TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Hadoop ecosystem

Instructor: A. Prof. Binh-Minh Nguyen
Slides by Dr. Viet-Trung Tran
School of Information and Communication Technology

1

Outline

- Apache Hadoop
- Hệ thống tệp tin Hadoop (HDFS)
- Mô thức xử lý dữ liệu MapReduce
- Các thành phần khác trong hệ sinh thái Hadoop

2

History of Hadoop (2002-2004: Lucene and Nutch)

- Early 2000s: Doug Cutting develops two open-source search projects:
  - Lucene: Search indexer
    - Used e.g., by Wikipedia
  - Nutch: A spider/crawler (with Mike Carafella, now a Prof . at UMich)
- Nutch
  - Goal: Web-scale, crawler-based search
  - Written by a few part-time developers
  - Distributed, 'by necessity'
  - Demonstrated 100M web pages on 4 nodes, but true 'web scale' still very distant

3

2004-2006: GFS and MapReduce

- 2003/04: GFS, MapReduce papers published
  - Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: "The Google File System", SOSP 2003
  - Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004-2008
  - Directly addressed Nutch's scaling issues
- GFS & MapReduce added to Nutch
  - Two part-time developers over two years (2004-2006)
  - Crawler & indexer ported in two weeks
  - Ran on 20 nodes at IA and UW
  - Much easier to program and run, scales to several 100M web pages, but still far from web scale

4

2006-2008: Yahoo

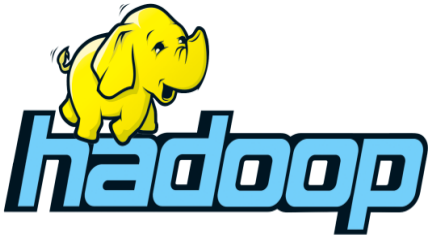
- 2006: Yahoo hires Cutting
  - Provides engineers, clusters, users, ...
  - Big boost for the project; Yahoo spends tens of M$
  - Not without a price: Yahoo has a slightly different focus (e.g., security) than the rest of the project; delays result
- Hadoop project split out of Nutch
  - Finally hit web scale in early 2008
- Cutting is now at Cloudera
  - Startup; started by three top engineers from Google, Facebook, Yahoo, and a former executive from Oracle
  - Has its own version of Hadoop; software remains free, but company sells support and consulting services
  - Was elected chairman of Apache Software Foundation

Who uses Hadoop?

- Hadoop is running search on some of the Internet's largest sites:
  - Amazon Web Services: Elastic MapReduce
  - AOL: Variety of uses, e.g., behavioral analysis & targeting
  - EBay: Search optimization (532-node cluster)
  - Facebook: Reporting/analytics, machine learning
  - Fox Interactive Media: MySpace, Photobucket, Rotten T.
  - Last.fm: Track statistics and charts
  - IBM: Blue Cloud Computing Clusters
  - LinkedIn: People You May Know (2x50 machines)
  - Rackspace: Log processing
  - Twitter: Store + process tweets, log files, other data
  - Yahoo: >36,000 nodes; biggest cluster is 4,000 nodes

Mục tiêu của Hadoop

- Mục tiêu chính
  - **Lưu trữ dữ liệu khả mở, tin cậy**
  - **Powerful data processing**
  - Efficient visualization
- Với thách thức
  - Thiết bị lưu trữ tốc độ chậm, máy tính thiếu tin cậy, lập trình song song phân tán không dễ dàng



7

7

Giới thiệu về Apache Hadoop

- **Lưu trữ và xử lý dữ liệu khả mở, tiết kiệm chi phí**
  - Xử lý dữ liệu phân tán với mô hình lập trình đơn giản, thân thiện hơn như MapReduce
  - Hadoop thiết kế để mở rộng thông qua kỹ thuật scale-out, tăng số lượng máy chủ
  - Thiết kế để vận hành trên phần cứng phổ thông, có khả năng chống chịu lỗi phần cứng
- Lấy cảm hứng từ kiến trúc dữ liệu của Google

8

8

Các thành phần chính của Hadoop

- Lưu trữ dữ liệu: Hệ thống tệp tin phân tán Hadoop (HDFS)
- Xử lý dữ liệu: MapReduce framework
- Các tiện ích hệ thống:
  - Hadoop Common: Các tiện ích chung hỗ trợ các thành phần của Hadoop.
  - Hadoop YARN: Một framework quản lý tài nguyên và lập lịch trong cụm Hadoop.

9

9

Hadoop giải quyết bài toán khả mở

- Thiết kế hướng "phân tán" ngay từ đầu
  - Hadoop mặc định thiết kế để triển khai trên cụm máy chủ
- Các máy chủ tham gia vào cụm được gọi là các Nodes
  - Mỗi node tham gia vào cả 2 vai trò lưu trữ và tính toán
- Hadoop mở rộng bằng kỹ thuật scale-out
  - Có thể tăng cụm Hadoop lên hàng chục ngàn nodes

10

10

Hadoop giải quyết bài toán chịu lỗi

- Với việc triển khai trên cụm máy chủ phổ thông
  - Hỏng hóc phần cứng là chuyện thường ngày, không phải là ngoại lệ
  - Hadoop chịu lỗi thông qua kỹ thuật "dư thừa"
- Các tệp tin trong HDFS được phân mảnh, nhân bản ra các nodes trong cụm
  - Nếu một node gặp lỗi, dữ liệu ứng với nodes đó được tái nhân bản qua các nodes khác
- Công việc xử lý dữ liệu được phân mảnh thành các tác vụ độc lập
  - Mỗi tác vụ xử lý một phần dữ liệu đầu vào
  - Các tác vụ được thực thi song song với các tác vụ khác
  - Tác vụ lỗi sẽ được tái lập lịch thực thi trên node khác
- Hệ thống Hadoop thiết kế sao cho các lỗi xảy ra trong hệ thống được xử lý tự động, không ảnh hưởng tới các ứng dụng phía trên

11

11

Tổng quan về HDFS

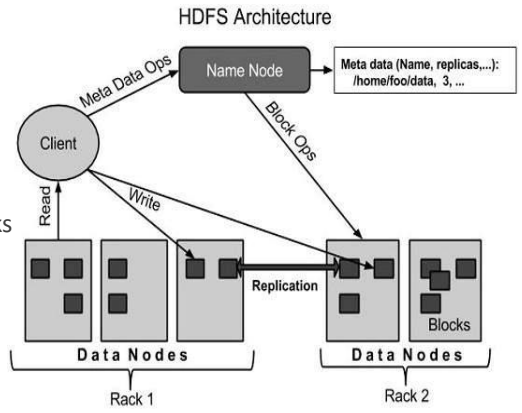
- HDFS cung cấp khả năng lưu trữ tin cậy và chi phí hợp lý cho khối lượng dữ liệu lớn
- Tối ưu cho các tập tin kích thước lớn (từ vài trăm MB tới vài TB)
- HDFS có không gian cây thư mục phân cấp như UNIX (vd., /hust/soict/hello.txt)
  - Hỗ trợ cơ chế phân quyền và kiểm soát người dùng như của UNIX
- Khác biệt so với hệ thống tập tin trên UNIX
  - Chỉ hỗ trợ thao tác ghi thêm dữ liệu vào cuối tệp (APPEND)
  - Ghi một lần và đọc nhiều lần

12

12

Kiến trúc của HDFS

- Kiến trúc Master/Slave
- HDFS master: name node
  - Quản lý không gian tên và siêu dữ liệu ánh xạ tệp tin tới vị trí các chunks
  - Giám sát các data node
- HDFS slave: data node
  - Trực tiếp thao tác I/O các chunks



13

13

Nguyên lý thiết kế cốt lõi của HDFS

- I/O pattern
  - Chỉ ghi thêm (Append)→ giảm chi phí điều khiển tương tranh
- Phân tán dữ liệu
  - Tệp được chia thành các chunks lớn (64 MB)
  - → Giảm kích thước metadata
  - → Giảm chi phí truyền dữ liệu
- Nhân bản dữ liệu
  - Mỗi chunk thông thường được sao làm 3 nhân bản
- Cơ chế chịu lỗi
  - Data node: sử dụng cơ chế tái nhân bản
  - Name node
    - Sử dụng Secondary Name Node
    - SNN hỏi data nodes khi khởi động thay vì phải thực hiện cơ chế đồng bộ phức tạp với primary NN

14

Mô thức xử lý dữ liệu MapReduce

- MapReduce là mô thức xử lý dữ liệu mặc định trong Hadoop
- MapReduce không phải là ngôn ngữ lập trình, được đề xuất bởi Google
- Đặc điểm của MapReduce
  - Đơn giản (Simplicity)
  - Linh hoạt (Flexibility)
  - Khả mở (Scalability)

15

15

A MR job = {Isolated Tasks}n

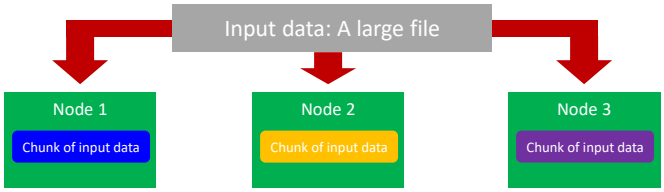
- Mỗi chương trình MapReduce là một công việc (job) được phân rã làm nhiều tác vụ độc lập (task) và các tác vụ này được phân tán trên các nodes khác nhau của cụm để thực thi
- Mỗi tác vụ được thực thi độc lập với các tác vụ khác để đạt được tính khả mở
  - Giảm truyền thông giữa các node máy chủ
  - Tránh phải thực hiện cơ chế đồng bộ giữa các tác vụ

16

16

Dữ liệu cho MapReduce

- MapReduce trong môi trường Hadoop thường làm việc với dữ liệu đã có sẵn trên HDFS
- Khi thực thi, mã chương trình MapReduce được gửi tới các node đã có dữ liệu tương ứng

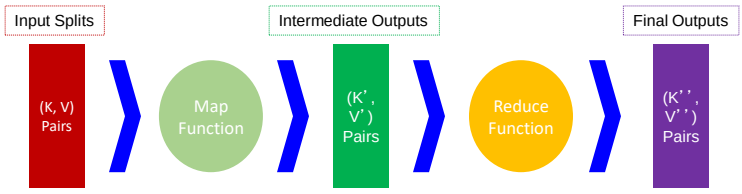


17

17

Chương trình MapReduce

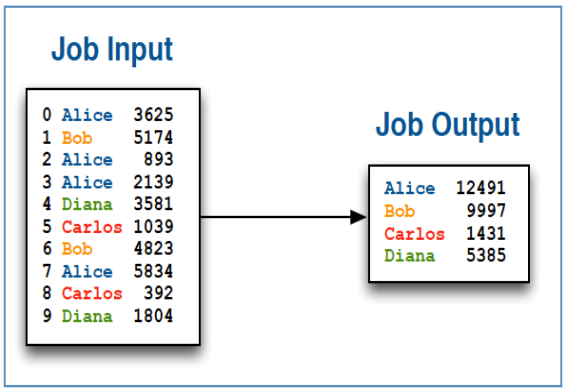
- Lập trình với MapReduce cần cài đặt 2 hàm Map và Reduce
  - 2 hàm này được thực thi bởi các tiến trình Mapper và Reducer tương ứng.
- Trong chương trình MapReduce, dữ liệu được nhìn nhận như là các cặp khóa – giá trị (key – value)
- Các hàm Map và Reduce nhận đầu vào và trả về đầu ra các cặp (key – value)



18

Ví dụ về MapReduce

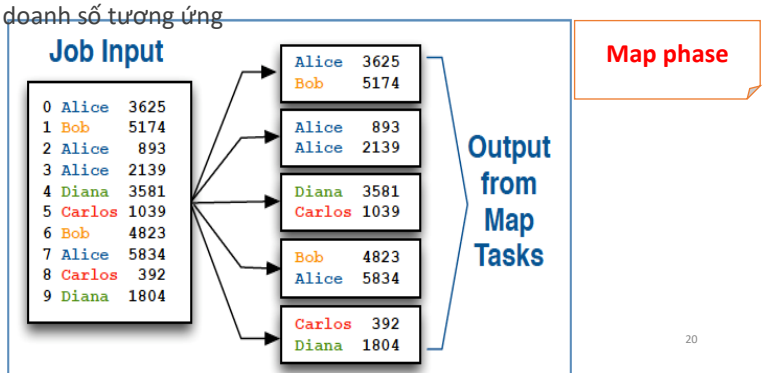
- Đầu vào: tệp văn bản chứa thông tin về order ID, employee name, and sale amount
- Đầu ra : Doanh số bán (sales) theo từng nhân viên (employee)



19

Bước Map

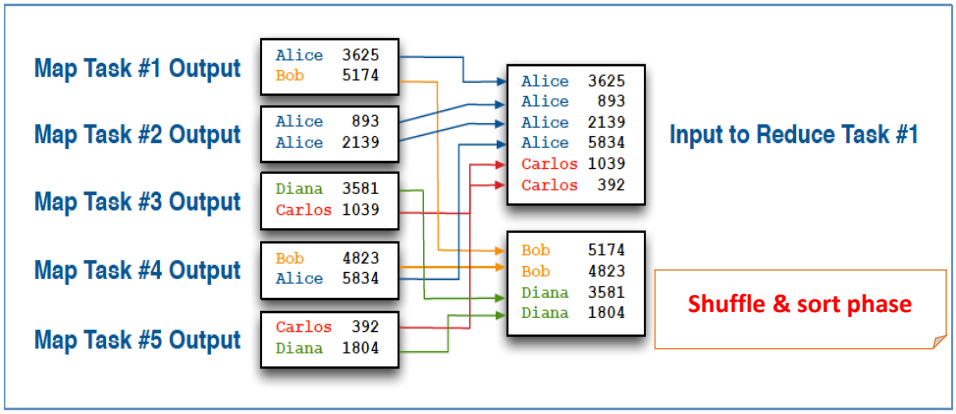
- Dữ liệu đầu vào được xử lý bởi nhiều tác vụ Mapping độc lập
  - Số lượng các tác vụ Mapping được xác định theo lượng dữ liệu đầu vào (~ số chunks)
  - Mỗi tác vụ Mapping xử lý một phần dữ liệu (chunk) của khối dữ liệu ban đầu
- Với mỗi tác vụ Mapping, Mapper xử lý lần lượt từng bản ghi đầu vào
  - Với mỗi bản ghi đầu vào (key-value), Mapper đưa ra 0 hoặc nhiều bản ghi đầu ra (key – value trung gian)
- Trong ví dụ này, tác vụ Mapping đơn giản đọc từng dòng văn bản và đưa ra tên nhân viên và doanh số tương ứng



20

Bước shuffle & sort

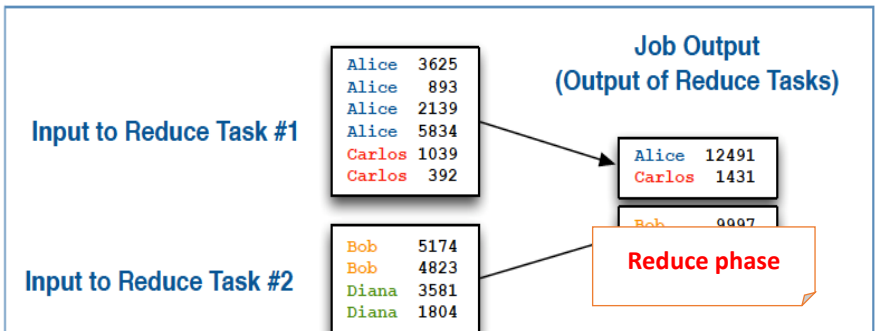
- Hadoop tự động sắp xếp và gộp đầu ra của các Mappers theo các partitions
  - Mỗi partitions là đầu vào cho một Reducer



21

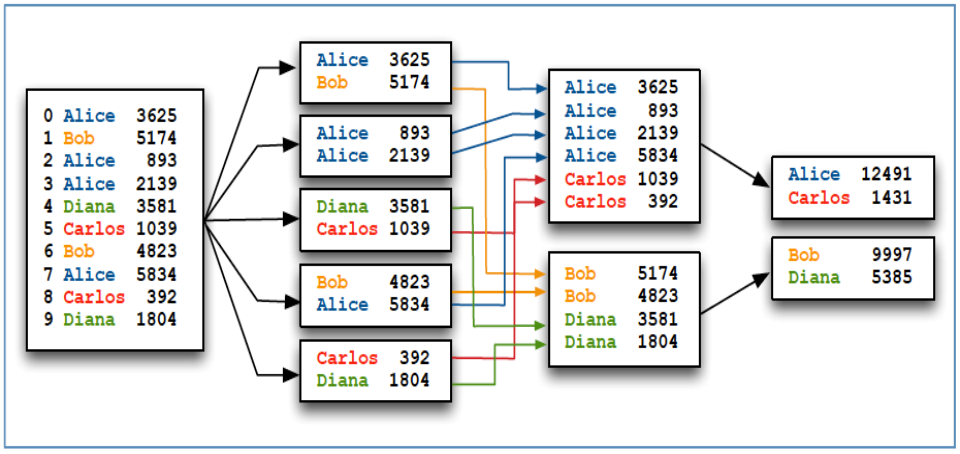
Bước Reduce

- Reducer nhận dữ liệu đầu vào từ bước shuffle & sort
  - Tất cả các bản ghi key – value tương ứng với một key được xử lý bởi một Reducer duy nhất
  - Giống bước Map, Reducer xử lý lần lượt từng key, mỗi lần với toàn bộ các values tương ứng
- Trong ví dụ, hàm reduce đơn giản là tính tổng doanh số cho từng nhân viên, đầu ra là các cặp key – value tương ứng với tên nhân viên – doanh số tổng



22

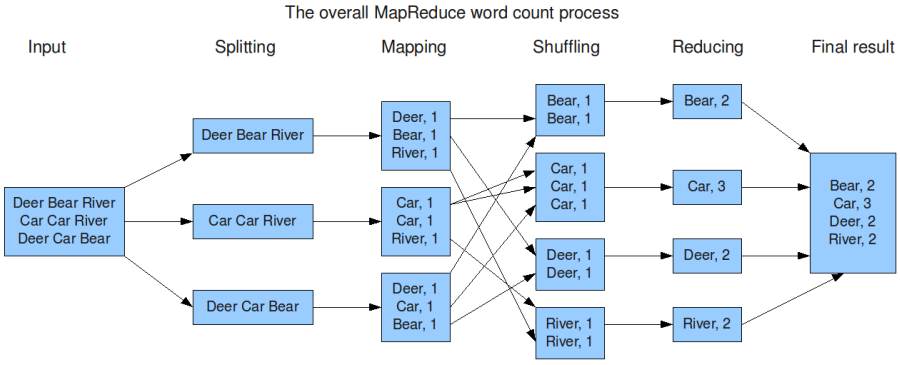
Luồng dữ liệu cho ví dụ MapReduce



23

23

Luồng dữ liệu với bài toán Word Count



24

Chương trình Word Count thực tế (1)

```
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.GenericOptionsParser;



public class WordCount {
public static void main(String [] args) throws Exception
{
Configuration c=new Configuration();
String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
Path input=new Path(files[0]);
Path output=new Path(files[1]);
Job j=new Job(c,"wordcount");
j.setJarByClass(WordCount.class);
j.setMapperClass(MapForWordCount.class);
j.setReducerClass(ReduceForWordCount.class);
j.setOutputKeyClass(Text.class);
j.setOutputValueClass(IntWritable.class);
FileInputFormat.addInputPath(j, input);
FileOutputFormat.setOutputPath(j, output);
System.exit(j.waitForCompletion(true)?0:1);
}
```

25

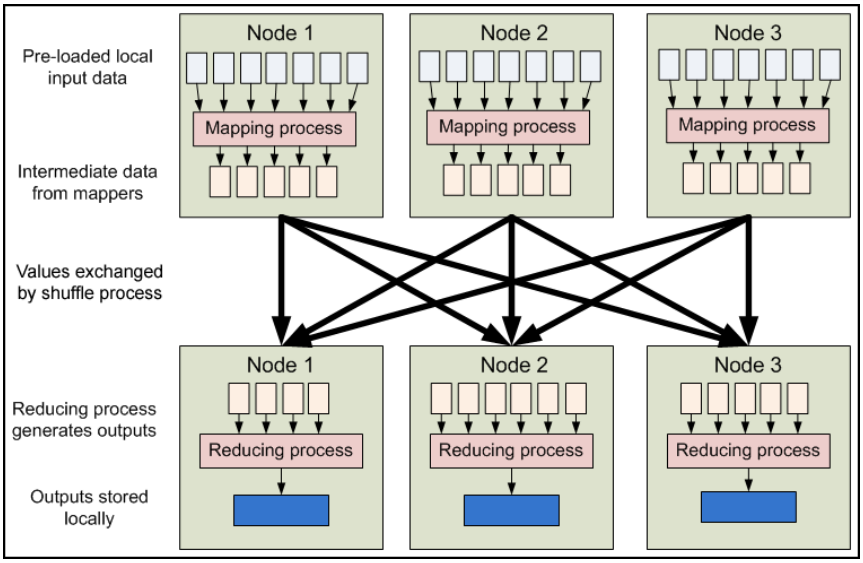
Chương trình Word Count thực tế (2)

```
public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{
public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException
{
String line = value.toString();
String[] words=line.split(",");
for(String word: words )
{
      Text outputKey = new Text(word.toUpperCase().trim());
  IntWritable outputValue = new IntWritable(1);
  con.write(outputKey, outputValue);
}
}
}

public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>
{
public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedE
{
int sum = 0;
   for(IntWritable value : values)
   {
   sum += value.get();
   }
   con.write(word, new IntWritable(sum));
}
```

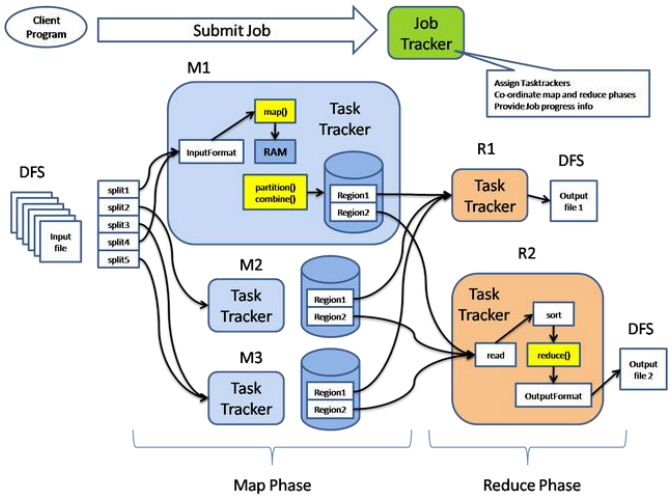
26

MapReduce trên môi trường phân tán



27

Vai trò của Job tracker và Task tracker



28

Các thành phần khác trong hệ sinh thái Hadoop

- Ngoài HDFS và MapReduce, hệ sinh thái Hadoop còn nhiều hệ thống, thành phần khác phục vụ
  - Phân tích dữ liệu
  - Tích hợp dữ liệu
  - Quản lý luồng
  - Vvv
- Các thành phần này không phải là 'core Hadoop' nhưng là 1 phần của hệ sinh thái Hadoop
  - Hầu hết là mã nguồn mở trên Apache

29

29

Apache Pig

- Apache Pig cung cấp giao diện xử lý dữ liệu mức cao
  - Pig đặc biệt tốt cho các phép toán Join và Transformation
- Trình biên dịch của Pig chạy trên máy client
  - Biến đổi PigLatin script thành các jobs của MapReduce
  - Đệ trình các công việc này lên cụm tính toán

```
people = LOAD '/user/training/customers' AS (cust_id, name);
orders = LOAD '/user/training/orders' AS (ord_id, cust_id, cost);
groups = GROUP orders BY cust_id;
totals = FOREACH groups GENERATE group, SUM(orders.cost) AS t;
result = JOIN totals BY group, people BY cust_id;
DUMP result;
```

