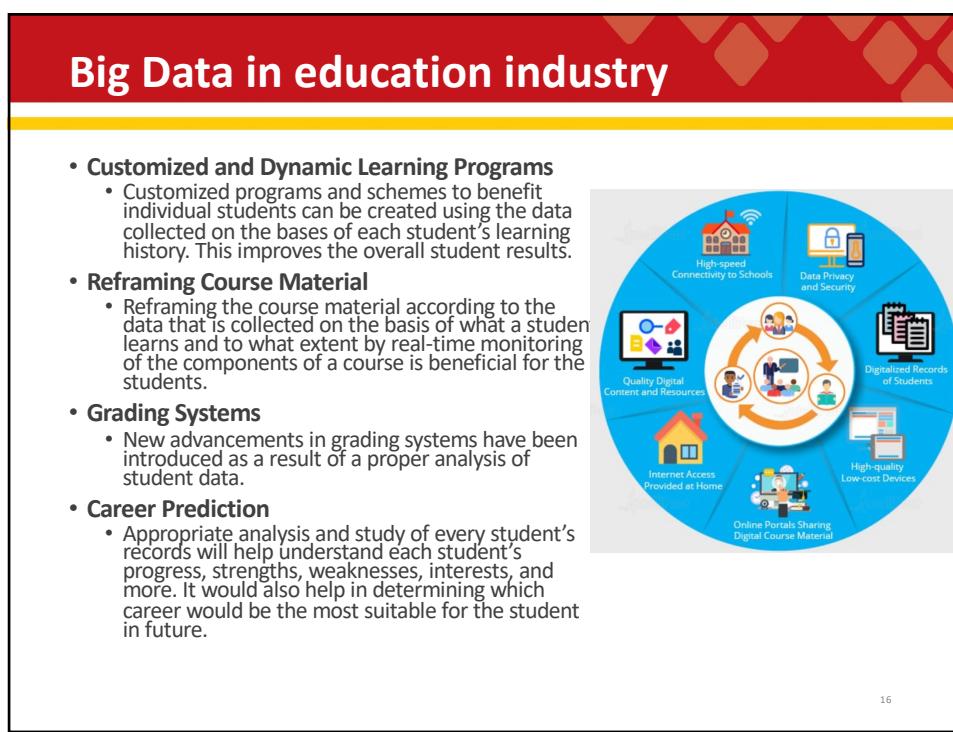


15



16

Edtech

- Coursera
- VioEdu
- <https://byjus.com/>
 - Engaging Video Lessons
 - Personalized Learning Journeys
 - Mapped to the Syllabus
 - In-depth Analysis
 - Engaging Interactive Questions

17

Big Data in healthcare industry

- Big data reduces costs of treatment since there is less chances of having to perform unnecessary diagnosis.
- It helps in predicting outbreaks of epidemics and also in deciding what preventive measures could be taken to minimize the effects of the same.
- It helps avoid preventable diseases by detecting them in early stages. It prevents them from getting any worse which in turn makes their treatment easy and effective.
- Patients can be provided with evidence-based medicine which is identified and prescribed after doing research on past medical results.

18

Big Data in government sector

• Welfare Schemes

- In making faster and informed decisions regarding various political programs
- To identify areas that are in immediate need of attention
- To stay up to date in the field of agriculture by keeping track of all existing land and livestock.
- To overcome national challenges such as unemployment, terrorism, energy resources exploration, and much more.

• Cyber Security

- Big Data is hugely used for deceit recognition.
- It is also used in catching tax evaders.



19

19

Big Data in media and entertainment industry

- Predicting the interests of audiences
- Optimized or on-demand scheduling of media streams in digital media distribution platforms
- Getting insights from customer reviews
- Effective targeting of the advertisements
- Example
 - Spotify, an on-demand music providing platform, uses Big Data Analytics, collects data from all its users around the globe, and then uses the analyzed data to give informed music recommendations and suggestions to every individual user.
 - Amazon Prime that offers, videos, music, and Kindle books in a one-stop shop is also big on using big data.

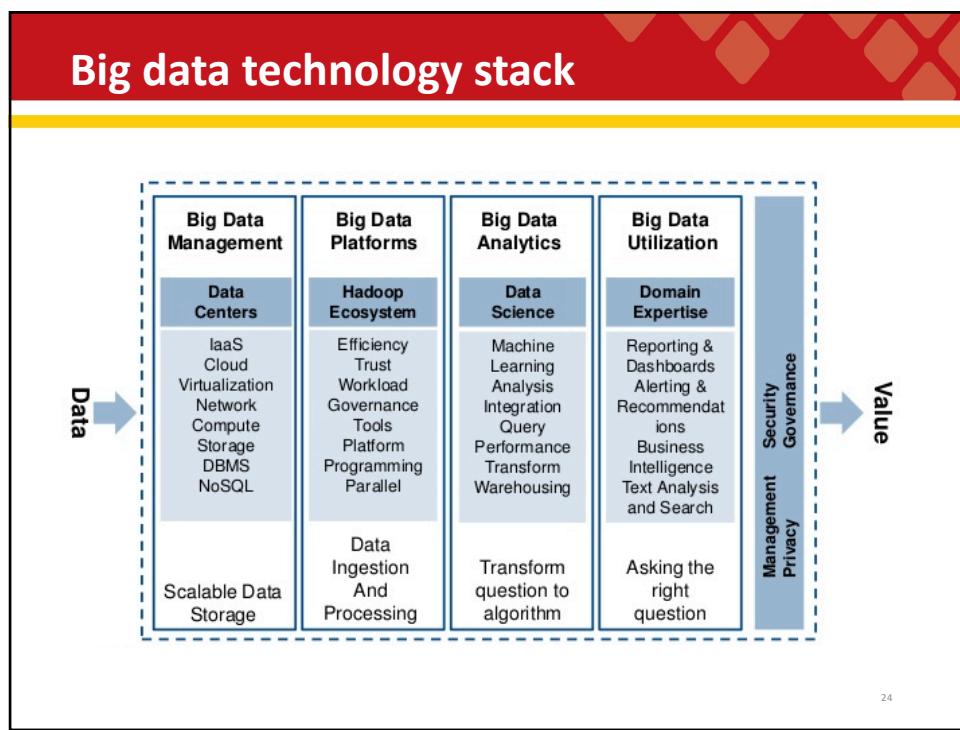


20

20



23



24

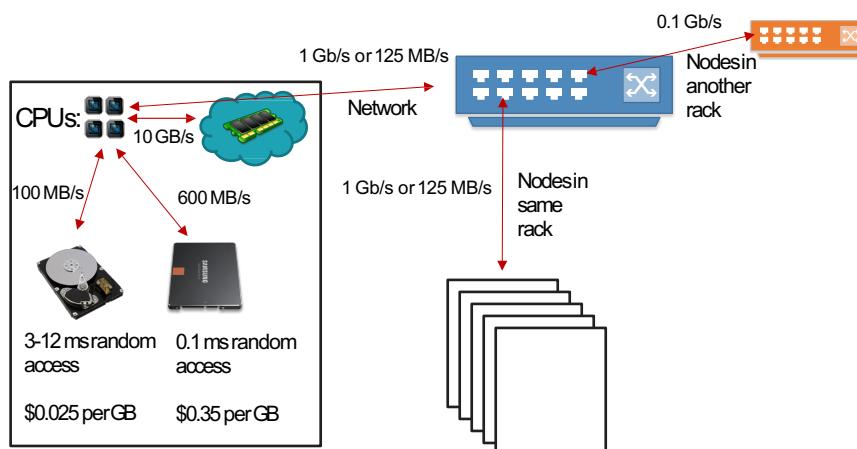
Scalable data management

- Scalability
 - Able to manage increasingly big volume of data
- Accessibility
 - Able to maintain efficiency in reading and writing data (I/O) into data storage systems
- Transparency
 - In distributed environment, users should be able to access data over the network as easily as if the data were stored locally.
 - Users should not have to know the physical location of data to access it.
- Availability
 - Fault tolerance
 - The number of users, system failures, or other consequences of distribution shouldn't compromise the availability.

25

25

Data I/O landscape



26

26

Scalable data ingestion and processing

- Data ingestion
 - Data from different complementing information systems is **to be combined to gain a more comprehensive basis** to satisfy the need
 - How to ingest data efficiently from various, distributed heterogeneous sources?
 - Different data formats
 - Different data models and schemas
 - Security and privacy
- Data processing
 - How to process massive volume of data in a timely fashion?
 - How to process massive stream of data in a real-time fashion?
 - Traditional parallel, distributed processing (OpenMP, MPI)
 - Big learning curve
 - Scalability is limited
 - Fault tolerance is hard to achieve
 - Expensive, high performance computing infrastructure
 - Novel realtime processing architecture
 - Eg. Mini-batch in Spark streaming
 - Eg. Complex event processing in Apache Flink

27

27

Scalable analytic algorithms

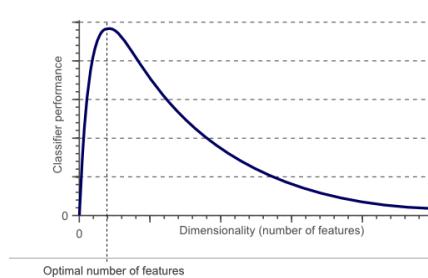
- Challenges
 - Big volume
 - Big dimensionality
 - Realtime processing
- Scaling-up Machine Learning algorithms
 - Adapting the algorithm to handle Big Data in a single machine.
 - Eg. Sub-sampling
 - Eg. Principal component analysis
 - Eg. feature extraction and feature selection
 - Scaling-up algorithms by parallelism
 - Eg. k-nn classification based on MapReduce
 - Eg. scaling-up support vector machines (SVM) by a divide and-conquer approach

28

28

Eg. Curse of dimensionality

- The required number of samples (to achieve the same accuracy) grows exponentially with the number of variables!
- In practice: number of training examples is fixed!
=> the classifier's performance usually will degrade for a large number of features!



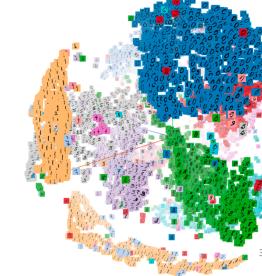
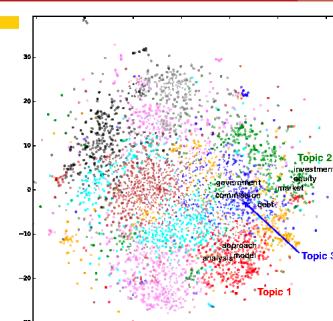
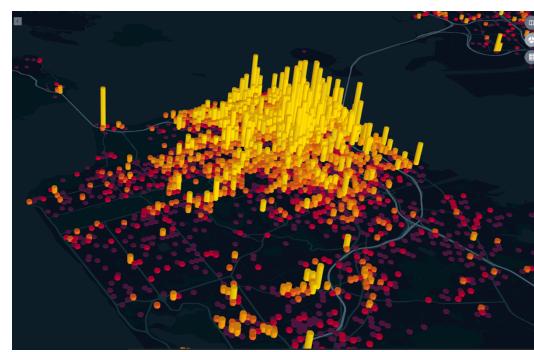
In fact, after a certain point, increasing the dimensionality of the problem by adding new features would actually degrade the performance of classifier.

29

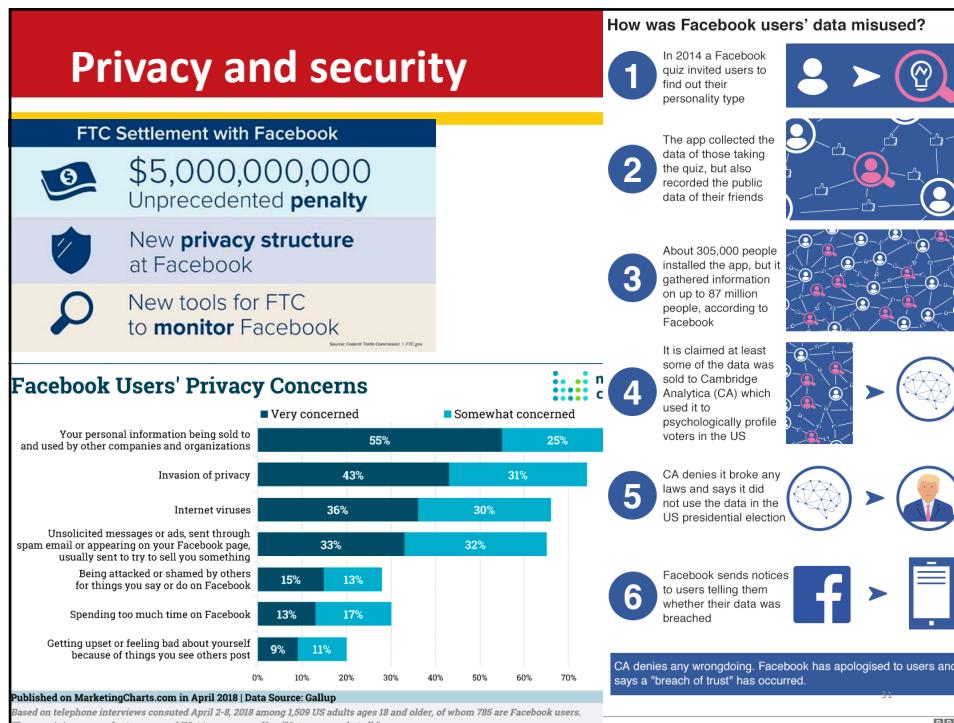
29

Utilization and interpretability of big data

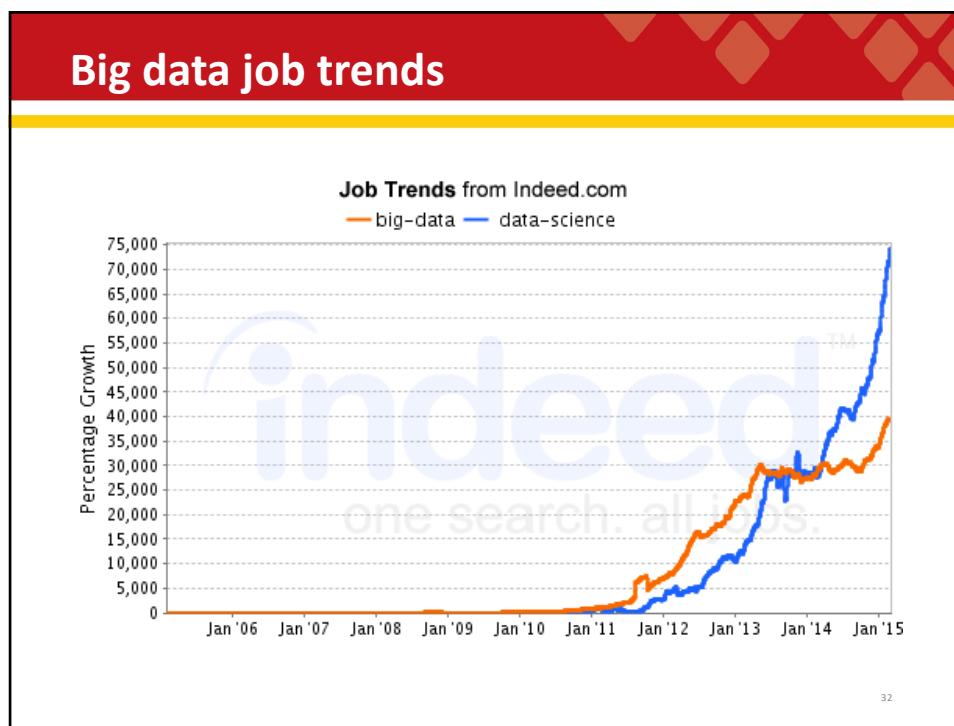
- Domain expertise to findout problems and interpret analytics results
- Scalable visualization and interpretability of million data points
 - to facilitate their interpretability and understanding

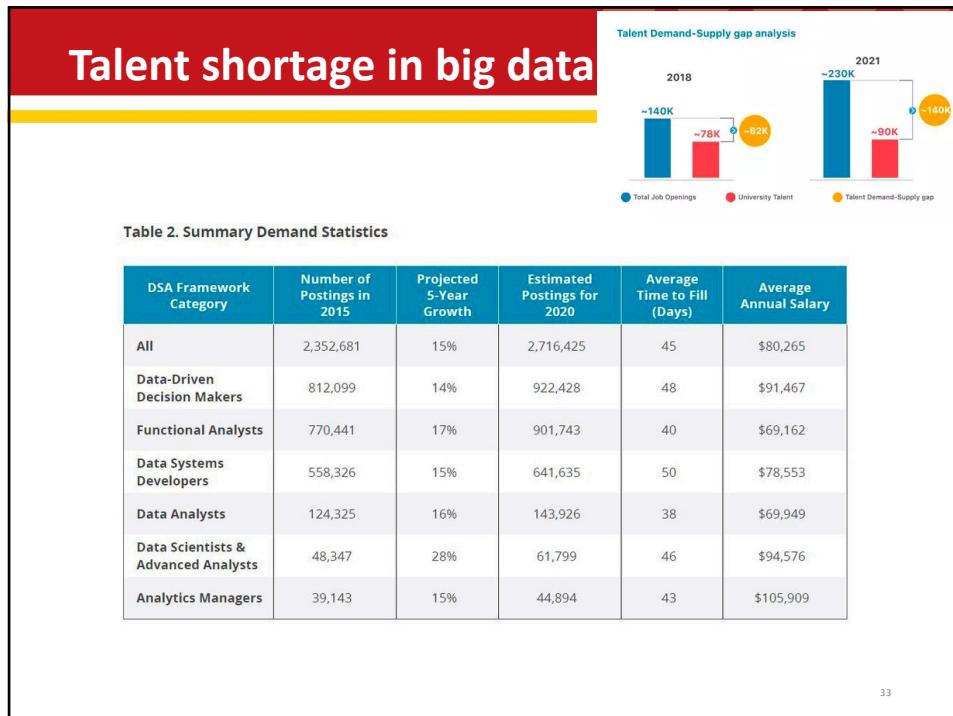


30



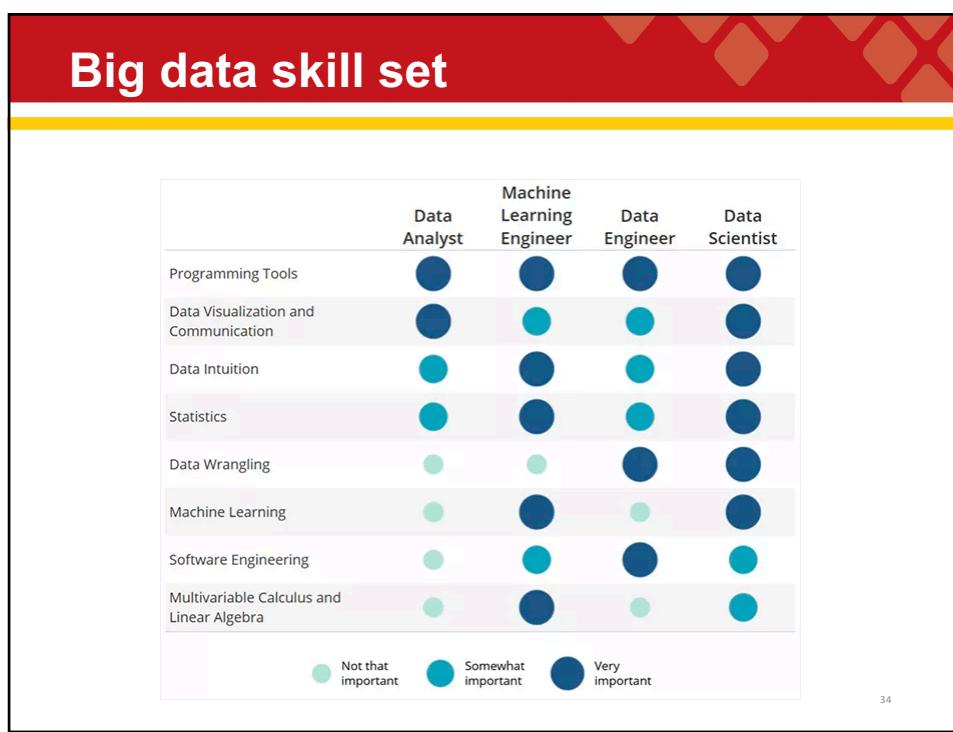
31





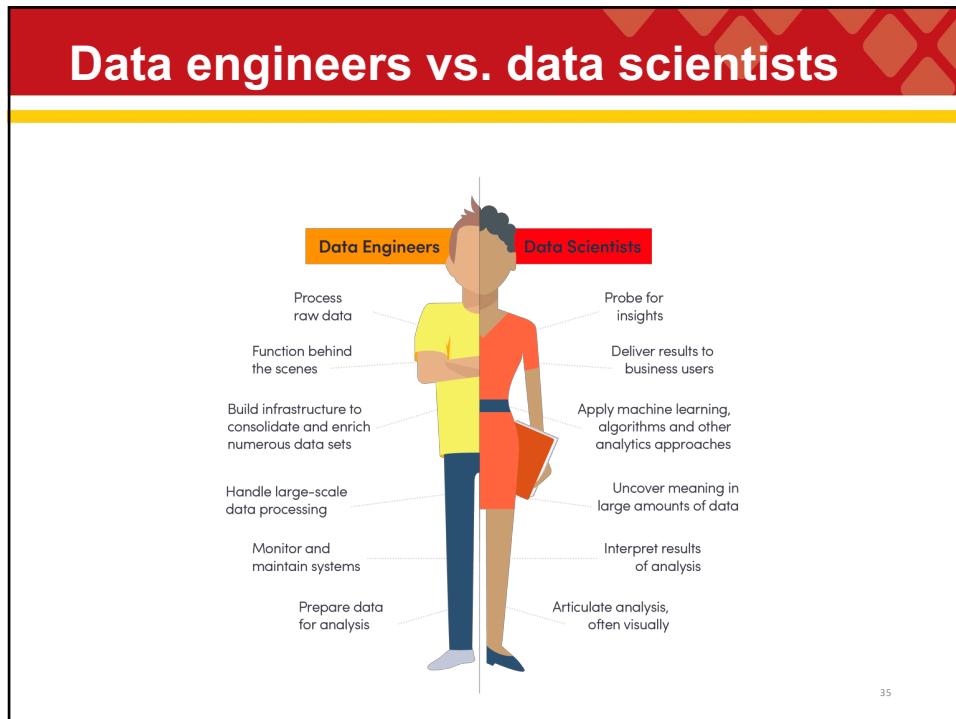
33

33



34

34



35

How to land big data related jobs

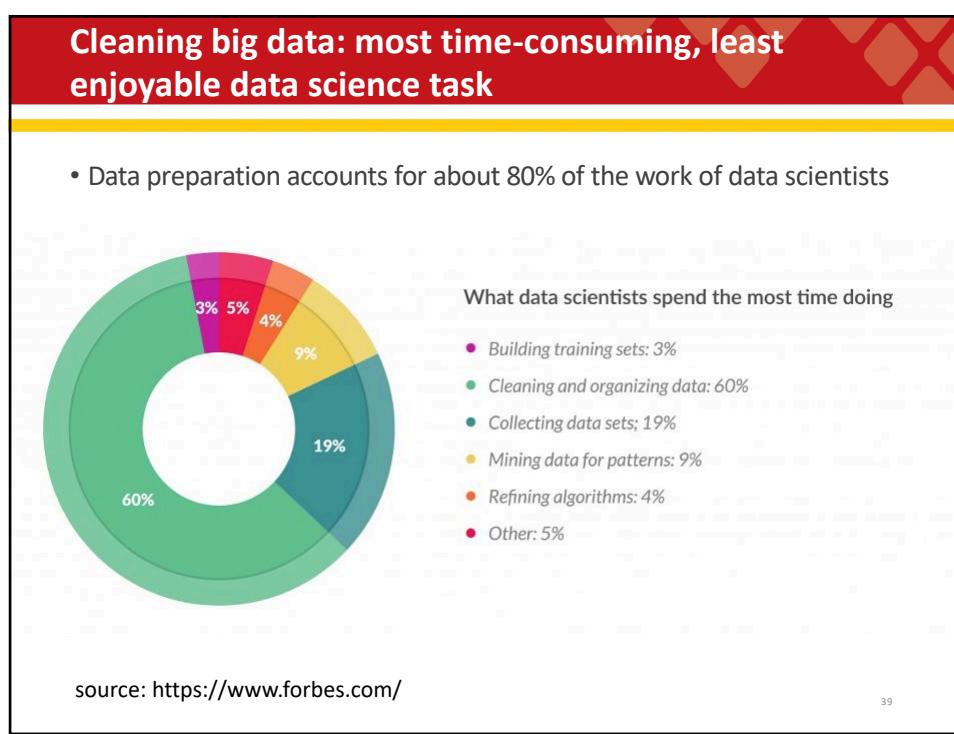
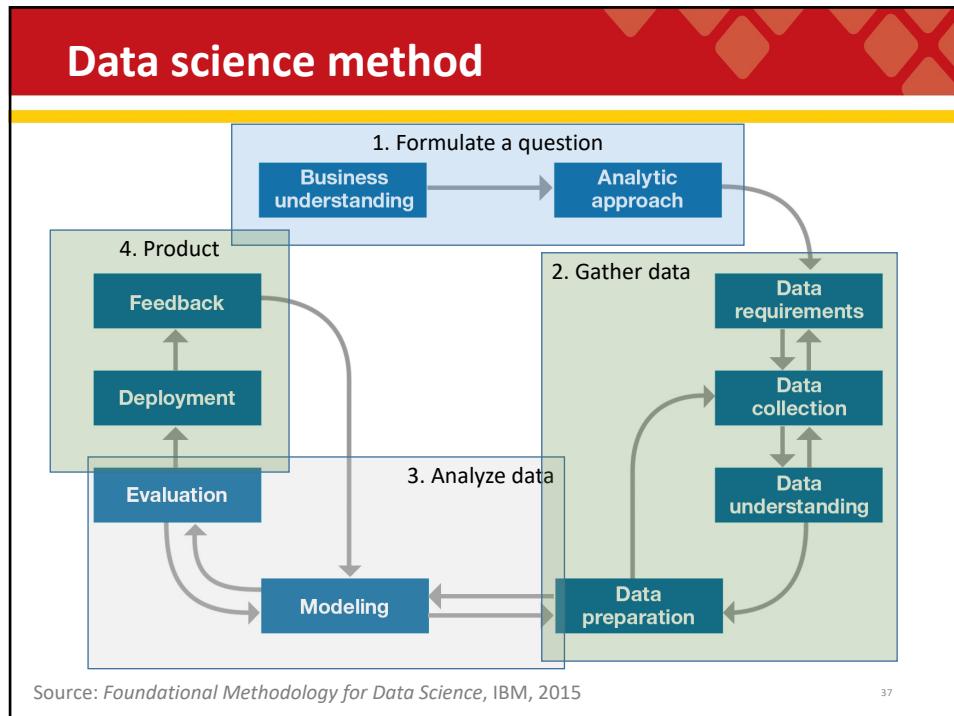
- Learn to code
 - Coursera
 - Udacity
 - Freecodecamp
 - Codecademy
- Math, Stats and machine learning
 - Kaggle
- Hadoop, NoSQL, Spark
- Visualization and Reporting
 - Tableau
 - Pentaho
- Meetup & Share
- Find a mentor
- Internships, projects

The Home of Data Science & Machine Learning
Kaggle helps you learn, work, and play

Create an account or Host a competition

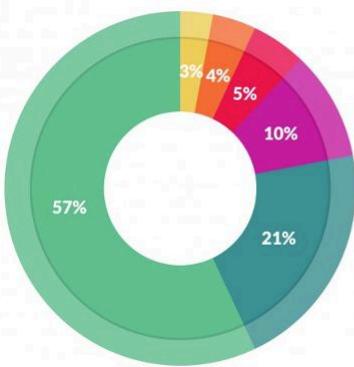
36

36



Cleaning big data: most time-consuming, least enjoyable data science task

- 57% of data scientists regard cleaning and organizing data as the least enjoyable part of their work and 19% say this about collecting data sets.



Part of Data Science	Percentage
Cleaning and organizing data	57%
Collecting data sets	21%
Mining data for patterns	10%
Refining algorithms	4%
Building training sets	3%
Other	5%

What's the least enjoyable part of data science?

