follows LSM tree design

- ▶ Early version used LSM tree based engine like LevelDB, which optimized for write
- ▶ It also used BoltDB, an engine based on memory mapped B+ tree, which is optimized for read
- ▶ Eventually build a special LSM tree engine, called Time Structured Merge Tree.

■ The main issue with general LSM tree based engine as identified by InfluxDB is delete performance

- ▶ Data retention policy means large amount of data may expire at any time point
- ▶ The original LSM approach to delete with tombstone marker makes it more expensive than other writes



InfluxDB Time Structured Merge Tree

■ TSM tree is very similar to LSM tree

- ▶ It has **WAL** to log the write query for durability and recovery
- ▶ It has **cache** for recent updates
- ▶ Data are stored as read-only files similar to SSTable format, called TSM files
 - Each contains data blocks
 - Data belonging to the same series would be saved in the same block(s) in timestamp order, the *series key* will be indexed, instead of incorporating it in every data point.
- ▶ Most write processing are similar to LSM based engine
 - Only WAL and cache are updated

■ Handling delete query

- ▶ Write a tombstone file for each TSM file that contains relevant data. These tombstone files are used at start up time to ignore blocks as well as during compactions to remove deleted entries.



References

- Pelkonen, T., et al. (2015). "Gorilla: A fast, scalable, in-memory time series database." Proceedings of the VLDB Endowment 8(12): 1816-1827.
 - ▶ Opensource project
 - <https://github.com/facebookarchive/beringei>
- **What the heck is time-series data (and why do I need a time-series database)?**
 - ▶ <https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/>
- InfluxDB data elements
- <https://docs.influxdata.com/influxdb/v2.0/reference/key-concepts/data-elements/>