30

30

Apache Hive

- Cũng là một lớp trừu tượng mức cao của MapReduce
  - Giảm thời gian phát triển
  - Cung cấp ngôn ngữ HiveQL: SQL-like language
- Trình biên dịch Hive chạy trên máy client
  - Chuyển HiveQL script thành MapReduce jobs
  - Đệ trình các công việc này lên cụm tính toán


```
SELECT customers.cust_id, SUM(cost) AS total
  FROM customers
  JOIN orders
    ON customers.cust_id = orders.cust_id
GROUP BY customers.cust_id
ORDER BY total DESC;
```

31

31

Apache Hbase

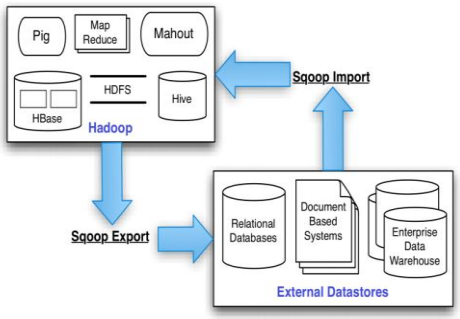
- HBase là một CSDL cột mở rộng phân tán, lưu trữ dữ liệu trên HDFS
  - Được xem như là hệ quản trị CSDL của Hadoop
- Dữ liệu được tổ chức về mặt logic là các bảng, bao gồm rất nhiều dòng và cột
  - Kích thước bảng có thể lên đến hàng Terabyte, Petabyte
  - Bảng có thể có hàng ngàn cột
- Có tính khả mở cao, đáp ứng băng thông ghi dữ liệu tốc độ cao
  - Hỗ trợ hàng trăm ngàn thao tác INSERT mỗi giây (/s)
- Tuy nhiên về các chức năng thì còn rất hạn chế khi so sánh với hệ QTCSDL truyền thống
  - Là NoSQL : không có ngôn ngữ truy vấn mức cao như SQL
  - Phải sự dụng API để scan/ put/ get/ dữ liệu theo khóa

32

32

Apache Sqoop

- Sqoop là một công cụ cho phép trung chuyển dữ liệu theo khối từ Apache Hadoop và các CSDL có cấu trúc như CSDL quan hệ
- Hỗ trợ import tất cả các bảng, một bảng hay 1 phần của bảng vào HDFS
  - Thông qua Map only hoặc MapReduce job
  - Kết quả là 1 thư mục trong HDFS chứa các tập tin văn bản phân tách các trường theo ký tự phân tách (vd. , hoặc \t)
- Hỗ trợ export dữ liệu ngược trở lại từ Hadoop ra bên ngoài



33

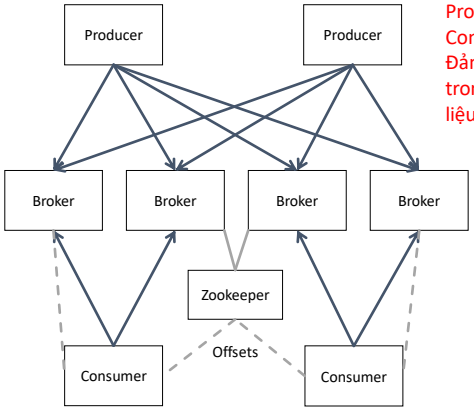
33

Apache Kafka

Producers

Kafka Cluster

Consumers



Producers không cần biết Consumers
Đảm bảo sự linh hoạt và tin cậy trong quá trình trung chuyển dữ liệu giữa các bên

Kafka cho phép phân tách mạch lạc các thành phần tham gia vào luồng dữ liệu

34

Apache Oozie

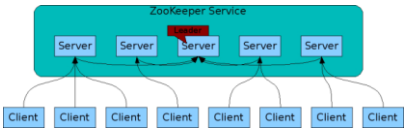
- Oozie là một hệ thống lập lịch luồng công việc để quản lý các công việc thực thi trên cụm Hadoop
- Luồng workflow của Oozie là đồ thị vòng có hướng (Directed Acyclical Graphs (DAGs)) của các khối công việc
- Oozie hỗ trợ đa dạng các loại công việc
  - Thực thi MapReduce jobs
  - Thực thi Pig hay Hive scripts
  - Thực thi các chương trình Java hoặc Shell
  - Tương tác với dữ liệu trên HDFS
  - Chạy chương trình từ xa qua SSH
  - Gửi nhận email

35

35

Apache Zookeeper

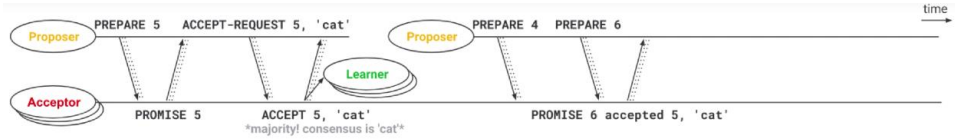
- Apache ZooKeeper là một dịch vụ cung cấp các chức năng phối hợp phân tán độ tin cậy cao
  - Quản lý thành viên trong nhóm máy chủ
  - Bầu cử leader
  - Quản lý thông tin cấu hình động
  - Giám sát trạng thái hệ thống
- Đây là các service lõi, tối quan trọng trong các hệ thống phân tán



36

36

PAXOS algorithm



⇨ **Proposer** wants to propose a certain value:
It sends **PREPARE IDp** to a majority (or all) of **Acceptors**.
IDp must be unique, e.g. slotted timestamp in nanoseconds.
e.g. **Proposer** 1 chooses IDs 1, 3, 5...
Proposer 2 chooses IDs 2, 4, 6..., etc.
Timeout? retry with a new (higher) IDp.

⇨ **Acceptor** receives a **PREPARE** message for IDp:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Will promise to ignore any request lower than IDp.
Has it ever accepted anything? (assume accepted ID=**IDa**)
Yes ->Reply with **PROMISE IDp accepted IDa, value**.
No -> Reply with **PROMISE IDp**.
If a majority of acceptors promise, no ID<IDp can make it through.

⇨ **Proposer** gets majority of **PROMISE** messages for a specific IDp:
It sends **ACCEPT-REQUEST IDp, VALUE** to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes - > It picks the value with the highest **IDa** that it got.
No -> It picks any value it wants.

⇨ **Acceptor** receives an **ACCEPT-REQUEST** message for IDp, value:
Did it promise to ignore requests with this IDp?
Yes -> then ignore
No -> Reply with **ACCEPT IDp, value**. Also send it to all **Learners**.
If a majority of acceptors accept **IDp, value**, consensus is reached.
Consensus is and will always be on **value** (not necessarily IDp).

⇨ **Proposer** or **Learner** get **ACCEPT** messages for IDp, value:
If a proposer/learner gets majority of accept for a specific **IDp**, they know that consensus has been reached on **value** (not IDp).

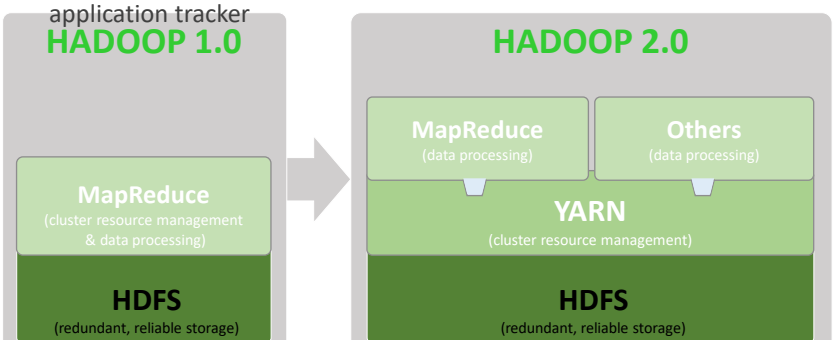
https://www.youtube.com/watch?v=d7nAGI_NZPk

37

37

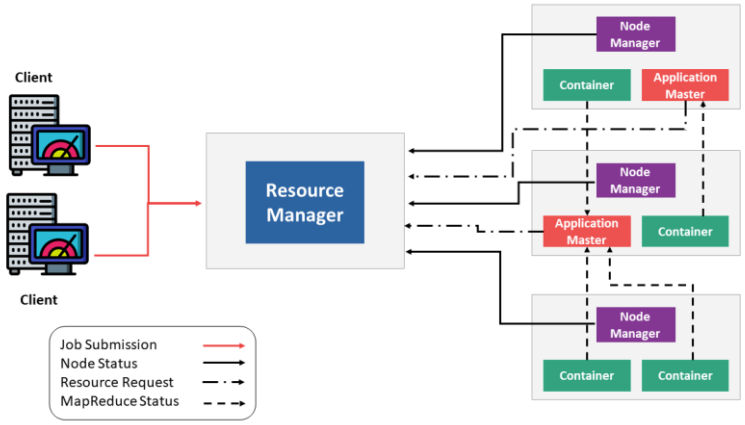
YARN – Yet Another Resource Negotiator

- Nodes có tài nguyên là – bộ nhớ và CPU cores
- YARN đóng vai trò cấp phát lượng tài nguyên phù hợp cho các ứng dụng khi có yêu cầu
- YARN được đưa ra từ Hadoop 2.0
  - Cho phép MapReduce và non MapReduce cùng chạy trên 1 cụm Hadoop
  - Với MapReduce job, vai trò của job tracker được thực hiện bởi application tracker



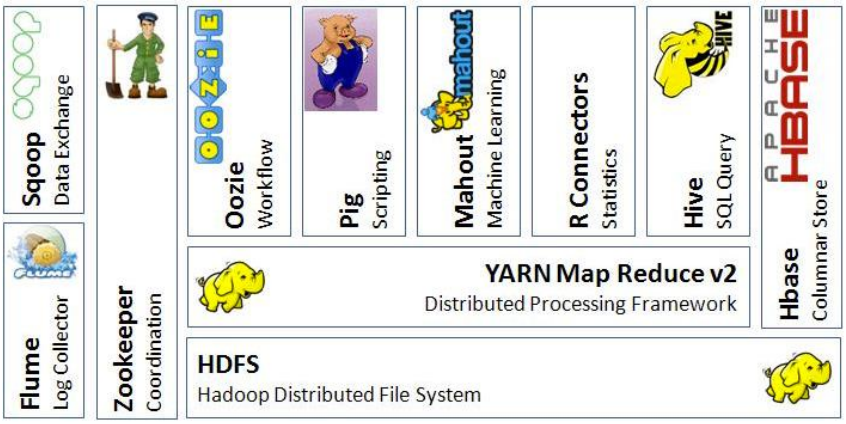
38

Ví dụ về cấp phát trên YARN



39

Bức tranh tổng thể hệ sinh thái Hadoop



40

Các platform quản lý dữ liệu lớn

41

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Thank you for your attention!
Q&A

42

Hadoop File System

Asoc. Prof. Nguyen binh minh
SoICT - HUST

1

Reference

- The Hadoop Distributed File System: Architecture and Design by Apache Foundation Inc.

2

Basic Features: HDFS

- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data
- Can be built out of commodity hardware

3

Fault tolerance

- Failure is the norm rather than exception
- A HDFS instance may consist of thousands of server machines, each storing part of the file system's data.
- Since we have huge number of components and that each component has non-trivial probability of failure means that there is always some component that is non-functional.
- Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

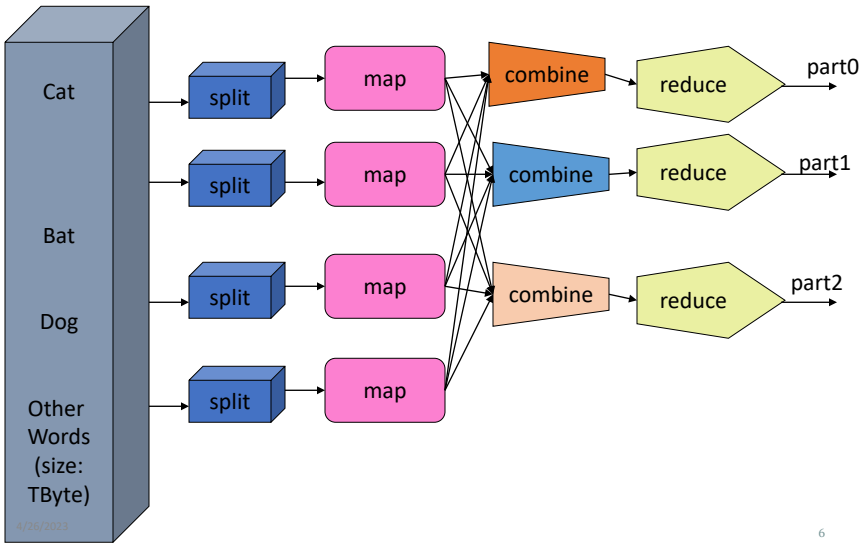
4

Data Characteristics

- Streaming data access
- Applications need streaming access to data
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- Write-once-read-many: a file once created, written and closed need not be changed – this assumption simplifies coherency
- A map-reduce application or web-crawler application fits perfectly with this model.

5

MapReduce



6

Architecture

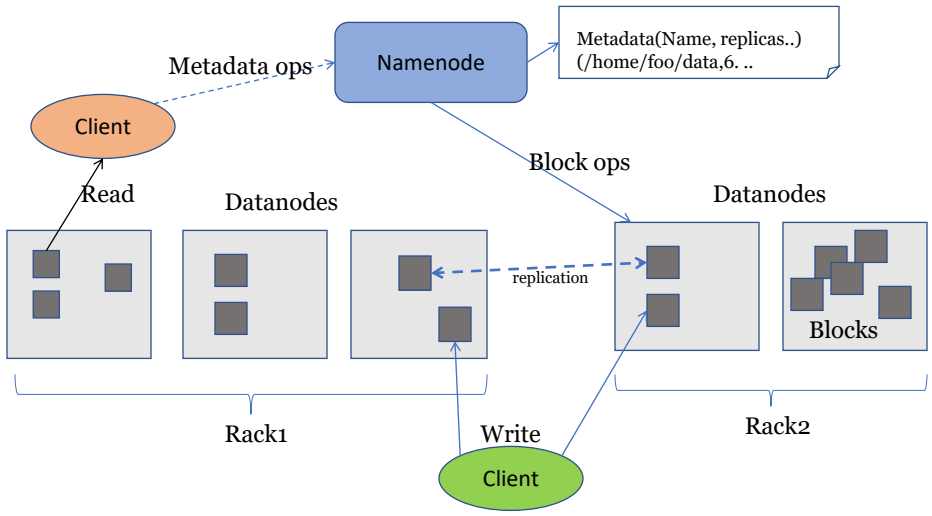
7

Namenode and Datanodes

- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes that they run on.
- HDFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in DataNodes.
- DataNodes: serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

8

HDFS Architecture



9

File system Namespace

- Hierarchical file system with directories and files
- Create, remove, move, rename etc.
- Namenode maintains the file system
- Any meta information changes to the file system recorded by the Namenode.
- An application can specify the number of replicas of the file needed: replication factor of the file. This information is stored in the Namenode.

10

Data Replication

- HDFS is designed to store very large files across machines in a large cluster.
- Each file is a sequence of blocks.
- All blocks in the file except the last are of the same size.
- Blocks are replicated for fault tolerance.
- Block size and replicas are configurable per file.
- The Namenode receives a Heartbeat and a BlockReport from each DataNode in the cluster.
- BlockReport contains all the blocks on a Datanode.

11

Replica Placement

- The placement of the replicas is critical to HDFS reliability and performance.
- Optimizing replica placement distinguishes HDFS from other distributed file systems.
- Rack-aware replica placement:
  - Goal: improve reliability, availability and network bandwidth utilization
  - Research topic
- Many racks, communication between racks are through switches.
- Network bandwidth between machines on the same rack is greater than those in different racks.
- Namenode determines the rack id for each DataNode.
- Replicas are typically placed on unique racks
  - Simple but non-optimal
  - Writes are expensive
  - Replication factor is 3
  - Another research topic?
- Replicas are placed: one on a node in a local rack, one on a different node in the local rack and one on a node in a different rack.
- 1/3 of the replica on a node, 2/3 on a rack and 1/3 distributed evenly across remaining racks.

12

Replica Selection

- Replica selection for READ operation: HDFS tries to minimize the bandwidth consumption and latency.
- If there is a replica on the Reader node then that is preferred.
- HDFS cluster may span multiple data centers: replica in the local data center is preferred over the remote one.

13

Safemode Startup

- On startup Namenode enters Safemode.
- Replication of data blocks do not occur in Safemode.
- Each DataNode checks in with Heartbeat and BlockReport.
- Namenode verifies that each block has acceptable number of replicas
- After a configurable percentage of safely replicated blocks check in with the Namenode, Namenode exits Safemode.
- It then makes the list of blocks that need to be replicated.
- Namenode then proceeds to replicate these blocks to other Datanodes.

14

Filesystem Metadata

- The HDFS namespace is stored by Namenode.
- Namenode uses a transaction log called the EditLog to record every change that occurs to the filesystem meta data.
  - For example, creating a new file.
  - Change replication factor of a file
  - EditLog is stored in the Namenode's local filesystem
- Entire filesystem namespace including mapping of blocks to files and file system properties is stored in a file FsImage. Stored in Namenode's local filesystem.

15

Namenode

- Keeps image of entire file system namespace and file Blockmap in memory.
- 4GB of local RAM is sufficient to support the above data structures that represent the huge number of files and directories.
- When the Namenode starts up it gets the FsImage and Editlog from its local file system, update FsImage with EditLog information and then stores a copy of the FsImage on the filesytstem as a checkpoint.
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.

16

Datanode

- A Datanode stores data in files in its local file system.
- Datanode has no knowledge about HDFS filesystem
- It stores each block of HDFS data in a separate file.
- Datanode does not create all files in the same directory.
- It uses heuristics to determine optimal number of files per directory and creates directories appropriately:
  - Research issue?
- When the filesystem starts up it generates a list of all HDFS blocks and send this report to Namenode: Blockreport.

17

Protocol

18

The Communication Protocol

- All HDFS communication protocols are layered on top of the TCP/IP protocol
- A client establishes a connection to a configurable TCP port on the Namenode machine. It talks ClientProtocol with the Namenode.
- The Datanodes talk to the Namenode using Datanode protocol.
- RPC abstraction wraps both ClientProtocol and Datanode protocol.
- Namenode is simply a server and never initiates a request; it only responds to RPC requests issued by DataNodes or clients.

19

Robustness

20

Objectives

- Primary objective of HDFS is to store data reliably in the presence of failures.
- Three common failures are: Namenode failure, Datanode failure and network partition.

21

DataNode failure and heartbeat

- A network partition can cause a subset of Datanodes to lose connectivity with the Namenode.
- Namenode detects this condition by the absence of a Heartbeat message.
- Namenode marks Datanodes without Hearbeat and does not send any IO requests to them.
- Any data registered to the failed Datanode is not available to the HDFS.
- Also the death of a Datanode may cause replication factor of some of the blocks to fall below their specified value.

22

Re-replication

- The necessity for re-replication may arise due to:
  - A Datanode may become unavailable,
  - A replica may become corrupted,
  - A hard disk on a Datanode may fail, or
  - The replication factor on the block may be increased.

23

Cluster Rebalancing

- HDFS architecture is compatible with data rebalancing schemes.
- A scheme might move data from one Datanode to another if the free space on a Datanode falls below a certain threshold.
- In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster.
- These types of data rebalancing are not yet implemented: research issue.

24

Data Integrity

- Consider a situation: a block of data fetched from Datanode arrives corrupted.
- This corruption may occur because of faults in a storage device, network faults, or buggy software.
- A HDFS client creates the checksum of every block of its file and stores it in hidden files in the HDFS namespace.
- When a clients retrieves the contents of file, it verifies that the corresponding checksums match.
- If does not match, the client can retrieve the block from a replica.

25

Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS.
- A corruption of these files can cause a HDFS instance to be non-functional.
- For this reason, a Namenode can be configured to maintain multiple copies of the FsImage and EditLog.
- Multiple copies of the FsImage and EditLog files are updated synchronously.
- Meta-data is not data-intensive.
- The Namenode could be single point failure => secondary namenode

26

Data Organization

27

Data Blocks

- HDFS support write-once-read-many with reads at streaming speeds.
- A typical block size is 64MB (or even 128 MB).
- A file is chopped into 64MB chunks and stored.

28

Staging

- A client request to create a file does not reach Namenode immediately.
- HDFS client caches the data into a temporary file. When the data reached a HDFS block size the client contacts the Namenode.
- Namenode inserts the filename into its hierarchy and allocates a data block for it.
- The Namenode responds to the client with the identity of the Datanode and the destination of the replicas (Datanodes) for the block.
- Then the client flushes it from its local memory.

29

Staging (contd.)

- The client sends a message that the file is closed.
- Namenode proceeds to commit the file for creation operation into the persistent store.
- If the Namenode dies before file is closed, the file is lost.
- This client side caching is required to avoid network congestion; also it has precedence is AFS (Andrew file system).

30

Replication Pipelining

- When the client receives response from Namenode, it flushes its block in small pieces (4K) to the first replica, that in turn copies it to the next replica and so on.
- Thus data is pipelined from Datanode to the next.

31

API (Accessibility)

32

Application Programming Interface

- HDFS provides Java API for application to use.
- Python access is also used in many applications.
- A C language wrapper for Java API is also available.
- A HTTP browser can be used to browse the files of a HDFS instance.

33

FS Shell, Admin and Browser Interface

- HDFS organizes its data in files and directories.
- It provides a command line interface called the FS shell that lets the user interact with data in the HDFS.
- The syntax of the commands is similar to bash and csh.
- Example: to create a directory /foodir

/bin/hadoop dfs –mkdir /foodir

- There is also DFSAdmin interface available
- Browser interface is also available to view the namespace.

34

Space Reclamation

- When a file is deleted by a client, HDFS renames file to a file in be the /trash directory for a configurable amount of time.
- A client can request for an undelete in this allowed time.
- After the specified time the file is deleted and the space is reclaimed.
- When the replication factor is reduced, the Namenode selects excess replicas that can be deleted.
- Next heartbeat(?) transfers this information to the Datanode that clears the blocks for use.

35

Summary

- We discussed the features of the Hadoop File System, a peta-scale file system to handle big-data sets.
- What discussed: Architecture, Protocol, API, etc.
- Missing element: Implementation
  - The Hadoop file system (internals)
  - An implementation of an instance of the HDFS (for use by applications such as web crawlers).