- Building training sets: 10%
- Cleaning and organizing data: 57%
- Collecting data sets: 21%
- Mining data for patterns: 3%
- Refining algorithms: 4%
- Other: 5%

40

40

References

- [1] Tiwari, Shashank. Professional NoSQL. John Wiley & Sons, 2011.
- [2] Lam, Chuck. Hadoop in action. Manning Publications Co., 2010.
- [3] Miner, Donald, and Adam Shook. MapReduce design patterns: building effective algorithms and analytics for Hadoop and other systems. "O'Reilly Media, Inc.", 2012.
- [4] Karau, Holden. Fast Data Processing with Spark. Packt Publishing Ltd, 2013.
- [5] Penchikala, Srinivas. Big data processing with apache spark. Lulu. com, 2018.
- [6] White, Tom. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.
- [7] Gandomi, Amir, and Murtaza Haider. "Beyond the hype: Big data concepts, methods, and analytics." International Journal of Information Management 35.2 (2015): 137-144.
- [8] Cattell, Rick. "Scalable SQL and NoSQL data stores." Acm Sigmod Record 39.4 (2011): 12-27.
- [9] Gessert, Felix, et al. "NoSQL database systems: a survey and decision guidance." Computer Science-Research and Development 32.3-4 (2017): 353-365.
- [10] George, Lars. HBase: the definitive guide: random access to your planet-size data. " O'Reilly Media, Inc.", 2011.
- [11] Sivasubramanian, Swaminathan. "Amazon dynamoDB: a seamlessly scalable non-relational database service." Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, 2012.
- [12] Chan, L. "Presto: Interacting with petabytes of data at Facebook." (2013).
- [13] Garg, Nishant. Apache Kafka. Packt Publishing Ltd, 2013.
- [14] Karau, Holden, et al. Learning spark: lightning-fast big data analysis. " O'Reilly Media, Inc.", 2015.
- [15] Iqbal, Muhammad Hussain, and Tariq Rahim Soomro. "Big data analysis: Apache storm perspective." International journal of computer trends and technology 19.1 (2015): 9-14.
- [16] Toshniwal, Ankit, et al. "Storm@twitter." Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014.
- [17] Lin, Jimmy. "The lambda and the kappa." IEEE Internet Computing 21.5 (2017): 60-66.

41

41



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



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

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
 - Hỗn hối phần cứng là chuyện thường ngày, không phải là ngoại lệ
 - Hadoop chịu lỗi thông qua kỹ thuật “dư thừa”
- Các tập 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 **input => split => map => ...**
 - 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

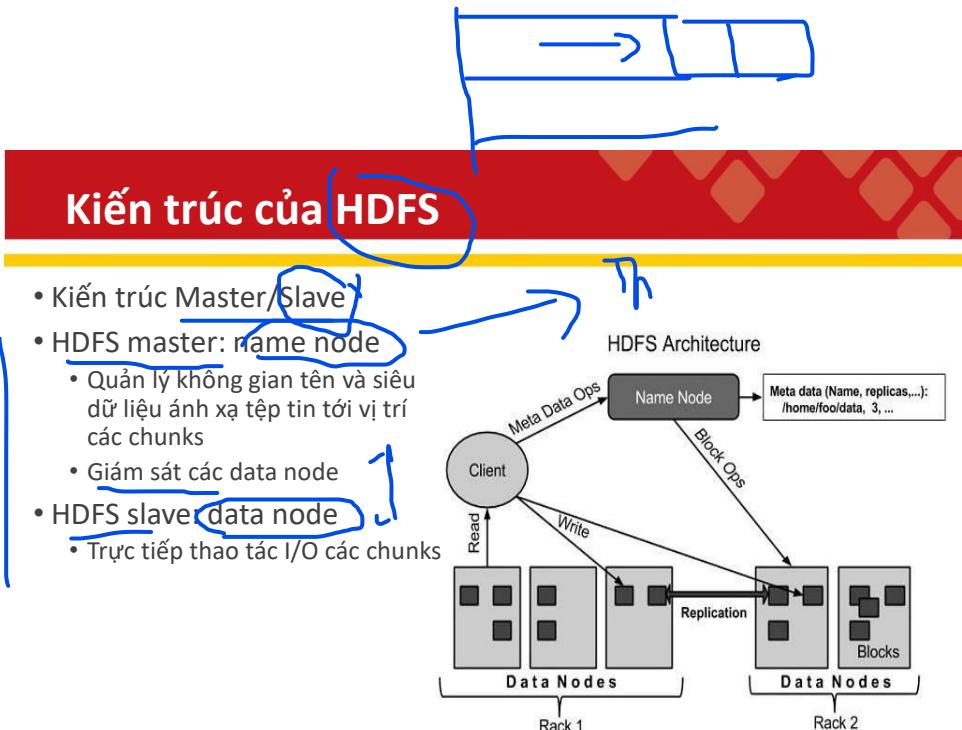
11

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

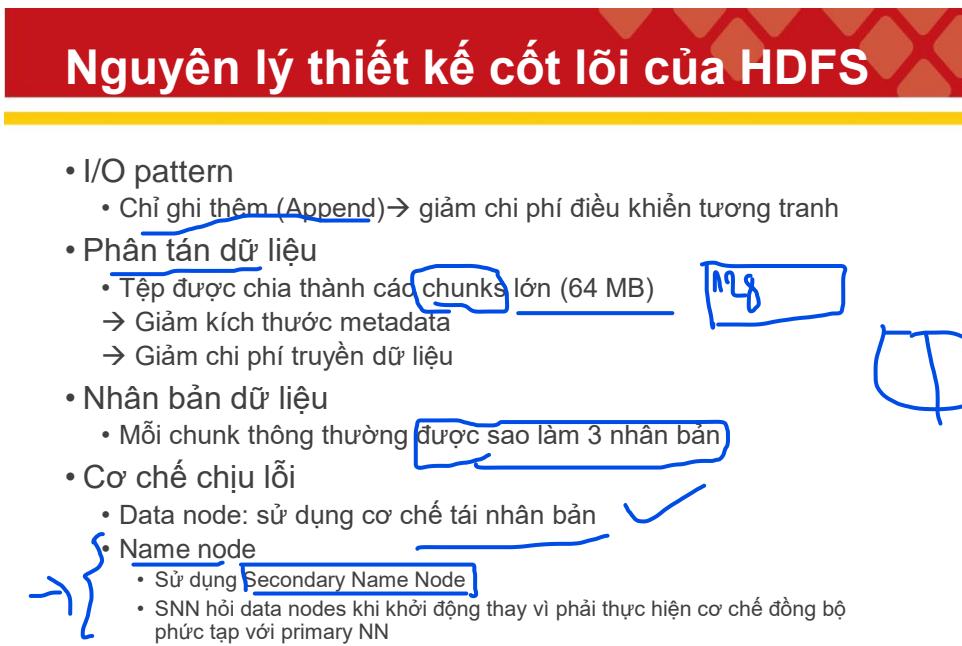
12

12



13

13



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

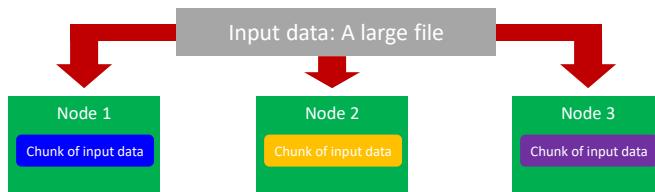
- Mỗi chương trình MapReduce là một công việc (job) được phân rã làm nhiều tác vụ độc lập (task) và các tác vụ này được phân tán trên các nodes khác nhau của cụm để thực thi
- Mỗi tác vụ được thực thi độc lập với các tác vụ khác để đạt được tính khả mở
 - Giảm truyền thông giữa các node máy chủ
 - Tránh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

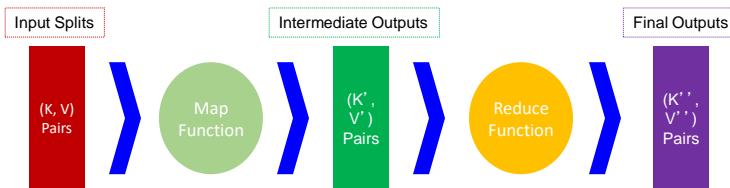


17

17

Chương trình MapReduce

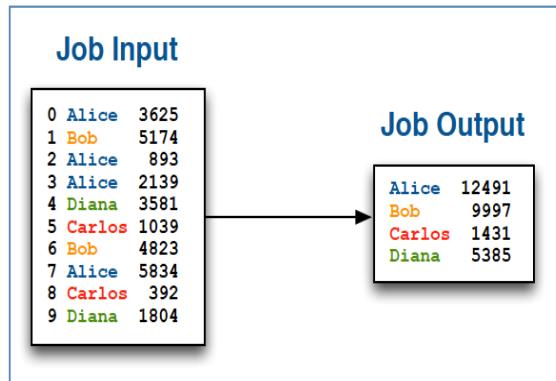
- Lập trình với MapReduce cần cài đặt 2 hàm Map và Reduce
 - 2 hàm này được thực thi bởi các tiến trình Mapper và Reducer tương ứng.
- Trong chương trình MapReduce, dữ liệu được nhìn nhận như là các cặp khóa – giá trị (key – value)
- Các hàm Map và Reduce nhận đầu vào và trả về đầu ra các cặp (key –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

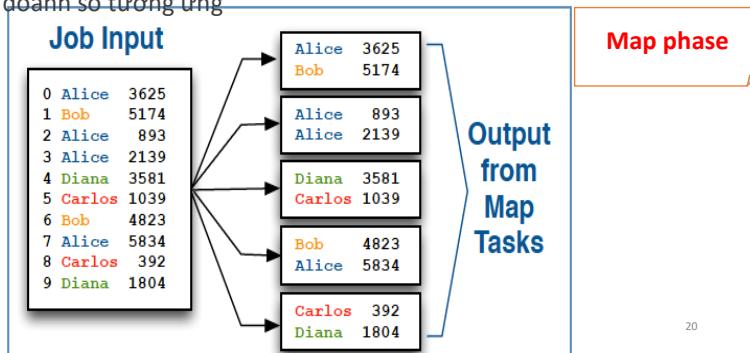


19

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

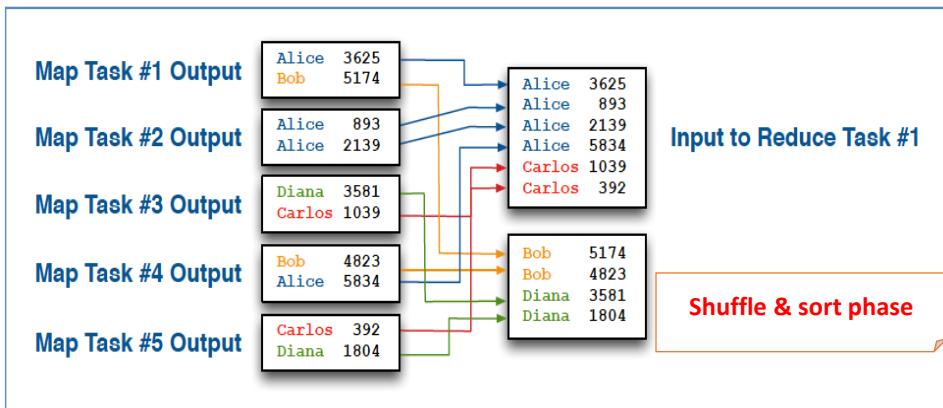


20

20

Bước shuffle & sort

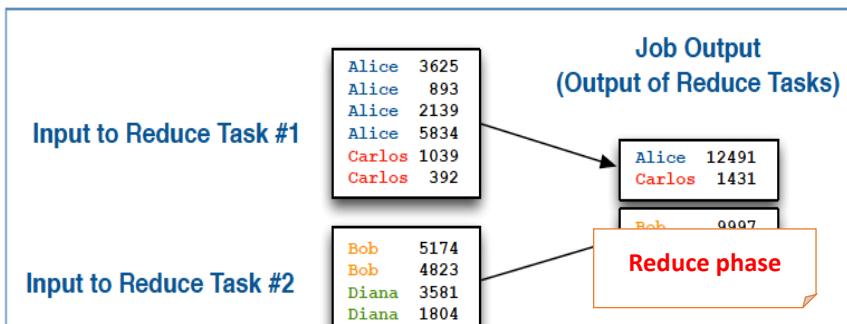
- Hadoop tự động sắp xếp và gộp đầu ra của các Mappers theo các partitions
 - Mỗi partitions là đầu vào cho một Reducer



21

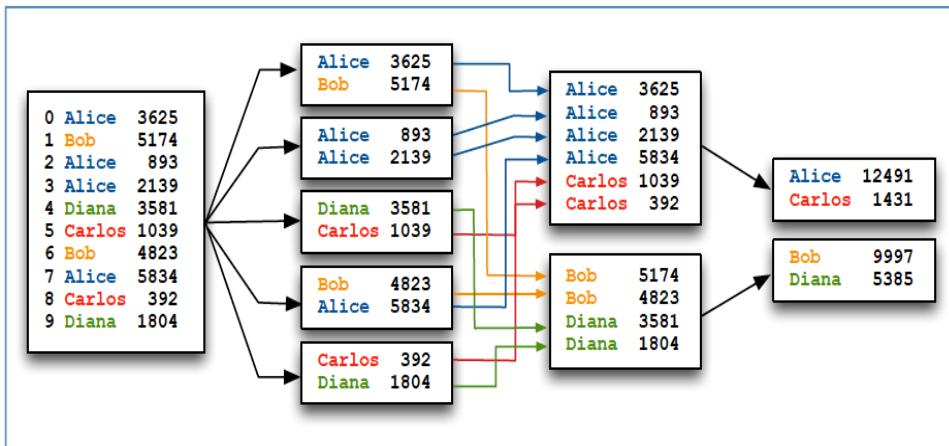
Bước Reduce

- 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

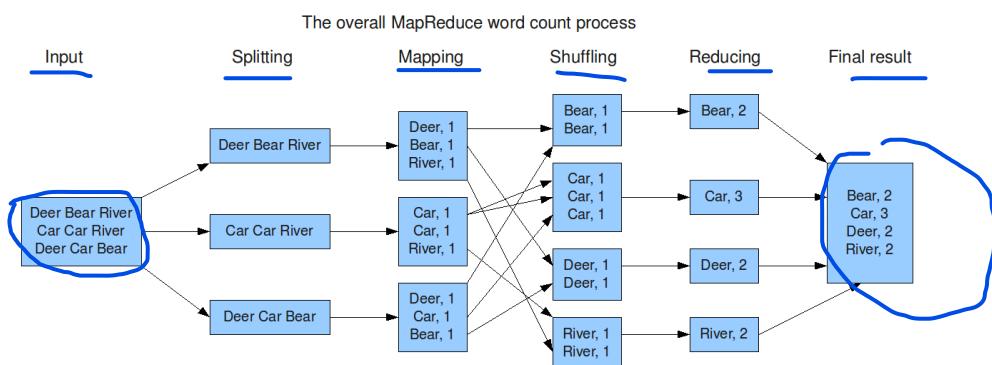
Luồng dữ liệu cho ví dụ MapReduce



23

23

Luồng dữ liệu với bài toán Word Count



24

Chương trình Word Count thực tế (1)

```

9 import org.apache.hadoop.mapreduce.Job;
10 import org.apache.hadoop.mapreduce.Mapper;
11 import org.apache.hadoop.mapreduce.Reducer;
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.util.GenericOptionsParser;
15
16
17
18
19 public class WordCount {
20 public static void main(String [] args) throws Exception
21 {
22 Configuration c=new Configuration();
23 String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
24 Path input=new Path(files[0]);
25 Path output=new Path(files[1]);
26 Job j=new Job(c,"wordcount");
27 j.setJarByClass(WordCount.class);
28 j.setMapperClass(MapForWordCount.class);
29 j.setReducerClass(ReduceForWordCount.class);
30 j.setOutputKeyClass(Text.class);
31 j.setOutputValueClass(IntWritable.class);
32 FileInputFormat.addInputPath(j, input);
33 FileOutputFormat.setOutputPath(j, output);
34 System.exit(j.waitForCompletion(true)?0:1);
35 }

```

25

Chương trình Word Count thực tế (2)

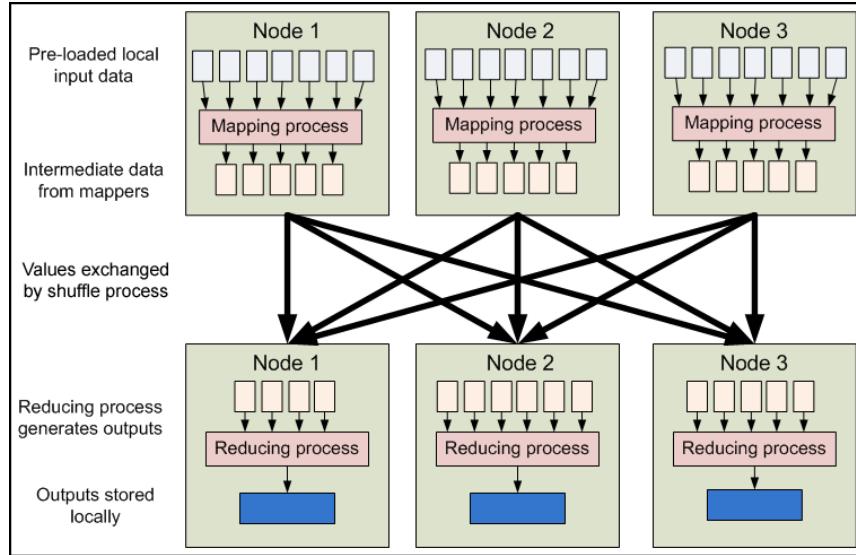
```

36 public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{
37 public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException
38 {
39 String line = value.toString();
40 String[] words=line.split(",");
41 for(String word: words )
42 {
43     Text outputKey = new Text(word.toUpperCase().trim());
44     IntWritable outputValue = new IntWritable(1);
45     con.write(outputKey, outputValue);
46 }
47 }
48 }
49
50 public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>
51 {
52 public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedException
53 {
54 int sum = 0;
55 for(IntWritable value : values)
56 {
57 sum += value.get();
58 }
59 con.write(word, new IntWritable(sum));
60 }

```

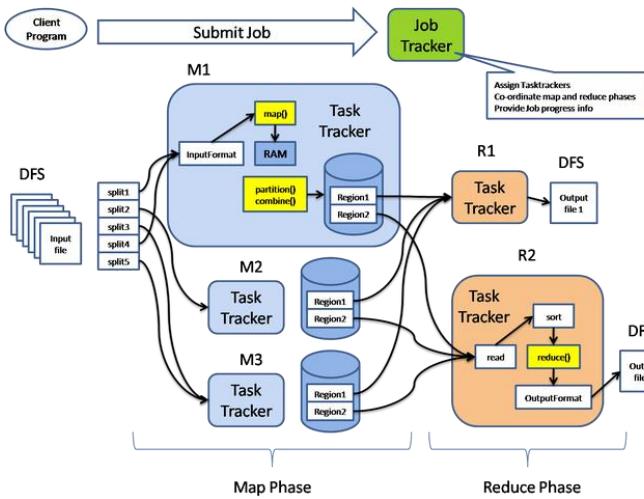
26

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
 - Độ trìngh 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
 - Độ trễ 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 QTCSL truyền thống
 - Là NoSQL : không có ngôn ngữ truy vấn mức cao như SQL
 - Phải sử dụng API để scan/ put/ get/ dữ liệu theo khóa

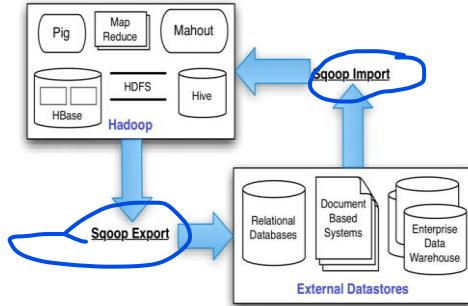


32

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

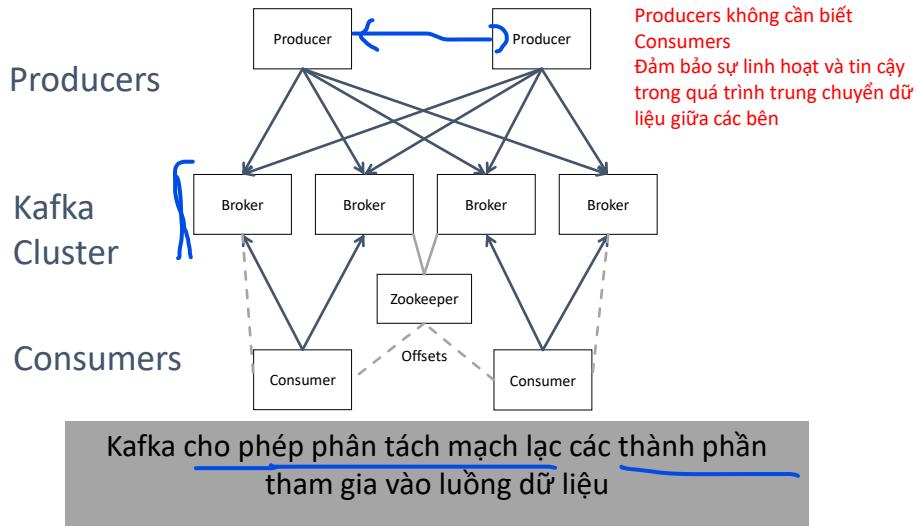


sqoop

33

33

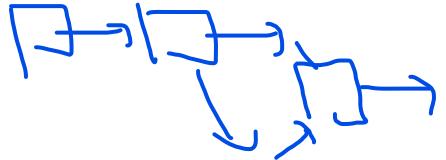
Apache Kafka



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

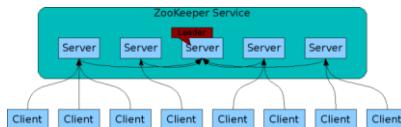


35

35

Apache Zookeeper

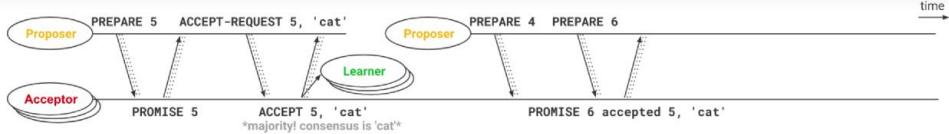
- Apache ZooKeeper là một dịch vụ cung cấp các chức năng phối hợp phân tán độ tin cậy cao
 - Quản lý thành viên trong nhóm máy chủ
 - Bầu cử leader
 - Quản lý thông tin cấu hình động
 - Giám sát trạng thái hệ thống
- Đây là các service lõi, tối quan trọng trong các hệ thống phân tán



36

36

PAXOS algorithm



- ⇒ **Proposer** wants to propose a certain value:
It sends **PREPARE** ID_p to a majority (or all) of **Acceptors**.
 ID_p 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) ID_p .
- ⇒ **Acceptor** receives a **PREPARE** message for ID_p :
Did it promise to ignore requests with this ID_p ?
Yes → then ignore
No → Will promise to ignore any request lower than ID_p .
Has it ever accepted anything? (assume accepted $ID = ID_a$)
Yes → Reply with **PROMISE** ID_p accepted $ID_a, value$.
No → Reply with **PROMISE** ID_p .
- ★ If a majority of acceptors promise, no $ID < ID_p$ can make it through.

- ⇒ **Proposer** gets majority of **PROMISE** messages for a specific ID_p :
It sends **ACCEPT-REQUEST** $ID_p, value$ to a majority (or all) of **Acceptors**.
Has it got any already accepted value from promises?
Yes → It picks the value with the highest ID_a that it got.
No → It picks any value it wants.
- ⇒ **Acceptor** receives an **ACCEPT-REQUEST** message for ID_p , value:
Did it promise to ignore requests with this ID_p ?
Yes → then ignore
No → Reply with **ACCEPT** $ID_p, value$. Also send it to all **Learners**.
- ★ If a majority of acceptors accept $ID_p, value$, consensus is reached.
Consensus is and will always be on **value** (not necessarily ID_p).
- ⇒ **Proposer** or **Learner** get **ACCEPT** messages for ID_p , value:
★ If a proposer/learner gets majority of accept for a specific ID_p , they know that consensus has been reached on **value** (not ID_p).

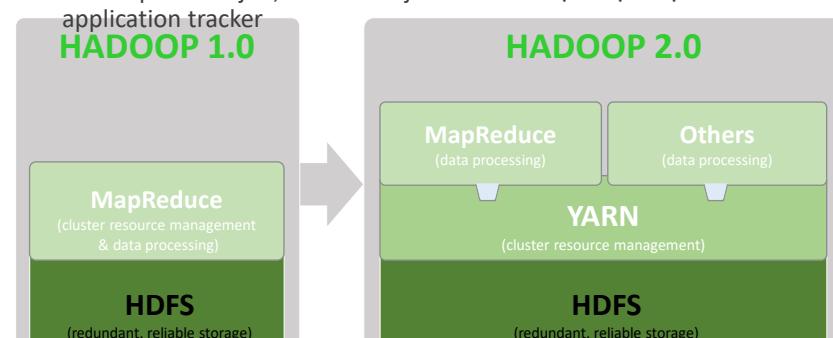
https://www.youtube.com/watch?v=d7nAGI_NZPk

37

37

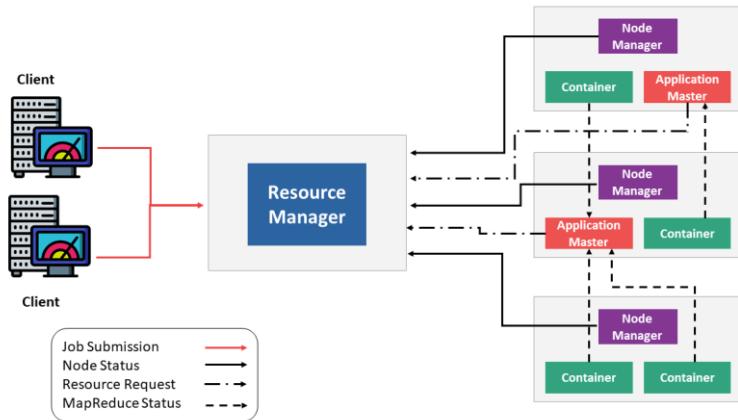
YARN – Yet Another Resource Negotiator

- 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

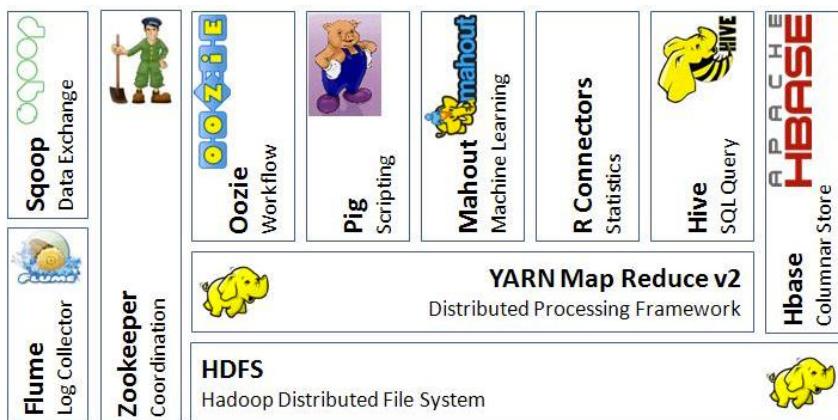


trungtv@soict.hust.edu.vn

39

39

Bức tranh tổng thể hệ sinh thái Hadoop



40

Các platform quản lý dữ liệu lớn

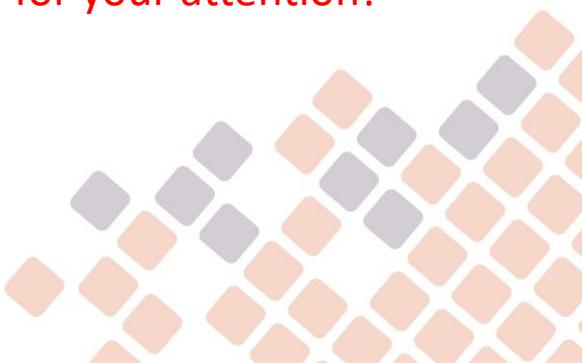


41



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

Thank you for your attention!
Q&A



42

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

4/26/2023

3

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.

4/26/2023

4

4

Data Characteristics

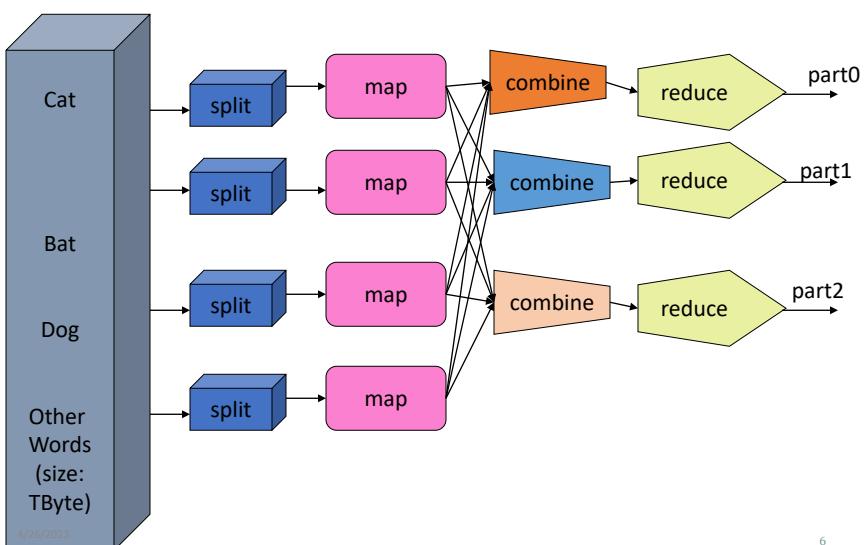
- Streaming data access
- Applications need streaming access to data
- Batch processing rather than interactive user access.
- Large data sets and files: gigabytes to terabytes size
- High aggregate data bandwidth
- Scale to hundreds of nodes in a cluster
- Tens of millions of files in a single instance
- 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.