36

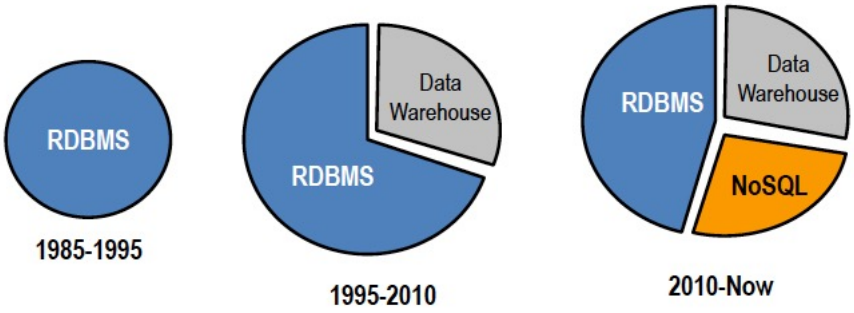
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

NoSQL part 1

Viet-Trung Tran
School of Information and Communication Technology

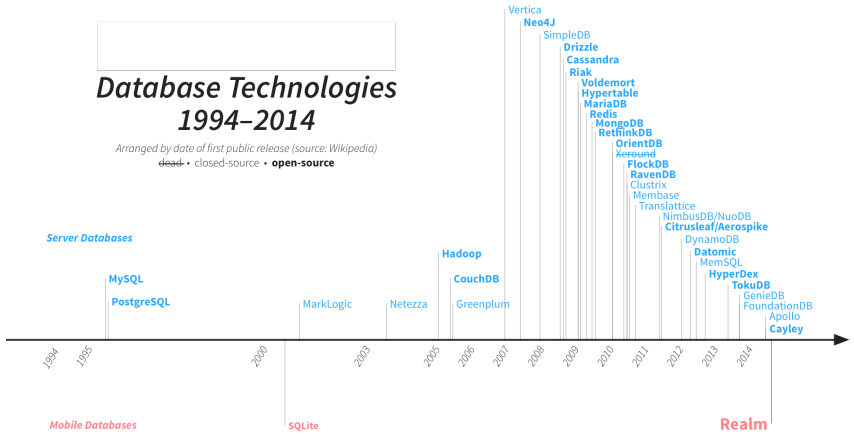
1

Eras of Databases



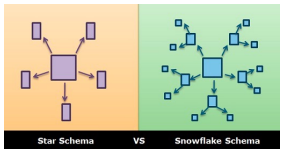
2

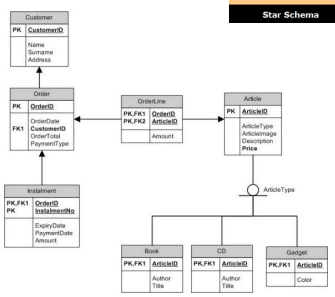
Eras of Databases



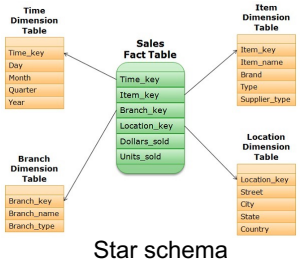
3

Before NoSQL

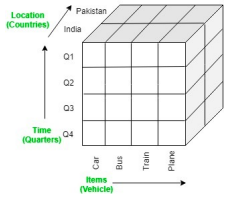




OLTP



Star schema

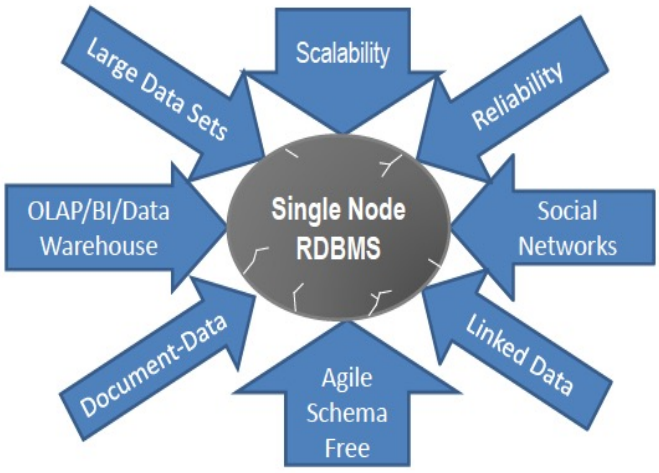


OLAP cube

4

4

RDBMS: one size fits all needs



5

5

ICDE 2005 conference

"One Size Fits All": An Idea Whose Time Has Come and Gone

Authors: Michael Stonebraker StreamBase Systems, Inc.
Ugur Cetintemel Brown University and StreamBase Systems, Inc.

2005 Article

Published in:
· Proceeding
ICDE '05 Proceedings of the 21st International Conference on Data Engineering
Pages 2-11

April 05 - 08, 2005
IEEE Computer Society Washington, DC, USA ©2005
table of contents ISBN:0-7695-2285-8 doi>10.1109/ICDE.2005.1

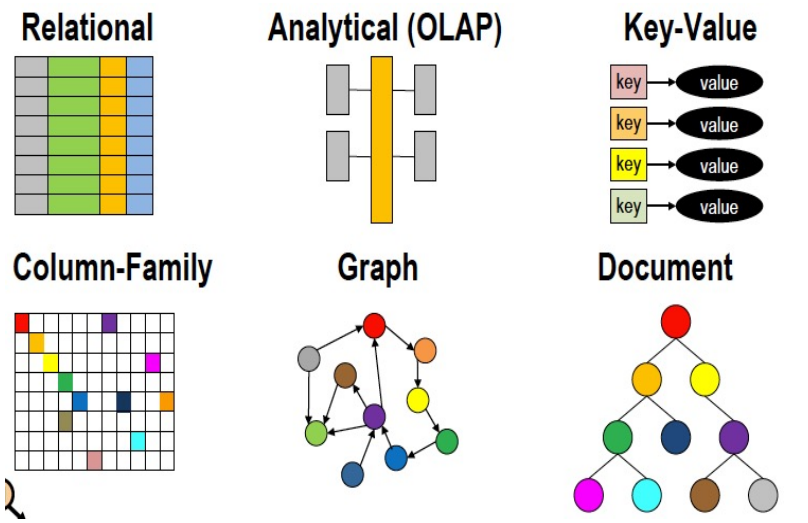
Bibliometrics
· Citation Count: 73
· Downloads (cumulative): 0
· Downloads (12 Months): 0
· Downloads (6 Weeks): 0

The last 25 years of commercial DBMS development can be summed up in a single phrase: "one size fits all". This phrase refers to the fact that **the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric application**s with widely varying characteristics and requirements. In this paper, we argue that this concept is no longer applicable to the database market, and that the commercial world will fracture into a collection of independent database engines ...

6

6

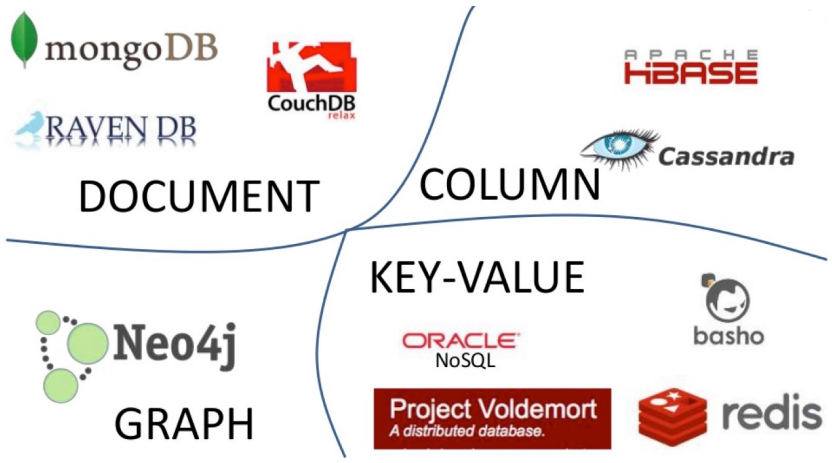
After is NoSQL



7

7

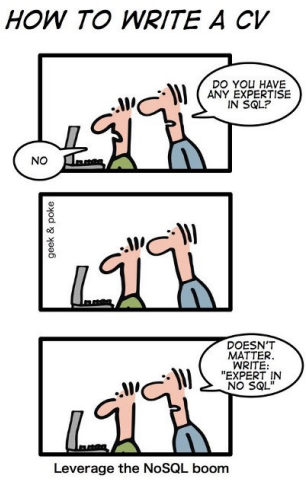
NoSQL landscape



8

8

How to write a CV



Leverage the NoSQL boom

9

9

Why NoSQL

- Web applications have different needs
  - Horizontal scalability – lowers cost
  - Geographically distributed
  - Elasticity
  - Schema less, flexible schema for semi-structured data
  - Easier for developers
  - Heterogeneous data storage
  - High Availability/Disaster Recovery
- Web applications do not always need
  - Transaction
  - Strong consistency
  - Complex queries

10

10

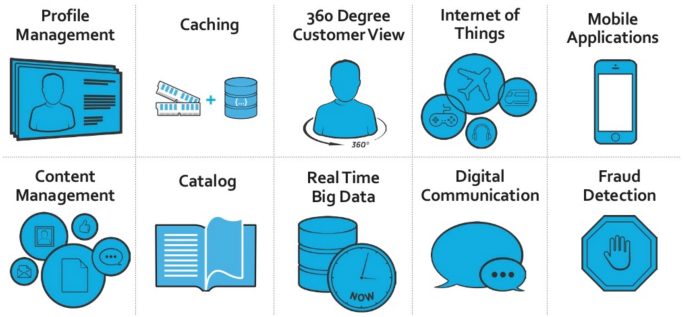
SQL vs NoSQL

SQL	NoSQL
Gigabytes to Terabytes	Petabytes(1kTB) to Exabytes(1kPB) to Zetabytes(1kEB)
Centralized	Distributed
Structured	Semi structured and Unstructured
Structured Query Language	No declarative query language
Stable Data Model	Schema less
Complex Relationships	Less complex relationships
ACID Property	Eventual Consistency
Transaction is priority	High Availability, High Scalability
Joins Tables	Embedded structures

11

NoSQL use cases

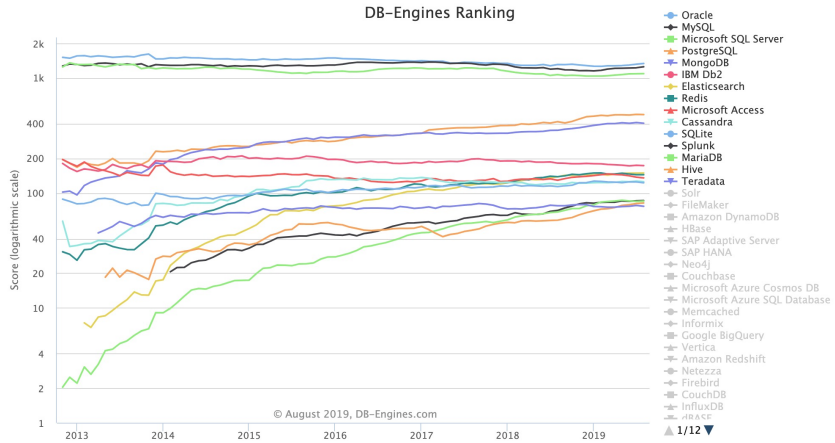
- Massive data volume at scale (Big volume)
  - Google, Amazon, Yahoo, Facebook – 10-100K servers
- Extreme query workload (Big velocity)
- High availability
- Flexible, schema evolution



12

12

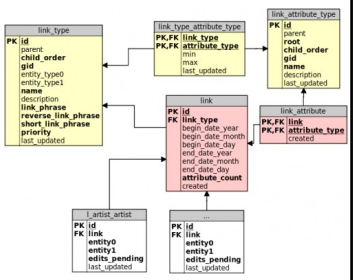
DB engines ranking according to their popularity (2019)



13

Relational data model revisited

- Data is usually stored in row by row manner (row store)
- Standardized query language (SQL)
- Data model defined **before** you add data
- Joins merge data from multiple tables
  - Results are tables
- **Pros:** Mature ACID transactions with fine-grain security controls, widely used
- **Cons:** Requires up front data modeling, does not scale well



Oracle, MySQL, PostgreSQL, Microsoft SQL Server, IBM DB/2

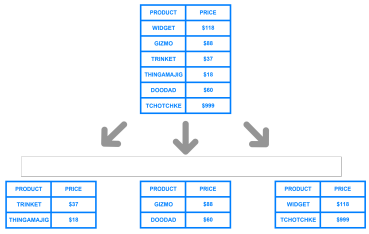
14

14

Key/value data model

- Simple key/value interface
  - GET, PUT, DELETE
- Value can contain any kind of data
- Super fast and easy to scale (no joins)
- Examples
  - Berkley DB, Memcache, DynamoDB, Redis, Riak

key	value
firstName	Bugs
lastName	Bunny
location	Earth

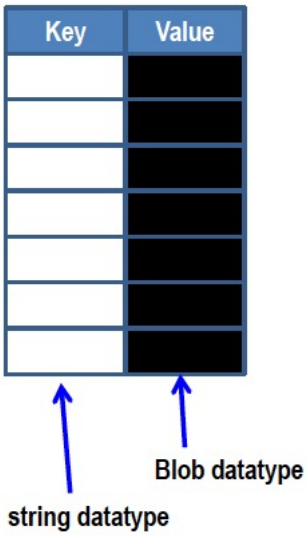


15

15

Key/value vs. table

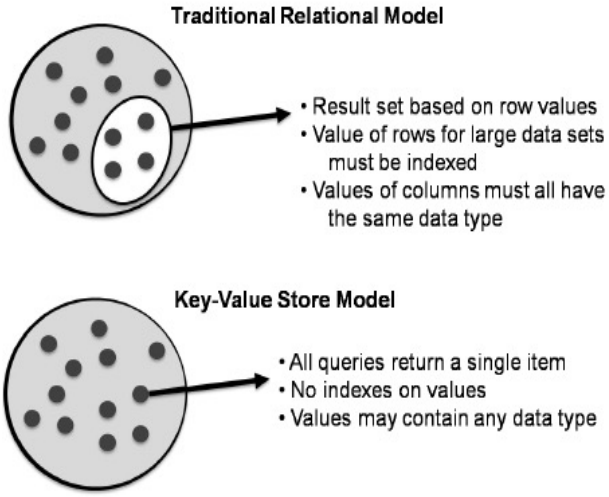
- A table with two columns and a simple interface
  - Add a key-value
  - For this key, give me the value
  - Delete a key



16

16

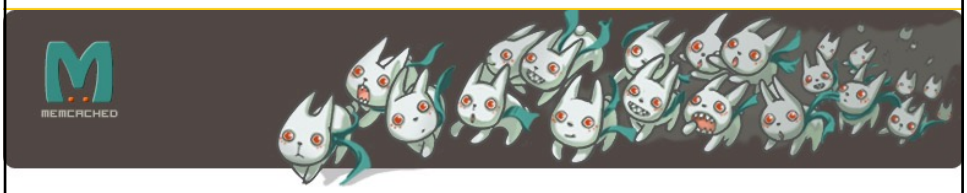
Key/value vs. Relational data model



17

17

Memcached



- Open source in-memory key-value caching system
- Make effective use of RAM on many distributed web servers
- Designed to speed up dynamic web applications by alleviating database load
  - Simple interface for highly distributed RAM caches
  - 30ms read times typical
- Designed for quick deployment, ease of development
- APIs in many languages

18

18

Redis

- Open source in-memory key-value store with optional durability
- Focus on high speed reads and writes of common data structures to RAM
- Allows simple lists, sets and hashes to be stored within the value and manipulated
- Many features that developers like expiration, transactions, pub/sub, partitioning

19

19

Amazon DynamoDB

- Scalable key-value store
- Fastest growing product in Amazon's history
- Focus on throughput on storage and predictable read and write times
- Strong integration with S3 and Elastic MapReduce

20

20

Riak

- Open source distributed key-value store with support and commercial versions by Basho
- A "Dynamo-inspired" database
- Focus on availability, fault-tolerance, operational simplicity and scalability
- Support for replication and auto-sharding and rebalancing on failures
- Support for MapReduce, fulltext search and secondary indexes of value tags
- Written in ERLANG

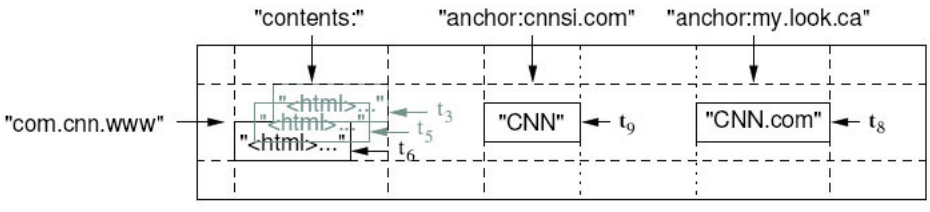
riak

21

21

Column family store

- Dynamic schema, column-oriented data model
- Sparse, distributed persistent multi-dimensional sorted map
- (row, column (family), timestamp) -> cell contents

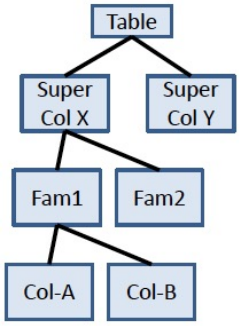


22

22

Column families

- Group columns into "Column families"
- Group column families into "Super-Columns"
- Be able to query all columns with a family or super family
- Similar data grouped together to improve speed

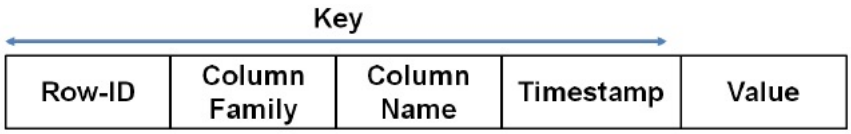


23

23

Column family data model vs. relational

- Sparse matrix, preserve table structure
  - One row could have millions of columns but can be very sparse
- Hybrid row/column stores
- Number of columns is extendible
  - New columns to be inserted without doing an "alter table"

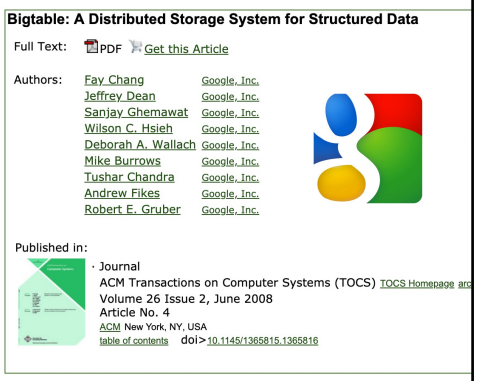


24

24

Bigtable

- ACM TOCS 2008
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance



Bigtable: A Distributed Storage System for Structured Data

Full Text: PDF Get this Article

Authors:

Fay Chang	Google, Inc.
Jeffrey Dean	Google, Inc.
Sanjay Ghemawat	Google, Inc.
Wilson C. Hsieh	Google, Inc.
Deborah A. Wallach	Google, Inc.
Mike Burrows	Google, Inc.
Tushar Chandra	Google, Inc.
Andrew Fikes	Google, Inc.
Robert E. Gruber	Google, Inc.

Published in:

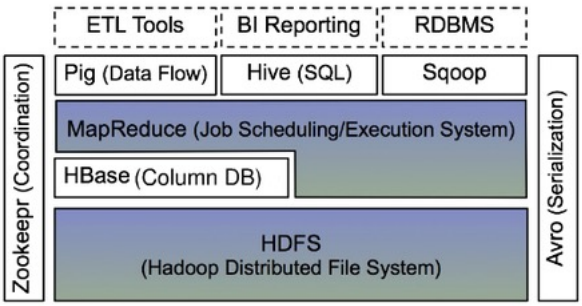
Journal
ACM Transactions on Computer Systems (TOCS) TOCS Homepage arc
Volume 26 Issue 2, June 2008
Article No. 4
ACM New York, NY, USA
table of contents doi>10.1145/1365815.1365816

25

25

Apache Hbase

- Open-source Bigtable, written in JAVA
- Part of Apache Hadoop project



APACHE HBASE

26

26

Apache Cassandra

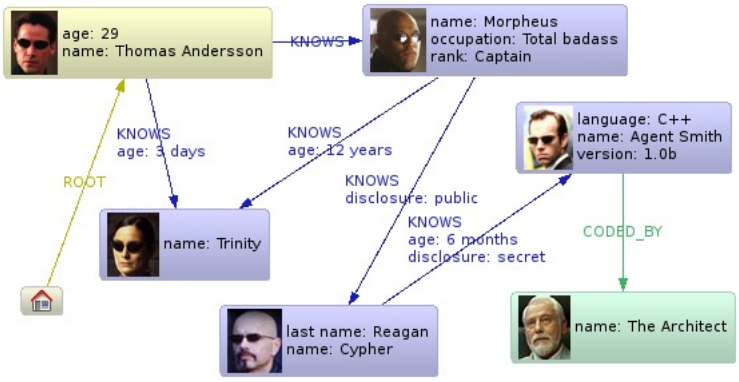
- Apache open source column family database
- Supported by DataStax
- Peer-to-peer distribution model
- Strong reputation for linear scale out (millions of writes/second)
- Written in Java and works well with HDFS and MapReduce

27

27

Graph data model

- Core abstractions: Nodes, Relationships, Properties on both



28

28

Graph database store

- A database stored data in an explicitly graph structure
- Each node knows its adjacent nodes
- Queries are really graph traversals

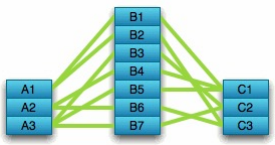
sones InfiniteGraph Powered by Objectivity *dex Neo4j OrientDB RDF AllegroGraph HYPERGRAPHDB

29

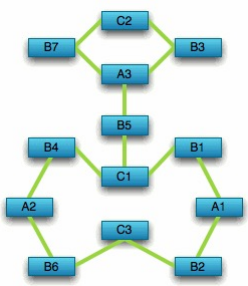
29

Compared to Relational Databases

Optimized for aggregation



Optimized for connections

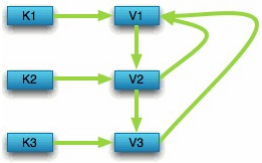


30

Compared to Key Value Stores

Optimized for simple look-ups

Optimized for traversing connected data

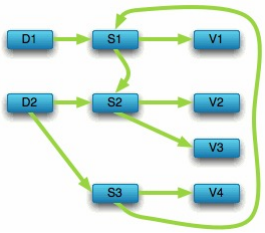


31

Compared to Document Stores

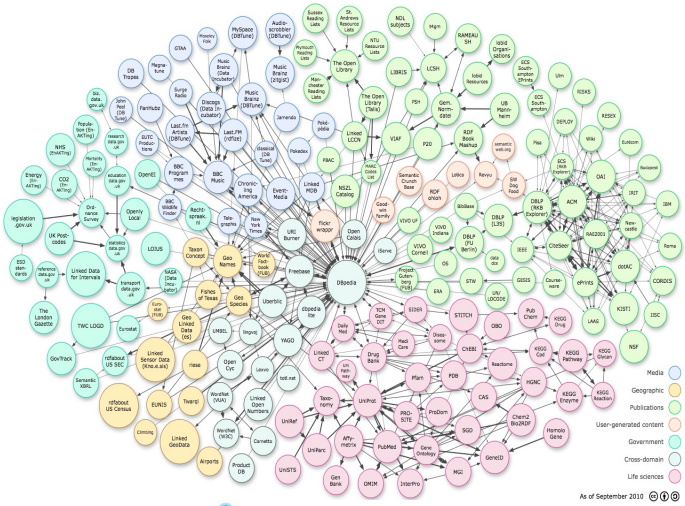
Optimized for “trees” of data

Optimized for seeing the forest and the trees, and the branches, and the trunks



32

Linking open data



LINKINGOPENDATA
W3C SWEO Community Project

33

33

Neo4j

- Graph database designed to be easy to use by Java developers
- Disk-based (not just RAM)
- Full ACID
- High Availability (with Enterprise Edition)
- 32 Billion Nodes, 32 Billion Relationships, 64 Billion Properties
- Embedded java library
- REST API

Neo4j

34

34

Document store

- Documents, not value, not tables
- JSON or XML formats
- Document is identified by ID
- Allow indexing on properties

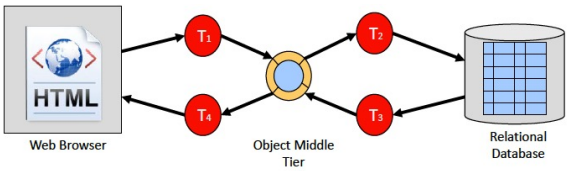
```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```

35

35

Relational data mapping

- T1–HTML into Objects
- T2–Objects into SQL Tables
- T3–Tables into Objects
- T4–Objects into HTML

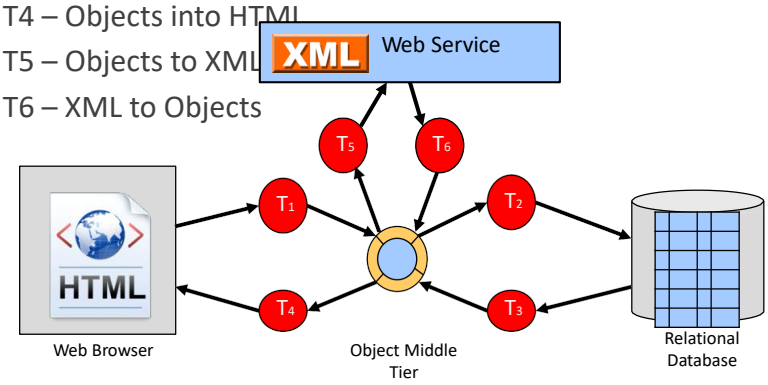


36

36

Web Service in the middle

- T1 – HTML into Java Objects
- T2 – Java Objects into SQL Tables
- T3 – Tables into Objects
- T4 – Objects into HTML
- T5 – Objects to XML
- T6 – XML to Objects



37

37

Discussion

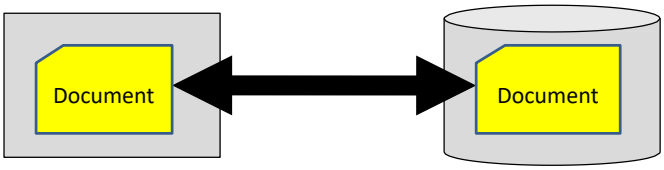
- Object-relational mapping has become one of the most complex components of building applications today
  - Java Hibernate Framework
  - JPA
- To avoid complexity is to keep your architecture very simple

38

38

Document mapping

- Documents in the database
- Documents in the application
- No object middle tier
- No "shredding"
- No reassembly
- Simple!



39

39

MongoDB

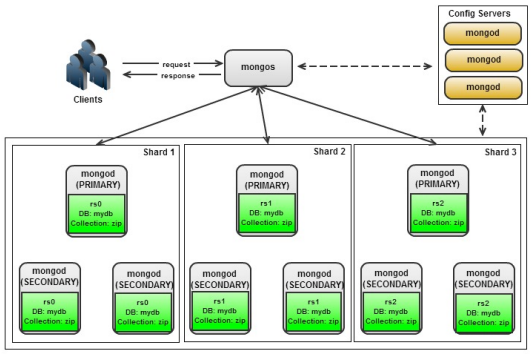
- Open Source JSON data store created by 10gen
- Master-slave scale out model
- Strong developer community
- Sharding built-in, automatic
- Implemented in C++ with many APIs (C++, JavaScript, Java, Perl, Python etc.)

40

40

MongoDB architecture

- Replica set
  - Copies of the data on each node
  - Data safety
  - High availability
  - Disaster recovery
  - Maintenance
  - Read scaling
- Sharding
  - "Partitions" of the data
  - Horizontal scale

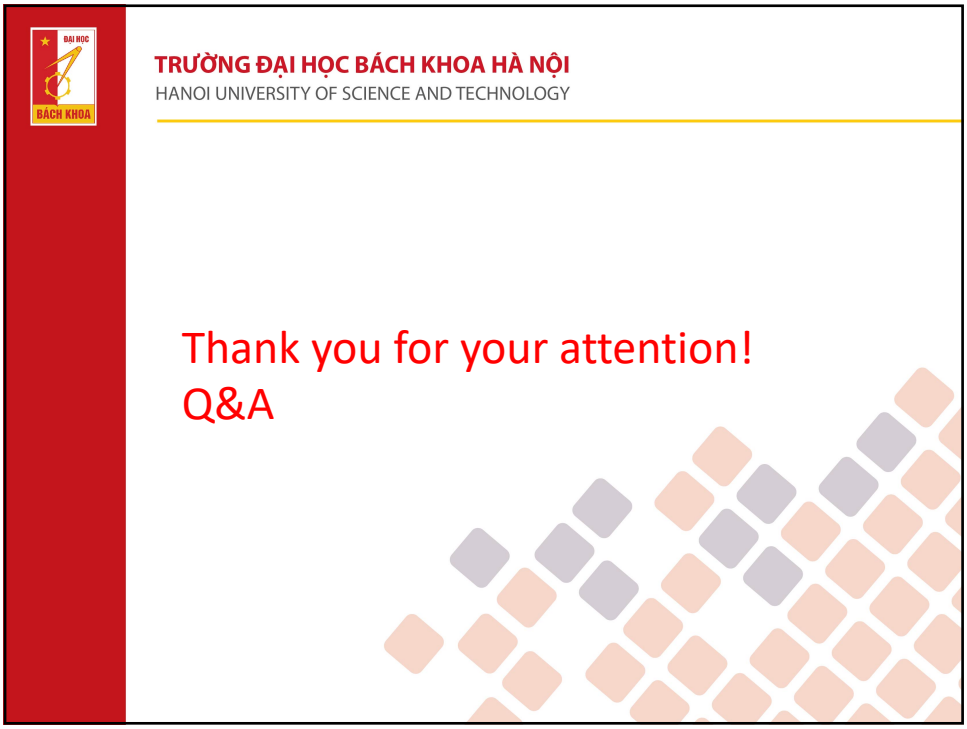


41

Apache CouchDB

- Apache project
- Open source JSON data store
- Written in ERLANG
- RESTful JSON API
- B-Tree based indexing, shadowing b-tree versioning
- ACID fully supported
- View model
- Data compaction
- Security

42


TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
Thank you for your attention!
Q&A

43

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

CAP theorem

Lecturer: Thanh-Chung Dao
Slides by Viet-Trung Tran
School of Information and Communication Technology

1

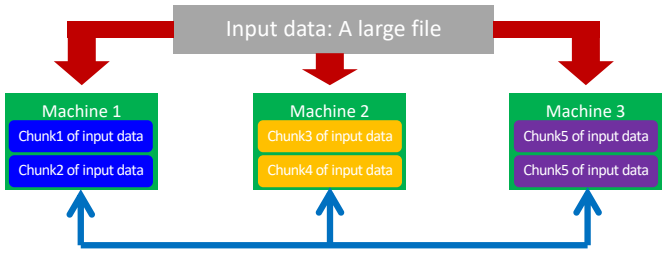
Scaling Traditional Databases

- Traditional RDBMSs can be either scaled:
  - Vertically (or Up)
    - Can be achieved by hardware upgrades (e.g., faster CPU, more memory, or larger disk)
    - Limited by the amount of CPU, RAM and disk that can be configured on a single machine
  - Horizontally (or Out)
    - Can be achieved by adding more machines
    - Requires database sharding and probably replication
    - Limited by the Read-to-Write ratio and communication overhead

2

Data sharding

- Data is typically sharded (or striped) to allow for concurrent/parallel accesses
- Will it scale for complex query processing?

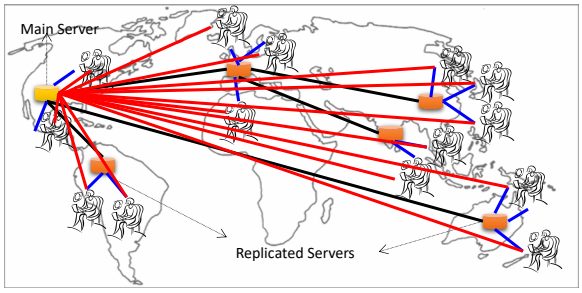


E.g., Chunks 1, 3 and 5 can be accessed in parallel

3

Data replicating

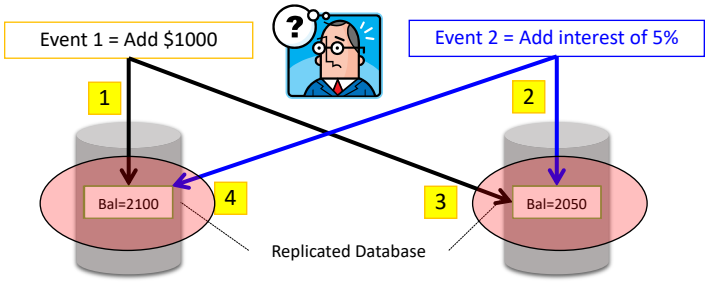
- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability



4

But, Consistency Becomes a Challenge

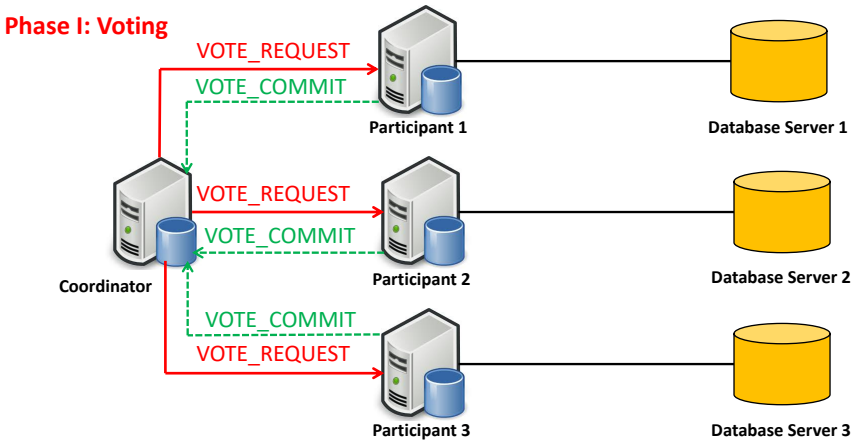
- An example:
  - In an e-commerce application, the bank database has been replicated across two servers
  - Maintaining consistency of replicated data is a challenge



5

The Two-Phase Commit Protocol

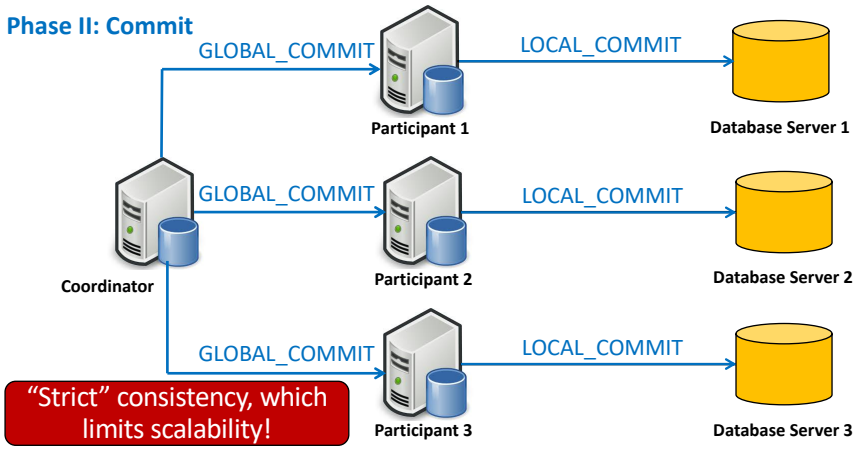
- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



6

The Two-Phase Commit Protocol

- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



7

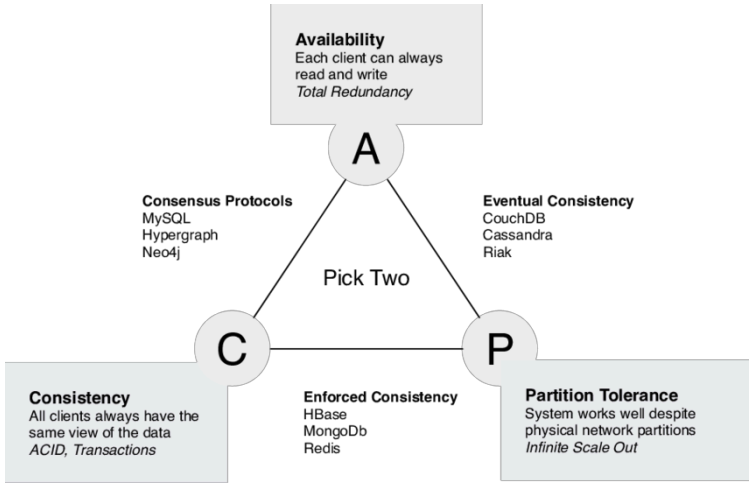
The CAP Theorem

- The limitations of distributed databases can be described in the so called the CAP theorem
  - Consistency: every node always sees the same data at any given instance (i.e., strict consistency)
  - Availability: the system continues to operate, even if nodes in a cluster crash, or some hardware or software parts are down due to upgrades
  - Partition Tolerance: the system continues to operate in the presence of network partitions

CAP theorem: any distributed database with shared data, can have *at most two* of the three desirable properties, C, A or P. These are trade-offs involved in distributed system by Eric Brewer in PODC 2000.

8

CAP Theorem

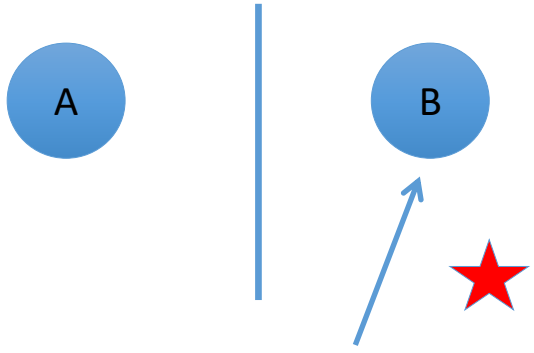


9

9

CAP Theorem: Proof

- A simple proof using two nodes:

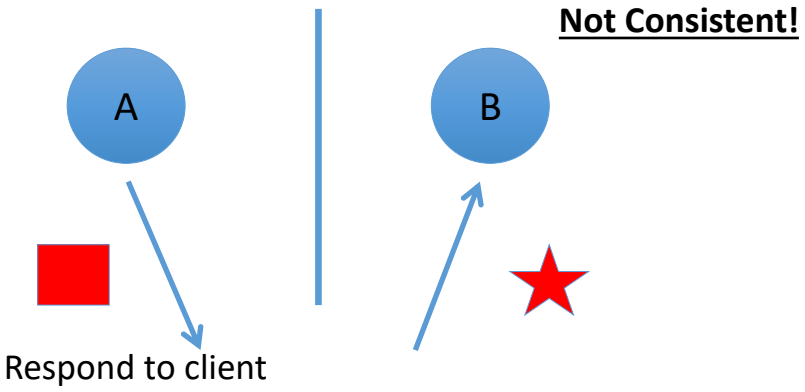


10

10

CAP Theorem: Proof

- A simple proof using two nodes:

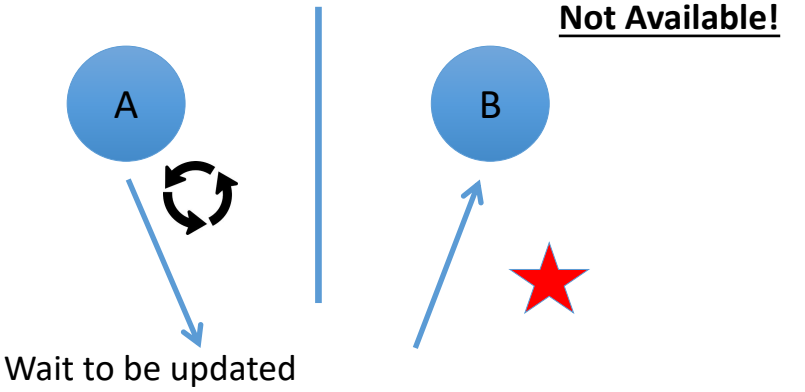


11

11

CAP Theorem: Proof

- A simple proof using two nodes:

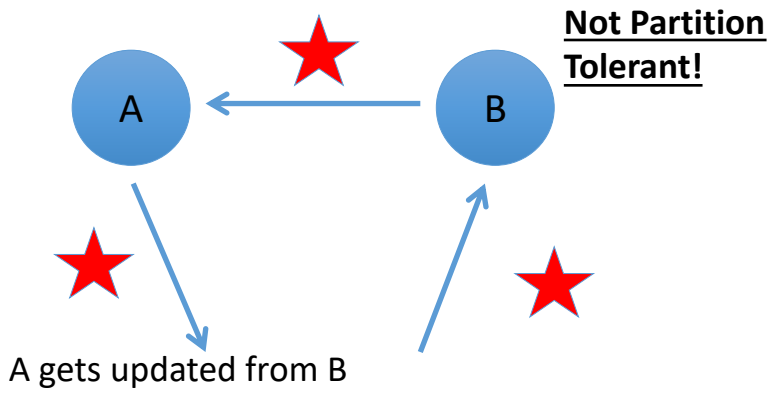


12

12

CAP Theorem: Proof

- A simple proof using two nodes:



13

13

Scalability of relational databases

- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which unfortunately is **impossible** …

14

14

Large-Scale Databases

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
  - A few minutes of downtime means lost revenue
- When horizontally scaling databases to 1000s of machines, the likelihood of a node or a network failure increases tremendously
- Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to sacrifice "strict" Consistency (implied by the CAP theorem)

15

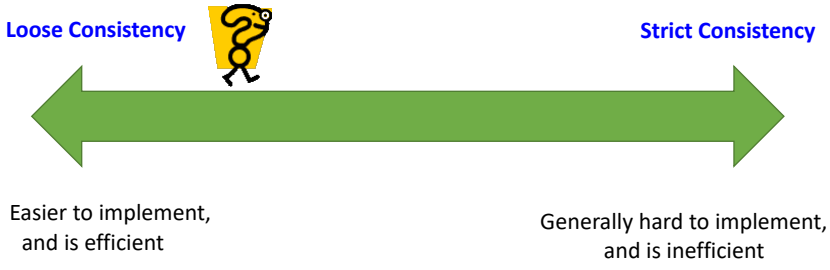
Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency depends on your application

16

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency depends on your application



Easier to implement, and is efficient

Generally hard to implement, and is inefficient

17

The BASE Properties

- The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions
- This resulted in databases with relaxed ACID guarantees
- In particular, such databases apply the BASE properties:
  - Basically Available: the system guarantees Availability
  - Soft-State: the state of the system may change over time
  - Eventual Consistency: the system will eventually become consistent

18

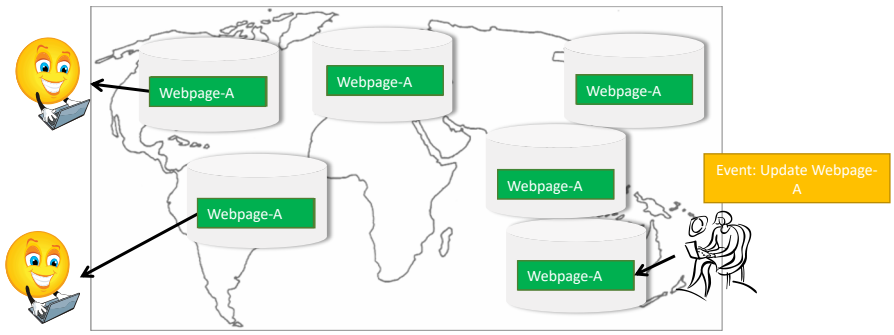
Eventual Consistency

- A database is termed as Eventually Consistent if:
  - All replicas will gradually become consistent in the absence of new updates

19

Eventual Consistency

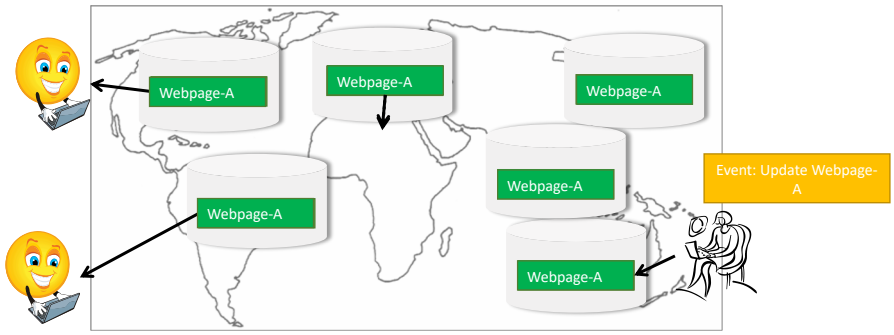
- A database is termed as Eventually Consistent if:
  - All replicas will gradually become consistent in the absence of new updates



20

Read-after-write consistency (eg. Amazon S3)

- But, what if the client accesses the data from different replicas?



Protocols like Read Your Own Writes (RYOW) can be applied!

21

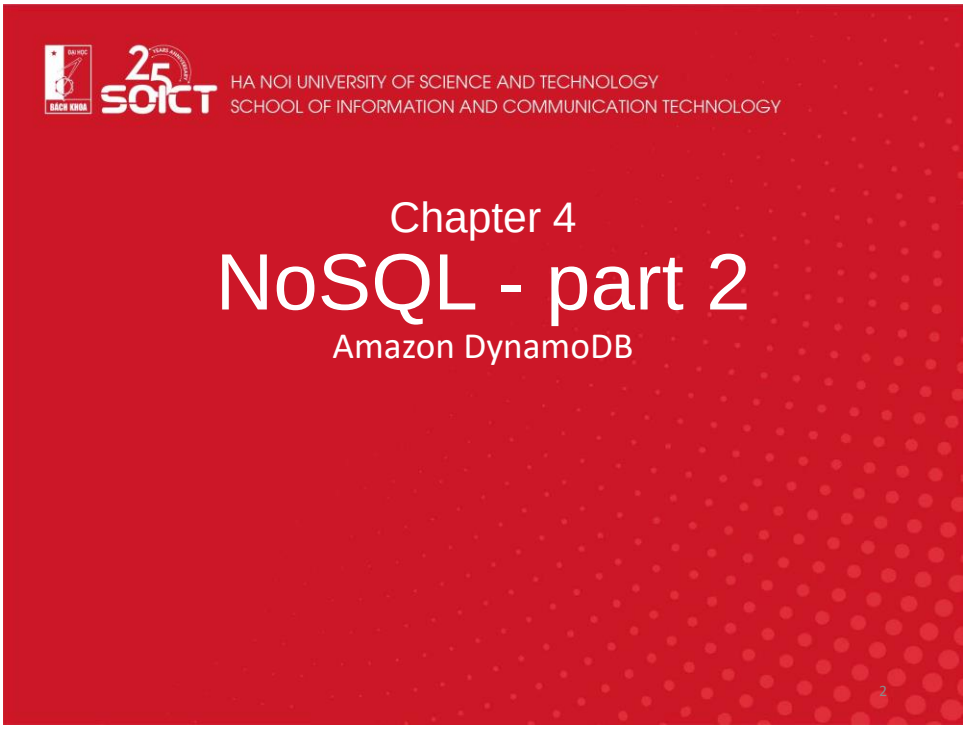
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Thank you for your attention!
Q&A

22


ĐẠI HỌC
BÁCH KHOA
25 YEARS ANNIVERSARY
SOICT
HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

1


HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY
Chapter 4
NoSQL - part 2
Amazon DynamoDB
2

2

Amazon DynamoDB

- Simple interface
  - Key/value store
- Sacrifice strong consistency for availability
- “always writeable” data store
  - no updates are rejected due to failures or concurrent writes
- Conflict resolution is executed during read instead of write
- An infrastructure within a single administrative domain where all nodes are assumed to be trusted.

amazon
DynamoDB

Design consideration

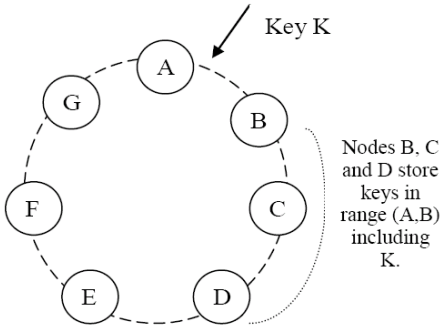
- Incremental scalability
- Symmetry
  - Every node in Dynamo should have the same set of responsibilities as its peers.
- Decentralization
  - In the past, centralized control has resulted in outages and the goal is to avoid it as much as possible
- Heterogeneity
  - This is essential in adding new nodes with higher capacity without having to upgrade all hosts at once

System architecture

- Partitioning
- High Availability for writes
- Handling temporary failures
- Recovering from permanent failures
- Membership and failure detection

Partition algorithm

- Consistent hashing: the output range of a hash function is treated as a fixed circular space or “ring”
- DynamoDB is a zero-hop DHT



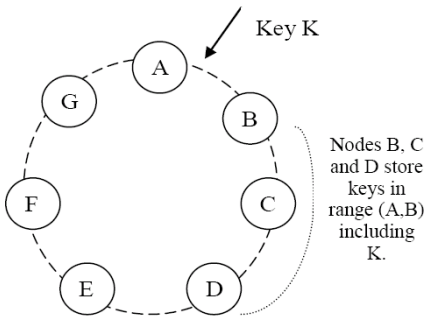
Grand challenge: every nodes must maintain an up-to-date view of the ring! How?

Virtual nodes

- Each node can be responsible for more than one virtual node.
  - Each physical node has multiple virtual nodes
  - More powerful machines have more virtual nodes
  - Distribute virtual nodes across the ring
- Advantages of using virtual nodes
  - If a node becomes unavailable, the load handled by this node is evenly dispersed across the remaining available nodes.
  - When a node becomes available again, or a new node is added to the system, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes.
  - The number of virtual nodes that a node is responsible can decided based on its capacity, accounting for heterogeneity in the physical infrastructure.

Replication

- Each data item is replicated at N hosts.
  - N is the “preference list”: The list of nodes that is responsible for storing a particular key.

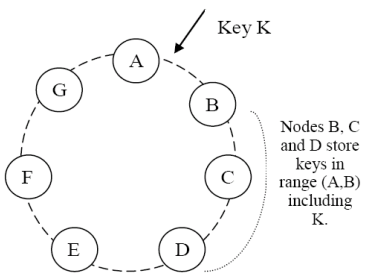


Quorum

- N: total number of replicas per each key/value pair
- R: minimum number of nodes that must participate in a sucessful reading
- W: minimum number of nodes that must participate in a sucessful writing
- Quorum-like system
  - R + W > N
  - In this model, the latency of a get (or put) operation is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.

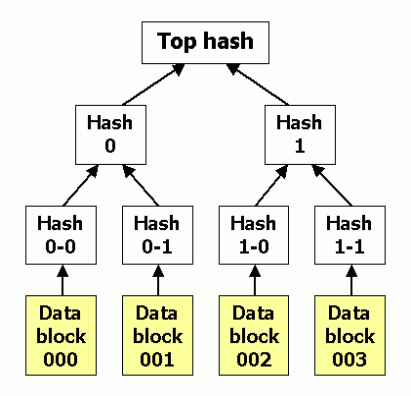
Temporary failures: Sloppy quorum and hinted handoff

- Assume N = 3. When B is temporarily down or unreachable during a write, send replica to E.
- E is hinted that the replica belongs to B and it will deliver to B when B is recovered.
- Again: “always writeable”



Replica synchronization

- Merkle tree
  - a hash tree where leaves are hashes of the values of individual keys
  - Parent nodes higher in the tree are hashes of their respective children
- Advantage of Merkle tree
  - Each branch of the tree can be checked independently without requiring nodes to download the entire tree
  - Help in reducing the amount of data that needs to be transferred while checking for inconsistencies among replicas



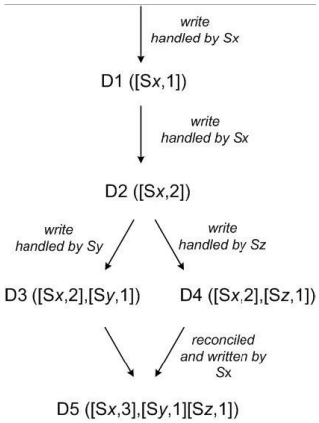
Data versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- Key Challenge: distinct version sub-histories - need to be reconciled.
  - Solution: uses vector clocks in order to capture causality between different versions of the same object.

Vector clock

- A vector clock is a list of (node, counter) pairs.
- Every version of every object is associated with one vector clock.
- 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.

Vector clock example



When the number of (node, counter) pairs in the vector clock reaches a threshold (say 10), the oldest pair is removed from the clock.

Technical summary

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.

DynamoDB sum up

- Dynamo is a highly available and scalable data store for Amazon.com's e-commerce platform.
- Dynamo has been successful in handling server failures, data center failures and network partitions.
- Dynamo is incrementally scalable and allows service owners to scale up and down based on their current request load.
- Dynamo allows service owners to customize their storage system by allowing them to tune the parameters N, R,and W.

References

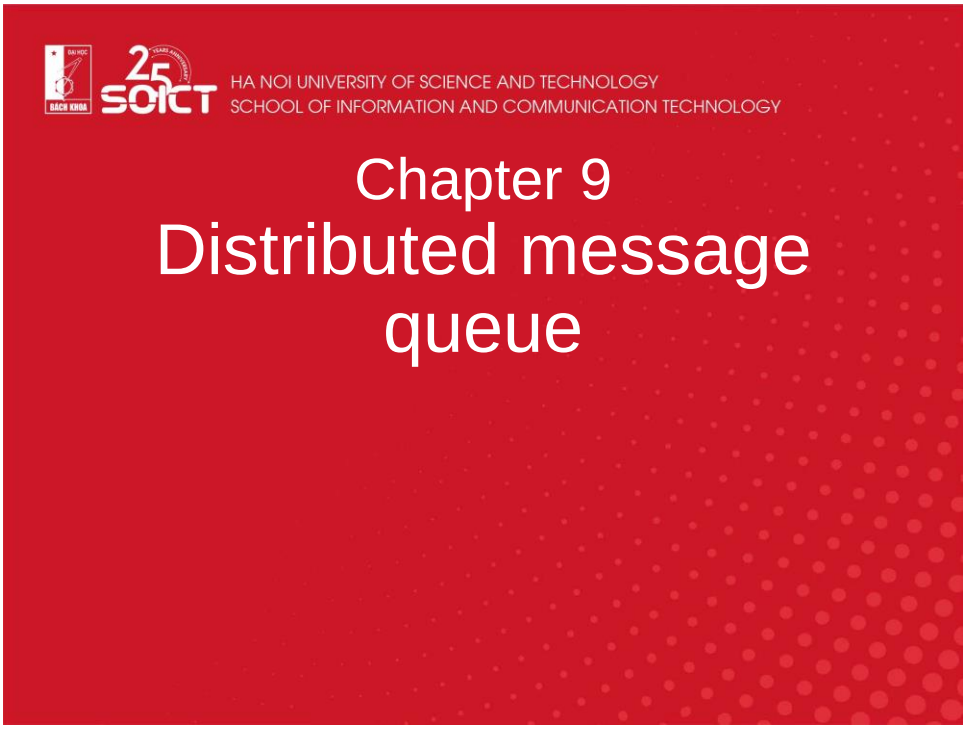
- Sivasubramanian, Swaminathan. "Amazon dynamoDB: a seamlessly scalable non-relational database service." *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012.
- Stoica, Ion, et al. "Chord: A scalable peer-to-peer lookup service for internet applications." *ACM SIGCOMM Computer Communication Review* 31.4 (2001): 149-160.

17

18

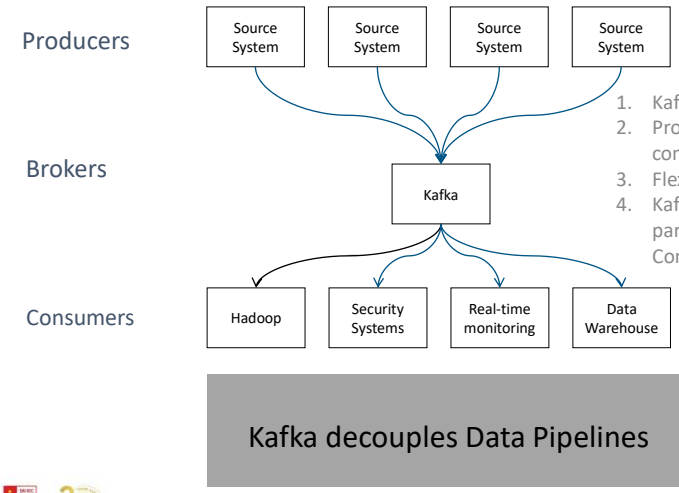

ĐẠI HỌC
BÁCH KHOA
25 YEARS ANNIVERSARY
SOICT
HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

1


HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY
Chapter 9
Distributed message queue

2

Why Kafka



1. Kafka decouple data streams
2. Producers don't know about consumers
3. Flexible message consumption
4. Kafka broker delegates log partition offset (location) to Consumers (clients)

Kafka decouples Data Pipelines

What is Kafka?

- Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
  - Publish and Subscribe to streams of records
  - Fault tolerant storage
    - Replicates Topic Log Partitions to multiple servers
  - Process records as they occur
  - Fast, efficient IO, batching, compression, and more
- Used to decouple data streams
- Kafka is often used instead of JMS, RabbitMQ and AMQP
  - higher throughput, reliability and replication

Kafka possibility

- Build real-time streaming applications that react to streams
  - Feeding data to do real-time analytic systems
  - Transform, react, aggregate, join real-time data flows (eg. Metrics gathering)
  - Feed events to CEP for complex event processing
  - Feeding of high-latency daily or hourly data analysis into Spark, Hadoop, etc.
    - (eg. External commit log for distributed systems. Replicated data between nodes, re-sync for nodes to restore state)
  - Up to date dashboards and summaries
- Build real-time streaming data pipe-lines
  - Enable in-memory microservices (actors, Akka, Vert.x, Qbit, RxJava)

Kafka adoption

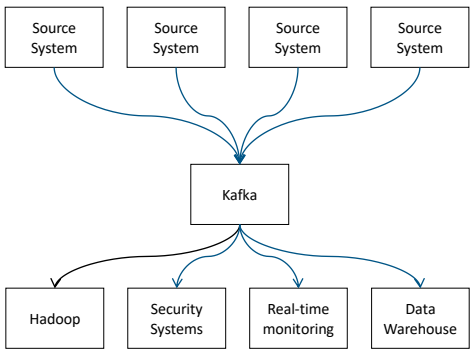
- 1/3 of all Fortune 500 companies
- Top ten travel companies, 7 of ten top banks, 8 of ten top insurance companies, 9 of ten top telecom companies
- LinkedIn, Microsoft and Netflix process 1 billion messages a day with Kafka
- Real-time streams of data, used to collect big data or to do real time analysis (or both)

Why is Kafka popular?

- Great performance
- Operational simplicity, easy to setup and use, easy to reason
- Stable, reliable durability,
- Flexible publish-subscribe/queue (scales with N-number of consumer groups),
- Robust replication,
- Producer tunable consistency guarantees,
- Ordering preserved at shard level (topic partition)
- Works well with systems that have data streams to process, aggregate, transform & load into other stores

Concepts

Basic Kafka Concepts