4/26/2023

5

5

MapReduce



4/26/2023

6

6

Architecture

4/26/2023

7

7

Namenode and Datanodes

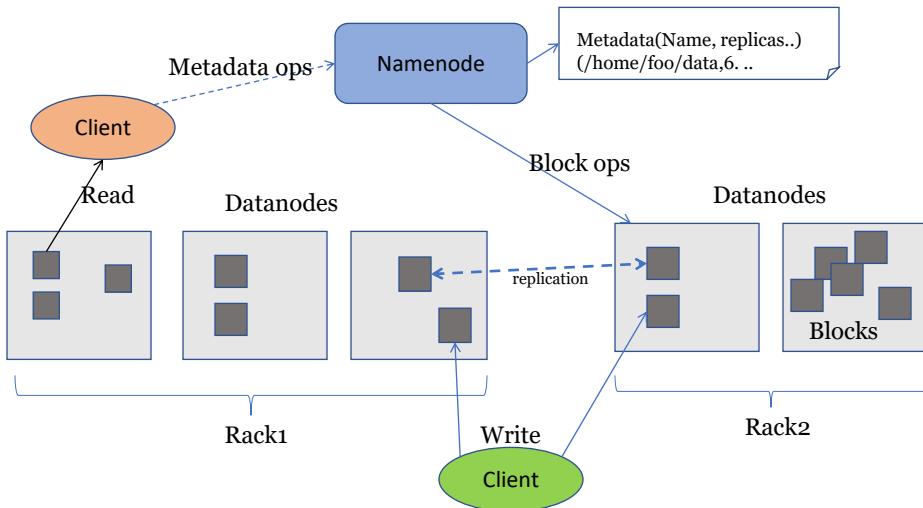
- Master/slave architecture
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **DataNodes** usually one per node in a cluster.
- The DataNodes manage storage attached to the nodes 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.

4/26/2023

8

8

HDFS Architecture



4/26/2023

9

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.

4/26/2023

10

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.

4/26/2023

11

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.

4/26/2023

12

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.

4/26/2023

13

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.

4/26/2023

14

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 FslImage. Stored in Namenode's local filesystem.

4/26/2023

15

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 FslImage and Editlog from its local file system, update FslImage with EditLog information and then stores a copy of the FslImage on the filesystem as a checkpoint.
- Periodic checkpointing is done. So that the system can recover back to the last checkpointed state in case of a crash.

4/26/2023

16

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

Protocol

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.

4/26/2023

19

19

Robustness

4/26/2023

20

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.

4/26/2023

21

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.

4/26/2023

22

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.

4/26/2023

23

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**.

4/26/2023

24

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 client 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.

4/26/2023

25

25

Metadata Disk Failure

- FslImage 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 FslImage and EditLog.
- Multiple copies of the FslImage and EditLog files are updated synchronously.
- Meta-data is not data-intensive.
- The Namenode could be single point failure => secondary namenode

4/26/2023

26

26

Data Organization

4/26/2023

27

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.

4/26/2023

28

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.

4/26/2023

29

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).

4/26/2023

30

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.

4/26/2023

31

31

API (Accessibility)

4/26/2023

32

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.

4/26/2023

33

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.

4/26/2023

34

34

Space Reclamation

- When a file is deleted by a client, HDFS renames file to a file in 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.

4/26/2023

35

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).

4/26/2023

36

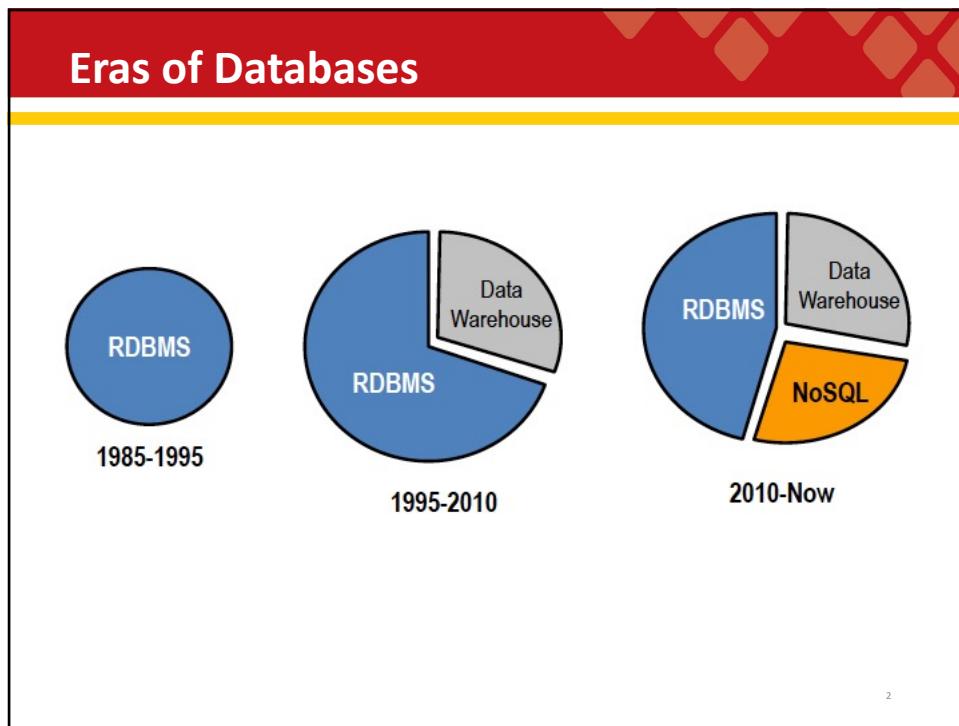
36

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

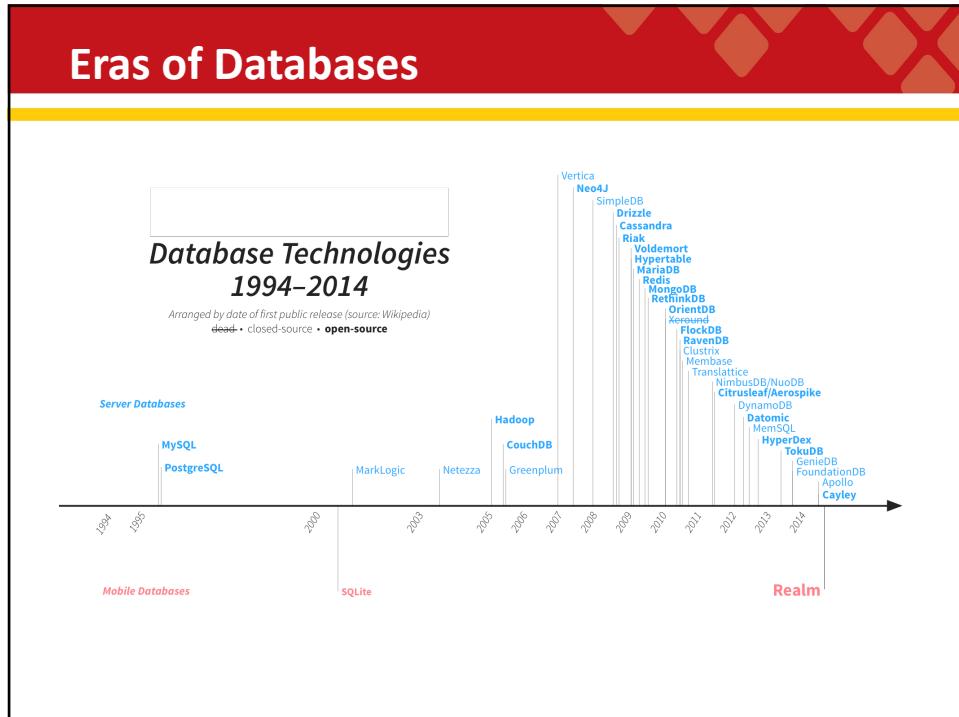
NoSQL part 1

Viet-Trung Tran
School of Information and Communication Technology

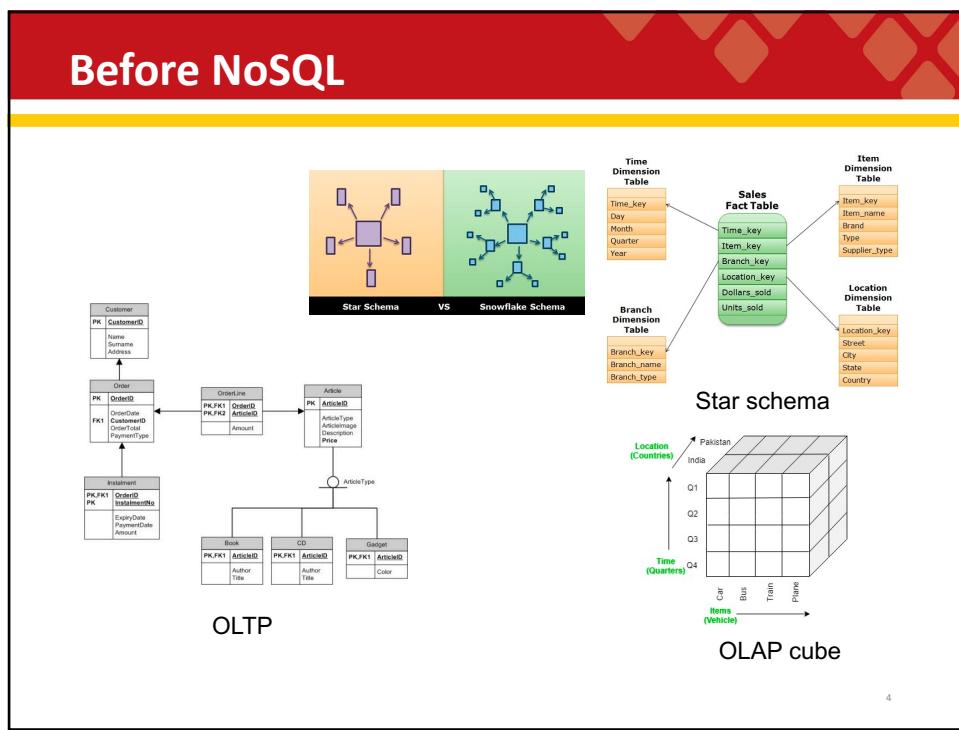
1



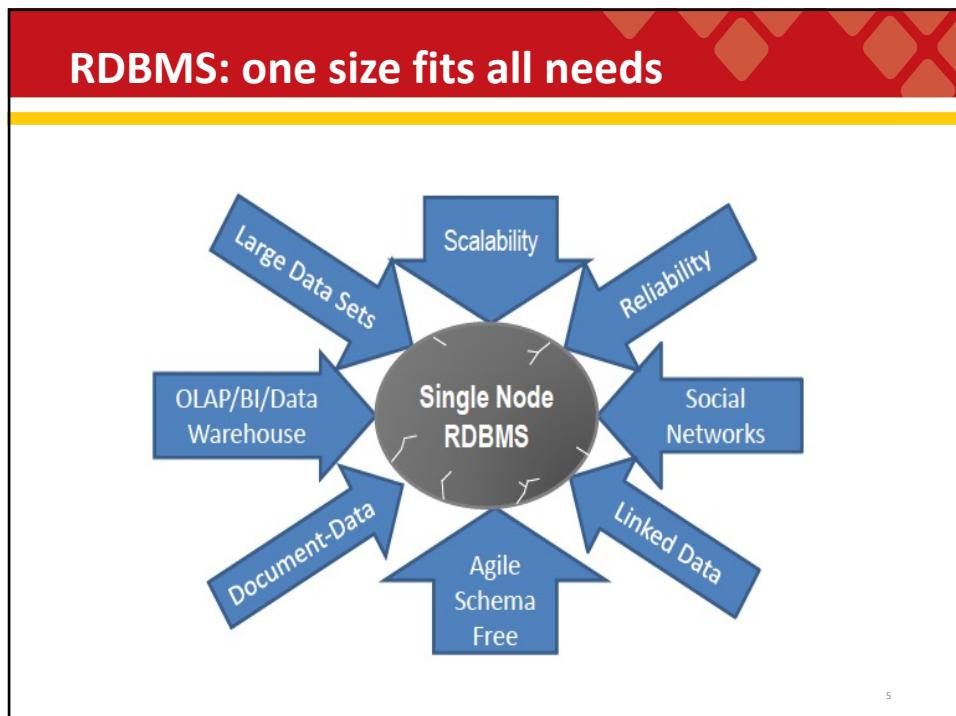
2



3



4



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.

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](#)



2005 Article

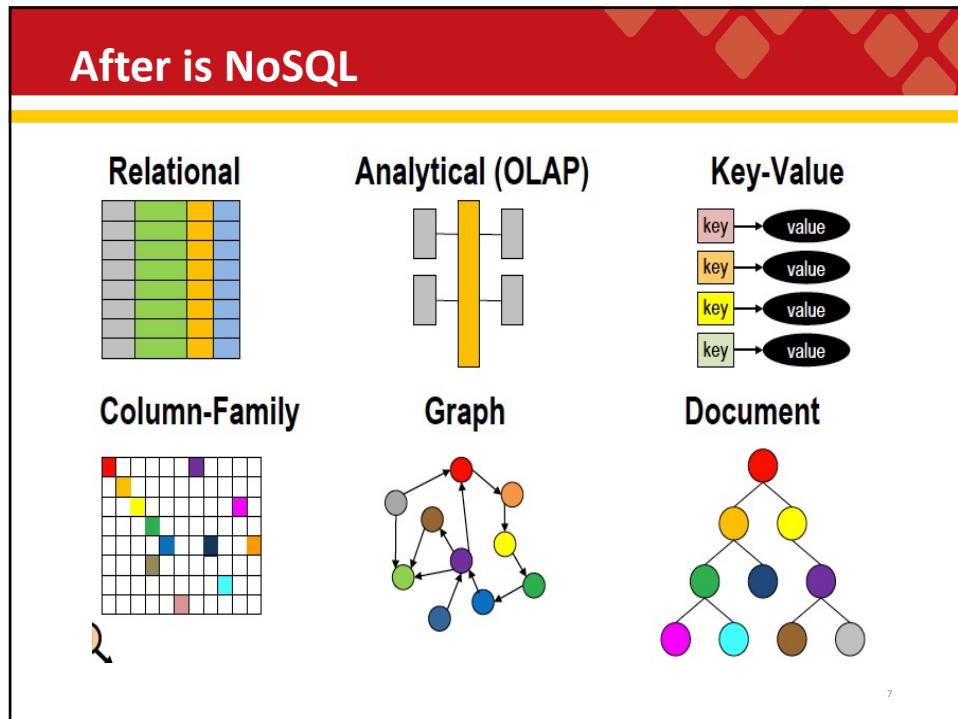
 **Bibliometrics**

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

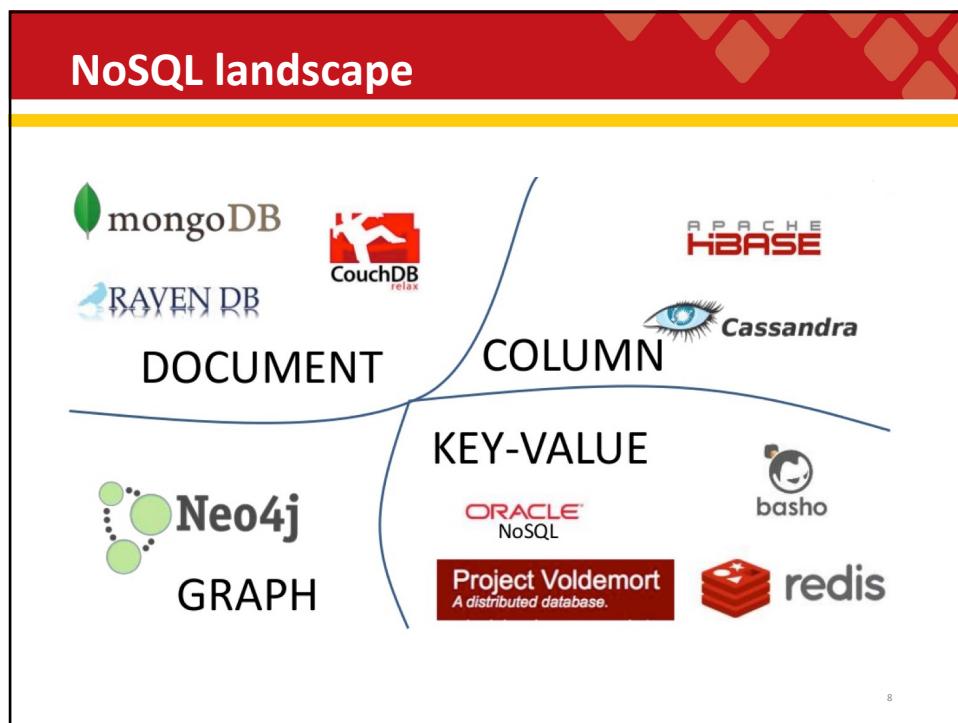
The last 25 years of commercial DBMS development can be summed up in a single phrase: "one size fits all". This phrase refers to the fact that **the traditional DBMS architecture (originally designed and optimized for business data processing) has been used to support many data-centric applications** 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



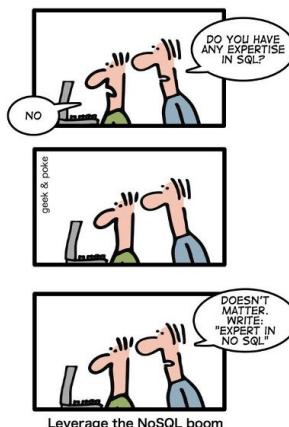
7



8

How to write a CV

HOW TO WRITE A CV



9

các bng + qh
qh => khai báo qh trc khi nhp dl
ng truy vn => SQL : select * from...
where join

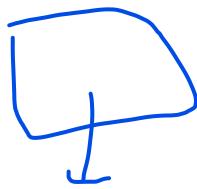
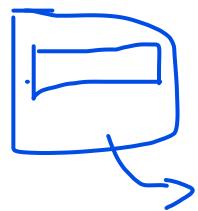
Why NoSQL

- 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

bng => ko qh => ko có câu lệnh truy vn

10

10



```
new Table(..)  
a = Table::get(..)  
if( a.id = b.id_a ) {  
    return...  
}
```

25/03/2021

SQL vs NoSQL

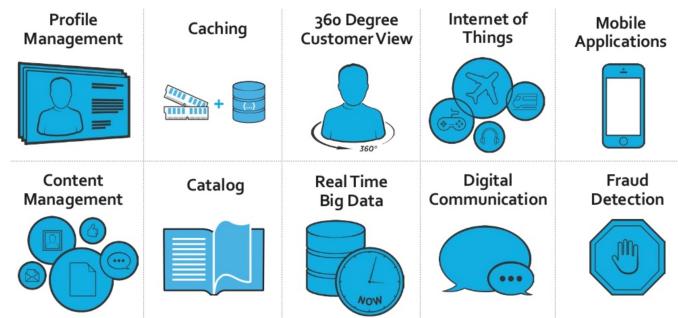
SQL	NoSQL
Gigabytes to Terabytes	Petabytes(1kTB) to Exabytes(1kPB) to Zetabytes(1kEB)
Centralized	Distributed
Structured	<u>Semi structured and Unstructured</u>
<u>Structured Query Language</u>	<u>No declarative query language</u>
Stable Data Model	Schema less
Complex Relationships	Less complex relationships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



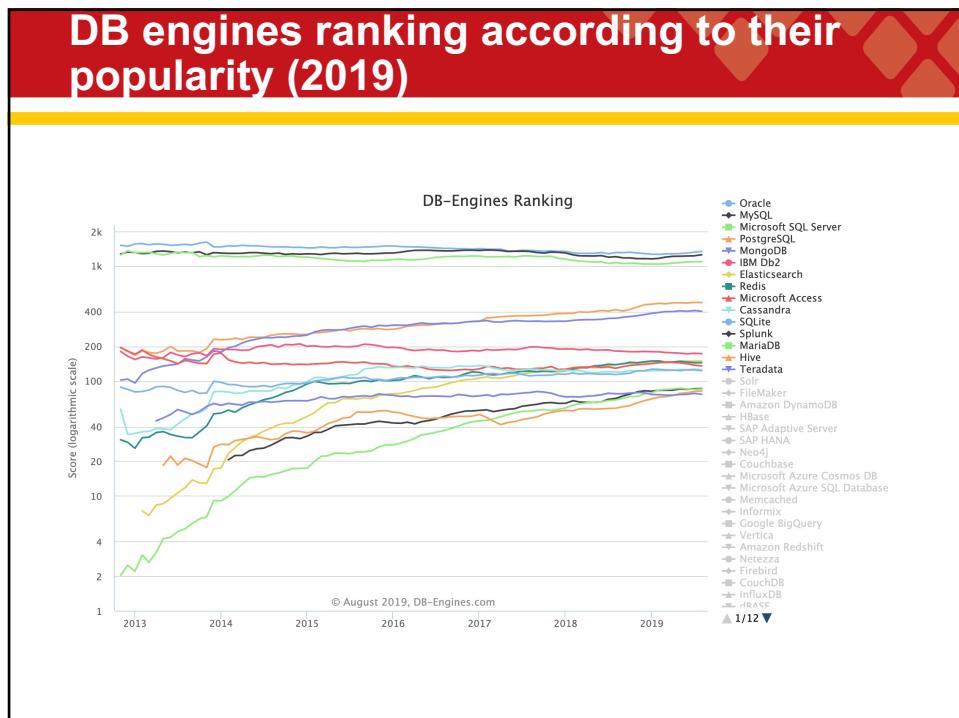
11

NoSQL use cases

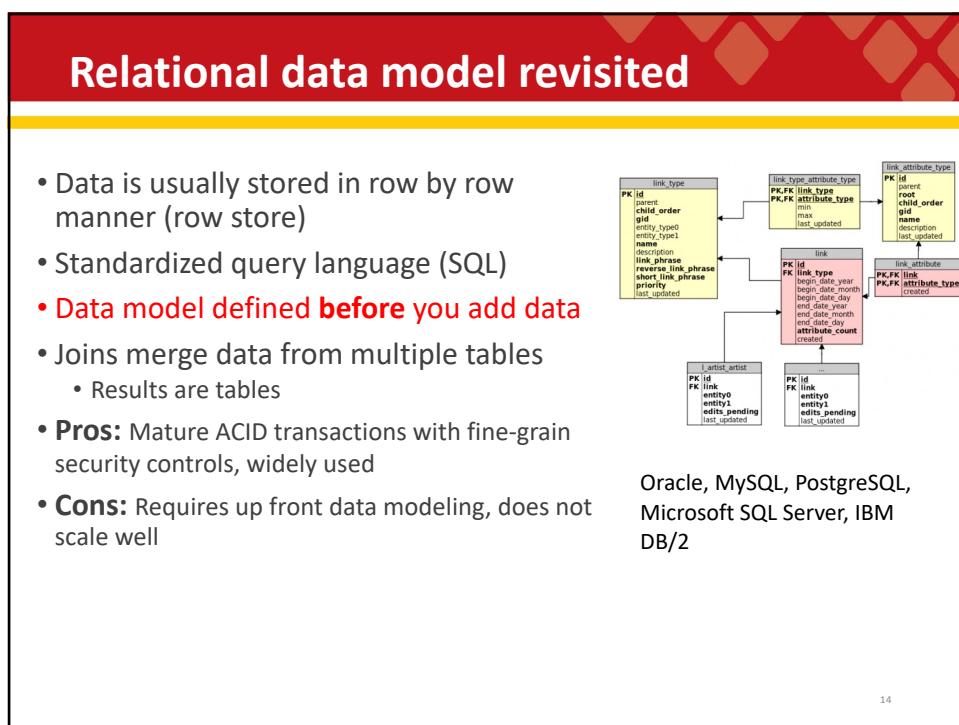
- Massive data volume at scale (Big volume)
 - Google, Amazon, Yahoo, Facebook – 10-100K servers
- Extreme query workload (Big velocity)
- High availability
- Flexible, schema evolution



12



13



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

The diagram illustrates how a single large table is partitioned into three smaller tables. The original table has columns 'PRODUCT' and 'PRICE'. It contains rows for various products: WIDGET (\$11), GIZMO (\$8), TRINKET (\$37), THINGAMAJIG (\$18), DOODAD (\$40), and TCHOTCHKE (\$899). Three arrows point from this original table to three separate smaller tables, each containing a subset of the original data. The first shard contains TRINKET and THINGAMAJIG. The second shard contains GIZMO and DOODAD. The third shard contains WIDGET and TCHOTCHKE.

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

The diagram shows a row of blue lockers. A callout bubble points to one specific locker, illustrating the concept of a key-value store. The bubble contains the text "Key: [key icon]" and "Value: An arbitrary container data [box icon]", where the box icon represents a blob datatype.

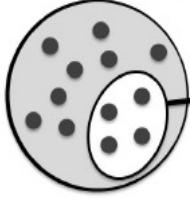
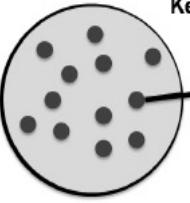
Key	Value

string datatype

Blob datatype

16

Key/value vs. Relational data model

 <p>Traditional Relational Model</p> <ul style="list-style-type: none"> • Result set based on row values • Value of rows for large data sets must be indexed • Values of columns must all have the same data type 	 <p>Key-Value Store Model</p> <ul style="list-style-type: none"> • All queries return a single item • No indexes on values • Values may contain any data type
--	--

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

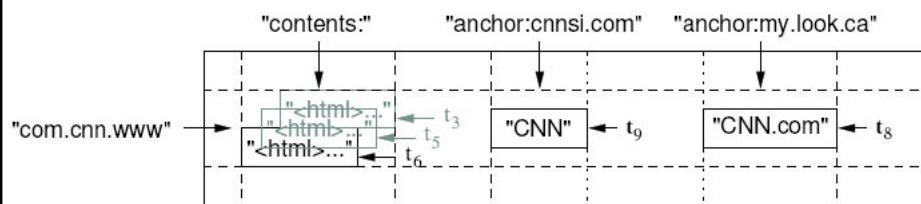


21

21

Column family store

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

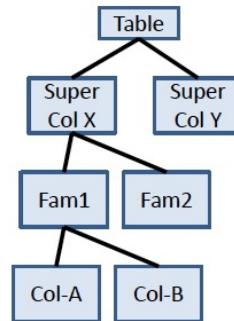


22

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

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"

Key

Row-ID	Column Family	Column Name	Timestamp	Value
--------	---------------	-------------	-----------	-------

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.

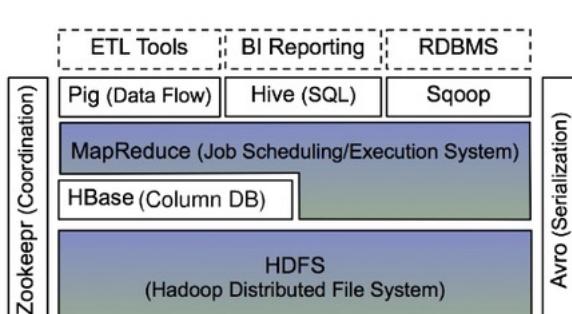


Published in:
 - Journal
 ACM Transactions on Computer Systems (TOCS) [TOCS Homepage](#) and
 Volume 26 Issue 2, June 2008
 Article No. 4
 ACM New York, NY, USA
[table of contents](#) doi:>[10.1145/1365815.1365816](#)

25

Apache Hbase

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



The diagram illustrates the Apache Hbase architecture, structured into several layers:

- Zookeeper (Coordination)**: On the far left, a vertical column labeled "Zookeeper (Coordination)" contains "MapReduce (Job Scheduling/Execution System)" and "HBase (Column DB)".
- ETL Tools**: A dashed-line box containing "Pig (Data Flow)", "Hive (SQL)", and "Sqoop".
- BI Reporting**: A dashed-line box.
- RDBMS**: A dashed-line box.
- HDFS (Hadoop Distributed File System)**: The bottom-most layer, shown in blue.
- Avro (Serialization)**: On the far right, a vertical column labeled "Avro (Serialization)" contains "HBase (Column DB)" and "HDFS (Hadoop Distributed File System)".