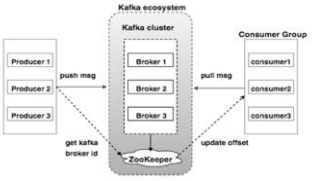


Key terminology

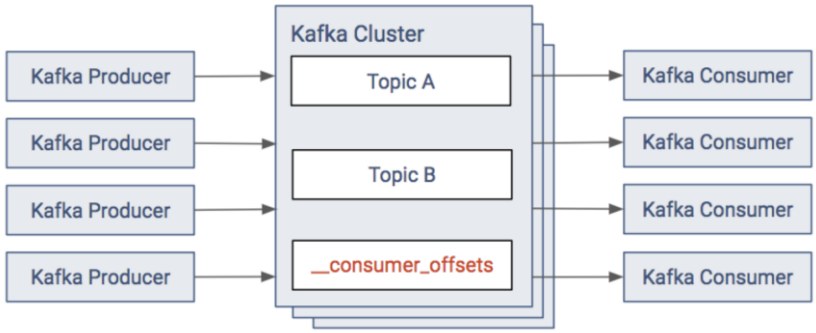
- Kafka maintains feeds of messages in categories called topics.
  - a stream of records (“/orders”, “/user-signups”), feed name
  - Log topic storage on disk
  - Partition / Segments (parts of Topic Log)
- Records have a key (optional), value and timestamp; Immutable
- Processes that publish messages to a Kafka topic are called **producers**.
- Processes that subscribe to topics and process the feed of published messages are called **consumers**.
- Kafka is run as a cluster comprised of one or more servers each of which is called a **broker**.

Kafka architecture

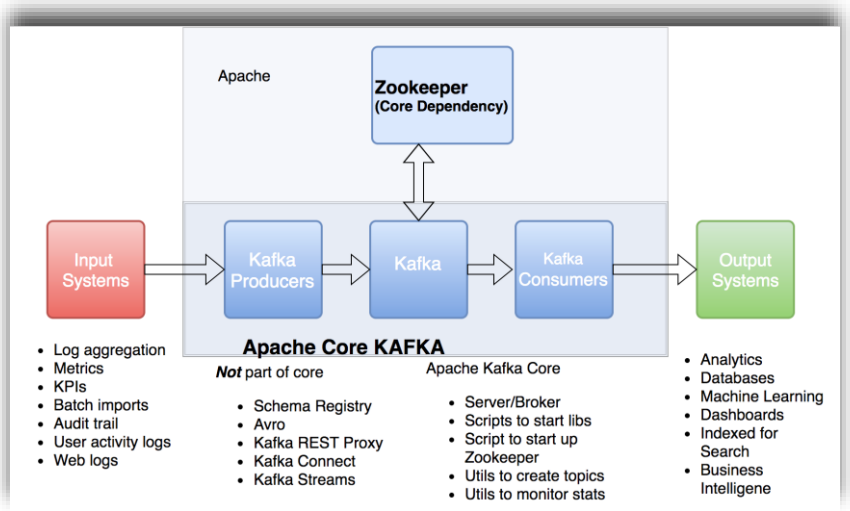
- Kafka cluster consists of mutliple brokers and zookeeper
- Communication between all components is done via a high performance simple binary API over TCP protocol
- Zookeeper provides in-sync view of Kafka Cluster configuration
  - Leadership election of Kafka Broker and Topic Partition pairs
  - manages service discovery for Kafka Brokers that form the cluster
- Zookeeper sends changes to Kafka
  - New Broker join, Broker died, etc.
  - Topic removed, Topic added, etc.



Topics, producers, and consumers



Apache Kafka



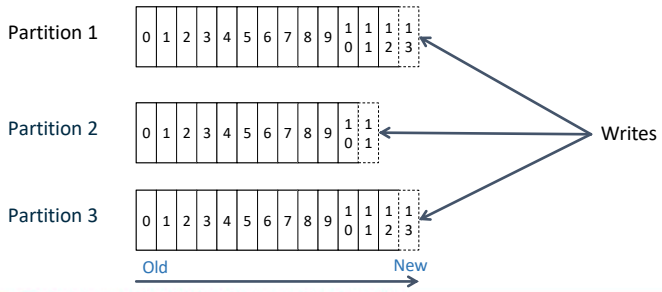
Kafka topics architecture

Kafka topics, logs, partitions

- Kafka topic is a stream of records
- Topics stored in log
- Topic is a category or stream name or feed
- Topics are pub/sub
  - Can have zero or many subscribers - consumer groups

Topic partitions

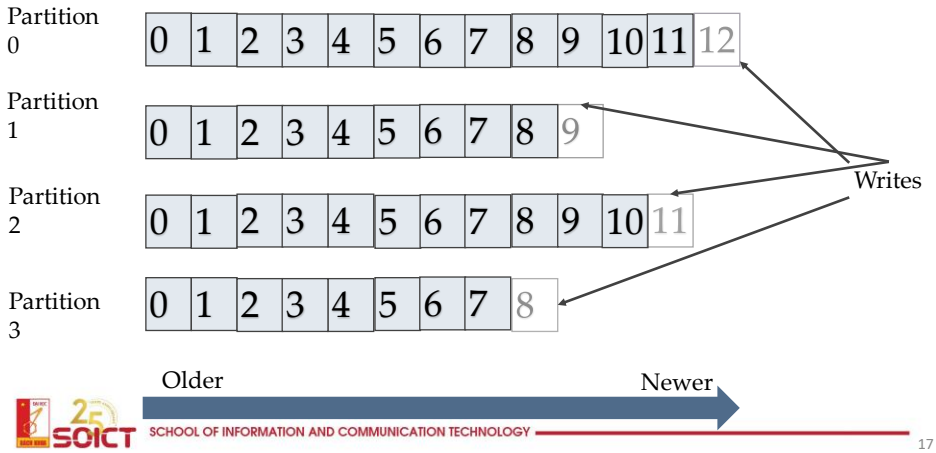
- Topics are broken up into partitions, decided usually by key of record
- Partitions are used to scale Kafka across many servers
  - Record sent to correct partition by key
- Partitions can be replicated to multiple brokers



Topic partition log

- Order is maintained only in a single partition
  - Partition is ordered, immutable sequence of records that is continually appended to—a structured commit log
- Records in partitions are assigned sequential id number called the offset

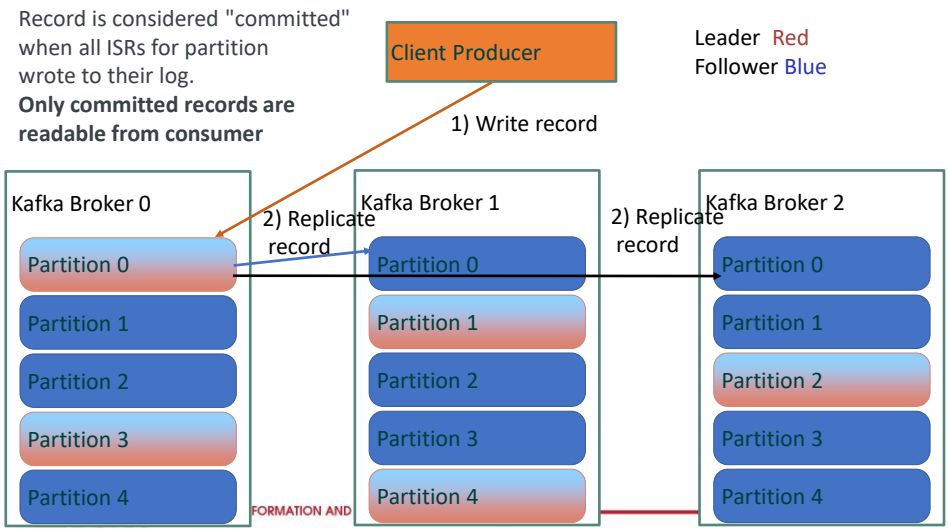
Kafka topic partitions layout



Kafka partition replication

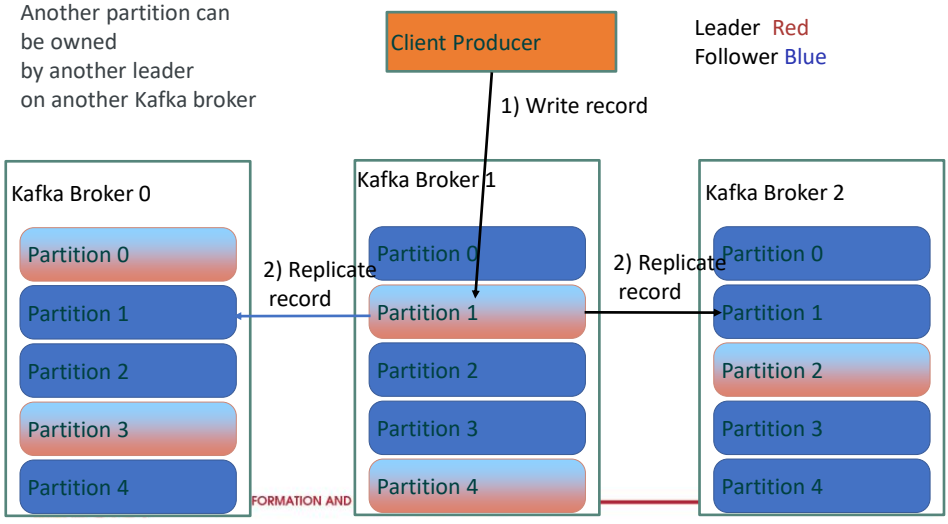
- Each partition has leader server and zero or more follower servers
  - Leader handles all read and write requests for partition
  - Followers replicate leader
  - An follower that is in-sync is called an ISR (in-sync replica)
  - If a partition leader fails, one ISR is chosen as new leader
- Partitions of log are distributed over the servers in the Kafka cluster with each server handling data and requests for a share of partitions
- Each partition can be replicated across a configurable number of Kafka servers
  - Used for fault tolerance

Kafka replication to partition (1)



19

Kafka replication to partitions (2)



20

Guarantees

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent
- Minimum available ISR can also be configured such that an error is returned if enough replicas are not available to replicate data
- A consumer instance sees messages in the order they are stored in the log
- For a topic with replication factor N, Kafka can tolerate up to N-1 server failures without “losing” any messages committed to the log

Kafka record retention

- Kafka cluster retains all published records
  - Time based – configurable retention period
  - Size based - configurable based on size
  - Compaction - keeps latest record
- Retention policy of three days or two weeks or a month
- It is available for consumption until discarded by time, size or compaction
- Consumption speed not impacted by size

Durable writes

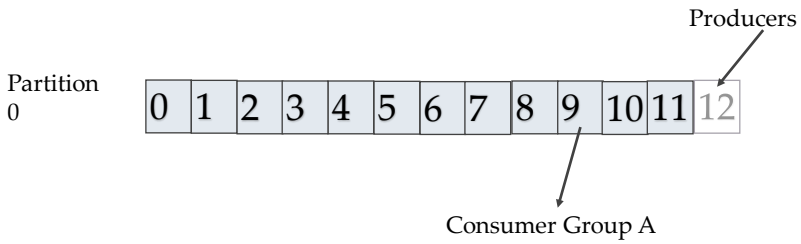
- Producers can choose to trade throughput for durability of writes:
- Note: throughput can also be raised with more brokers…

Durability	Behaviour	Per Event Latency	Required Acknowledgements (request.required.acks)
Highest	ACK all ISRs have received	Highest	-1
Medium	ACK once the leader has received	Medium	1
Lowest	No ACKs required	Lowest	0

Producers

- Producers publish to a topic of their choosing (push)
  - Producer(s) append Records at end of Topic log
- Load can be distributed in number of partitions
  - Typically by “round-robin”
  - Can also do “semantic partitioning” based on a key in the message
    - Example have all the events of a certain 'employeeId' go to same partition
  - Important: Producer picks partition
- All nodes can answer metadata requests about
  - Which servers are alive
  - Where leaders are for the partitions of a topic

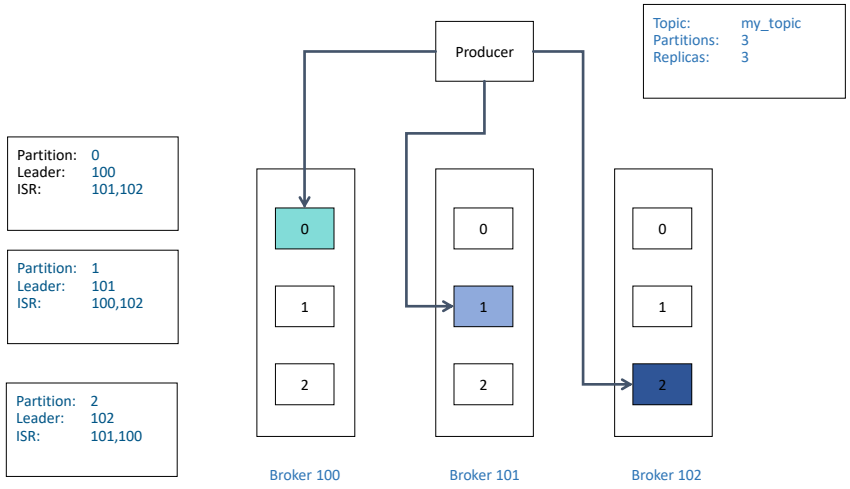
Kafka producers and consumers



Producers are writing at Offset 12

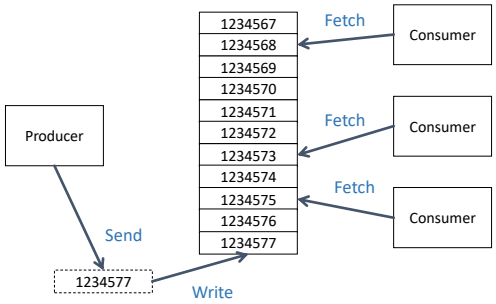
Consumer Group A is Reading from Offset 9.

Producer – Load balancing and ISRs



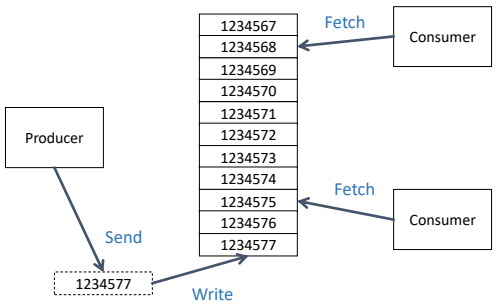
Consumer (1)

- Multiple Consumers can read from the same topic
- Each Consumer is responsible for managing it's own offset
- Messages stay on Kafka…they are not removed after they are consumed



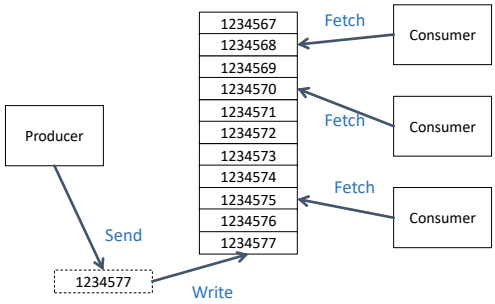
Consumer (2)

- Consumers can go away



Consumer (3)

- And then come back



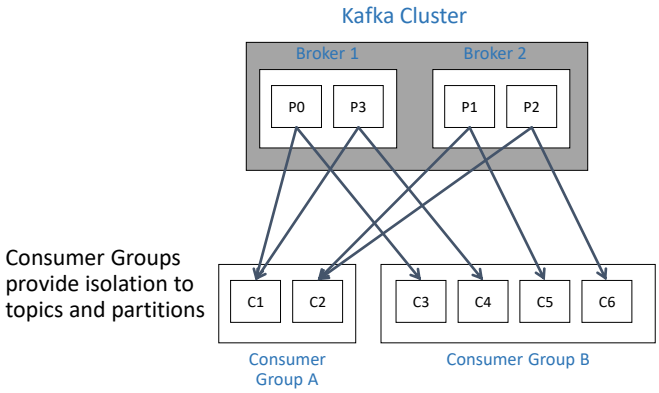
Consumer Group

- Consumers are grouped into a Consumer Group
  - Consumer group has a unique id
  - Each consumer group is a subscriber
  - Each consumer group maintains its own offset
- Multiple subscribers = multiple consumer groups
- Each has different function: one might delivering records to microservices while another is streaming records to Hadoop
- A record is delivered to one Consumer in a Consumer Group
- Each consumer in consumer groups takes records and only one consumer in group gets same record
- Consumers in Consumer Group load balance record consumption

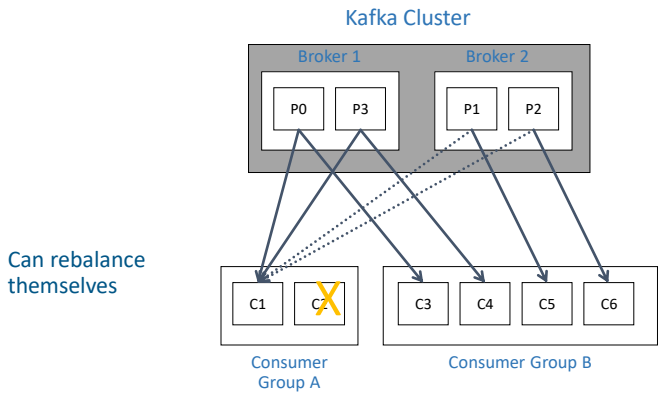
Common consumer group patterns

- All consumer instances in one group
  - Acts like a traditional queue with load balancing
- All consumer instances in different groups
  - All messages are broadcast to all consumer instances
- "Logical Subscriber" – Many consumer instances in a group
  - Consumers are added for scalability and fault tolerance
  - Each consumer instance reads from one or more partitions for a topic
  - There cannot be more consumer instances than partitions

Consumer - Groups



Consumer - Groups



Kafka consumer load share

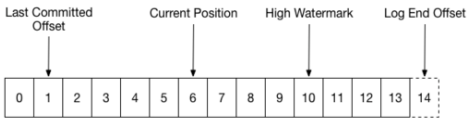
- Consumer membership in Consumer Group is handled by the Kafka protocol dynamically
- If new Consumers join Consumer group, it gets a share of partitions
- If Consumer dies, its partitions are split among remaining live Consumers in Consumer Group

Kafka consumer failover

- Consumers notify broker when it successfully processed a record
  - advances offset (“__consumer_offset”)
- If Consumer fails before sending commit offset to Kafka broker,
  - different Consumer can continue from the last committed offset
  - some Kafka records could be reprocessed
  - at least once behavior
  - messages should be idempotent

What can be consumed

- "Log end offset" is offset of last record written to log partition and where Producers write to next
- "High watermark" is offset of last record successfully replicated to all partitions followers
- Consumer only reads up to “high watermark”. Consumer can’t read un-replicated data



Consumer to partition cardinality

- Only a single Consumer from the same Consumer Group can access a single Partition
- If Consumer count exceeds Partition count:
  - Extra Consumers remain idle; can be used for failover
- If more Partitions than Consumer instances,
  - Some Consumers will read from more than one partition

Kafka brokers

- Kafka Cluster is made up of multiple Kafka Brokers
- Each Broker has an ID (number)
- Brokers contain topic log partitions
- Connecting to one broker bootstraps client to entire cluster
- Start with at least three brokers, cluster can have, 10, 100, 1000 brokers if needed

Kafka scale and speed

- How can Kafka scale if multiple producers and consumers read/write to same Kafka Topic log?
- Writes fast: Sequential writes to filesystem are fast (700 MB or more a second)
- Scales writes and reads by sharding:
  - Topic logs into Partitions (parts of a Topic log)
  - Topics logs can be split into multiple Partitions different machines/different disks
  - Multiple Producers can write to different Partitions of the same Topic
  - Multiple Consumers Groups can read from different partitions efficiently

Kafka scale and speed (2): high throughput and low latency

- Batching of individual messages to amortize network overhead and append/consume chunks together
  - end to end from Producer to file system to Consumer
  - Provides More efficient data compression. Reduces I/O latency
- Zero copy I/O using sendfile (Java's NIO FileChannel transferTo method).
  - Implements linux sendfile() system call which skips unnecessary copies
  - Heavily relies on Linux PageCache
    - The I/O scheduler will batch together consecutive small writes into bigger physical writes which improves throughput.
    - The I/O scheduler will attempt to re-sequence writes to minimize movement of the disk head which improves throughput.
    - It automatically uses all the free memory on the machine

Delivery semantics

Default

- At least once
  - Messages are never lost but may be redelivered
- At most once
  - Messages are lost but never redelivered
- Exactly once
  - Messages are delivered once and only once

Delivery semantics

- At least once
  - Messages are never lost but may be redelivered
- At most once
  - Messages are lost but never redelivered
- Exactly once
  - Messages are delivered once and only once

Much Harder (Impossible??)

Getting exactly once semantics

- Must consider two components
  - Durability guarantees when publishing a message
  - Durability guarantees when consuming a message
- Producer
  - What happens when a produce request was sent but a network error returned before an ack?
  - Use a single writer per partition and check the latest committed value after network errors
- Consumer
  - Include a unique ID (e.g. UUID) and de-duplicate.
  - Consider storing offsets with data

https://dzone.com/articles/interpreting-kafkas-exactly-once-semantics

Kafka positioning

- For really large file transfers
  - Probably not, it's designed for "messages" not really for files. If you need to ship large files, consider good-ole-file transfer, or breaking up the files and reading per line to move to Kafka.
- As a replacement for MQ/Rabbit/Tibco
  - Probably. Performance Numbers are drastically superior. Also gives the ability for transient consumers. Handles failures pretty well.
- If security on the broker and across the wire is important?
  - Not right now. We can't really enforce much in the way of security. (KAFKA-1682)
- To do transformations of data
  - Not really by itself

References

- Garg, Nishant. *Apache Kafka*. Packt Publishing Ltd, 2013.
- Thein, Khin Me Me. "Apache kafka: Next generation distributed messaging system." *International Journal of Scientific Engineering and Technology Research* 3.47 (2014): 9478-9483.
- Dobbelaere, Philippe, and Kyumars Sheykh Esmaili. "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper." *Proceedings of the 11th ACM international conference on distributed and event-based systems*. 2017.

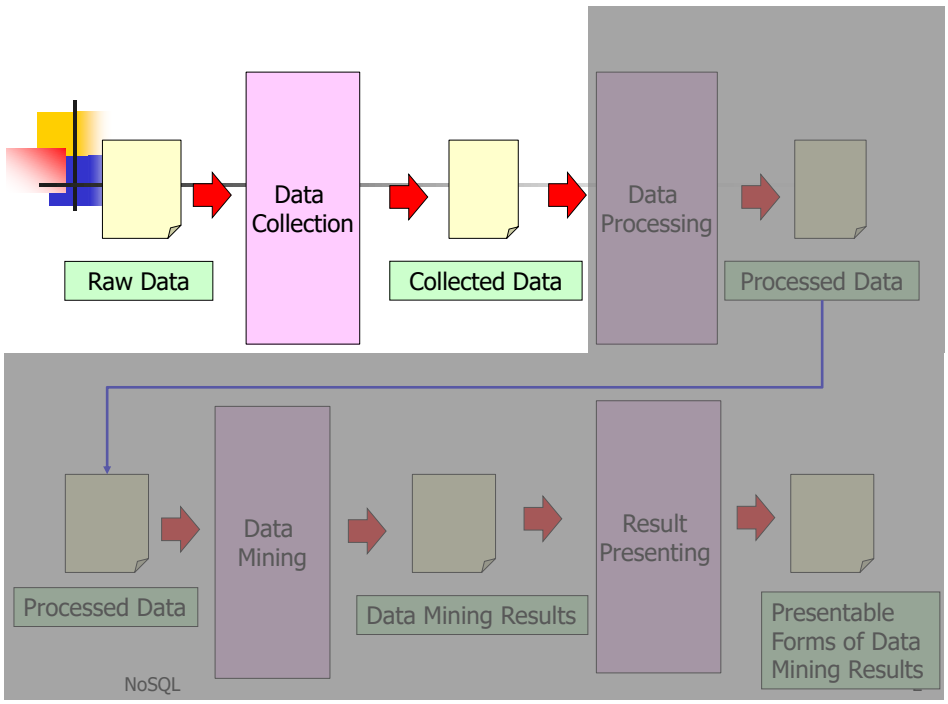
45

46

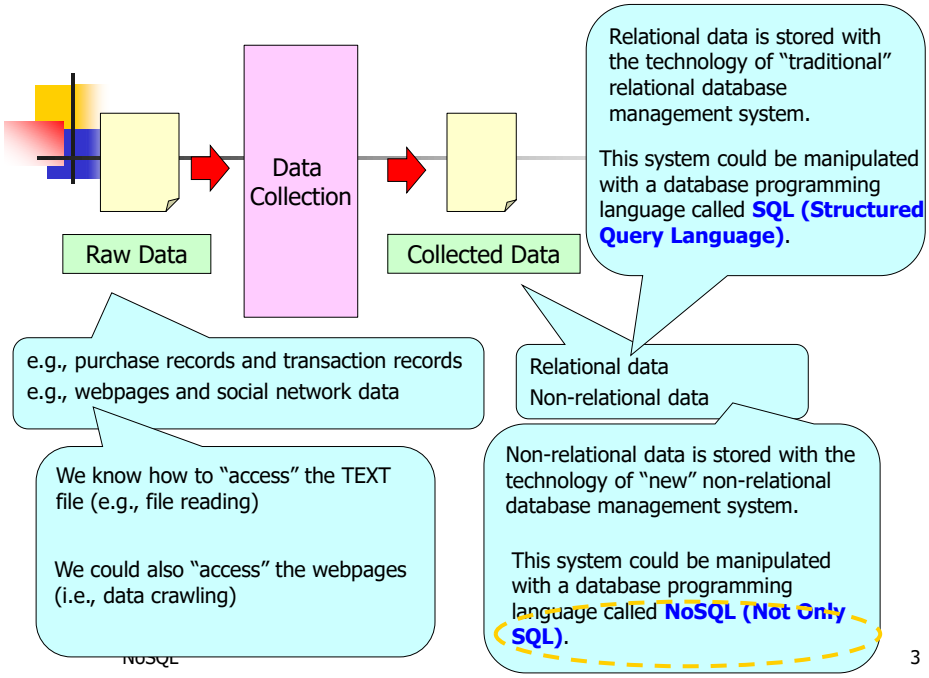
Bigdata Storage and Processing

NoSQL

1



2



Outline

1. What is NoSQL?
2. NoSQL

1. What is NoSQL?

- **NoSQL (Not Only SQL)**
- The NoSQL system was designed NOT to follow any relational schema.
  - The storage format is not in a table form.
  - We can insert data without first defining any schema.
  - It is preferred not to involve any time-consuming join operation.
- Large-scale web organizations such as Google and Amazon used
  - **NoSQL systems** focus on **narrow** operational goals (for fast operations) and
  - **relational databases** as adjuncts for **data consistency**

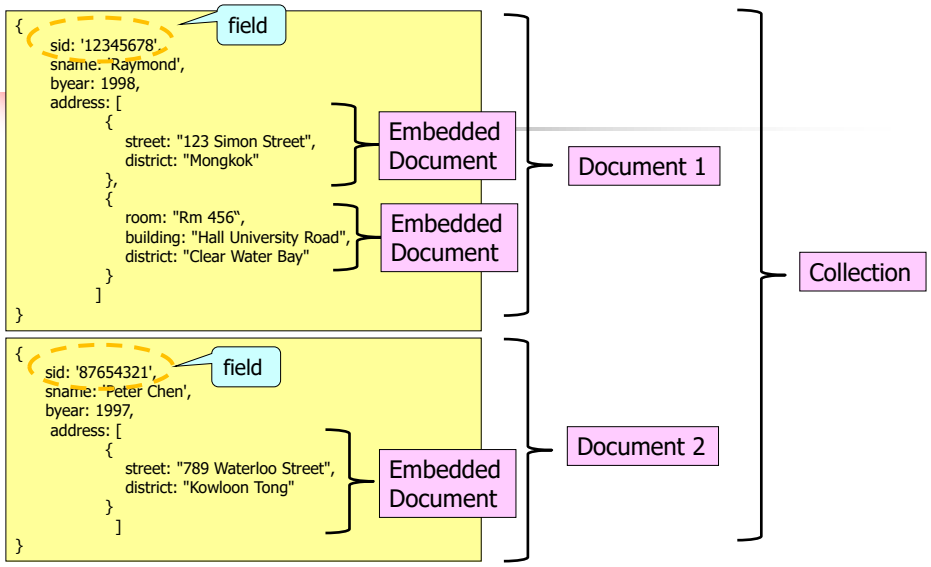
1. What is NoSQL?

- **MongoDB** (from the word "hu**mongo**us") is the most popular NoSQL system
- In the following, we will illustrate with MongoDB.
- Other NoSQL systems:
  - Amazon DynamoDB
  - Google BigTable
  - Apache Cassandra

1. What is NoSQL?

- **Advantage**
  - It could store a large amount of data **without much structure** or **with little structure**.
  - It could be used with **cloud computing and storage** for scalability.
  - It could be **developed very rapidly** (since we do not need to define any schema).

In the age of Big Data, these advantages become more important.



A number of different collections form a database

SQL (Relational DB)	NoSQL
Database	Database
Table	Collection
Tuple/Row/Record	Document
Column	Field
Table Join	Embedded Documents

- **Relational Database**
  - Table-based database
- MySQL is the most popular open-source relational database system
- Oracle is the most popular commercial relational database system
- **Advantage**
  - It could store a large amount of **"structured" data**
  - It **avoids** storing **duplicate data**.
    (Thus, it could guarantee data consistency).
  - It is **easier to write** database languages (i.e., SQL) in relational database than NoSQL

- When should we use NoSQL?
  - The structure of the data is not that clear.

    We know that NoSQL stores "unstructured" data (which has an "irregular" structure).
    NoSQL could also store "structured" data (which has a "regular" structure).

  - We want to avoid the time-consuming "join" operation (because the data stored in NoSQL could be a result of a "join" operation).

Sometimes, this may introduce storing duplicate data (i.e., data redundancy). Thus, it sometimes cannot guarantee data consistency.


```
{
      sid: '12345678',
      sname: 'Raymond',
      byear: 1998,
      course_list: [
              {
                 cid: "COMP4332",
                 cname: "Big Data Mining"
              },
              {
                 cid: "COMP5331",
                 cname: "Knowledge Discovery in Databases"
              }
            ]
}
```

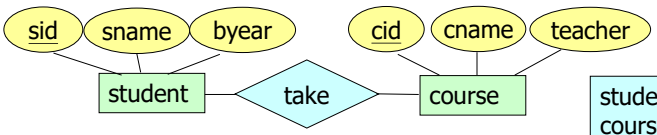
```
{
     sid: '87654321',
     sname: 'Peter Chen',
     byear: 1997,
      course_list: [
              {
                 cid: "COMP4332",
                 cname: "Big Data Mining"
              }
               ]
}
```

Data redundancy

- When should we use SQL?
  - The structure of the data is very clear.

    We know that SQL stores "structured" data (which has a "regular" structure).
    SQL could also store "unstructured" data (which has an "irregular" structure) but it stores many "null" values in the tables.
  - We want to store data without duplication (for data consistency)

Sometimes, this may introduce time-consuming join operations.



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

```
select T.cid
from student S, take T
where S.sid = T.sid and
      S.sname = 'Raymond'
```

Natural Join

It is time-consuming.

Outline

1. What is NoSQL?
2. NoSQL

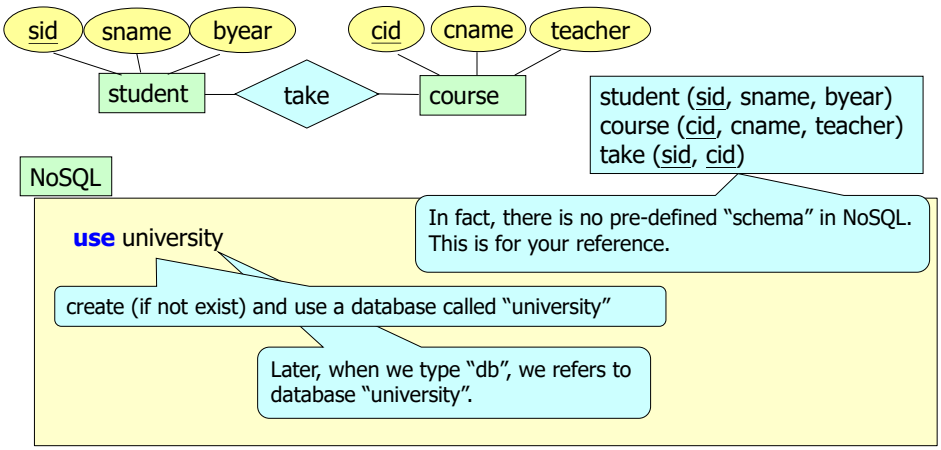
2. NoSQL

- Data Definition Language (DDL)
- Data Manipulation Language (DML)

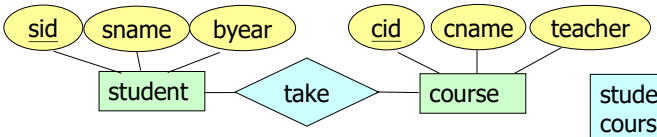
2. DDL

- Data Definition Language (DDL)
  - Although NoSQL does not require to define a schema, we could see how NoSQL could define the schema "automatically".

2. Database Creation



2. Collection Creation



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL | Version 1

db.createCollection("student")

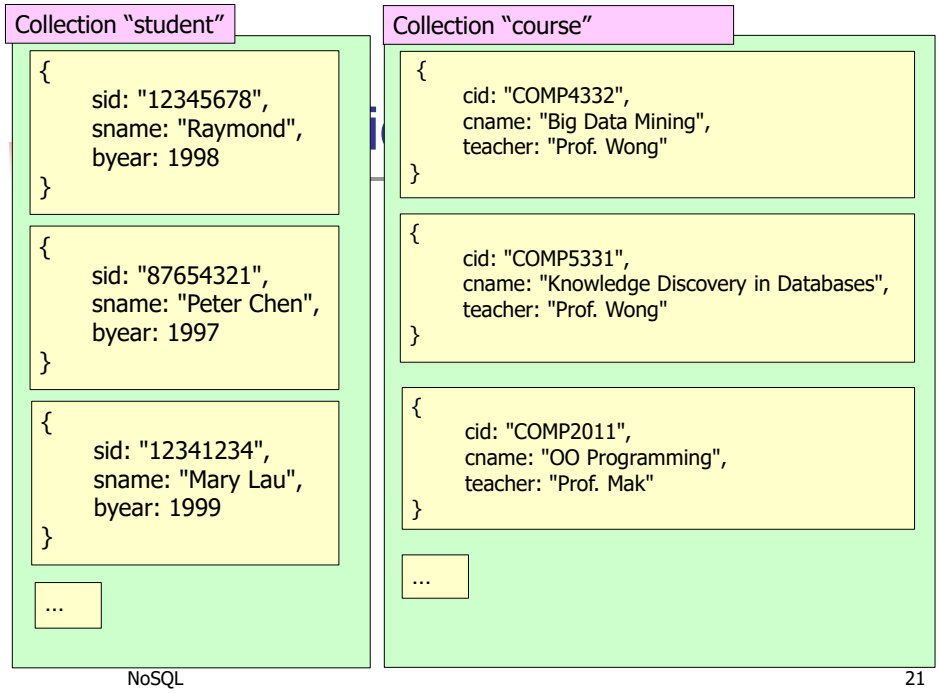
db.createCollection("course")

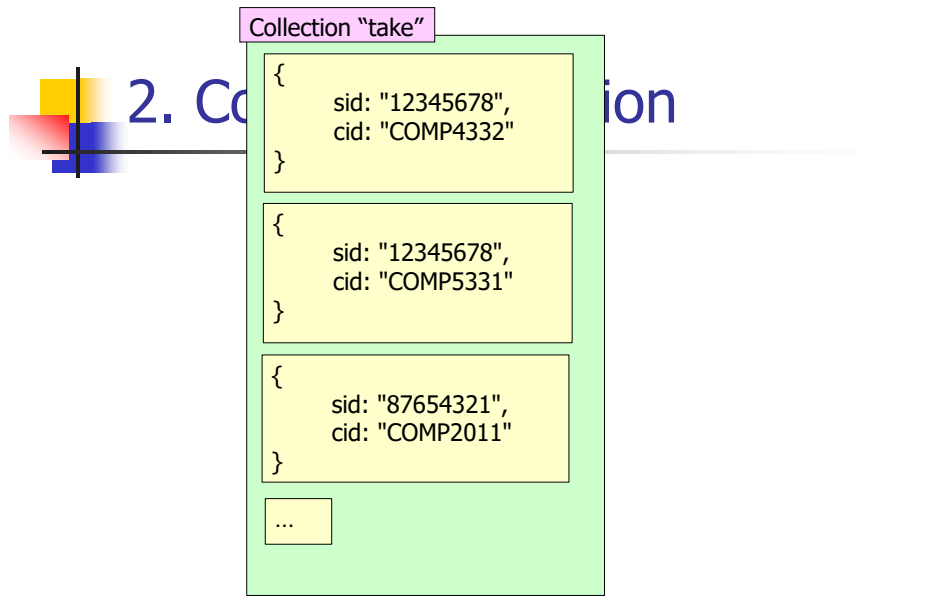
db.createCollection("take")

There is no need to specify any schema of each of these collections.

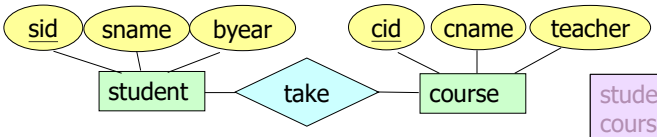
2. Collection Creation

- If we execute the previous commands, we plan to have our data (to be inserted later) like the following.


Collection "student"
{
sid: "12345678",
sname: "Raymond",
byear: 1998
}
{
sid: "87654321",
sname: "Peter Chen",
byear: 1997
}
{
sid: "12341234",
sname: "Mary Lau",
byear: 1999
}
...
Collection "course"
{
cid: "COMP4332",
cname: "Big Data Mining",
teacher: "Prof. Wong"
}
{
cid: "COMP5331",
cname: "Knowledge Discovery in Databases",
teacher: "Prof. Wong"
}
{
cid: "COMP2011",
cname: "OO Programming",
teacher: "Prof. Mak"
}
...


2. Co ion
Collection "take"
{
sid: "12345678",
cid: "COMP4332"
}
{
sid: "12345678",
cid: "COMP5331"
}
{
sid: "87654321",
cid: "COMP2011"
}
...

2. Collection Creation



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL Version 2

takeIncludeAll (sid, sname, byear, cid, cname, teacher)