APACHE HBASE

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

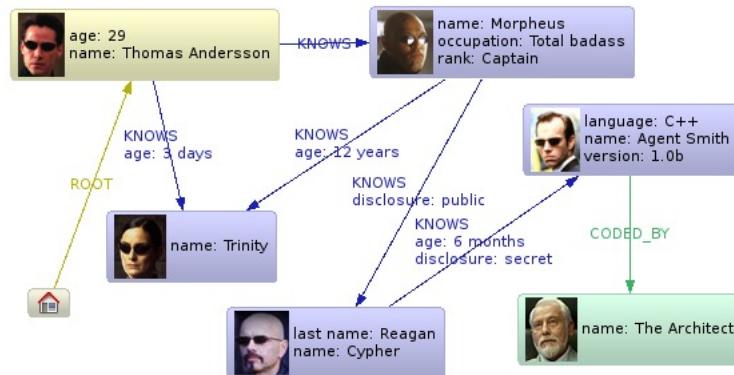


27

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

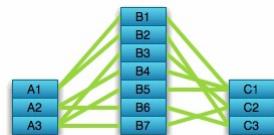


29

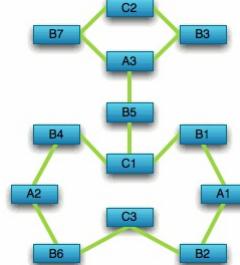
29

Compared to Relational Databases

Optimized for aggregation



Optimized for connections



30

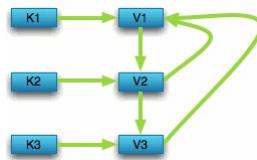
15

Compared to Key Value Stores

Optimized for simple look-ups



Optimized for traversing connected data



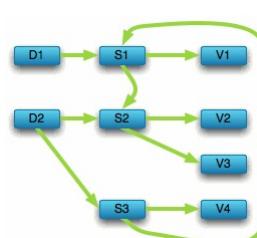
31

Compared to Document Stores

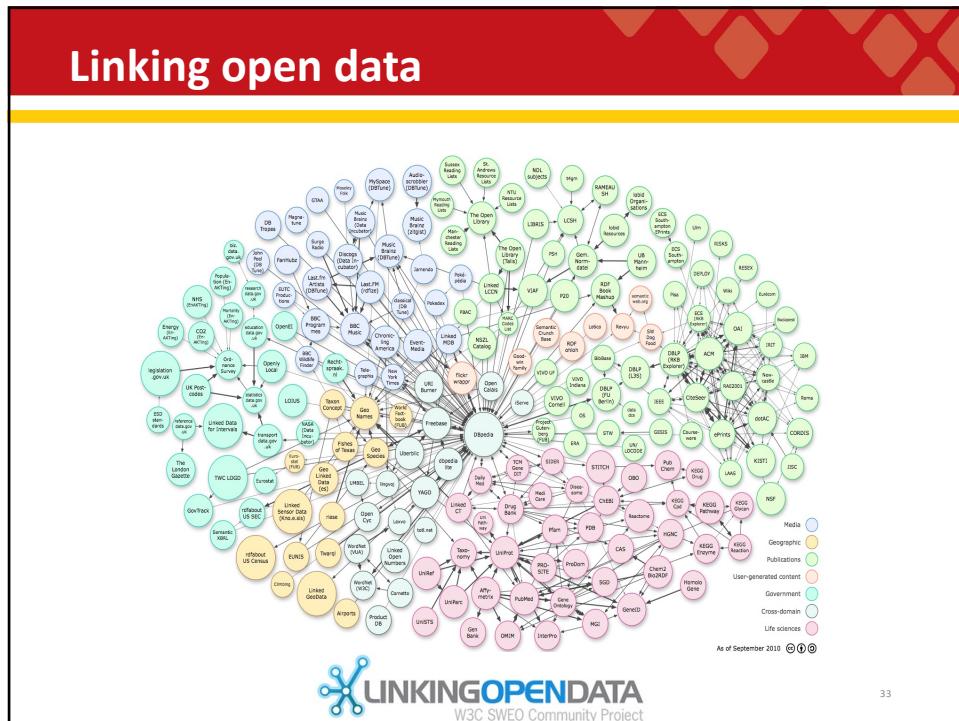
Optimized for “trees” of data



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



32



33



Document store

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

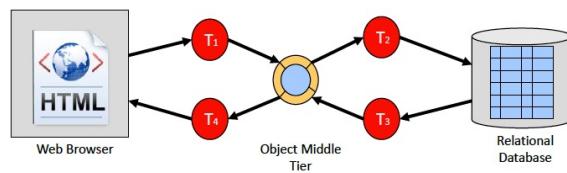
```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```

35

35

Relational data mapping

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

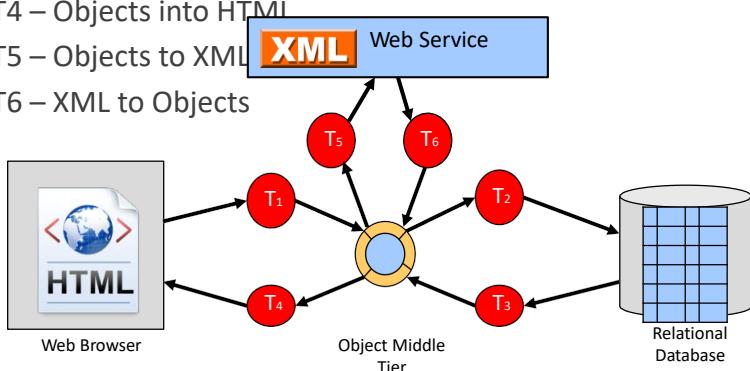


36

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

Discussion

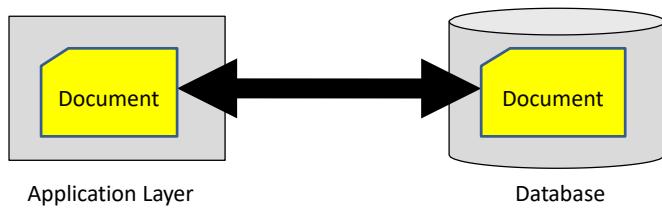
- Object-relational mapping has become one of the most complex components of building applications today
 - Java Hibernate Framework
 - JPA
- To avoid complexity is to keep your architecture very simple

38

38

Document mapping

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



39

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

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

The diagram illustrates the MongoDB architecture. At the top, a 'Clients' icon is connected to a 'mongos' instance. The 'mongos' instance is connected to three separate 'Shard' boxes. Each shard contains a 'mongod (PRIMARY)' instance and two 'mongod (SECONDARY)' instances. All these nodes are connected to a 'Config Servers' box containing three 'mongod' instances. Arrows indicate the flow of 'request' and 'response' between clients and the mongos, and between the mongos and the shards.

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

Apache CouchDB™ is a database that uses **JSON** for documents, **JavaScript** for **MapReduce** indexes, and regular **HTTP** for its **API**

42



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

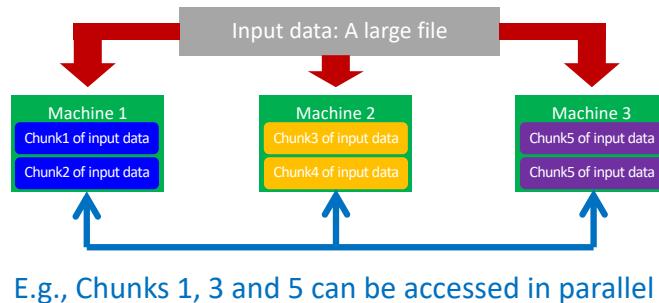
Scaling Traditional Databases

- 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?

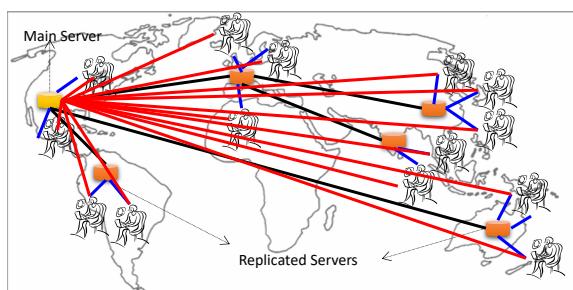


E.g., Chunks 1, 3 and 5 can be accessed in parallel

3

Data replicating

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

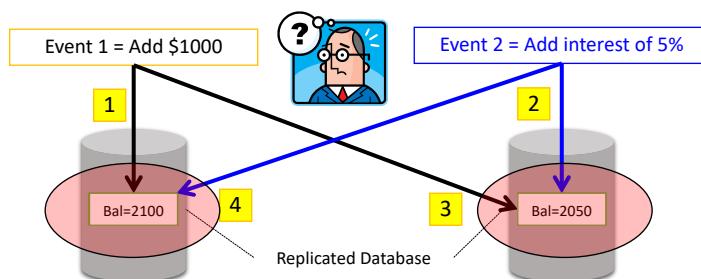


4

But, Consistency Becomes a Challenge

- An example:

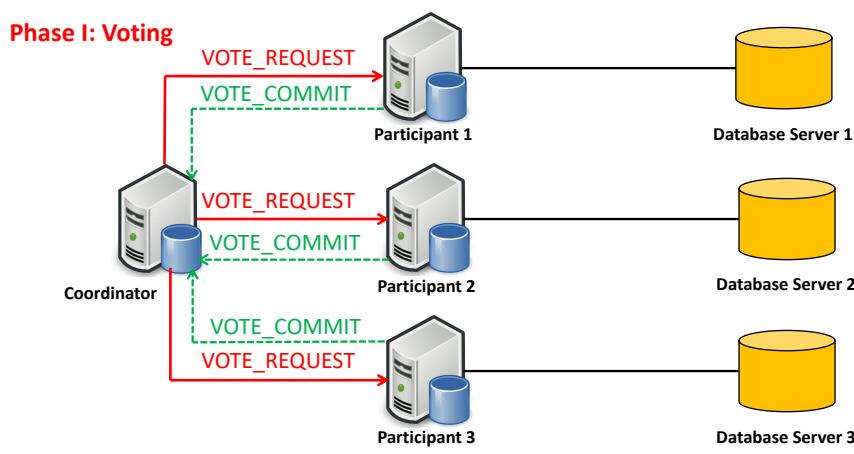
- In an e-commerce application, the bank database has been replicated across two servers
- Maintaining consistency of replicated data is a challenge



5

The Two-Phase Commit Protocol

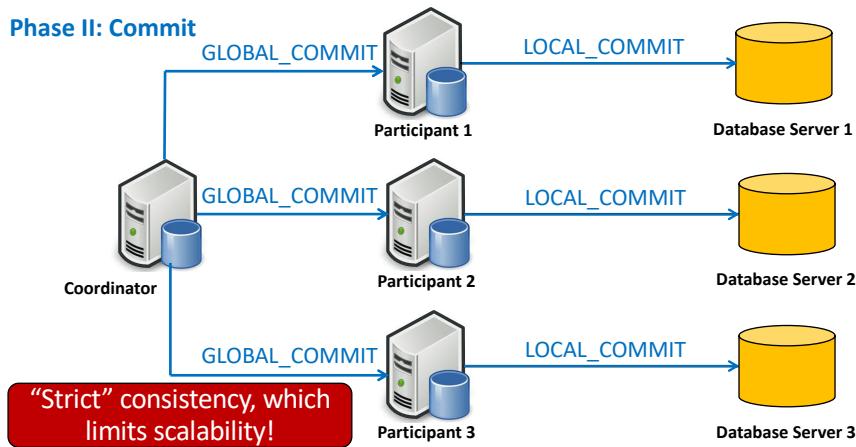
- The two-phase commit protocol (2PC) can be used to ensure atomicity and consistency



6

The Two-Phase Commit Protocol

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



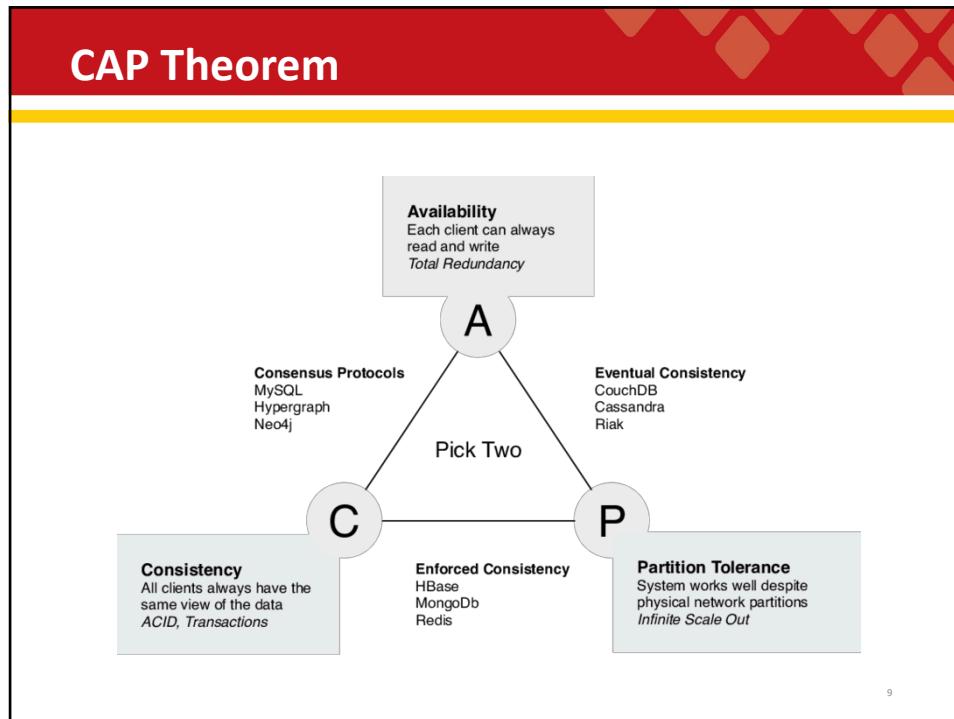
7

The CAP Theorem

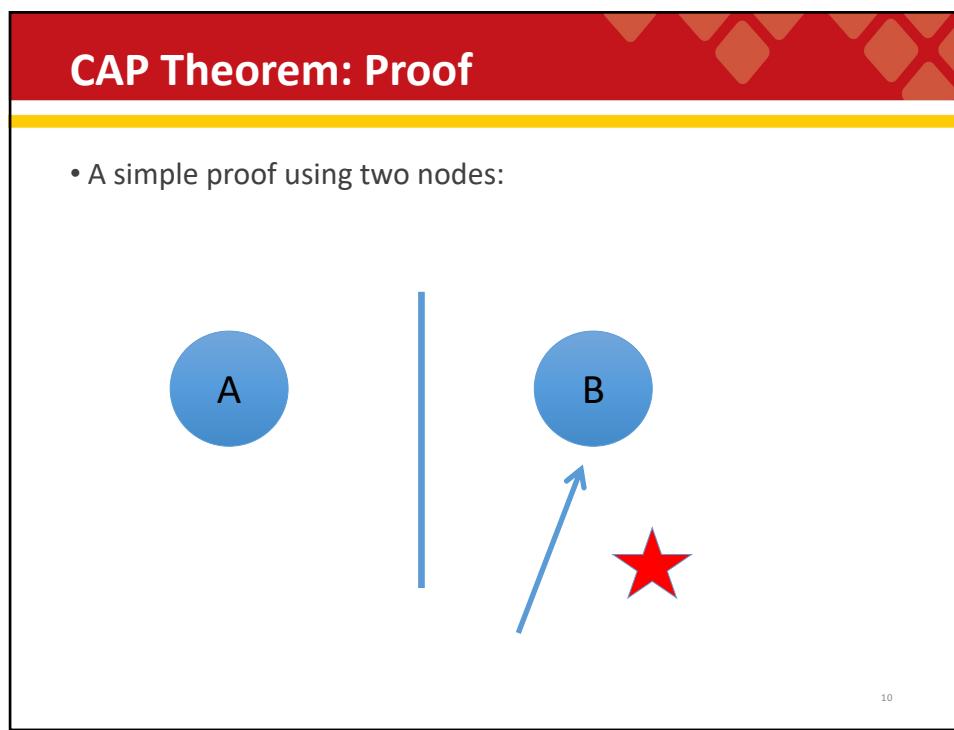
- The limitations of distributed databases can be described in the so called the CAP theorem
 - Consistency: every node always sees the same data at any given instance (i.e., strict consistency)
 - Availability: the system continues to operate, even if nodes in a cluster crash, or some hardware or software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



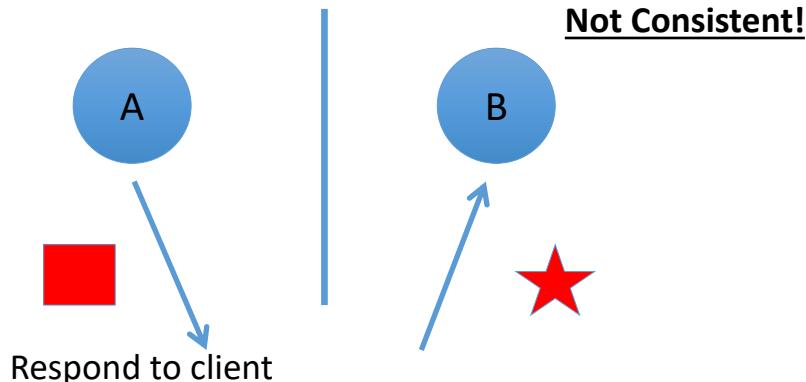
9



10

CAP Theorem: Proof

- A simple proof using two nodes:

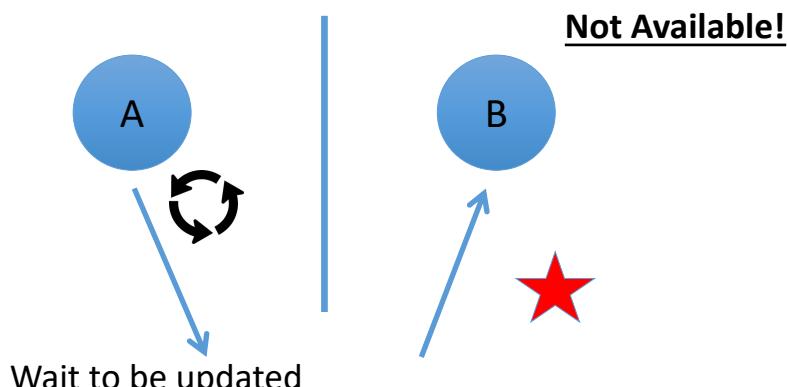


11

11

CAP Theorem: Proof

- A simple proof using two nodes:

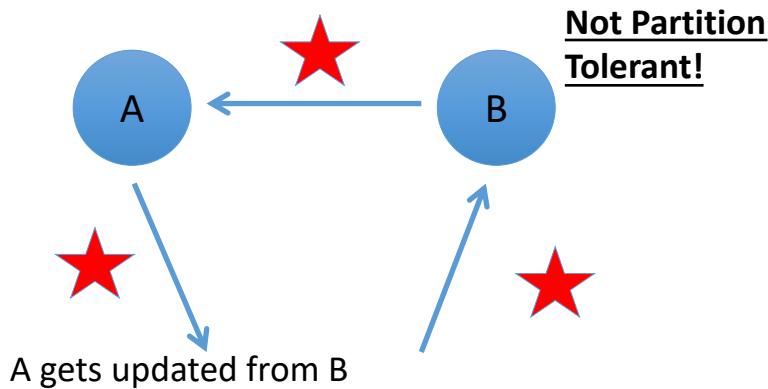


12

12

CAP Theorem: Proof

- A simple proof using two nodes:



13

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

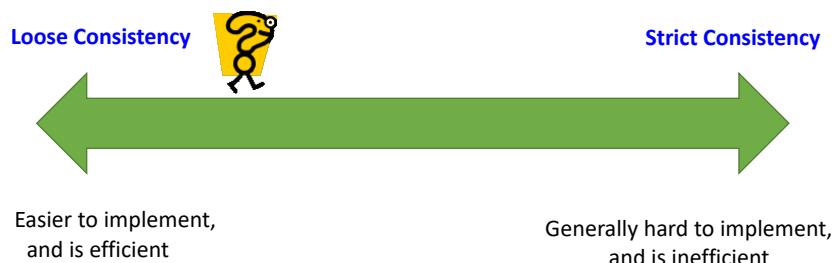
Trading-Off Consistency

- 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



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

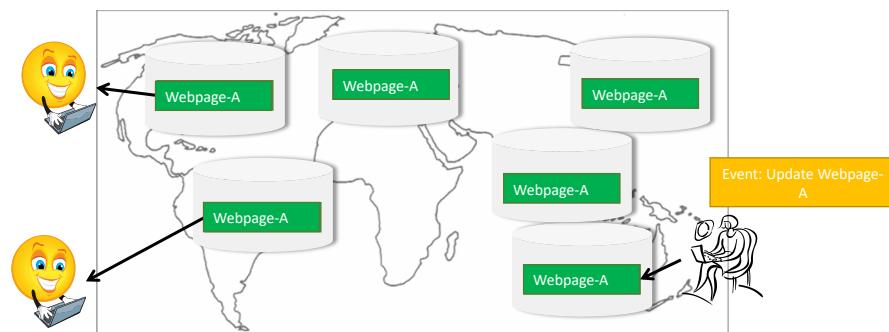
Eventual Consistency

- 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

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.



SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

3

3

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



4

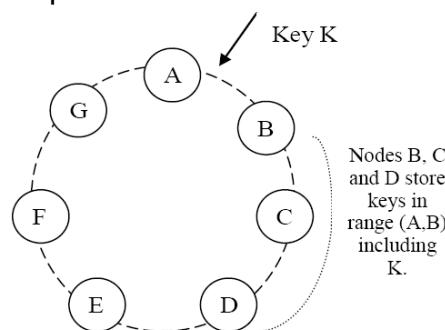
4

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



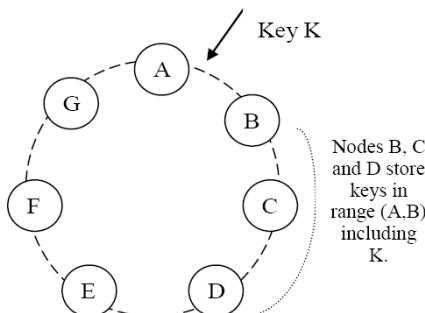
Grand challenge: every nodes must maintain an up-to-date view of the ring! How?

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 for can be 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 are responsible for storing a particular key.

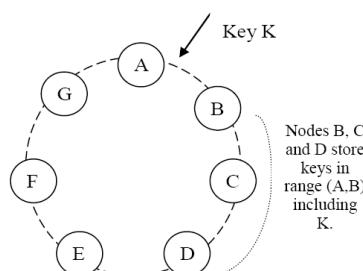


Quorum

- N: total number of replicas per each key/value pair
- R: minimum number of nodes that must participate in a successful reading
- W: minimum number of nodes that must participate in a successful 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.

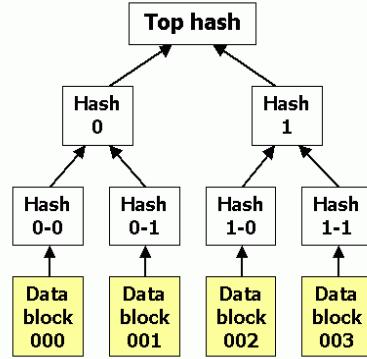
Temporary failures: Sloppy quorum and hinted handoff

- 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



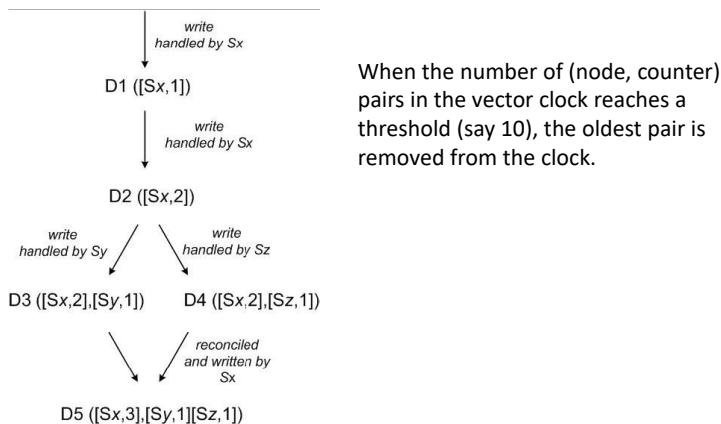
Data versioning

- 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



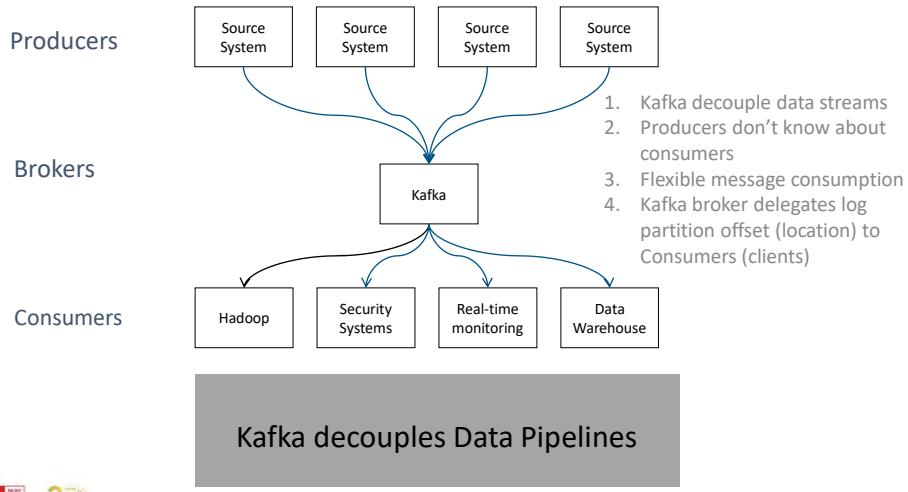
Technical summary

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.

Why Kafka



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

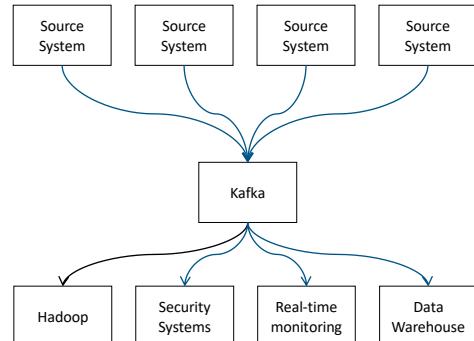
- 1/3 of all Fortune 500 companies
- Top ten travel companies, 7 of ten top banks, 8 of ten top insurance companies, 9 of ten top telecom companies
- LinkedIn, Microsoft and Netflix process 1 billion messages a day with Kafka
- Real-time streams of data, used to collect big data or to do real time analysis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

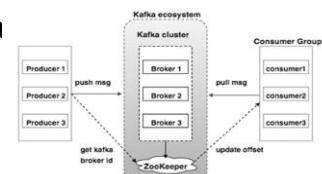


Key terminology

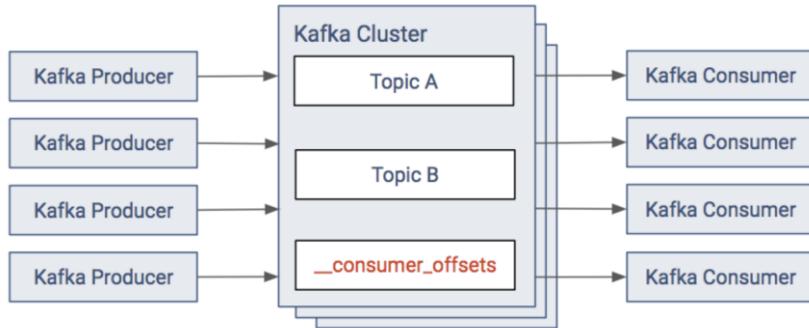
- Kafka maintains feeds of messages in categories called topics.
 - a stream of records (“/orders”, “/user-signups”), feed name
 - Log topic storage on disk
 - Partition / Segments (parts of Topic Log)
- Records have a key (optional), value and timestamp; Immutable
- Processes that publish messages to 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

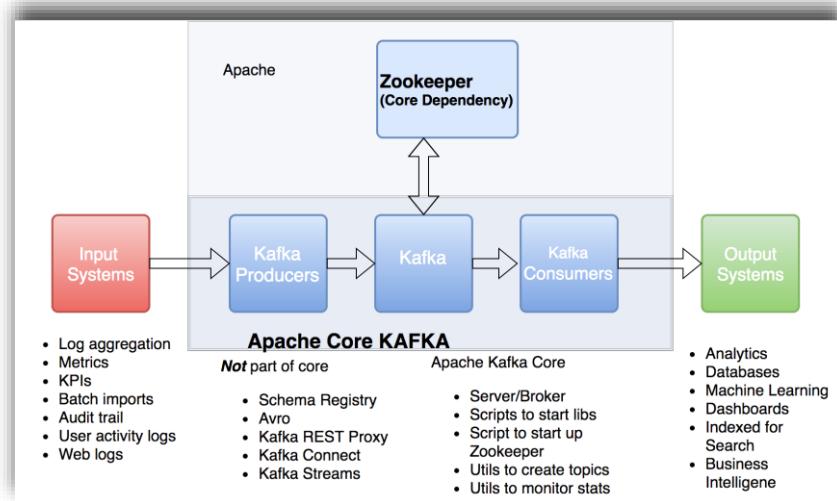
- Kafka cluster consists of multiple brokers and zookeeper
- Communication between all components is done via a high performance simple binary API over TCP protocol
- Zookeeper provides in-sync view of Kafka Cluster configuration
 - Leadership election of Kafka Broker and Topic Partition pairs
 - manages 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



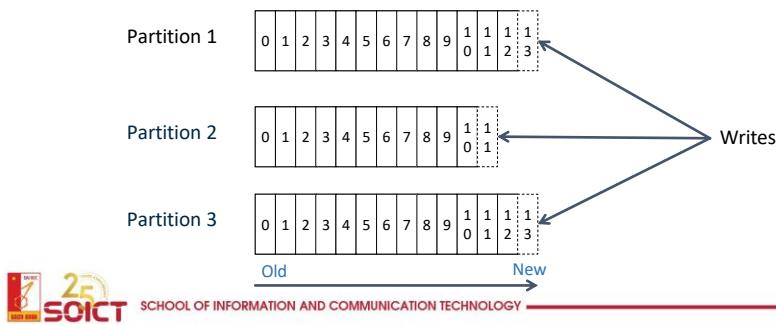
Kafka topics architecture

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



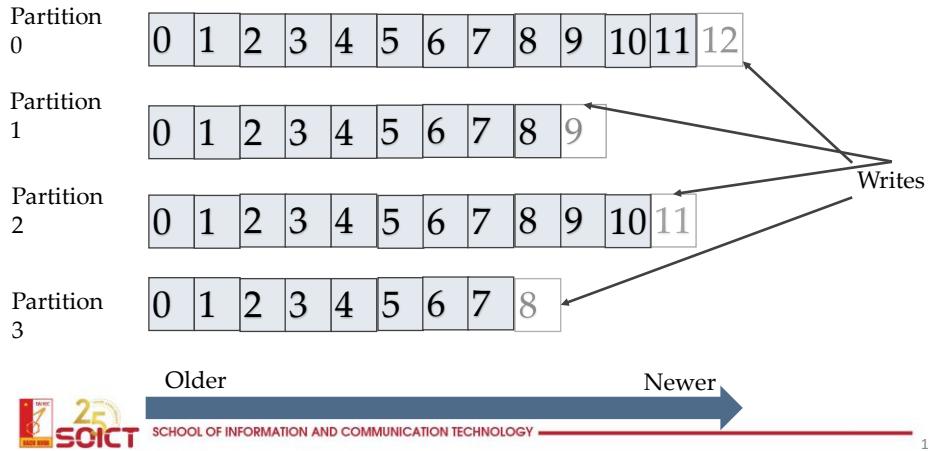
15

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

16

Kafka topic partitions layout



17

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
 - A 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)

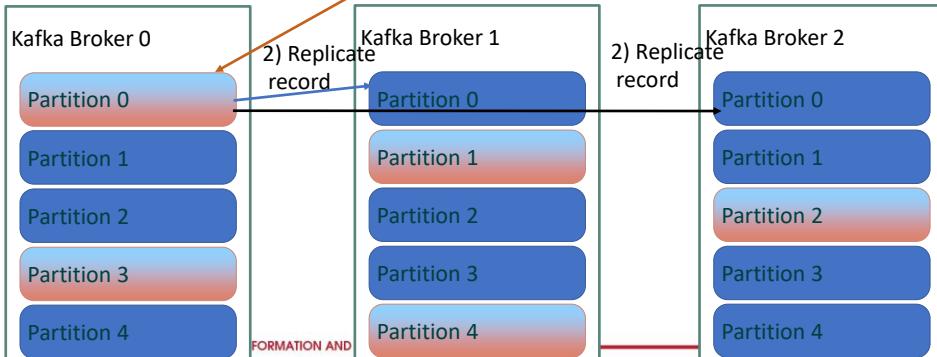
Record is considered "committed" when all ISR for partition wrote to their log.

Only committed records are readable from consumer

Client Producer

Leader Red
Follower Blue

1) Write record



19

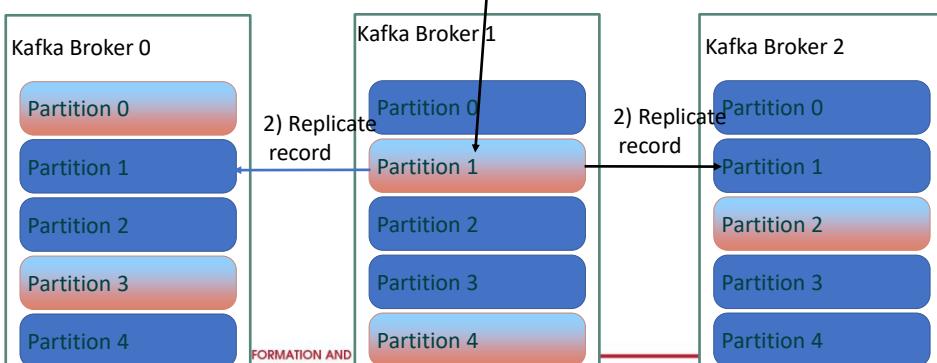
Kafka replication to partitions (2)

Another partition can be owned by another leader on another Kafka broker

Client Producer

Leader Red
Follower Blue

1) Write record



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

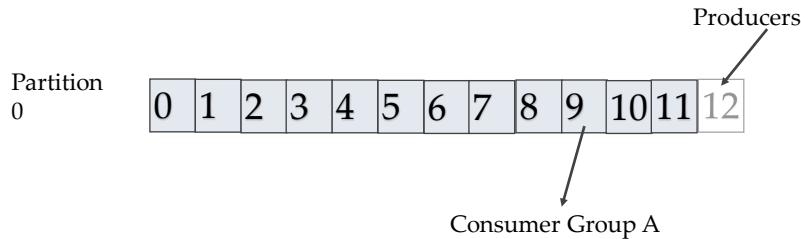
- Producers can choose to trade throughput for durability of writes:
- Note: throughput can also be raised with more brokers...

Durability	Behaviour	Per Event Latency	Required Acknowledgements (request.required.acks)
Highest	ACK all ISR 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 ‘employeeld’ 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

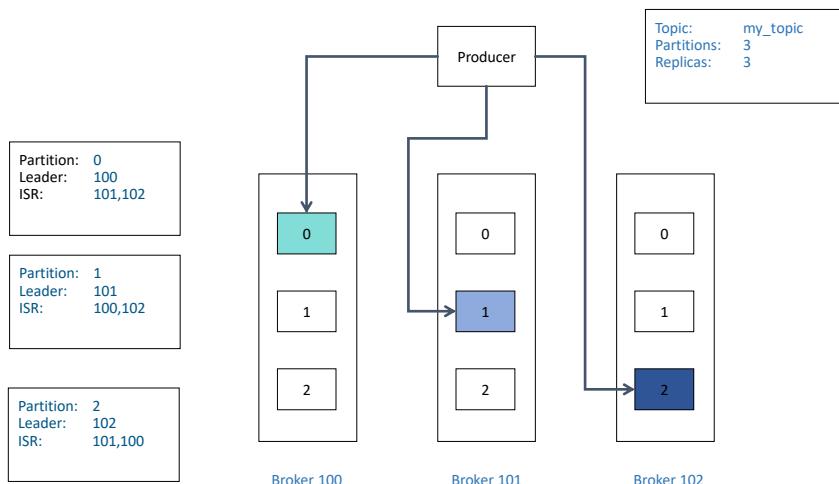
Kafka producers and consumers



Producers are writing at Offset 12

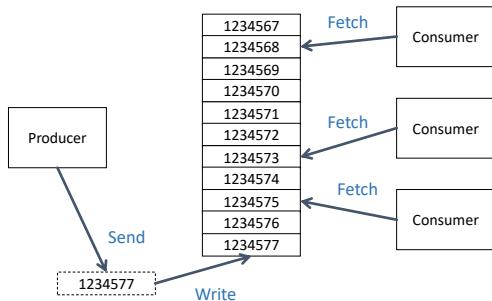
Consumer Group A is Reading from Offset 9.

Producer – Load balancing and ISRs



Consumer (1)

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



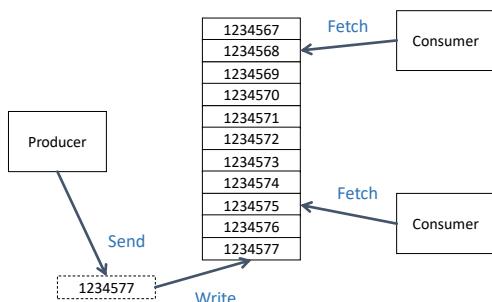
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

27

27

Consumer (2)

- Consumers can go away



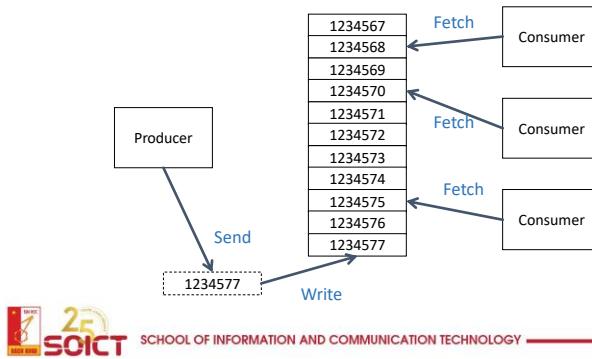
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

28

28

Consumer (3)

- And then come back



29

29

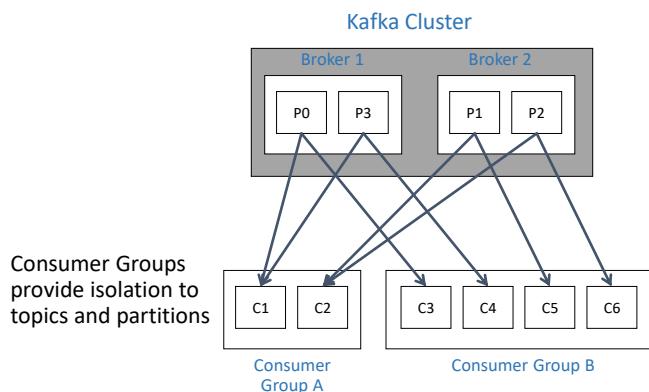
Consumer Group

- 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

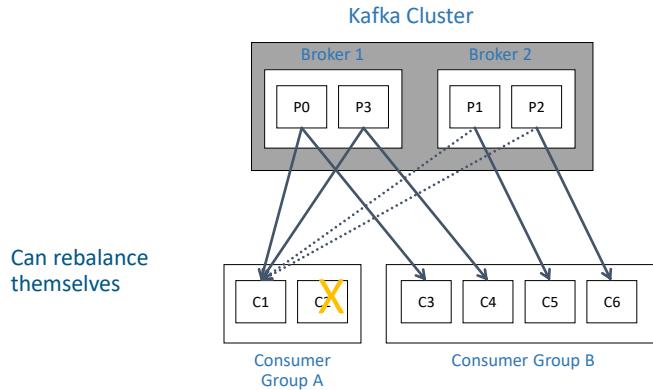
Common consumer group patterns

- 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

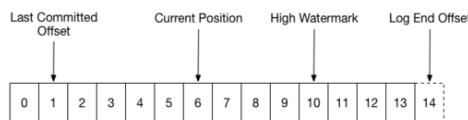
- Consumer membership in Consumer Group is handled by the Kafka protocol dynamically
- If new Consumers join Consumer group, it gets a share of partitions
- If Consumer dies, its partitions are split among remaining live Consumers in Consumer Group

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

Much Harder
(Impossible??)

- 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

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

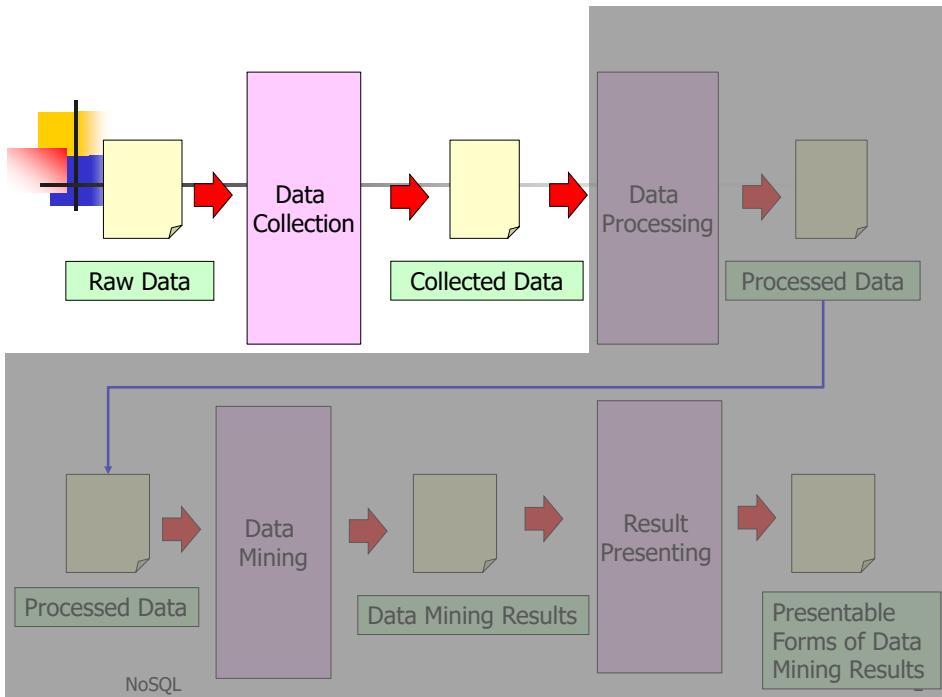
Bigdata Storage and Processing

NoSQL

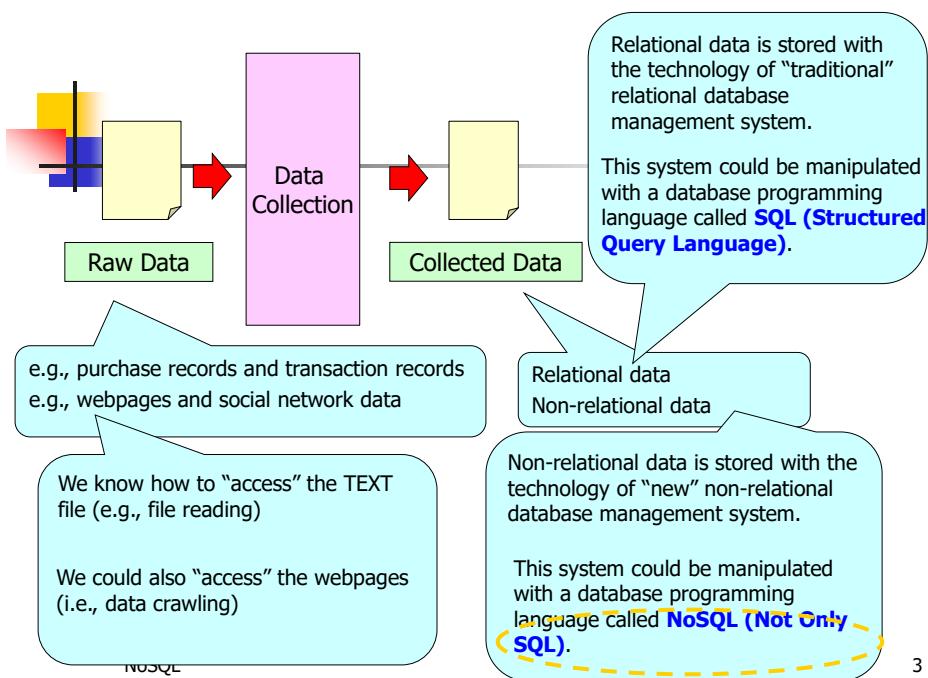
NoSQL

1

1

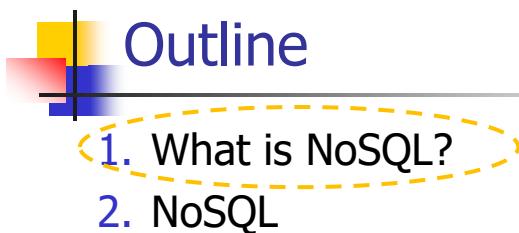


2



3

3



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**

NoSQL

5

5

1. What is NoSQL?

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

NoSQL

6

6

1. What is NoSQL?

- **Advantage**

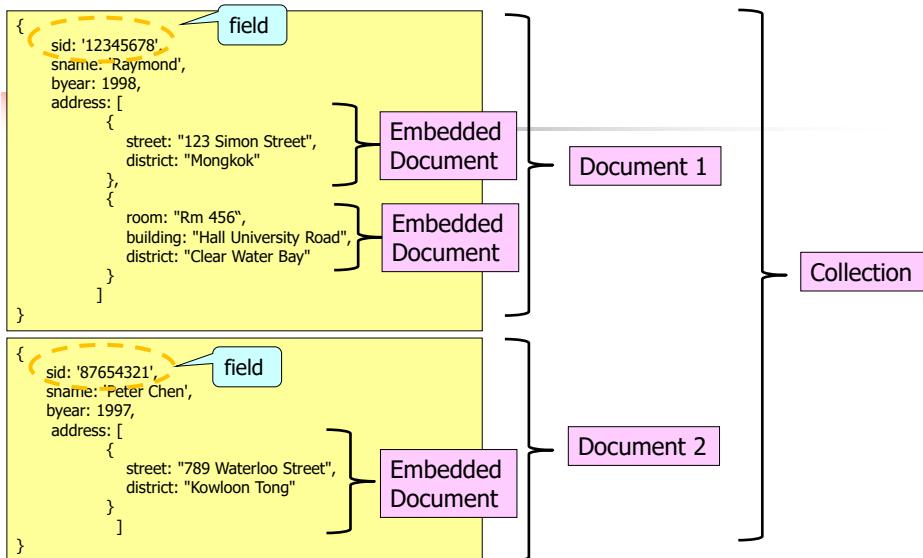
- It could store a large amount of data **without much structure** or **with little structure**.
- It could be used with **cloud computing and storage** for scalability.
- It could be **developed very rapidly** (since we do not need to define any schema).

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

NoSQL

7

7

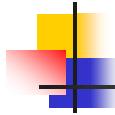


A number of different collections form a database

NoSQL

8

8



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

NoSQL

9

9



■ 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

NoSQL

10

10

- 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.

NoSQL

11

11

```
{
  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"
    }
  ]
}
```

NoSQL

12

12

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

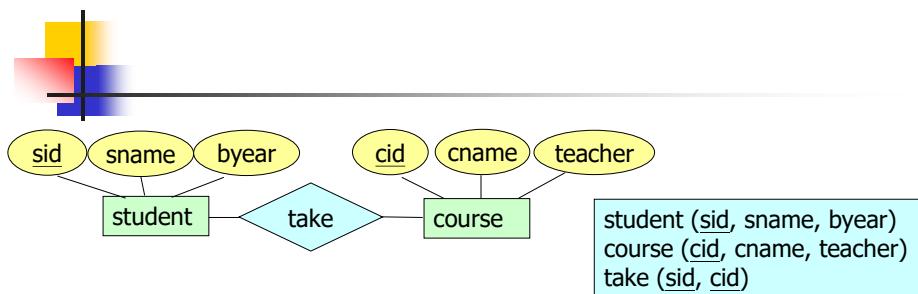
We know that SQL stores “structured” data (which has a “regular” structure).
 SQL could also store “unstructured” data (which has an “irregular” structure) but it stores many “null” values in the tables.
 - We want to store data without duplication (for data consistency)

Sometimes, this may introduce time-consuming join operations.

NoSQL

13

13



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

Natural Join

It is time-consuming.

NoSQL

14

14



Outline

1. What is NoSQL?

2. NoSQL

NoSQL

15

15



2. NoSQL

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

NoSQL

16

16

2. DDL

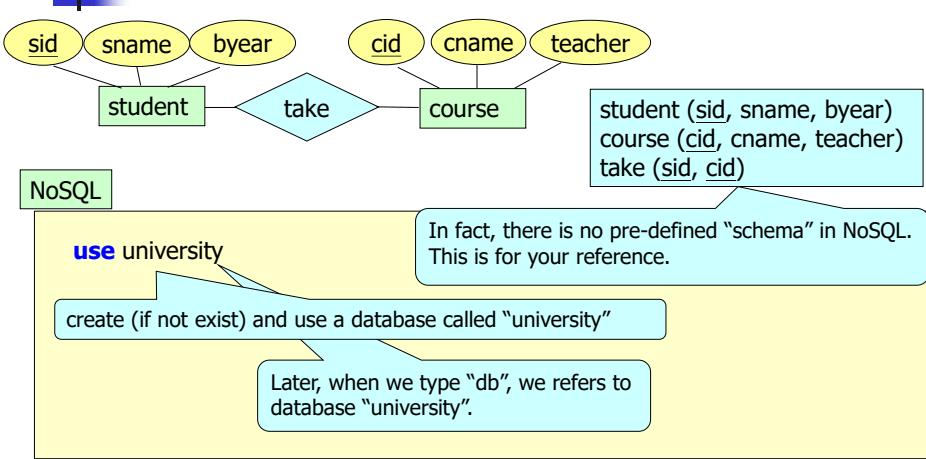
- Data Definition Language (DDL)
 - Although NoSQL does not require to define a schema, we could see how NoSQL could define the schema "automatically".

NoSQL

17

17

2. Database Creation

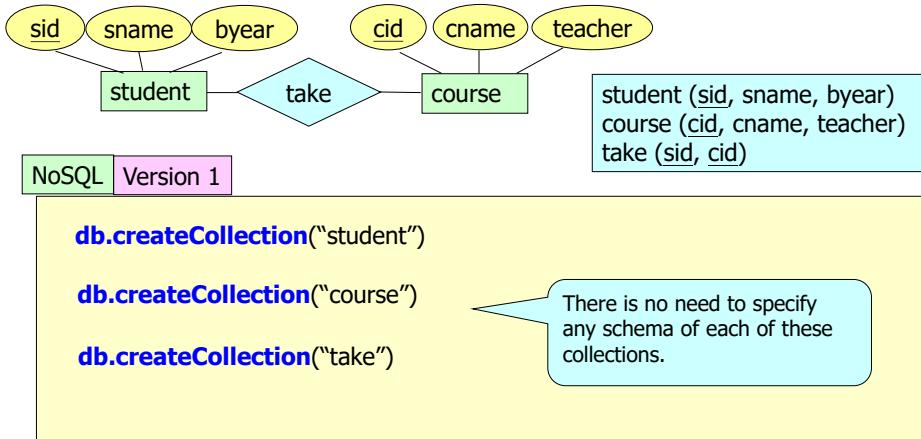


NoSQL

18

18

2. Collection Creation



NoSQL

19

19

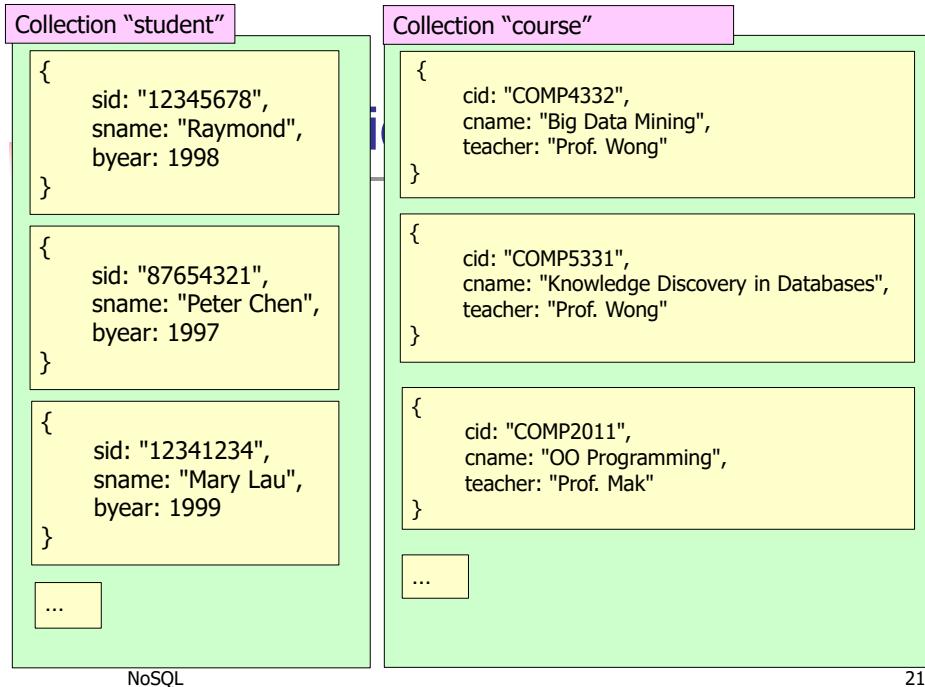
2. Collection Creation

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

NoSQL

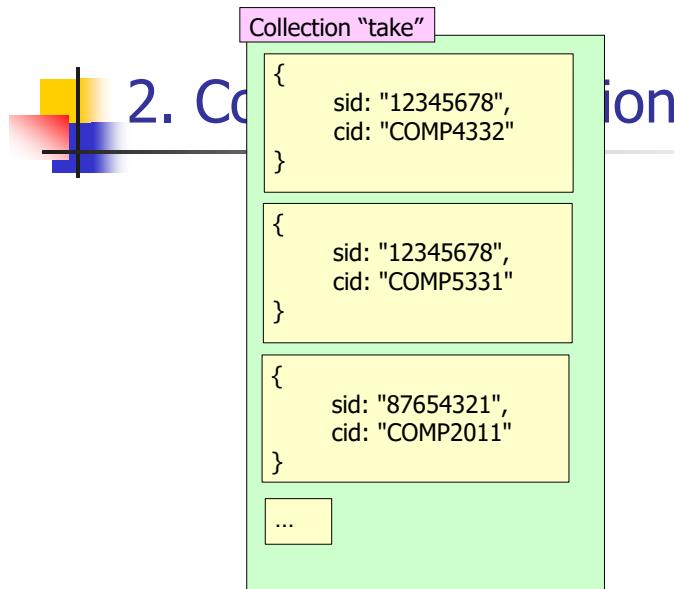
20

20



21

21

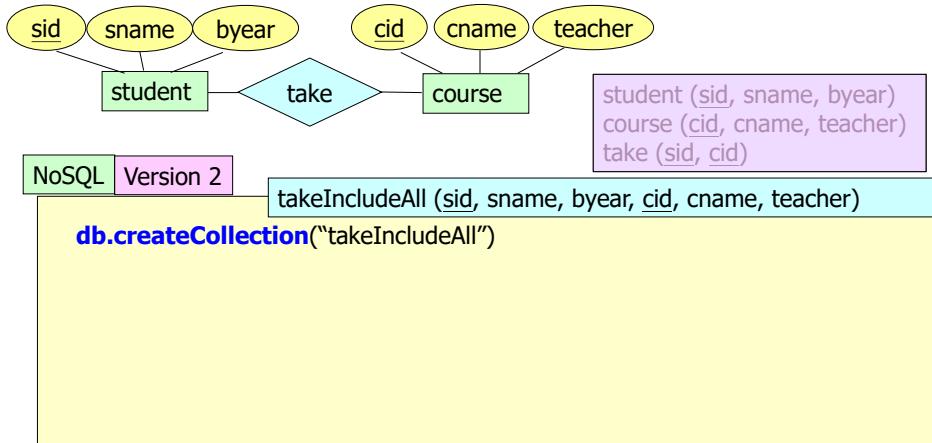


NoSQL

22

22

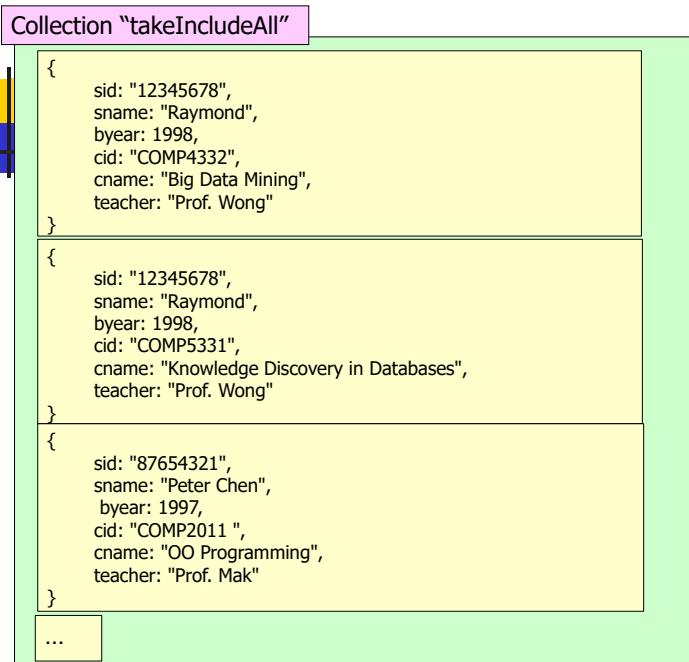
2. Collection Creation



NoSQL

23

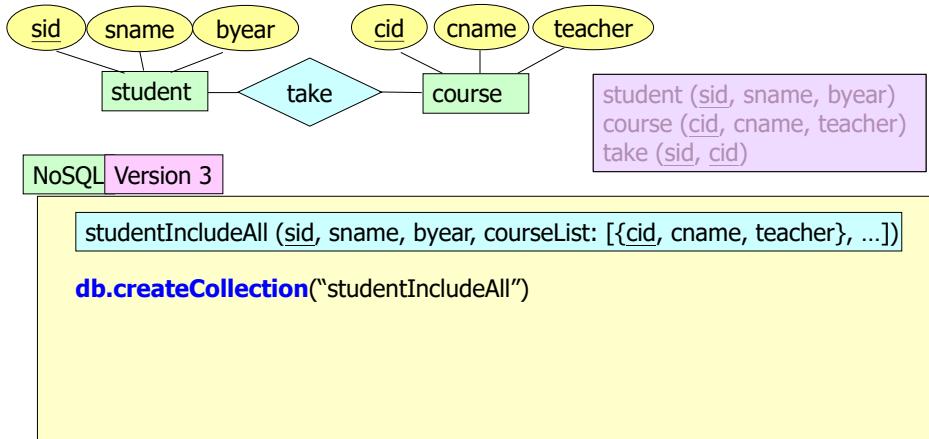
23



24

24

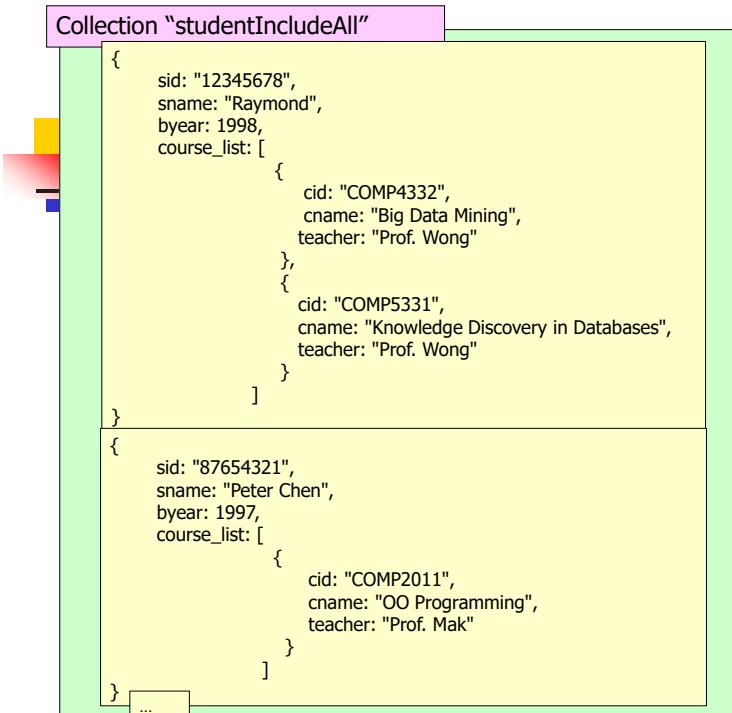
2. Collection Creation



NoSQL

25

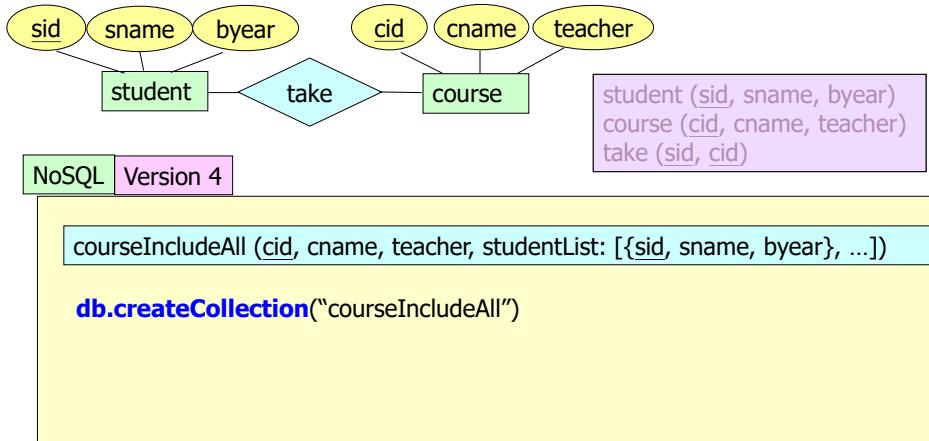
25



26

26

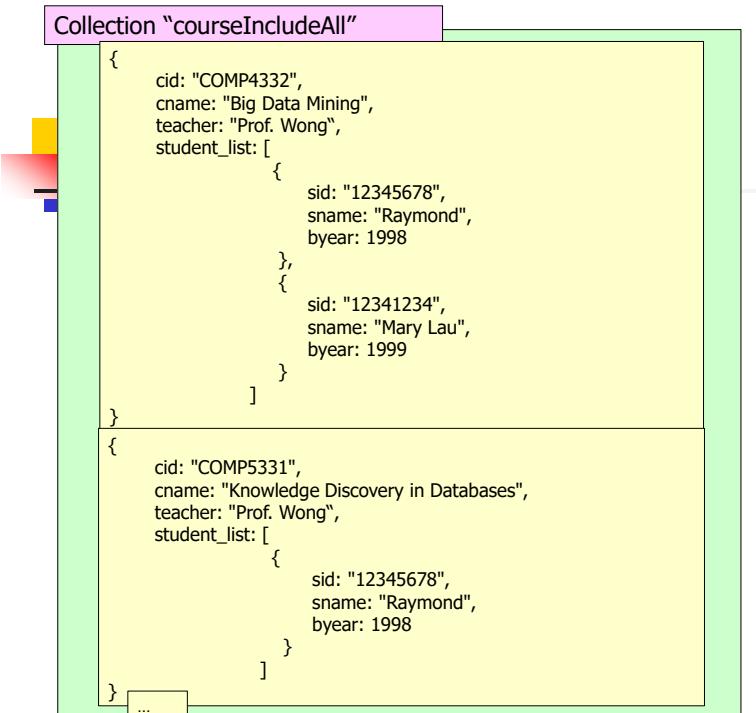
2. Collection Creation



NoSQL

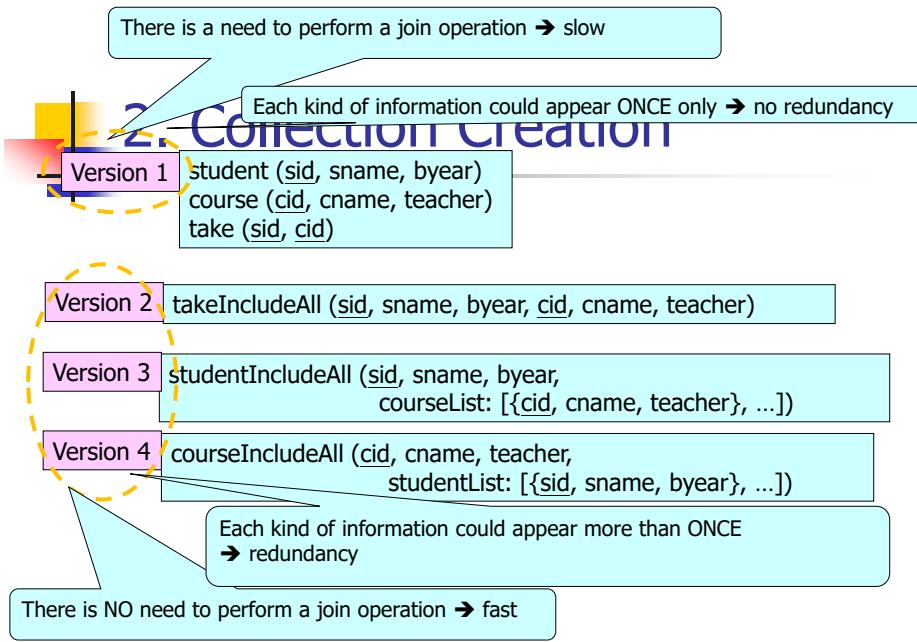
27

27



28

28



NoSQL

29

29

2. Collection Creation

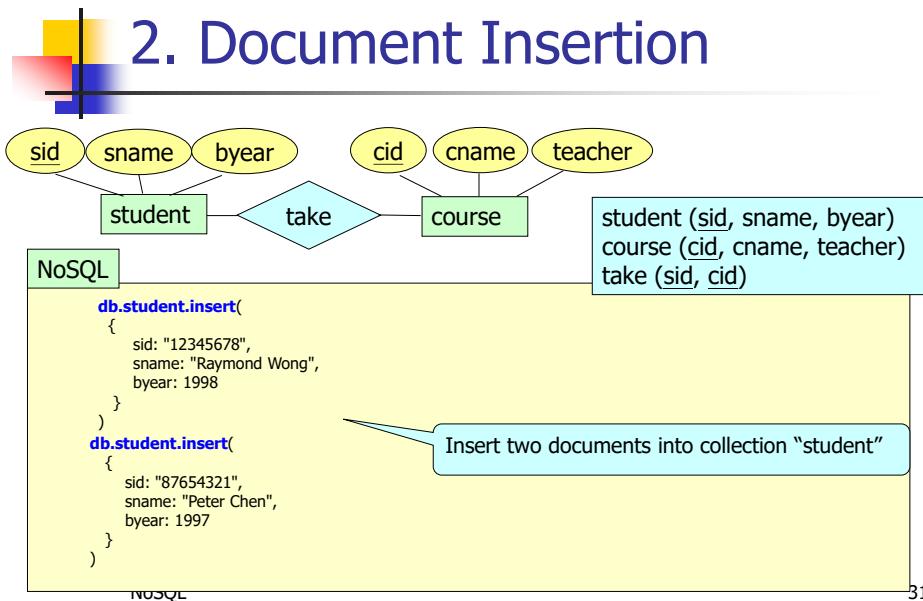
- We focus on Version 1
- In this version, sometimes, we need to perform a “so-called” join operation, which may be slow.
- Besides, the NoSQL programming language related to this “join” operation is more complicated than a “standard” SQL language

NoSQL

30

30

2. Document Insertion



31

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.

NoSQL

32

32

## 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

NoSQL

33

33

## 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-DDThh:mm:ssZ

Time zone offset from UTC  
(Coordinated Universal Time)

NoSQL

34

34

## 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)"
}
```