```
db.createCollection("takeIncludeAll")
```

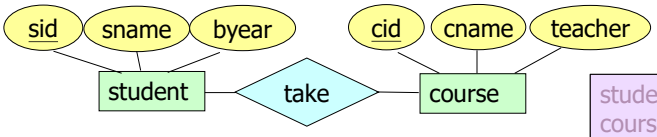

23

Collection "takeIncludeAll"

```
{
    sid: "12345678",
    sname: "Raymond",
    byear: 1998,
    cid: "COMP4332",
    cname: "Big Data Mining",
    teacher: "Prof. Wong"
}
{
    sid: "12345678",
    sname: "Raymond",
    byear: 1998,
    cid: "COMP5331",
    cname: "Knowledge Discovery in Databases",
    teacher: "Prof. Wong"
}
{
    sid: "87654321",
    sname: "Peter Chen",
     byear: 1997,
    cid: "COMP2011 ",
    cname: "OO Programming",
    teacher: "Prof. Mak"
}
...
```


24

2. Collection Creation



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL Version 3

studentIncludeAll (sid, sname, byear, courseList: [{cid, cname, teacher}, …])

db.createCollection("studentIncludeAll")

25

Collection "studentIncludeAll"

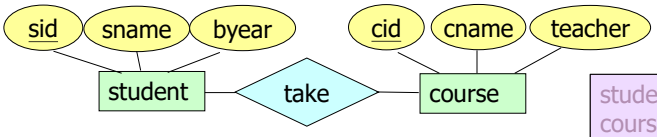
```
{
      sid: "12345678",
      sname: "Raymond",
      byear: 1998,
      course_list: [
                    {
                       cid: "COMP4332",
                       cname: "Big Data Mining",
                      teacher: "Prof. Wong"
                    },
                    {
                      cid: "COMP5331",
                      cname: "Knowledge Discovery in Databases",
                      teacher: "Prof. Wong"
                    }
                ]
}
```

```
{
      sid: "87654321",
      sname: "Peter Chen",
      byear: 1997,
      course_list: [
                    {
                       cid: "COMP2011",
                       cname: "OO Programming",
                       teacher: "Prof. Mak"
                    }
                ]
}
```

…

26

2. Collection Creation



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL Version 4

courseIncludeAll (cid, cname, teacher, studentList: [{sid, sname, byear}, …])

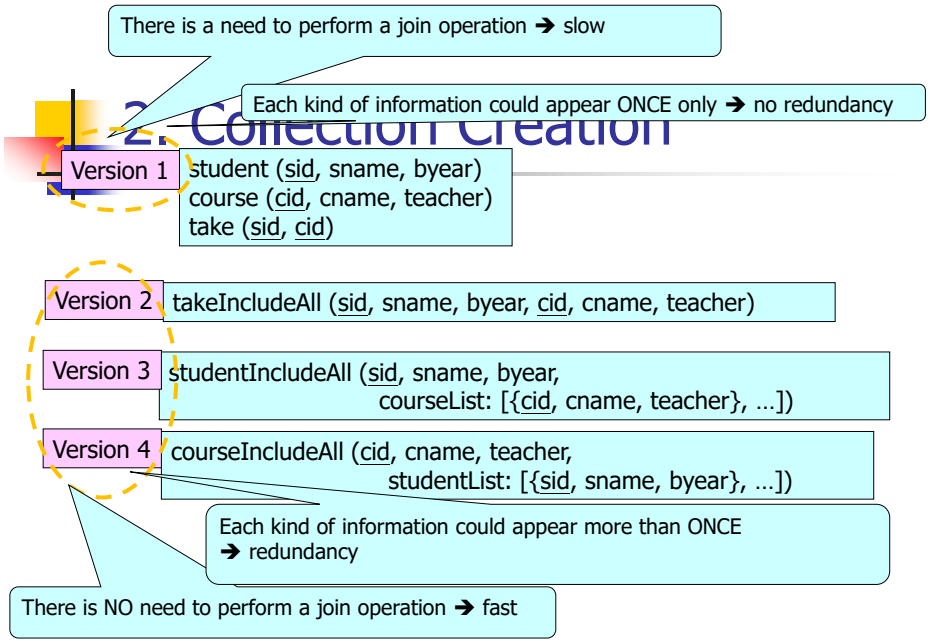
db.createCollection("courseIncludeAll")

Collection "courseIncludeAll"

```
{
      cid: "COMP4332",
      cname: "Big Data Mining",
      teacher: "Prof. Wong",
      student_list: [
                    {
                        sid: "12345678",
                        sname: "Raymond",
                        byear: 1998
                    },
                    {
                        sid: "12341234",
                        sname: "Mary Lau",
                        byear: 1999
                    }
                ]
}
```

```
{
      cid: "COMP5331",
      cname: "Knowledge Discovery in Databases",
      teacher: "Prof. Wong",
      student_list: [
                    {
                        sid: "12345678",
                        sname: "Raymond",
                        byear: 1998
                    }
                ]
}
```

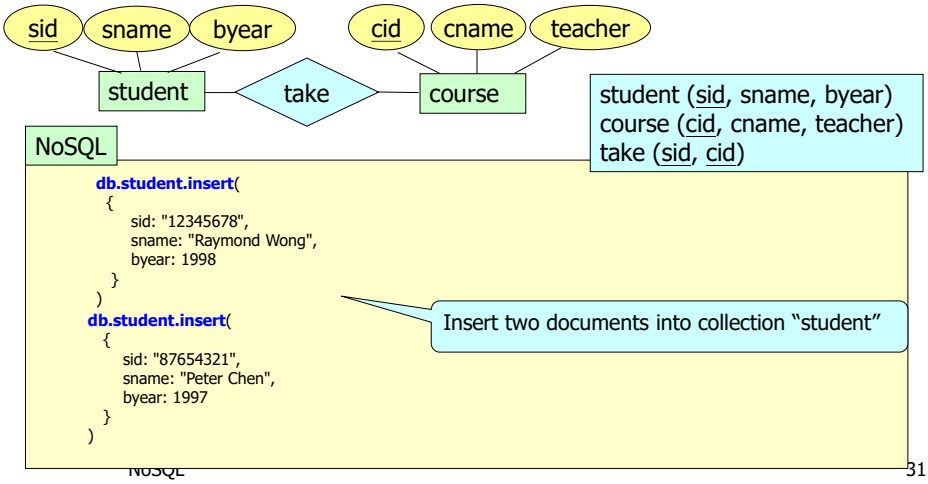
…



2. Collection Creation

- We focus on Version 1
- In this version, sometimes, we need to perform a "so-called" join operation, which may be slow.
- Besides, the NoSQL programming language related to this "join" operation is more complicated than a "standard" SQL language

2. Document Insertion



2. Document Insertion

- Each document is associated with a field called "_id".
- When we insert a document into a collection, if we do not specify the value of field "_id", a field value is generated automatically.
- Thus, the first document is stored in the following format.

```
{
    _id: ObjectId("5a23f26b967809308848df77"),
    sid: "12345678",
    sname: "Raymond Wong",
    byear: 1998
}
```

- But, we could also specify the "_id" field if we like. The insertion format is just like the above format.

2. Document Insertion – Data Types

- **Object ID**
  - E.g., ObjectId("5a23f26b967809308848df77")
- **string**
  - a string of any length
  - We could specify with the following.
    "Raymond"
- **number**
  - A numeric number
  - We could specify with the following.
    1998
    1.2345

2. Document Insertion – Data Types

- **Datetime**
  - Date and time (in ISO Date)
  - We could specify with the following.

```
new Date("2018-02-18T16:45:00Z")
new Date("2018-02-18T16:45:00+08:00")
```

Remember to include "new"

YYYY-MM-DD**T**hh:mm:ss**Z**

Time zone offset from UTC (Coordinated Universal Time)

2. Document Insertion – Data Types

NoSQL

```
db.testDate.insert({birthday1: new Date("2018-02-18T16:45:00Z")})
db.testDate.insert({birthday2: new Date("2018-02-18T16:45:00+08:00")})
db.testDate.insert({birthday3: Date("2018-02-18T16:45:00Z")})

db.testDate.find()
```

"new" is omitted here.

Output

```
{
        "_id" : ObjectId("5a260944ba2f887e3d01d757"),
        "birthday1" : ISODate("2018-02-18T16:45:00Z")
}
{
        "_id" : ObjectId("5a260944ba2f887e3d01d758"),
        "birthday2" : ISODate("2018-02-18T08:45:00Z")
}
{
        "_id" : ObjectId("5a260944ba2f887e3d01d759"),
        "birthday3" : "Tue Dec 05 2017 10:49:40 GMT+0800 (China Standard Time)"
}
```

Exactly what we specified

Re-adjusted time zone based on the offset

The "current" time (not the date/time specified)

35

2. Document Insertion – Data Types

- We could obtain the following field from the "datetime" type.
  - Year
  - Month
  - Day (of the month)
  - Hour
  - Minute
  - Second
  - Day Of the Year
  - Day of the Week (where Sunday is represented by "1")
  - Week number of the Year

NoSQL

```
db.testDate2.insert({birthday: new Date("2018-02-18T16:45:00Z")})
db.testDate2.aggregate(
  [
    {
      $project:
        {
          year: { $year: "$birthday" },
          month: { $month: "$birthday" },
          day: { $dayOfMonth: "$birthday" },
          hour: { $hour: "$birthday" },
          minutes: { $minute: "$birthday" },
          seconds: { $second: "$birthday" },
          dayOfYear: { $dayOfYear: "$birthday" },
          dayOfWeek: { $dayOfWeek: "$birthday" },
          week: { $week: "$birthday" }
        }
    }
  ]
)
```

"$project" is the name of an operation of outputting some attributes/fields

Output

```
{
    _id : ObjectId("5a260e23ba2f887e3d01d75a"),
    year : 2018,
    month : 2,
    day : 18,
    hour : 16,
    minutes : 45,
    seconds : 0,
    dayOfYear : 49,
    dayOfWeek : 1,
    week : 7
}
```

Sunday

37

37

2. Embedded Document Insertion

- We want to illustrate how to insert an embedded document into a document
- We consider the following example.

NoSQL

```
db.tempTable.insert({
                    sid: "12345678",
                    sname: "Raymond",
                    courseTaken: [
                                {cid: "COMP5331"},
                                {cid: "COMP4332"}
                              ]
                  })
```

NoSQL

38

38

2. Embedded Document Insertion

- If we type the following,

NoSQL

```
db.tempTable.update({sid:"12345678"}, {$push: {courseTaken: {cid: "COMP1942"}}})
```

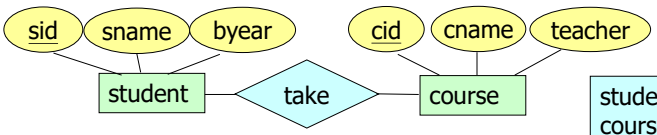
```
db.tempTable.find()
```

Output

```
{
    _id : ObjectId("5a26aef0ba2f887e3d01d761"),
    sid : "12345678",
    sname : "Raymond",
    courseTaken : [
                    { cid : "COMP5331" },
                    { cid : "COMP4332" },
                    { cid : "COMP1942" }
                ]
}
```

39

2. Collection Removal

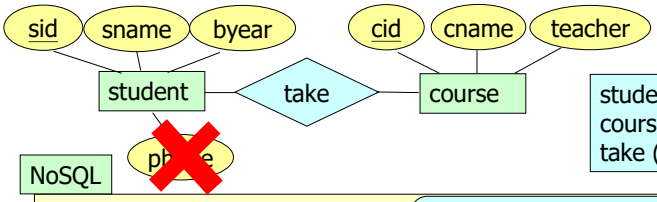


student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL

```
db.take.drop()
db.course.drop()
db.student.drop()
```

2. Collection Scheme Update



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL

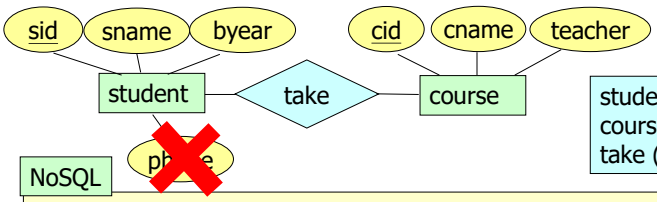
```
db.student.insert(
 {
    sid: "12341234",
    sname: "Mary Lau",
    byear: 1999,
    phone: "23581234"
 }
)
db.student.insert(
 {
    sid: "56785678",
    sname: "David Lee",
    byear: 1998
 }
)
```

Since there is no concept of defining a schema, we just insert a new document with a new field called "phone".

Since there is no concept of defining a schema, we just insert a new document without the field called "phone".

41

2. Collection Scheme Update



student (sid, sname, byear)
course (cid, cname, teacher)
take (sid, cid)

NoSQL

```
db.student.update({}, {$unset: {phone: 1}}, {multi: true})
```

We completely remove the field "phone" of all documents

42

2. NoSQL

- Data Definition Language (DDL)
- Data Manipulation Language (DML)

2. NoSQL

- Three Major Functions
  - find
  - distinct
  - Aggregate
- Operation of Datetime

The easiest operation

The easiest operation which is used in some cases

The most complicated (or the most powerful) operation which could be used in most cases. It can also perform exactly with the same output as "find" and "distinct" but with more complex script

2. NoSQL – find

NoSQL

db.student.find()

To output all documents in collection "student"

Output

```
{ _id : ObjectId("5a8a431472f2fbbf196e55bd"), sid : "12345678", sname : "Raymond", byear : 1998 }
{ _id : ObjectId("5a8a431472f2fbbf196e55be"), sid : "87654321", sname : "Peter Chen", byear : 1997 }
{ _id : ObjectId("5a8a431472f2fbbf196e55bf"), sid : "12341234", sname : "Mary Lau", byear : 1999 }
{ _id : ObjectId("5a8a431472f2fbbf196e55c0"), sid : "56785678", sname : "David Lee", byear : 1998 }
{ _id : ObjectId("5a8a431572f2fbbf196e55c1"), sid : "88888888", sname : "Test Test", byear : 1998 }
```


2. NoSQL – find

NoSQL

db.student.find({byear:1998})

To output all documents which have the field "byear" = 1998 in collection "student"

Output

```
{ _id : ObjectId("5a8a431472f2fbbf196e55bd"), sid : "12345678", sname : "Raymond", byear : 1998 }
{ _id : ObjectId("5a8a431472f2fbbf196e55c0"), sid : "56785678", sname : "David Lee", byear : 1998 }
{ _id : ObjectId("5a8a431572f2fbbf196e55c1"), sid : "88888888", sname : "Test Test", byear : 1998 }
```

2. NoSQL – find

NoSQL

```
db.student.find({byear:1998}, {sid:1, _id:0})
```

To output all documents which have the field "byear" = 1998 in collection "student" by showing the field "sid" of each document in the output.

Output

```
{ sid : "12345678" }
{ sid : "56785678" }
{ sid : "88888888" }
```


2. NoSQL – find

NoSQL

```
db.student.find({}, {sid:1, _id:0})
```

To output all documents in collection "student" by showing the field "sid" of each document in the output.

Output

```
{ sid : "12345678" }
{ sid : "87654321" }
{ sid : "12341234" }
{ sid : "56785678" }
{ sid : "88888888" }
```

2. NoSQL – find

NoSQL

The descending order could be used with sort({sid:-1}) instead of sort({sid:1}).

```
db.student.find({}, {sid:1, _id:0}).sort({sid:1})
```

To output all documents in collection "student" by showing the field "sid" of each document in the output.
Show them in ascending order of "sid"

Output

```
{ sid : "12341234" }
{ sid : "12345678" }
{ sid : "56785678" }
{ sid : "87654321" }
{ sid : "88888888" }
```

Sorted in ascending order of "sid"

2. NoSQL – find

- In the previous examples, the documents are structured.
- In some cases, the documents are unstructured.
- That is, some documents have some fields but some other documents do not have.

2. NoSQL – find

- Suppose that we have the following 2 documents.

```
{
    _id : ObjectId("5a23ede1967809308848df56"),
    sid: "12345678",
    sname: "Raymond",
    byear: 1998
}
{
    _id : ObjectId("5a23ee31967809308848df5b"),
    sid: "56785678",
    sname: "David Lee"
}
```

There is no "byear" field in the 2nd document.

We could perform the NoSQL query on field "byear" for the whole collection.

2. NoSQL – find

NoSQL

```
db.student.find({byear:1998})
```

To output all documents which have the field "byear" = 1998 in collection "student"

This query is valid even if some documents do not contain field "byear"

Output

```
{
    _id : ObjectId("5a23ede1967809308848df56"),
    sid: "12345678",
    sname: "Raymond",
    byear: 1998
}
```

2. NoSQL

- Three Major Functions
  - find
  - distinct
  - aggregate
- Operation of Datetime

The easiest operation

The easiest operation which is used in some cases

The most complicated (or the most powerful) operation which could be used in most cases. It can also perform exactly with the same output as "find" and "distinct" but with more complex script

2. NoSQL – distinct

NoSQL

```
db.student.distinct("sname", {byear:1998})
```

To output a list of distinct values of field "sname" of all documents which have the field "byear" = 1998 in collection "student"

Note that we should have a double-quote here. If we miss the double-quote, there is a syntax error.

Output

```
[ "Raymond", "David Lee", "Test Test" ]
```

If there are two or more documents with "sname" = "Raymond", only one "Raymond" will be shown.

2. NoSQL – distinct

- In the previous examples, the documents are structured.
- In some cases, the documents are unstructured.
- That is, some documents have some fields but some other documents do not have.

2. NoSQL – distinct

- Suppose that we have the following 2 documents.

```
{
    _id : ObjectId("5a23ede1967809308848df56"),
    sid: "12345678",
    sname: "Raymond",
    byear: 1998
}
{
    _id : ObjectId("5a23ee31967809308848df5b"),
    sid: "56785678",
    sname: "David Lee"
}
```

There is no "byear" field in the 2nd document.

We could perform the NoSQL query on field "byear" for the whole collection.

2. NoSQL – distinct

NoSQL

```
db.student.distinct("byear")
```

To output a list of distinct values of field "byear" of all documents in collection "student"

This query is valid even if some documents do not contain field "byear"

Output

```
[1998]
```

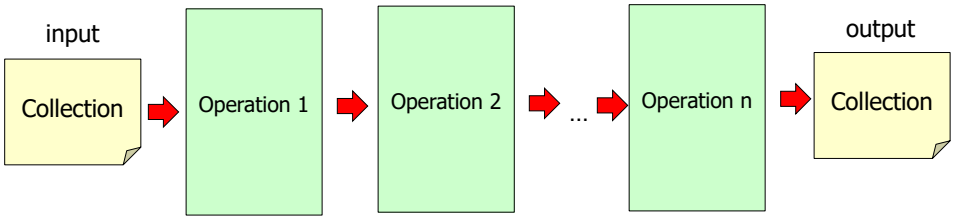
Thus, non-specified values will not be shown here.

2. NoSQL

- Three Major Functions
  - find — The easiest operation
  - distinct — The easiest operation which is used in some cases
  - Aggregate — The most complicated (or the most powerful) operation which could be used in most cases. It can also perform exactly with the same output as "find" and "distinct" but with more complex script
- Operation of Datetime

2. NoSQL – aggregation

- "aggregation" involves a "pipeline" process where the output from each step/operation in the pipeline provides the input of the next step/operation



2. NoSQL – aggregation

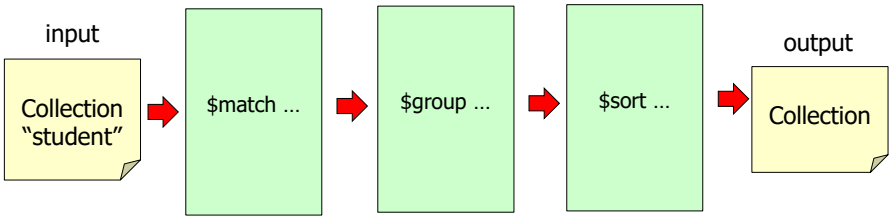
- Aggregation include the following operations
  - $project — Specify a list of fields to be shown in the output (formally called "project")
  - $match — Specify a list of conditions to be matched by the documents in the output (similar to "find")
  - $unwind — Specify an array field to expand the array of each document, generating one output document for each array entry
  - $group — Specify a list of fields used for grouping documents from the input
  - $sort — Specify a list of fields used for sorting the documents in the output
  - $out — Specify a new collection name to write the output result
  - $lookup — Specify another collection to be joined

2. NoSQL – aggregation

Note that there is a pair of "[" and "]"

NoSQL

```
db.student.aggregate([  {$match: …}, {$group: …}, {$sort: …}  ])
```



2. NoSQL – aggregation

SQL	Aggregation Operation
select	$project $group functions: $sum, $min, $avg, …
from	db.student.aggregate(…) $lookup
where	$match
group by	$group
having	$match
order by	$sort

"$out" and "$unwind" are not shown here since they do not match the above SQL query exactly

2. NoSQL – aggregation

- Next, we give the following aggregation examples
  - Examples (which could perform exactly with the same output as "find")
  - Examples (which could perform exactly with the same output as "distinct")
  - Examples (which could perform in the way that "find" and "distinct" could not perform)

2. NoSQL – aggregation (same as "find")

NoSQL

db.student.aggregate()

To output all documents in collection "student"

Same as "db.student.find()"

Output

```
{ _id : ObjectId("5a8a431472f2fbbf196e55bd"), sid : "12345678", sname : "Raymond", byear : 1998 }
{ _id : ObjectId("5a8a431472f2fbbf196e55be"), sid : "87654321", sname : "Peter Chen", byear : 1997 }
{ _id : ObjectId("5a8a431472f2fbbf196e55bf"), sid : "12341234", sname : "Mary Lau", byear : 1999 }
{ _id : ObjectId("5a8a431472f2fbbf196e55c0"), sid : "56785678", sname : "David Lee", byear : 1998 }
{ _id : ObjectId("5a8a431572f2fbbf196e55c1"), sid : "88888888", sname : "Test Test", byear : 1998 }
```

2. NoSQL – aggregation (same as "find")

NoSQL

To output all documents which have the field "byear" = 1998 in collection "student"

```
db.student.aggregate([{$match: {byear:1998} }])
```

Same as "db.student.find({byear:1998})"

Output