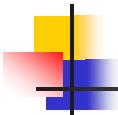
Exactly what we specified

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

36

36



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" }
 }
 }
]
)

```

**a**

Output

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

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()
```

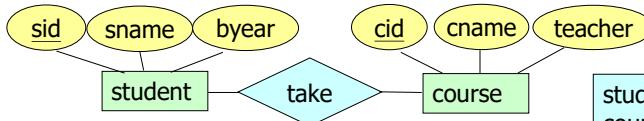
Output

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

39

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()

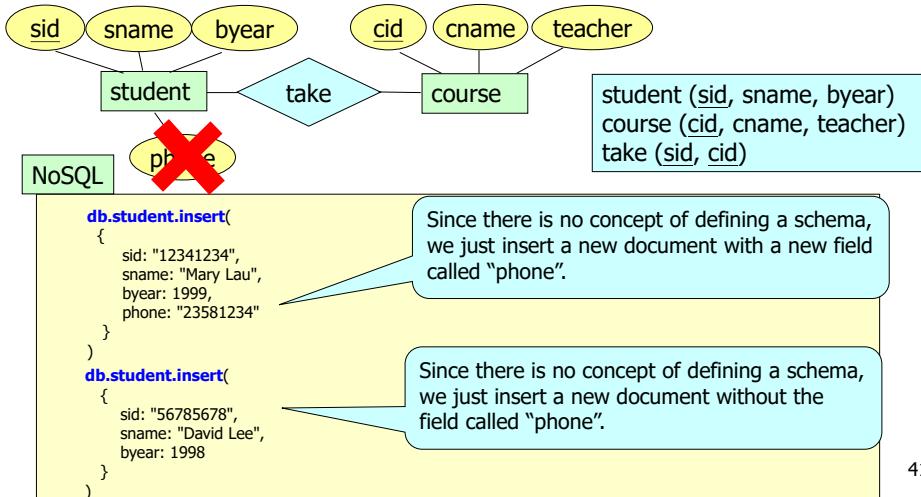
```

NoSQL

40

40

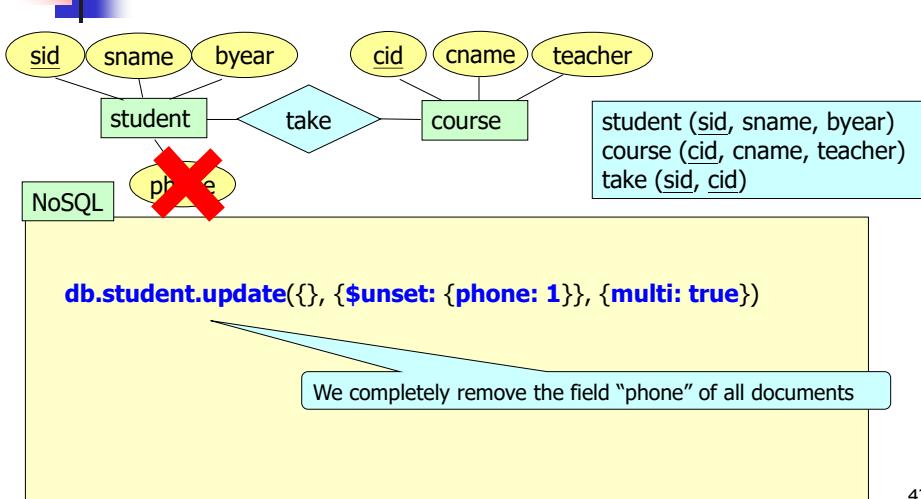
## 2. Collection Scheme Update



41

41

## 2. Collection Scheme Update



42

42

## 2. NoSQL

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

NoSQL

43

43

## 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

NoSQL

44

44

## 2. NoSQL – find

NoSQL

`db.student.find()`

To output all documents in collection "student"

Output

```
{
 "_id": ObjectId("5a8a431472f2fbf196e55bd"),
 "sid": "12345678",
 "sname": "Raymond",
 "byear": 1998
},
{
 "_id": ObjectId("5a8a431472f2fbf196e55be"),
 "sid": "87654321",
 "sname": "Peter Chen",
 "byear": 1997
},
{
 "_id": ObjectId("5a8a431472f2fbf196e55bf"),
 "sid": "12341234",
 "sname": "Mary Lau",
 "byear": 1999
},
{
 "_id": ObjectId("5a8a431472f2fbf196e55c0"),
 "sid": "56785678",
 "sname": "David Lee",
 "byear": 1998
},
{
 "_id": ObjectId("5a8a431572f2fbf196e55c1"),
 "sid": "88888888",
 "sname": "Test Test",
 "byear": 1998
}
```

NoSQL

45

45

## 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("5a8a431472f2fbf196e55bd"),
 "sid": "12345678",
 "sname": "Raymond",
 "byear": 1998
},
{
 "_id": ObjectId("5a8a431472f2fbf196e55c0"),
 "sid": "56785678",
 "sname": "David Lee",
 "byear": 1998
},
{
 "_id": ObjectId("5a8a431572f2fbf196e55c1"),
 "sid": "88888888",
 "sname": "Test Test",
 "byear": 1998
}
```

NoSQL

46

46

## 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" }
```

NoSQL

47

47

NoSQL

## 2. NoSQL – find

```
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" }
```

NoSQL

48

48



## 2. NoSQL – find

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"

49

49



## 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.

NoSQL

50

50

## 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"
}
```

NoSQL

51

There is no "byear" field in the 2<sup>nd</sup> document.

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

51

## 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("5a23ede1967809308848df56"),
 "sid": "12345678",
 "sname": "Raymond",
 "byear": 1998
}
```

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

NoSQL

52

52

## 2. NoSQL

- Three Major Functions

- find      → The easiest operation which is used in some cases
- distinct    → The easiest operation
- aggregate

- Operation of Datetime

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

NoSQL

53

53

## 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.

NoSQL

54

54

## 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.

NoSQL

55

55

## 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 2<sup>nd</sup> document.

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

NoSQL

56

56

## 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.

NoSQL

57

57



## 2. NoSQL

### ■ Three Major Functions

- find

The easiest operation

- distinct

The easiest operation which is used in some cases

- ➡ ■ Aggregate

### ■ Operation of Datetime

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

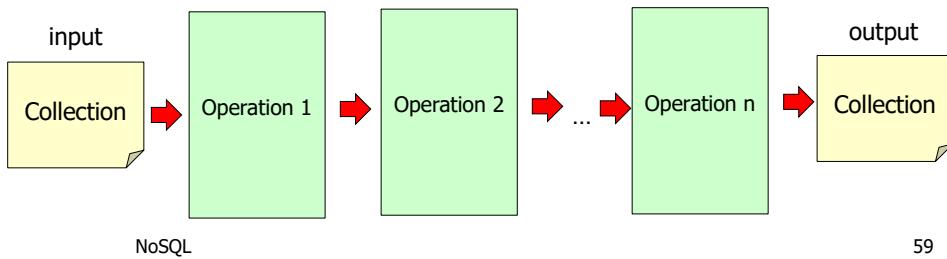
NoSQL

58

58

## 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



59

59

## 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

NoSQL

60

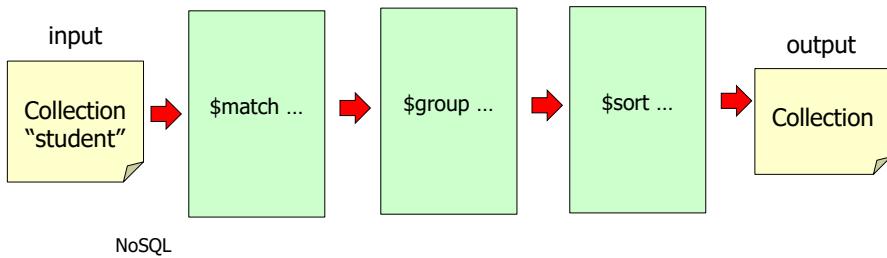
60

## 2. NoSQL – aggregation

NoSQL

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

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



61

61

## 2. NoSQL – aggregation

| SQL      | Aggregation Operation                                    |
|----------|----------------------------------------------------------|
| select   | \$project<br>\$group functions: \$sum, \$min, \$avg, ... |
| from     | db.student.aggregate(...)<br>\$lookup                    |
| where    | \$match                                                  |
| group by | \$group                                                  |
| having   | \$match                                                  |
| order by | \$sort                                                   |

NoSQL

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

62

## 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)

NoSQL

63

63

## 2. NoSQL – aggregation (same as “find”)

NoSQL

db.student.aggregate()

To output all documents in collection “student”

Output

Same as “db.student.find()”

```
{
 "_id": ObjectId("5a8a431472f2fbfb196e55bd"), "sid": "12345678", "sname": "Raymond", "byear": 1998 },
 {"_id": ObjectId("5a8a431472f2fbfb196e55be"), "sid": "87654321", "sname": "Peter Chen", "byear": 1997 },
 {"_id": ObjectId("5a8a431472f2fbfb196e55bf"), "sid": "12341234", "sname": "Mary Lau", "byear": 1999 },
 {"_id": ObjectId("5a8a431472f2fbfb196e55c0"), "sid": "56785678", "sname": "David Lee", "byear": 1998 },
 {"_id": ObjectId("5a8a431572f2fbfb196e55c1"), "sid": "88888888", "sname": "Test Test", "byear": 1998 }
```

NoSQL

64

64

## 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("5a8a431472f2fbff196e55bd"), sid : "12345678", sname : "Raymond", byear : 1998 }
{ _id : ObjectId("5a8a431472f2fbff196e55c0"), sid : "56785678", sname : "David Lee", byear : 1998 }
{ _id : ObjectId("5a8a431572f2fbff196e55c1"), sid : "88888888", sname : "Test Test", byear : 1998 }
```

NoSQL

65

65

## 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

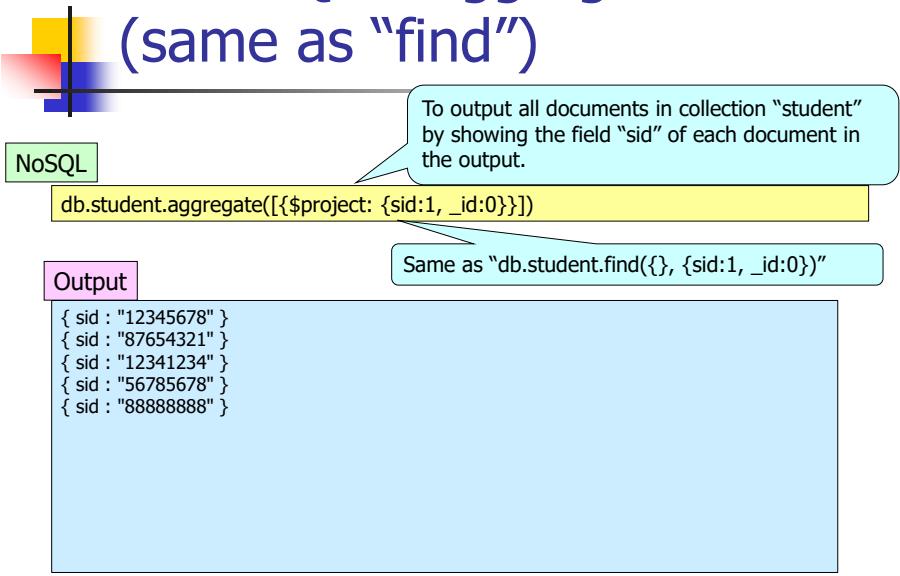
```
{ sid : "12345678"
{ sid : "56785678"
{ sid : "88888888"
```

NoSQL

66

66

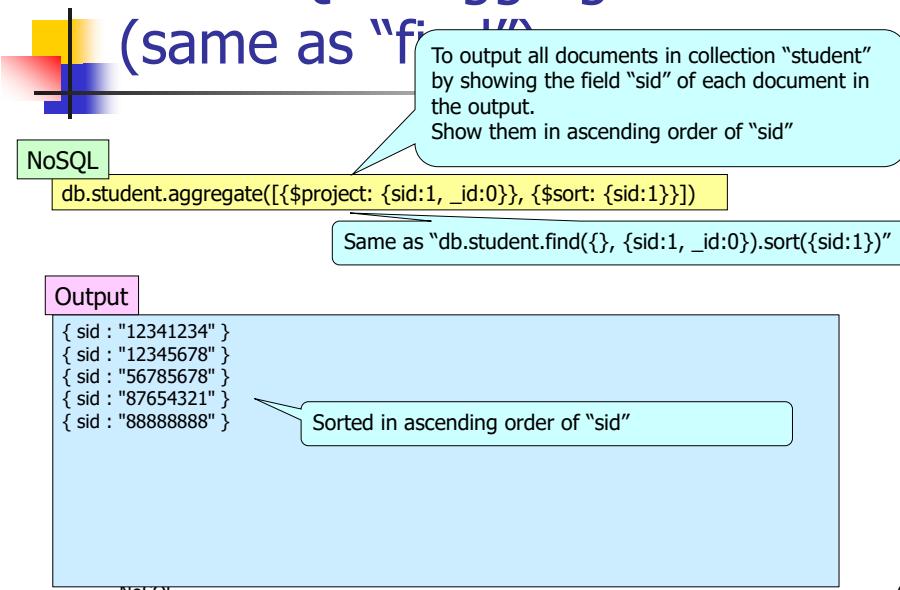
## 2. NoSQL – aggregation (same as “find”)



67

67

## 2. NoSQL – aggregation (same as “find”)



68

68

## 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)

NoSQL

69

69

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

NoSQL

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

```
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.)

```
{ _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).

NoSQL

70

70

## 2. NoSQL – aggregation (same to “distinct”)

NoSQL

Same as the previous NoSQL query but with a mapping function which maps the collection output to an array

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

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

Output

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

NoSQL

71

71

## 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)

NoSQL

72

72

## 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 }
```

73

73

## 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.

NoSQL

74

74

## 2. NoSQL - aggregation

- Suppose that we have the following 2 documents.

NoSQL

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

There is no "byear" field in the 2<sup>nd</sup> document.

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

75

75

## 2. NoSQL - agg

NoSQL

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

```
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.

NoSQL

76

76

## 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"}]
})
```

NoSQL

77

77

## 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" }
]
}
```

NoSQL

78

78



## 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 "coursesTaken" of collection "tempTable"

`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. 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 "coursesTaken" of collection "tempTable".  
The new non-array field should contain a single value.  
The "\_id" field should be removed.

`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. NoSQL

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 "coursesTaken" 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"

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

NoSQL

81

81



## 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: []
})
```

NoSQL

82

82

## 2. NoSQL - aggregation

NoSQL

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

Output

```
<no result>
```

83

83

## 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: []
})
```

84

## 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

85

## 2. NoSQL - aggregation

NoSQL

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.

```
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

87

## 2. NoSQL - aggregation

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

NoSQL

88

88

## 2. NoSQL - aggregation

NoSQL

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

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

NoSQL

89

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" }
]
}
```

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

```
{
 _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" }
]
}
```

...

90

## 2. NoSQL - aggregation

NoSQL

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

```
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}}
]
```

NoSQL

91

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"}
]
)
```

NoSQL

93

93

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" }
}
```

94

94

## 2. NoSQL - aggregation

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

NoSQL

95

95

## 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

NoSQL

96

96

## 2. NoSQL

- We could compare two Datetime concepts in NoSQL.

NoSQL

97

97

### 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")
}
```

98

## 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")
}
```

NoSQL

99

99

NoSQL

## 2. Operation of Datetime

```
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")
}
```

NoSQL

100

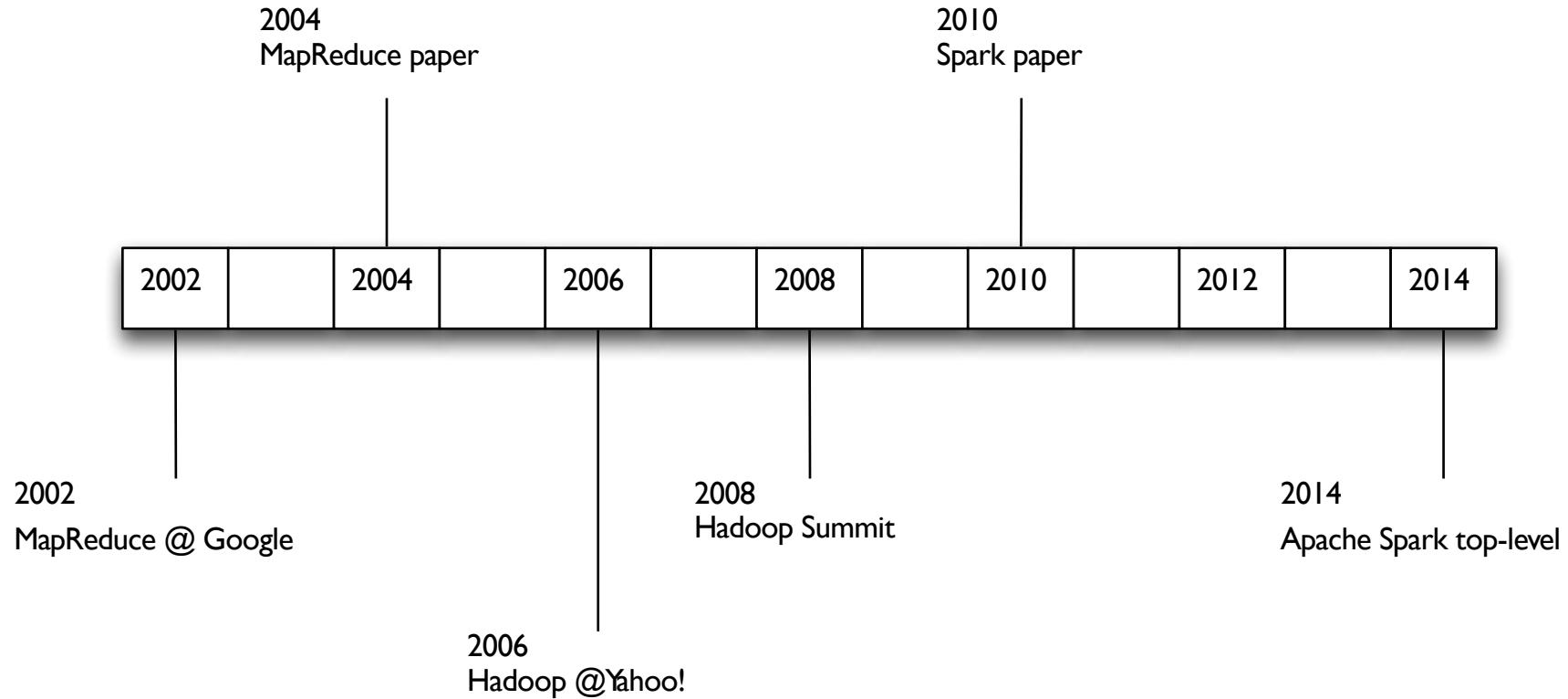
100

# Big Data Integration and Processing

05/2023

Thanh-Chung Dao Ph.D.

# History of Spark



# History of Spark

circa 1979 – **Stanford, MIT, CMU**, etc.

set/list operations in LISP, Prolog, etc., for parallel processing

[www-formal.stanford.edu/jmc/history/lisp/lisp.htm](http://www-formal.stanford.edu/jmc/history/lisp/lisp.htm)

circa 2004 – **Google**

*MapReduce: Simplified Data Processing on Large*

*Clusters* Jeffrey Dean and Sanjay Ghemawat

[research.google.com/archive/mapreduce.html](http://research.google.com/archive/mapreduce.html)

circa 2006 – **Apache**

*Hadoop*, originating from the Nutch Project Doug Cutting

[research.yahoo.com/files/cutting.pdf](http://research.yahoo.com/files/cutting.pdf)

circa 2008 – **Yahoo**

*web scale search indexing Hadoop Submit, HUG, etc.*

[developer.yahoo.com/hadoop/](http://developer.yahoo.com/hadoop/)

circa 2009 – **Amazon AWS**

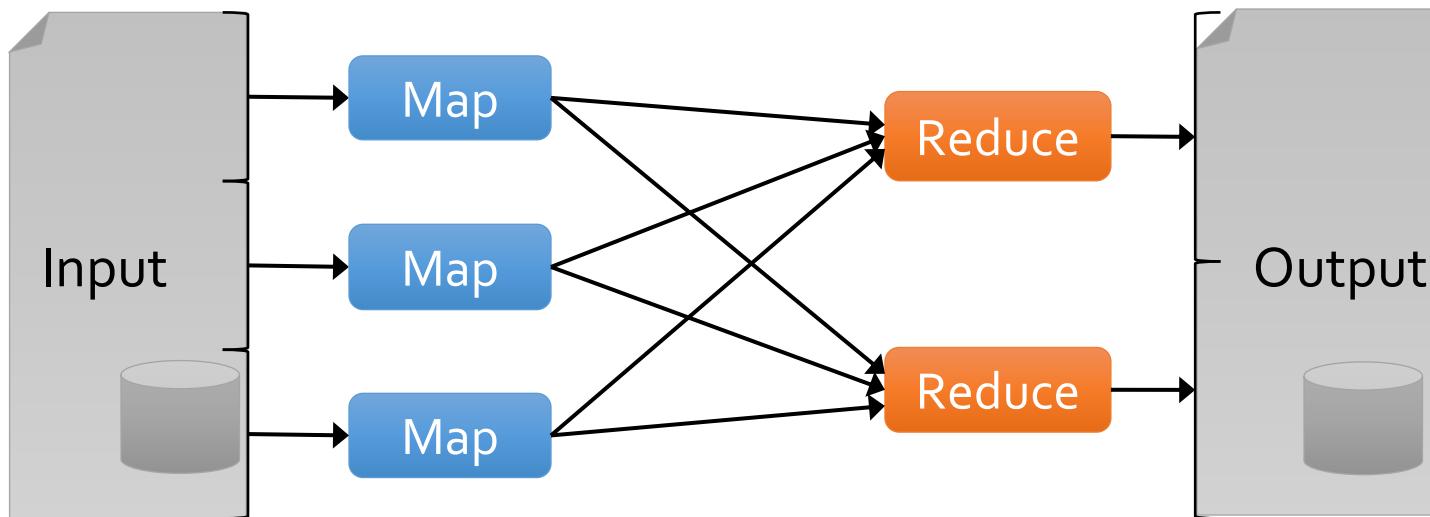
Elastic MapReduce

Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.

[aws.amazon.com/elasticmapreduce/](http://aws.amazon.com/elasticmapreduce/)

# MapReduce

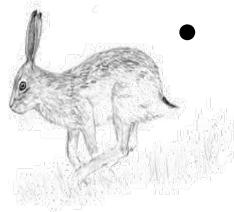
Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



# MapReduce

- Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining tools (R, Excel, Python)

# Data Processing Goals

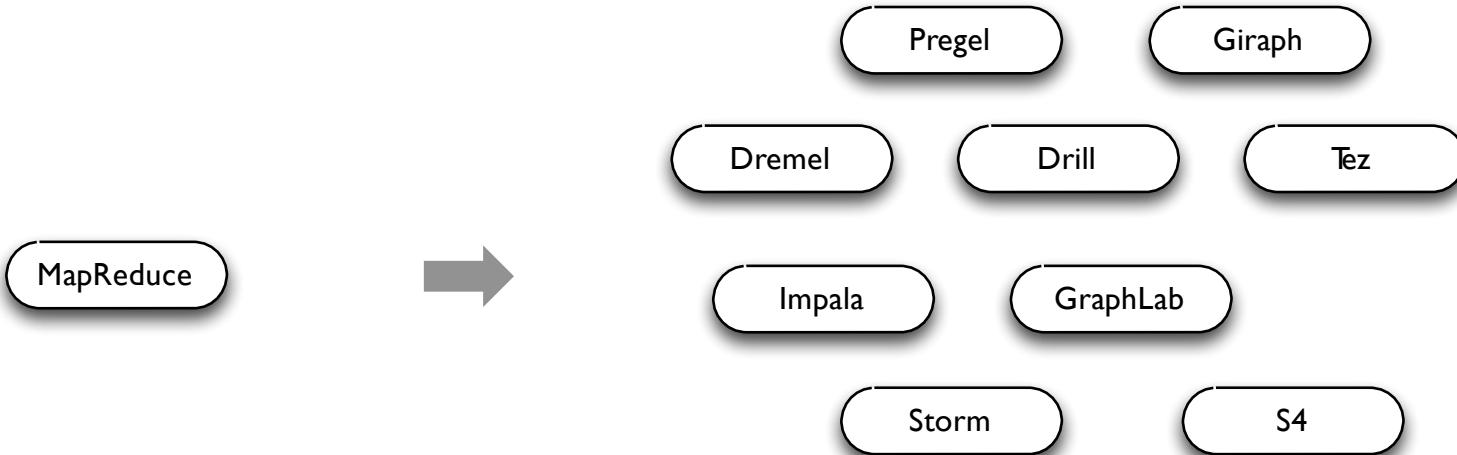


- **Low latency (interactive) queries on historical data:** enable faster decisions
  - E.g., identify why a site is slow and fix it
- **Low latency queries on live data (streaming):** enable decisions on real-time data
  - E.g., detect & block worms in real-time (a worm may infect **1mil** hosts in **1.3sec**)
- **Sophisticated data processing:** enable “better” decisions
  - E.g., anomaly detection, trend analysis



Therefore, people built specialized  
systems as workarounds...

# Specialized Systems



---

**General Batch Processing**

---

**Specialized Systems:**  
iterative, interactive, streaming, graph, etc.

*The State of Spark, and Where We're Going Next*

**Matei Zaharia**

Spark Summit (2013)

[youtu.be/nU6vO2EJAb4](https://youtu.be/nU6vO2EJAb4)

# Storage vs Processing Wars

## NoSQL battles

*Relational vs NoSQL*

HBase vs  
Cassandra

*Redis vs Memcached vs  
Riak*

*MongoDB vs CouchDB vs Couchbase*  
*Neo4j vs Titan vs  
Giraph vs OrientDB*  
*Solr vs Elasticsearch*

## Compute battles

*MapReduce vs  
Spark*

*Spark Streaming vs Storm*

*Hive vs Spark SQL vs  
Impala*

*Mahout vs MLlib vs H2O*

# Storage vs Processing Wars

## NoSQL battles

*Relational* vs *NoSQL*

HBase vs  
Cassandra

Redis vs Memcached vs  
Riak

MongoDB vs CouchDB vs Couchbase

Neo4j vs Titan vs  
Giraph vs OrientDB

Solr vs Elasticsearch

## Compute battles

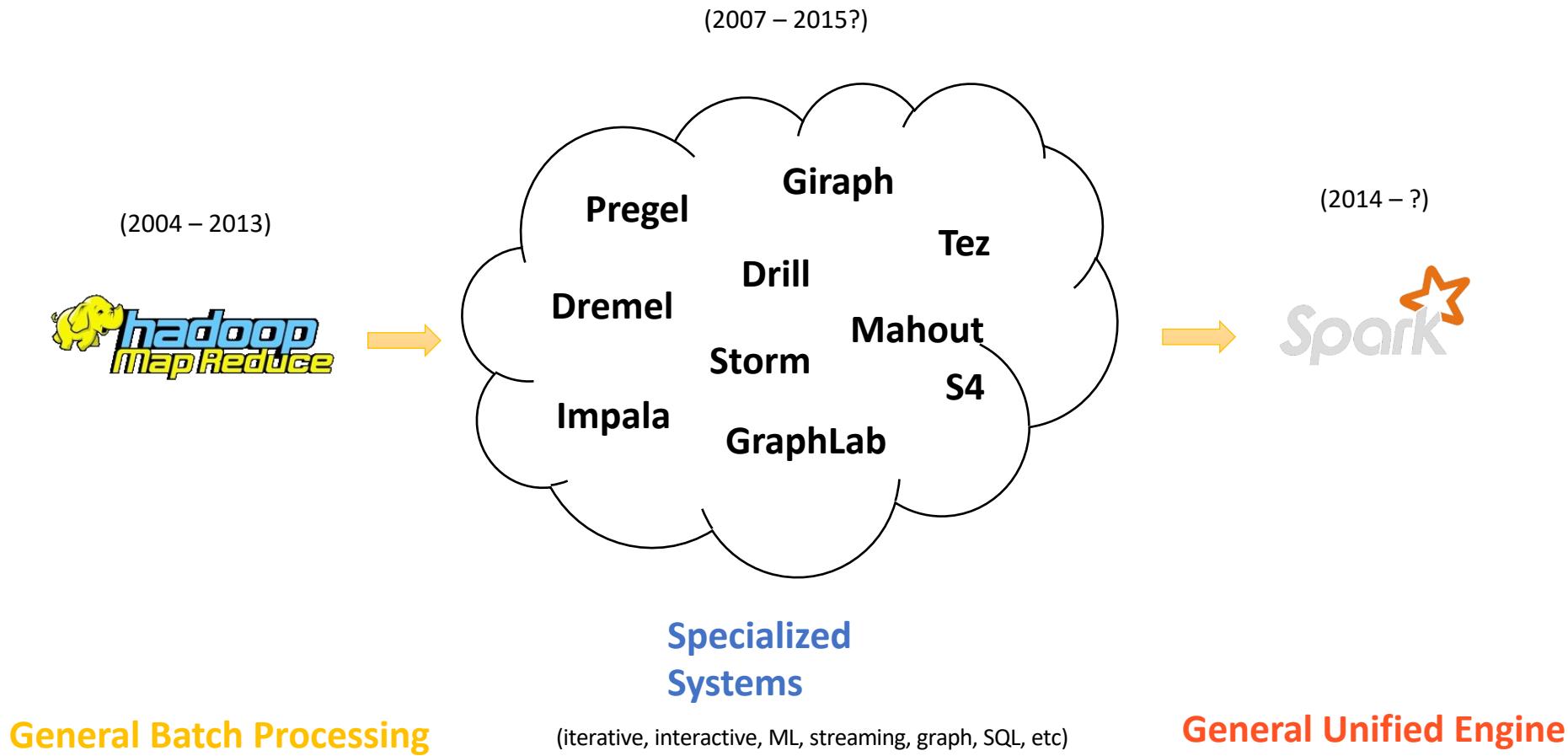
MapReduce vs  
Spark

Spark Streaming vs Storm

Hive vs Spark SQL vs  
Impala

Mahout vs MLlib vs H2O

# Specialized Systems





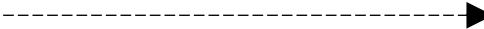
vs



YARN



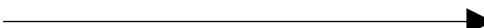
Mesos



Tachyon



SQL



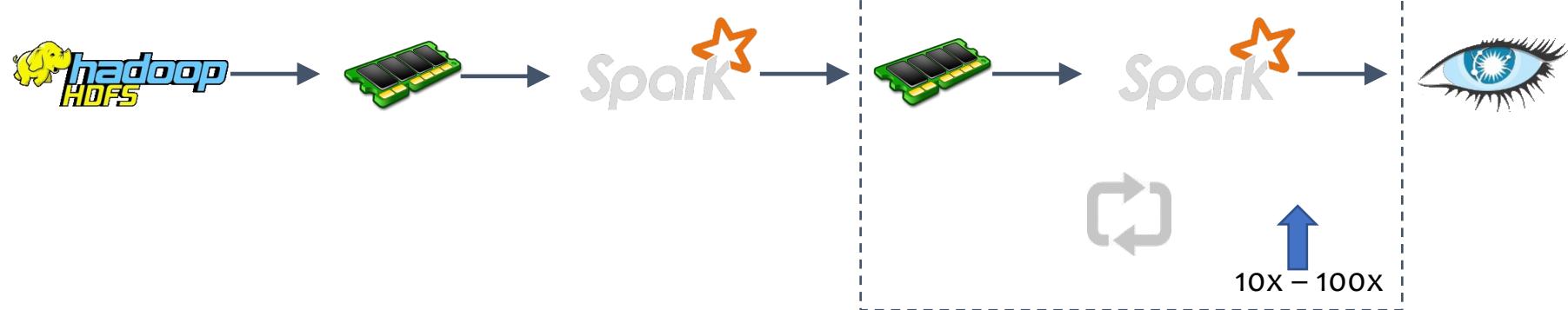
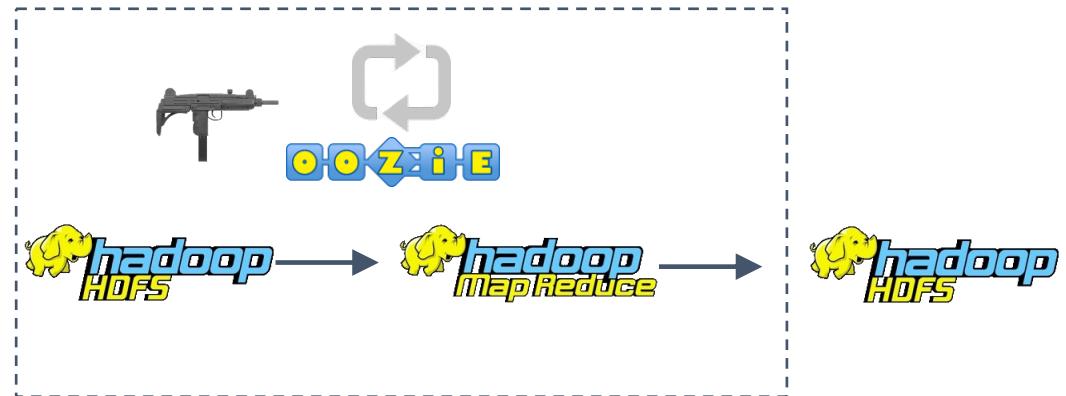
MLlib



STORM

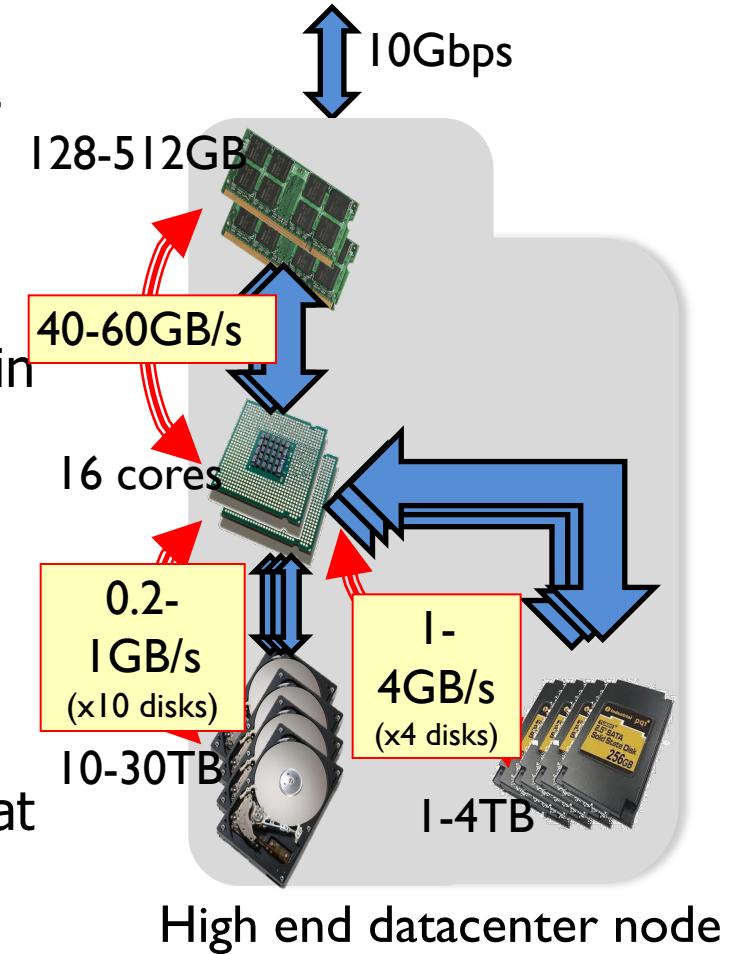


Streaming



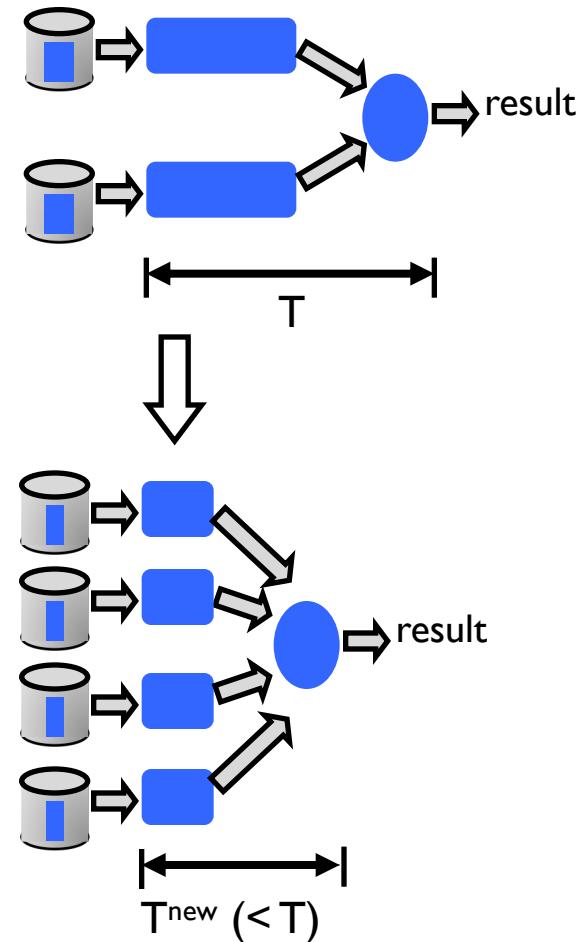
# Support Interactive and Streaming Comp.

- Aggressive use of ***memory***
- Why?
  1. Memory transfer rates >> disk or SSDs
  2. Many datasets already fit into memory
    - Inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
    - e.g., 1TB = 1 billion records @ 1KB each
  3. Memory density (still) grows with Moore's law
    - RAM/SSD hybrid memories at horizon



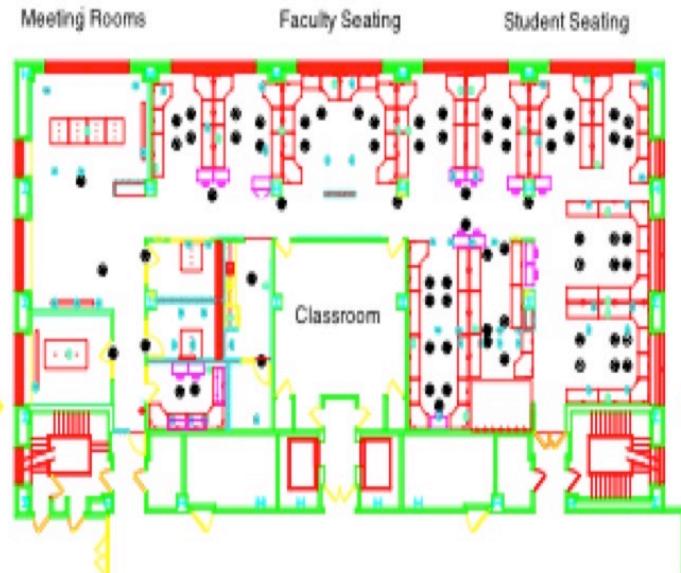
# Support Interactive and Streaming Comp.

- Increase ***parallelism***
- Why?
  - Reduce work per node → improve latency
- Techniques:
  - Low latency parallel **scheduler** that achieve high locality
  - Optimized **parallel communication patterns** (e.g., shuffle, broadcast)
  - Efficient **recovery** from failures and straggler mitigation



# Berkeley AMPLab

- “Launched” January 2011: 6 Year Plan
- 8 CS Faculty
- ~40 students
- 3 software engineers
- Organized for collaboration:



# Berkeley AMPLab

- Funding:

- 



XData,



CISE Expedition Grant

- Industrial, founding sponsors
  - 18 other sponsors, including



**Goal:** Next Generation of Analytics Data Stack for Industry & Research:

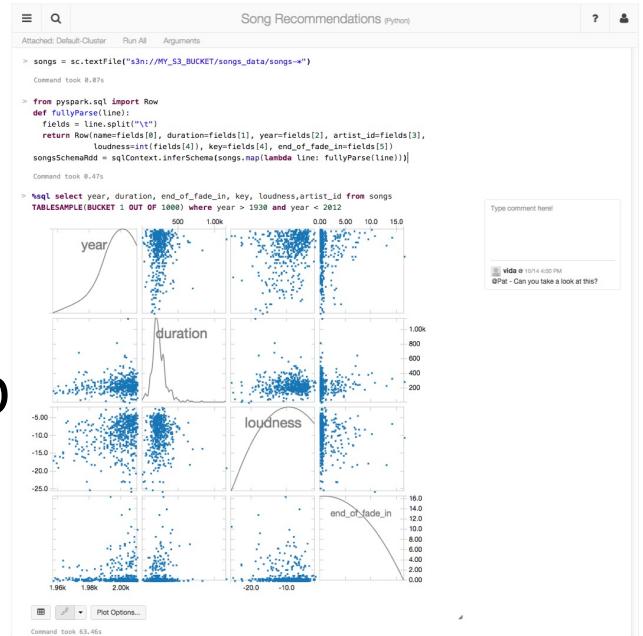
- Berkeley Data Analytics Stack (BDAS)
- Release as Open Source

# Databricks



# making big data simple

- Founded in late 2013
  - by the creators of Apache Spark
  - Original team from UC Berkeley AMPLab
  - Raised \$47 Million in 2 rounds



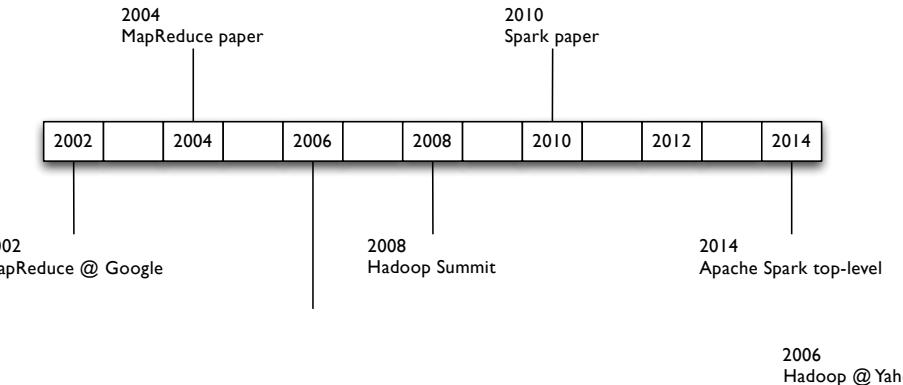
## Databricks Cloud:

**"A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products."**

The Databricks team contributed more than **75%** of the code  
added to Spark in the 2014



# History of Spark



## Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica  
University of California, Berkeley

### Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

### 1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and MapReduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by partitioning data and computation and then using an acyclic data flow graph to pass inputs through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper we focus on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs:** Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a

MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.

- **Interactive analytics:** Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Scala can be used interactively via a simplified version of the Scala interpreter, which allows the user to define RDDs, functional variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging.

We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

## Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury,  
Michael J. Franklin, Scott Shenker, Ion Stoica  
USENIX HotCloud (2010)

[people.csail.mit.edu/matei/papers/2010/hotcloud\\_spark.pdf](http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf)

## Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica NSDI (2012)

[usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf](http://usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf)

# History of Spark

## Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,  
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica  
*University of California, Berkeley*

### Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

### 1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that reuse intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serializa-

tion, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while Hadoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance efficiently. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.<sup>1</sup> If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

<sup>1</sup>Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

**“We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.**

**RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.**

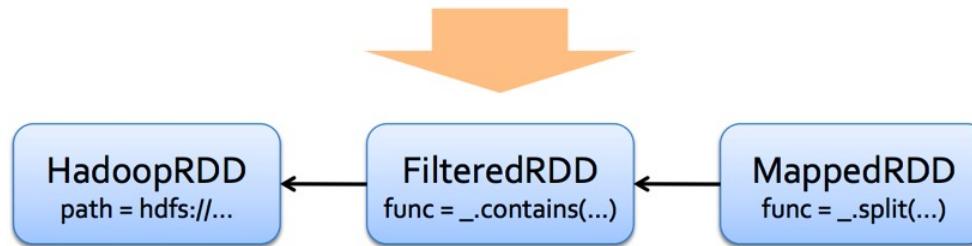
**In both cases, keeping data in memory can improve performance by an order of magnitude.”**

# History of Spark

## RDD Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data

E.g: `messages = textFile(...).filter(_.contains("error"))  
 .map(_.split('\t')(2))`



*The State of Spark, and Where We're Going Next*

**Matei Zaharia**

Spark Summit (2013)

[youtu.be/nU6vO2EJAb4](https://youtu.be/nU6vO2EJAb4)

# History of Spark



Analyze real time streams of data in  $\frac{1}{2}$  second intervals

[www.cs.berkeley.edu/~matei/papers/2013/sosp\\_spark\\_streaming.pdf](http://www.cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf)

**Discretized Streams: Fault-Tolerant Streaming Computation at Scale**

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica  
*University of California, Berkeley*

**Abstract**

Many “big data” applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

**1 Introduction**

Much of “big data” is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in

*faults and stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even more important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs  $2 \times$  the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed

```
TwitterUtils.createStream(...)
 .filter(_.getText.contains("Spark"))
 .countByWindow(Seconds(5))
```

# History of Spark



Seamlessly mix SQL queries with Spark programs.