```
{ _id : ObjectId("5a8a431472f2fbbf196e55bd"), sid : "12345678", sname : "Raymond", byear : 1998 }
{ _id : ObjectId("5a8a431472f2fbbf196e55c0"), sid : "56785678", sname : "David Lee", byear : 1998 }
{ _id : ObjectId("5a8a431572f2fbbf196e55c1"), sid : "88888888", sname : "Test Test", byear : 1998 }
```


2. NoSQL – aggregation (same as "find")

NoSQL

To output all documents which have the field "byear" = 1998 in collection "student" by showing the field "sid" of each document in the output.

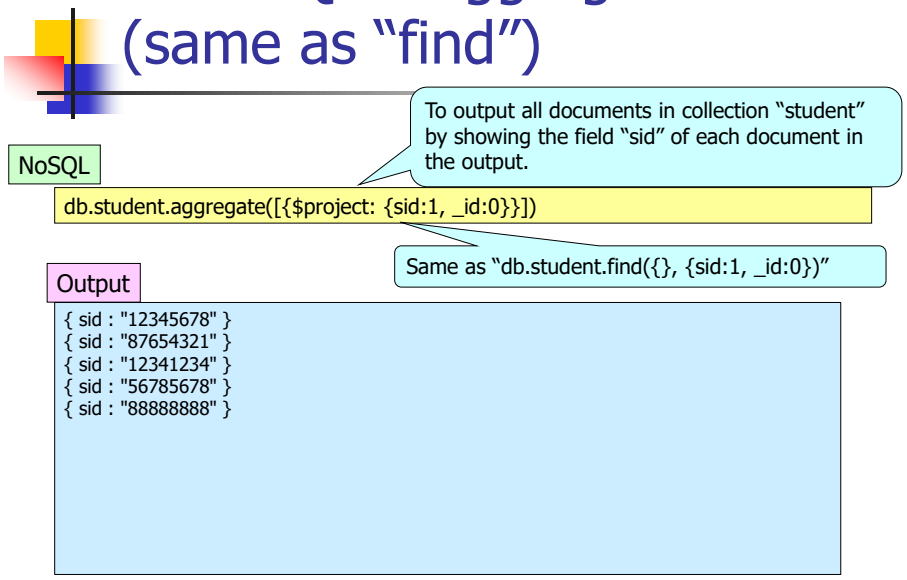
```
db.student.aggregate([{$match: {byear:1998}}, {$project: {sid:1, _id:0}}])
```

Same as "db.student.find({byear:1998}, {sid:1, _id:0})"

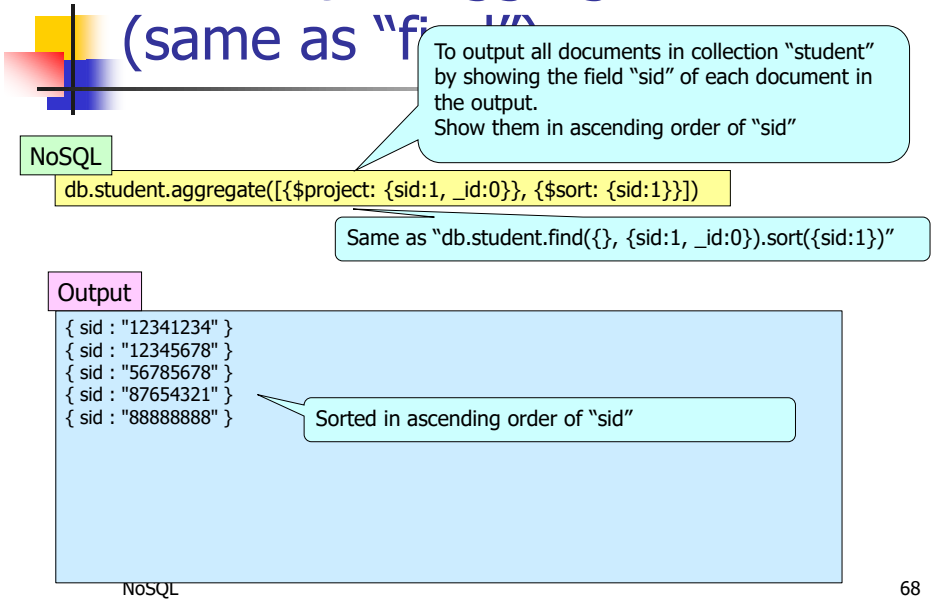
Output

```
{ sid : "12345678" }
{ sid : "56785678" }
{ sid : "88888888" }
```

2. NoSQL – aggregation (same as "find")



2. NoSQL – aggregation (same as "find")



2. NoSQL – aggregation

- Next, we give the following aggregation examples
  - Examples (which could perform exactly with the same output as "find")
  - Examples (which could perform exactly with the same output as "distinct")
  - Examples (which could perform in the way that "find" and "distinct" could not perform)

2. NoSQL – aggregation (same to "distinct")

To output a list of distinct values of field "sname" of all documents which have the field "byear" = 1998 in collection "student"

NoSQL

```
db.student.aggregate([{$match: {byear:1998}}, {$group: {_id: "$sname"}} ])
```

Note that we should have a double-quote here. If we miss the double-quote, there is a syntax error.

Same as "db.student.distinct("sname", {byear:1998})" (but the output of this "distinct" operation is an array.)

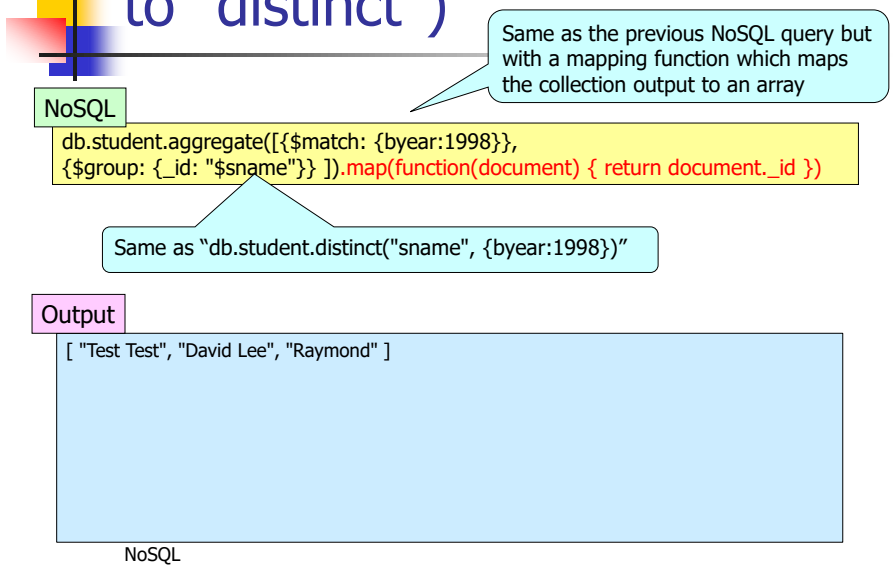
Output

```
{ _id : "Test Test" }
{ _id : "David Lee" }
{ _id : "Raymond" }
```

If there are two or more documents with "sname" = "Raymond", only one document will be shown.

The output here is a collection (not an array).

2. NoSQL – aggregation (same to "distinct")



2. NoSQL – aggregation

- Next, we give the following aggregation examples
  - Examples (which could perform exactly with the same output as "find")
  - Examples (which could perform exactly with the same output as "distinct")
  - Examples (which could perform in the way that "find" and "distinct" could not perform)

2. NoSQL - aggregation

NoSQL

Find the class size of each course and show the course id and the class size for each course.

```
db.take.aggregate([{ $group : {_id : "$cid", classSize : {$sum : 1}}  } ])
```

Output

```
{ _id : "COMP2711", classSize : 1 }
{ _id : "COMP2011", classSize : 2 }
{ _id : "RMBI4310", classSize : 2 }
{ _id : "COMP5331", classSize : 2 }
{ _id : "COMP4332", classSize : 2 }
```


2. NoSQL - aggregation

- In the previous examples, the documents are structured.
- In some cases, the documents are unstructured.
- That is, some documents have some fields but some other documents do not have.

2. NoSQL - aggregation

- Suppose that we have the following 2 documents.

```
{
      _id : ObjectId("5a23ede1967809308848df56"),
      sid: "12345678",
      sname: "Raymond",
      byear: 1998
}
{
      _id : ObjectId("5a23ee31967809308848df5b"),
      sid: "56785678",
      sname: "David Lee"
}
```

There is no "byear" field in the 2nd document.

We could perform the NoSQL query on field "byear" for the whole collection.

2. NoSQL - agg

To output the number of students for each possible value of "byear" in collection "student"

NoSQL

```
db.student.aggregate([ {$group: {_id: "$byear", size:  {$sum: 1}} } ])
```

This query is valid even if some documents do not contain field "byear"

Output

```
{
   _id : null,
   size : 1
}
{
   _id : 1998,
   size : 1
}
```

Note that the "null" value (i.e., the non-specified value) is shown here.

2. NoSQL - aggregation

- We will illustrate "$unwind" (and "$out") with a collection which contains a document including an array.
- Consider the following new collection.

NoSQL

```
db.tempTable.insert({
                        sid: "12345678",
                        sname: "Raymond",
                        courseTaken: [{cid: "COMP5331"}, {cid: "COMP4332"}]
                    })
```


2. NoSQL - aggregation

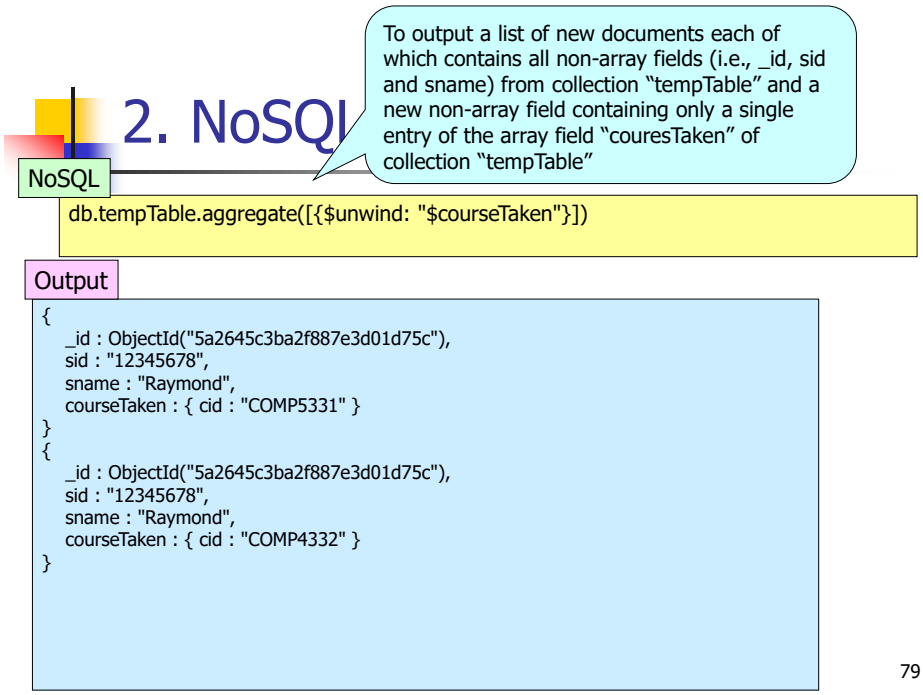
- If we type the following,

NoSQL

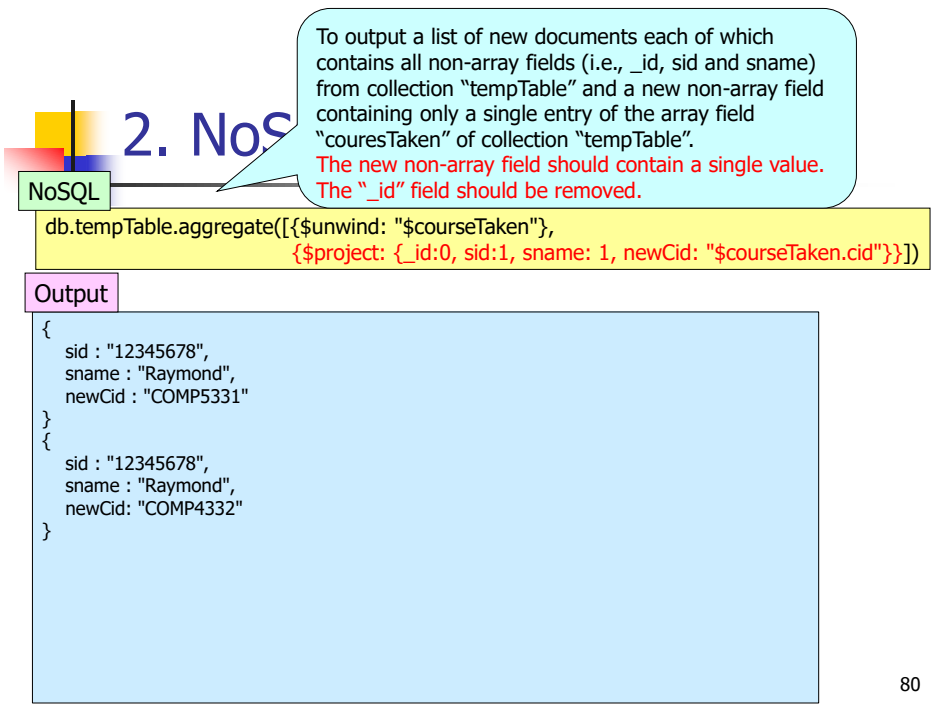
```
db.tempTable.find()
```

Output

```
{
   _id : ObjectId("5a2645c3ba2f887e3d01d75c"),
   sid : "12345678",
   sname : "Raymond",
   courseTaken : [
                    { cid : "COMP5331" },
                    { cid : "COMP4332" }
                ]
}
```


2. NoSQL
To output a list of new documents each of which contains all non-array fields (i.e., _id, sid and sname) from collection "tempTable" and a new non-array field containing only a single entry of the array field "couresTaken" of collection "tempTable"
NoSQL
db.tempTable.aggregate([{$unwind: "$courseTaken"}])
Output
{
_id : ObjectId("5a2645c3ba2f887e3d01d75c"),
sid : "12345678",
sname : "Raymond",
courseTaken : { cid : "COMP5331" }
}
{
_id : ObjectId("5a2645c3ba2f887e3d01d75c"),
sid : "12345678",
sname : "Raymond",
courseTaken : { cid : "COMP4332" }
}
79

79


2. NoS
To output a list of new documents each of which contains all non-array fields (i.e., _id, sid and sname) from collection "tempTable" and a new non-array field containing only a single entry of the array field "couresTaken" of collection "tempTable".
The new non-array field should contain a single value.
The "_id" field should be removed.
NoSQL
db.tempTable.aggregate([{$unwind: "$courseTaken"},
{$project: {_id:0, sid:1, sname: 1, newCid: "$courseTaken.cid"}}])
Output
{
sid : "12345678",
sname : "Raymond",
newCid : "COMP5331"
}
{
sid : "12345678",
sname : "Raymond",
newCid: "COMP4332"
}
80

80

2. NoSQ

To output a list of new documents each of which contains all non-array fields (i.e., _id, sid and sname) from collection "tempTable" and a new non-array field containing only a single entry of the array field "couresTaken" of collection "tempTable".
The new non-array field should contain a single value.
The "_id" field should be removed.
The output collection will be stored in a new collection called "tempOutput"

NoSQL

```
db.tempTable.aggregate([{$unwind: "$courseTaken"},
                        {$project: {_id:0, sid:1, sname: 1, newCid: "$courseTaken.cid"}},
                        {$out: "tempOutput"}])
```


2. NoSQL - aggregation

- Suppose that the example is changed as follows.
- That is, "tempTable" contains a student with an empty courseTaken list.

NoSQL

```
db.tempTable.insert({
                        sid: "87654321",
                        sname: "Peter",
                        courseTaken: []
                    })
```

2. NoSQL - aggregation

NoSQL

```
db.tempTable.aggregate([{$unwind: "$courseTaken"}])
```

Output

```
<no result>
```


2. NoSQL - aggregation

- Suppose that the example is changed again as follows.
- That is, "tempTable" contains 2 students where one has an empty courseTaken list but the other has an non-empty courseTaken list.

NoSQL

```
db.tempTable.insert({
                        sid: "12345678",
                        sname: "Raymond",
                        courseTaken: [{cid: "COMP5331"}, {cid: "COMP4332"}]
                    })

db.tempTable.insert({
                        sid: "87654321",
                        sname: "Peter",
                        courseTaken: []
                    })
```

2. NoSQL - aggregation

NoSQL

```
db.tempTable.aggregate([{$unwind: "$courseTaken"}])
```

Output

```
{
    _id : ObjectId("5a2645c3ba2f887e3d01d75c"),
    sid : "12345678",
    sname : "Raymond",
    courseTaken : { cid : "COMP5331" }
}
{
    _id : ObjectId("5a2645c3ba2f887e3d01d75c"),
    sid : "12345678",
    sname : "Raymond",
    courseTaken : { cid : "COMP4332" }
}
```

Same result as the previous result on the collection containing one student with a non-empty CourseTaken list.

85

2. NoSQL - aggregation

- Suppose that the example is changed again as follows.
- The example is the same as the previous example but we add one document at the end.

NoSQL

```
db.tempTable.insert({
                        sid: "12345678",
                        sname: "Raymond",
                        courseTaken: [{cid: "COMP5331"}, {cid: "COMP4332"}]
                    })

db.tempTable.insert({
                        sid: "87654321",
                        sname: "Peter",
                        courseTaken: []
                    })

db.tempTable.insert({
                        sid: "12341234",
                        sname: "Mary"
                    })
```

86

2. NoSQL - aggregation

NoSQL

```
db.tempTable.aggregate([{$unwind: "$courseTaken"}])
```

Output

```
{
    _id : ObjectId("5a2645c3ba2f887e3d01d75c"),
    sid : "12345678",
    sname : "Raymond",
    courseTaken : { cid : "COMP5331" }
}
{
    _id : ObjectId("5a2645c3ba2f887e3d01d75c"),
    sid : "12345678",
    sname : "Raymond",
    courseTaken : { cid : "COMP4332" }
}
```

Same result as the previous result on the collection containing one student with a non-empty CourseTaken list.

87

2. NoSQL - aggregation

- We will illustrate "$lookup" (which could be used in the "join" operation)
- Consider back Version 1.

2. NoSQL - aggregation

Output all students each of which is associated with a list of courses taken by him/her (in a "non-tidy" format)

NoSQL

```
db.student.aggregate(
[
  {
    $lookup: {
                    localField: "sid",
                     from: "take",
                     foreignField: "sid",
                     as: "list_course"
                  }
  }
]
)
```


89

Output

```
{
      _id : ObjectId("5a23ede1967809308848df56"),
      sid : "12345678",
      sname : "Raymond",
      byear : 1998,
      list_course : [
                { _id : ObjectId("5a23f088967809308848df66"), sid : "12345678", cid : "COMP4332" },
                { _id : ObjectId("5a23f088967809308848df67"), sid : "12345678", cid : "COMP5331" },
                { _id : ObjectId("5a23f088967809308848df68"), sid : "12345678", cid : "COMP2711" }
                   ]
}
{
      _id : ObjectId("5a23ee24967809308848df57"),
      sid : "87654321",
      sname : "Peter Chen",
      byear : 1997,
      list_course : [
                { _id : ObjectId("5a23f088967809308848df69"), sid : "87654321", cid : "COMP2011" },
                { _id : ObjectId("5a23f088967809308848df6a"), sid : "87654321", cid : "RMBI4310" }
                   ]
}

…
```

If "Raymond" does not take any courses (i.e., no documents stored in "take" related to "Raymond"), the output here becomes list_course : []

90

2. NoSQL - aggregation

Output all students each of which is associated with a list of courses taken by him/her (in a "tidy" format)

NoSQL

```
db.student.aggregate(
[
  {
    $lookup: {
                    localField: "sid",
                     from: "take",
                     foreignField: "sid",
                     as: "list_course"
                  }
  },
  { $project: {sid: 1, sname: 1, byear: 1, "list_course.cid" : 1, _id: 0}}
]
)
```


91

Output

```
{
      sid : "12345678",
      sname : "Raymond",
      byear : 1998,
      list_course : [
                { "cid" : "COMP4332" },
                { "cid" : "COMP5331" },
                { "cid" : "COMP2711" }
                   ]
}

{
      sid : "87654321",
      sname : "Peter Chen",
      byear : 1997,
      list_course : [
                { "cid" : "COMP2011" },
                { "cid" : "RMBI4310" }
                   ]
}

…
```

92

2. NoSQL - aggregation

Output all students each of which is associated with a list of courses taken by him/her (in a "tidy" format)

NoSQL

```
db.student.aggregate(
[
  {
    $lookup: {
                    localField: "sid",
                     from: "take",
                     foreignField: "sid",
                     as: "list_course"
                  }
  },
  { $project: {sid: 1, sname: 1, byear: 1, "list_course.cid" : 1, _id: 0}},
  { $unwind: "$list_course"}
]
)
```


Output

```
{
    sid : "12345678",
    sname : "Raymond",
    byear : 1998,
    list_course : { cid : "COMP4332" }
}
{
    sid : "12345678",
    sname : "Raymond",
    byear : 1998,
    list_course : { cid : "COMP5331" }
}
{
    sid : "12345678",
    sname : "Raymond",
    byear : 1998,
    list_course : { cid : "COMP2711" }
}
{
    sid : "87654321",
    sname : "Peter Chen",
    byear : 1997,
    list_course : { cid : "COMP2011" }
}
{
    sid : "87654321",
    sname : "Peter Chen",
    byear : 1997,
    list_course : { cid : "RMBI4310" }
}
...
```

2. NoSQL - aggregation

- There are more complicated examples related to aggregation
- We will illustrate with some queries in the next set of lecture notes.

2. NoSQL

- Three Major Functions
  - find
  - distinct
  - Aggregate
- Operation of Datetime

The easiest operation

The easiest operation which is used in some cases

The most complicated (or the most powerful) operation which could be used in most cases. It can also perform exactly with the same output as "find" and "distinct" but with more complex script

2. NoSQL

- We could compare two Datetime concepts in NoSQL.

NoSQL

```
db.testDate.insert({birthday1: new Date("2018-02-18T16:45:00Z")})
db.testDate.insert({birthday1: new Date("2018-02-19T16:45:00Z")})
db.testDate.insert({birthday1: new Date("2018-02-20T16:45:00Z")})
db.testDate.insert({birthday1: new Date("2018-02-21T16:45:00Z")})

db.testDate.find()
```

Output

```
{
    _id : ObjectId("5a6995be83d3283fd9392923"),
    birthday1 : ISODate("2018-02-18T16:45:00Z")
}
{
    _id : ObjectId("5a6995be83d3283fd9392924"),
    birthday1 : ISODate("2018-02-19T16:45:00Z")
}
{
    _id : ObjectId("5a6995be83d3283fd9392925"),
    birthday1 : ISODate("2018-02-20T16:45:00Z")
}
{
    _id : ObjectId("5a6995be83d3283fd9392926"),
    birthday1 : ISODate("2018-02-21T16:45:00Z")
}
```

2. Operation of Datetime

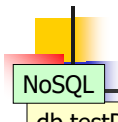
NoSQL

```
db.testDate.find( {birthday1: new Date("2018-02-19T16:45:00Z") } )
```

Find a list of birthdays in collection "testDate" which are equal to '2018-02-19 16:45:00'

Output

```
{
    _id : ObjectId("5a6995be83d3283fd9392924"),
    birthday1 : ISODate("2018-02-19T16:45:00Z")
}
```

2. Operation of Datetime

NoSQL

```
db.testDate.find( {
  $and: [    { birthday1: {$gte : new Date("2018-02-19T16:45:00Z") } },
             { birthday1: {$lte : new Date("2018-02-20T16:45:00Z") } }       ]
})
```

Find a list of birthdays in collection "testDate" which are between '2018-02-19 16:45:00' (inclusively) and '2018-02-20 16:45:00' (inclusively)

Output

```
{
    _id : ObjectId("5a6995be83d3283fd9392924"),
    birthday1 : ISODate("2018-02-19T16:45:00Z")
}
{
    _id : ObjectId("5a6995be83d3283fd9392925"),
    birthday1 : ISODate("2018-02-20T16:45:00Z")
}
```