## Spark SQL: Relational Data Processing in Spark

Michael Armbrust<sup>†</sup>, Reynold S. Xin<sup>†</sup>, Cheng Lian<sup>†</sup>, Yin Huai<sup>†</sup>, Davies Liu<sup>†</sup>, Joseph K. Bradley<sup>†</sup>, Xiangrui Meng<sup>†</sup>, Tomer Kaftan<sup>‡</sup>, Michael J. Franklin<sup>†‡</sup>, Ali Ghodsi<sup>†</sup>, Matei Zaharia<sup>\*</sup>

<sup>†</sup>Databricks Inc.    <sup>‡</sup>MIT CSAIL    <sup>\*</sup>AMPLab, UC Berkeley

### ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.*, declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.*, machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.*, schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

### Categories and Subject Descriptors

H.2 [Database Management]: Systems

### Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

### 1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
 "SELECT * FROM people")
names = results.map(lambda p:
p.name)
```

# History of Spark



Analyze networks of nodes and edges using graph processing

## GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley  
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

### ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has led to the development of new *graph-parallel* systems (e.g., Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

### 1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the

and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

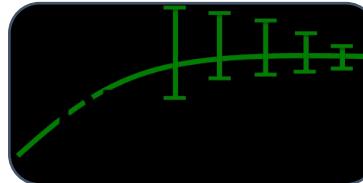
Alternatively, *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these

```
graph = Graph(vertices, edges)
messages =
spark.textFile("hdfs://...")
graph2 =
graph.joinVertices(messages) {
 (id, vertex, msg) => ...
}
```

[https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx\\_with\\_fonts.pdf](https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf)

# History of Spark



## SQL queries with Bounded Errors and Bounded Response Times

### BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal<sup>†</sup>, Barzan Mozafari<sup>◦</sup>, Aurojit Panda<sup>†</sup>, Henry Milner<sup>†</sup>, Samuel Madden<sup>◦</sup>, Ion Stoica<sup>\*†</sup>

<sup>†</sup>University of California, Berkeley

<sup>◦</sup>Massachusetts Institute of Technology

<sup>\*</sup>Conviva Inc.

{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

#### Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200x faster than Hive), within an error of 2-10%.

#### 1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks,

cessing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average SessionTime over all users originating in New York:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the Sessions table contains 100 million tuples for New York, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 New York tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

Queries with Time Bounds

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

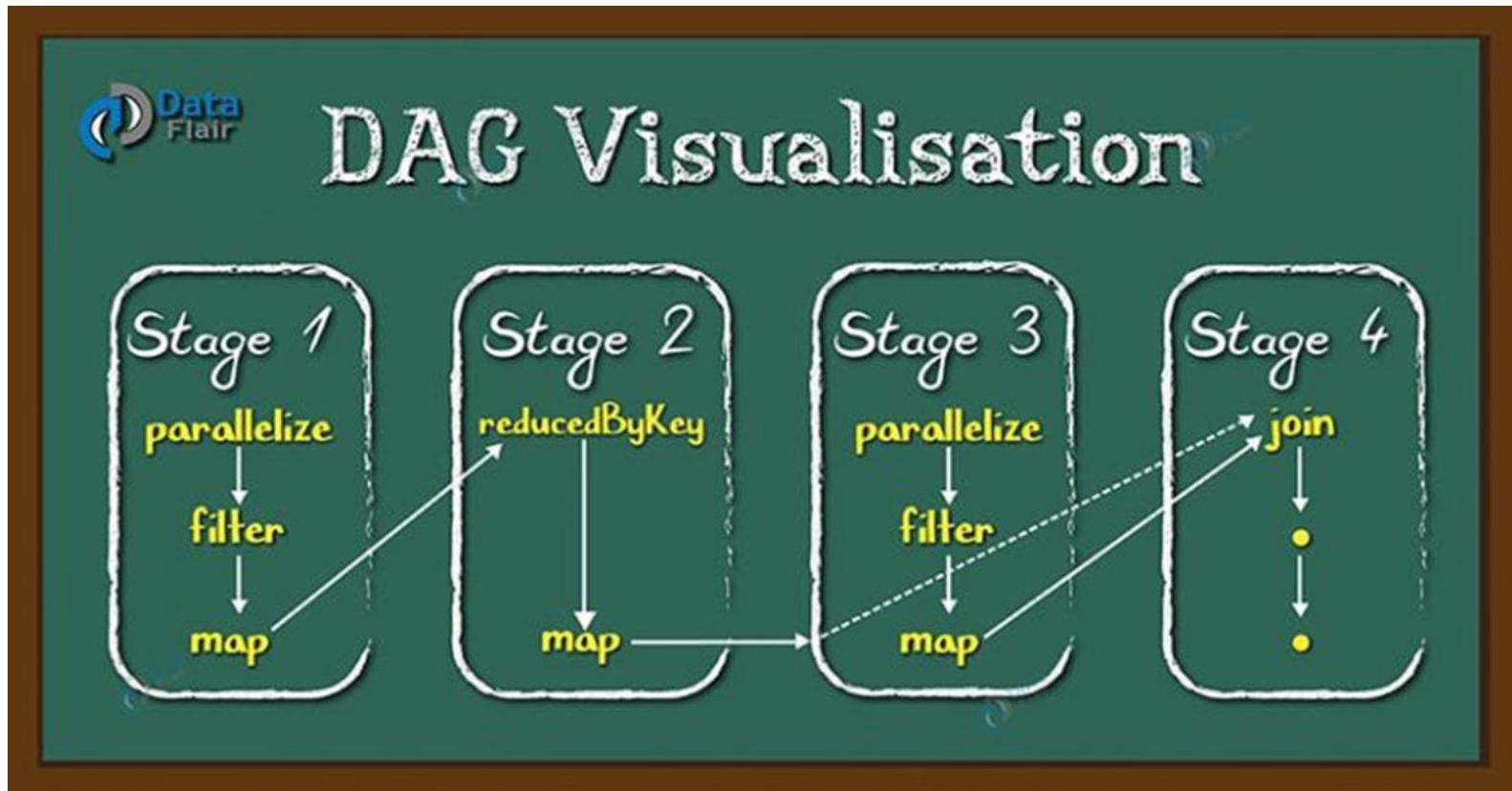
Queries with Error Bounds

[https://www.cs.berkeley.edu/~sameerag/blinkdb\\_eurosys13.pdf](https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf)

# History of Spark

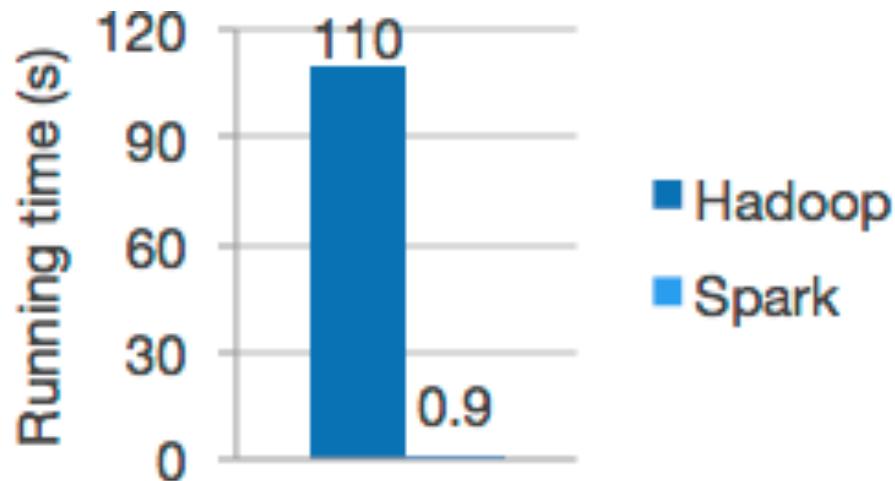
- Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine
- Two reasonably small additions are enough to express the previous models:
  - *fast data sharing*
  - *general DAGs*
- This allows for an approach which is more efficient for the engine, and much simpler for the end users

# Directed Acyclic Graph - DAG



# What is Apache Spark

- Spark is a unified **analytics** engine **for large-scale data processing**
- **Speed**: run workloads 100x faster
  - High performance for both batch and streaming data
  - Computations run in memory



Logistic regression in Hadoop and Spark

# What is Apache Spark

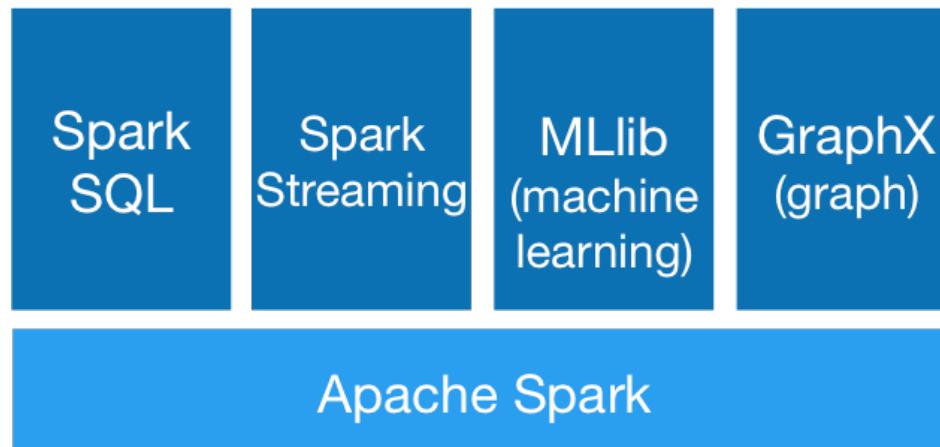
- **Ease of Use:** write applications quickly in Java, Scala, Python, R, SQL
  - Offer over 80 high-level operators
  - Use them interactively from Scala, Python, R, and SQL

```
df = spark.read.json("logs.json") df.
where("age > 21")
select("name.first").show()
```

Spark's Python DataFrame API  
Read JSON files with automatic schema inference

# What is Apache Spark

- **Generality:** combine SQL, Streaming, and complex analytics
  - Provide libraries including SQL and DataFrames, Spark Streaming, MLib, GraphX,
  - Wide range of workloads e.g., batch applications, interactive algorithms, interactive queries, streaming



# What is Apache Spark

- Run Everywhere:
  - run on Hadoop, Apache Mesos, Kubernetes, standalone or in the cloud.
  - access data in HDFS, Aluxio, Apache Cassandra, Apache Hbase, Apache Hive, etc.



# Comparison between Hadoop and Spark

	 <b>Hadoop MapReduce</b>	 <b>Spark</b>
<b>Strengths</b>	<ul style="list-style-type: none"><li>▪ Can collect any data</li><li>▪ Limitless in size</li></ul>	<ul style="list-style-type: none"><li>▪ Can work off any Hadoop collection</li><li>▪ Runs on Hadoop, or other clusters</li><li>▪ In-memory processing makes it very fast</li><li>▪ Supports Java, Scala, Python, and R*, and can be used with SQL.</li></ul>
<b>Used for</b>	<ul style="list-style-type: none"><li>▪ Initial data ingestion</li><li>▪ Data curation</li><li>▪ Large-scale “boil the ocean” analytics</li><li>▪ Data archiving</li></ul>	<ul style="list-style-type: none"><li>▪ Complex query processing of large amounts of data quickly</li><li>▪ Can handle ad hoc queries</li></ul>
<b>Limitations</b>	<ul style="list-style-type: none"><li>▪ MapReduce is hard to program</li><li>▪ Disk-based batch nature limits speed, agility.</li></ul>	<ul style="list-style-type: none"><li>▪ Limited only by processor speed, available memory, cores, and cluster size.</li></ul>

# 100TB Daytona Sort Competition

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Spark sorted the same data **3X faster**  
using **10X fewer machines**  
than Hadoop MR  
in 2013.

All the sorting took place on disk (HDFS)  
without using Spark's in-memory cache!

# Startup Crunches 100 Terabytes of Data in a Record 23 Minutes

BY KLINT FINLEY | 10.13.14 | 2:36 PM | PERMALINK

[Share](#) 1.1k [Tweet](#) 789 [G+1](#) 75 [in Share](#) 565 [Pin it](#)

The screenshot shows the Gigaom homepage with a prominent yellow sidebar on the left. The main content area features a large headline: "Startup Crunches 100 Terabytes of Data in a Record 23 Minutes". Below the headline, it says "BY KLINT FINLEY | 10.13.14 | 2:36 PM | PERMALINK". There are social sharing buttons for Facebook, Twitter, Google+, LinkedIn, and Pinterest. The share count is 1.1k for Facebook, 789 for Twitter, 75 for Google+, and 565 for LinkedIn. The LinkedIn button includes the text "in Share". Below the social sharing are three "Must Reads" articles with small thumbnail images and titles: "Google launches Contributor, a crowdfunding tool for publishers", "Net neutrality looks doomed in Europe before it even gets started", and "Five tech products that designers have fallen in love with". At the bottom of the main content area, there is a dark banner with white text: "Databricks demolishes big data benchmark to prove Spark is fast on disk, too". Below this banner, it says "by Derrick Harris Oct. 10, 2014 - 1:49 PM PST" and "Comment".

# The Spark stack

Spark SQL  
structured data

Spark Streaming  
real-time

MLib  
machine  
learning

GraphX  
graph  
processing

Spark Core

Standalone Scheduler

YARN

Mesos

# The Spark stack

- Spark Core:
  - contain basic functionality of Spark including task scheduling, memory management, fault recovery, etc.
  - provide APIs for building and manipulating RDDs
- SparkSQL
  - allow querying structured data via SQL, Hive Query Language
  - allow combining SQL queries and data manipulations in Python, Java, Scala

# The Spark stack

- Spark Streaming: enables processing of live streams of data via APIs
- Mlib:
  - contain common machine language functionality
  - provide multiple types of algorithms: classification, regression, clustering, etc.
- GraphX:
  - library for manipulating graphs and performing graph-parallel computations
  - extend Spark RDD API

# The Spark stack

- Cluster Managers
  - Hadoop Yarn
  - Apache Mesos, and
  - Standalone Scheduler (simple manager in Spark).

# Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

# RDD Basics

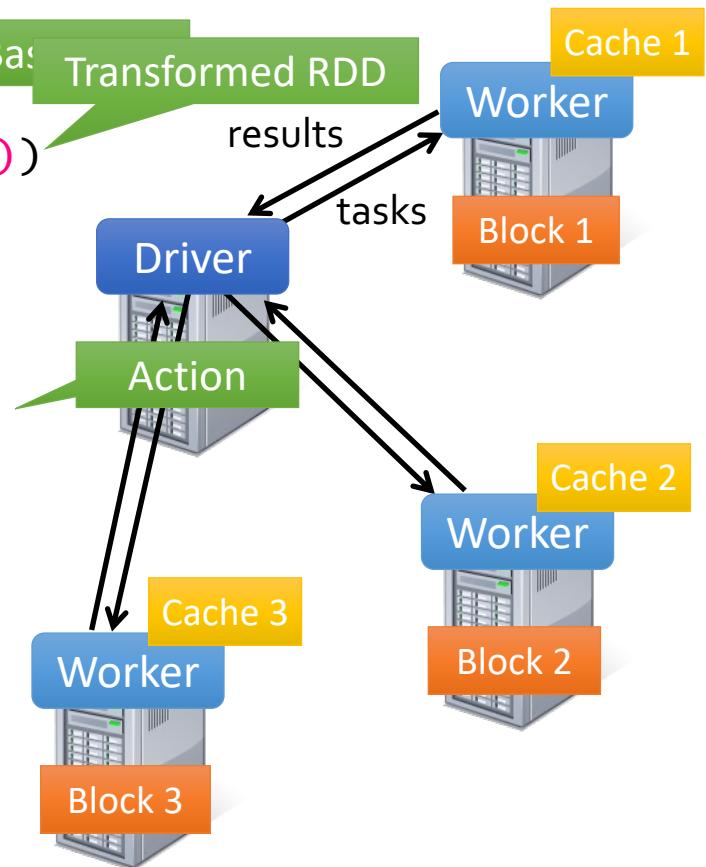
- RDD:
  - Immutable distributed collection of objects
  - Split into multiple partitions => can be computed on different nodes
- All work in Spark is expressed as
  - creating new RDDs
  - transforming existing RDDs
  - calling actions on RDDs

# Example

Load error messages from a log into memory,  
then interactively search for various patterns

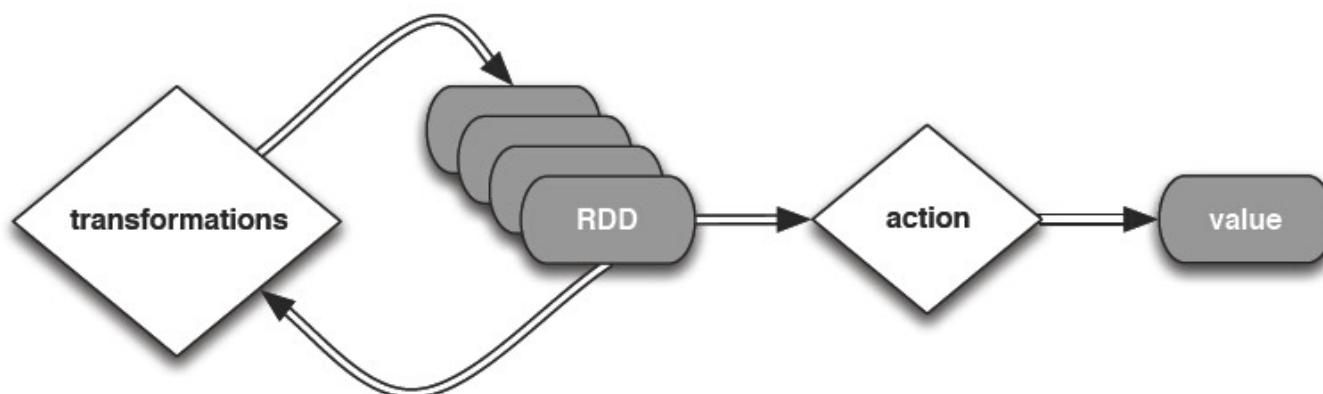
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t")(2))
cachedMsgs = messages.cache()
```

```
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```



# RDD Basics

- Two types of operations: transformations and actions
- Transformations: construct a new RDD from a previous one e.g., filter data
- Actions: compute a result base on an RDD e.g., count elements, get first element



# Transformations

- Create new RDDs from existing RDDs
- Lazy evaluation
  - See the whole chain of transformations
  - Compute just the data needed
- Persist contents:
  - persist an RDD in memory to reuse it in future
  - persist RDDs on disk is possible

# **Typical works of a Spark program**

1. Create some input RDDs from external data
2. Transform them to define new RDDs using transformations like filter()
3. Ask Spark to persist() any intermediate RDDs that will need to be reused
4. Launch actions such as count(), first() to kick off a parallel computation

# Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

# Two ways to create RDDs

## 1. Parallelizing a collection: uses parallelize()

- Python

```
lines = sc.parallelize(["pandas", "i like
pandas"])
```

- Scala

```
val lines = sc.parallelize(List("pandas", "i
like pandas"))
```

- Java

```
JavaRDD<String> lines =
sc.parallelize(Arrays.asList("pandas", "i
like pandas"));
```

## **Two ways to create RDDs**

### 2. Loading data from external storage

- Python

```
lines =
sc.textFile("/path/to/README.md")
```

- Scala

```
val lines =
sc.textFile("/path/to/README.md")
```

- Java

```
JavaRDD<String> lines =
sc.textFile("/path/to/README.md");
```

# Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- **RDD Operations**
- Common Transformation and Actions
- Persistence (Caching)

# RDD Operations

- Two types of operations
  - Transformations: operations that **return a new RDDs** e.g., `map()`, `filter()`
  - Actions: operations that return a **result** to the driver program or write it to storage such as `count()`, `first()`
- Treated differently by Spark
  - Transformation: lazy evaluation
  - Action: execution at any time

# Transformation

- Example 1. Use **filter()**

- Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line =>
 line.contains("error"))
```

- Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
 new Function<String, Boolean>() {
 public Boolean call(String x) {
 return x.contains("error"); } })
);
```

# Transformation

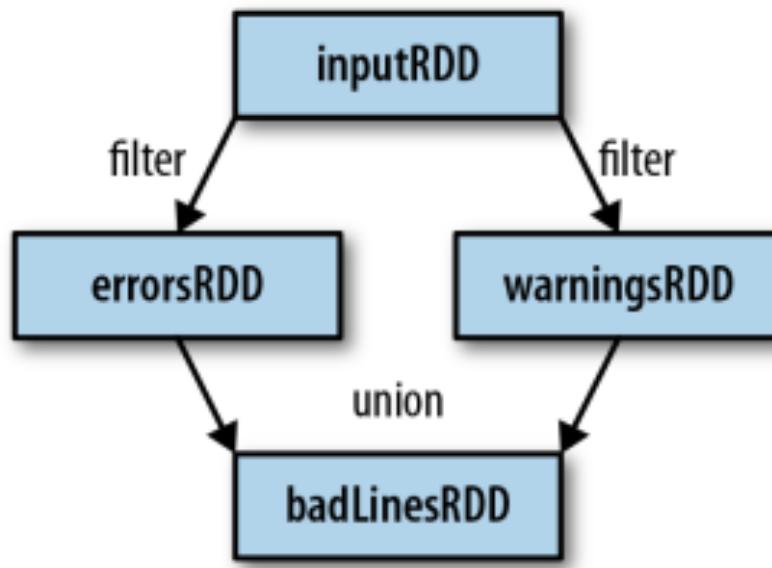
- `filter()`
  - does not change the existing *inputRDD*
  - returns a pointer to an entirely new RDD
  - *inputRDD* still can be reused
- `union()`

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- transformations can operate on any number of input RDDs

# Transformation

- Spark keeps track dependencies between RDDs, called the **lineage graph**
- Allow recovering lost data



# Actions

- Example. count the number of errors
- Python

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
 print line
```

- Scala

```
println("Input had " + badLinesRDD.count() + " concerning
lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

- Java

```
System.out.println("Input had " + badLinesRDD.count() + "
concerning lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
 System.out.println(line);
}
```

# Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)



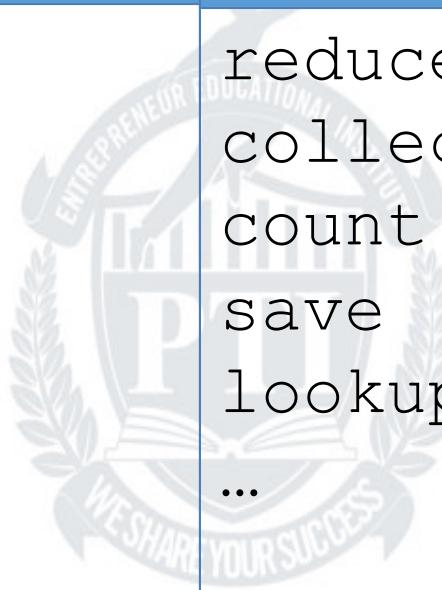
# RDD Basics

## Transformations

map  
flatMap  
filter  
sample  
union  
groupByKey  
reduceByKey  
join  
cache  
...

## Actions

reduce  
collect  
count  
save  
lookupKey  
...



# Transformations

<i>transformation</i>	<i>description</i>
<b>map(func)</b>	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
<b>filter(func)</b>	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
<b>flatMap(func)</b>	similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item)
<b>sample(withReplacement, fraction, seed)</b>	sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator <i>seed</i>
<b>union(otherDataset)</b>	return a new dataset that contains the union of the elements in the source dataset and the argument
<b>distinct([numTasks]))</b>	return a new dataset that contains the distinct elements of the source dataset

# Transformations

<i>transformation</i>	<i>description</i>
<b>groupByKey</b> ( [ <i>numTasks</i> ] )	when called on a dataset of (K, V) pairs, returns a dataset of (K, Seq[V]) pairs
<b>reduceByKey</b> ( <i>func</i> , [ <i>numTasks</i> ] )	when called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function
<b>sortByKey</b> ( [ <i>ascending</i> ], [ <i>numTasks</i> ] )	when called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
<b>join</b> ( <i>otherDataset</i> , [ <i>numTasks</i> ] )	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key
<b>cogroup</b> ( <i>otherDataset</i> , [ <i>numTasks</i> ] )	when called on datasets of type (K, V) and (K, W), returns a dataset of (K, Seq[V], Seq[W]) tuples – also called groupWith
<b>cartesian</b> ( <i>otherDataset</i> )	when called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements)

# Actions

<i>action</i>	<i>description</i>
<b>reduce(<i>func</i>)</b>	aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
<b>collect()</b>	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
<b>count()</b>	return the number of elements in the dataset
<b>first()</b>	return the first element of the dataset – similar to <i>take(1)</i>
<b>take(<i>n</i>)</b>	return an array with the first <i>n</i> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
<b>takeSample(<i>withReplacement</i>, <i>fraction</i>, <i>seed</i>)</b>	return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator seed

# Actions

<i>action</i>	<i>description</i>
<b>saveAsTextFile(path)</b>	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call <code>toString</code> on each element to convert it to a line of text in the file
<b>saveAsSequenceFile(path)</b>	write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
<b>countByKey()</b>	only available on RDDs of type (K, V). Returns a 'Map' of (K, Int) pairs with the count of each key
<b>foreach(func)</b>	run a function <code>func</code> on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

# Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

# Persistence levels

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

# Persistence

- Example

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(", "))
```

## **Acknowledgement and References**

# Agenda



## Zepelin notebook

What and why we need it?

Installation using Docker

Usage



## Load, inspect, and save data

Loading data from difference sources

Simple inspecting commands

Saving data

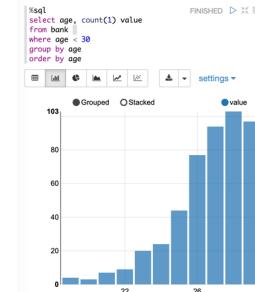
# **Zeppelin notebook**

- A web-based interface for interactive data analytics
  - Easy to write and access your code
  - Support many programming languages
    - Scala (with Apache Spark), Python (with Apache Spark),  
SparkSQL, Hive, Markdown, Angular, and Shell
    - Data visualization
- Monitoring Spark jobs

# Zeppelin usage

- Run the first node: “About this Build”
  - Check Spark version
- Check Spark running mode
  - <http://localhost:4040>
  - Need to start Spark first by running the first note
- Run the second node: “Tutorial/Basic Features (Spark)”
  - Load data into table
  - SQL example

The screenshot shows the Zeppelin web interface. At the top, there's a navigation bar with the Zeppelin logo, 'Notebook', and 'Job'. Below the header, a welcome message reads 'Welcome to Zeppelin!'. A sidebar on the left contains a 'Notebook' section with 'Import note' and 'Create new note' buttons, and a 'Filter' input field. A tree view lists nodes: 'About this Build' (selected), 'Zeppelin Tutorial' (expanded), 'Basic Features (Spark)', 'Matplotlib (Python + PySpark)', 'R (SparkR)', 'Using Flink for batch processing', 'Using Mahout', and 'Using Pig for querying data'. On the right, there are links for 'Help', 'Documentation', 'Community', 'Mailing list', 'Issues tracking', and 'Github'.



# **Useful Docker commands**

- Login to a container
  - docker ps (get any container id)
  - docker exec -it container\_id bash
- List all containers: docker ps -a
- Stop a container: docker stop container\_id
- Start a stopped container: docker start container\_id

## **Load, inspect, and save data**

- Data is always huge that does not fit on a single machine
  - Data is distributed on many storage nodes
- Data scientists can likely focus on the format that their data is already in
  - Engineers may wish to explore more output formats
- Spark supports a wide range of input and output sources

# Data sources

- File formats and filesystems
  - Local or distributed filesystem, such as NFS, HDFS, or Amazon S3
  - File formats including text, JSON, SequenceFiles, and protocol buffers
- Structured data sources through Spark SQL
  - Apache Hive
  - Parquet
  - JSON
  - From RDDs
- Databases and key/value stores
  - Cassandra, HBase, Elasticsearch, and JDBC dbs

# File Formats

- Formats range from unstructured, like text, to semistructured, like JSON, to structured, like SequenceFiles

*Table 5-1. Common supported file formats*

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

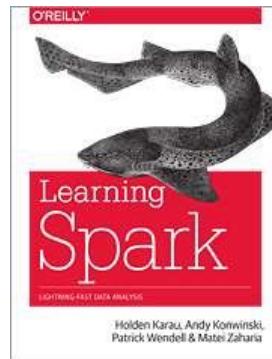
# **Lab: loading, inspecting, and saving data**

- On the Zeppelin notebook
  - <http://localhost:8080/#/notebook/2EAMFFAH7>

# From where to learn Spark ?

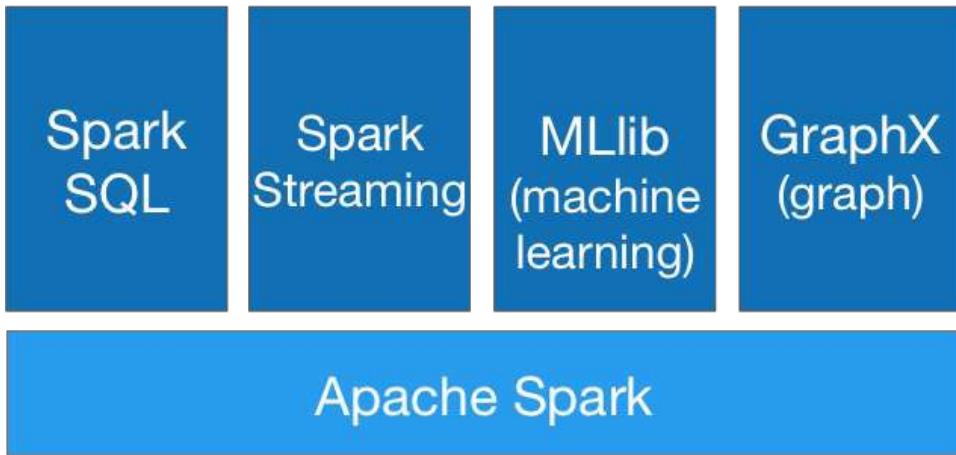


<http://spark.apache.org/>



<http://shop.oreilly.com/product/0636920028512.do>

# Spark architecture



# Easy ways to run Spark ?

- ★ your IDE (ex. Eclipse or IDEA)
- ★ Standalone Deploy Mode: simplest way to deploy Spark on a single machine
- ★ Docker & Zeppelin
- ★ EMR
- ★ Hadoop vendors (Cloudera, Hortonworks)  
Digital Ocean (Kubernetes cluster)

# Supported languages

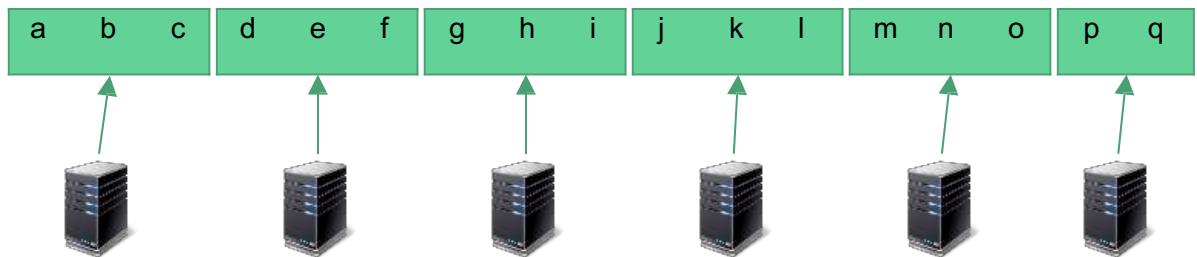


python



# RDD

An RDD is simply an immutable distributed collection of objects!



# RDD (Resilient Distributed Dataset)

## **RDD (Resilient Distributed Dataset)**

- Resilient: If data in memory is lost, it can be recreated
  - Distributed: Processed across the cluster
  - Dataset: Initial data can come from a source such as a file, or it can be created programmatically
- RDDs are the fundamental unit of data in Spark**
- Most Spark programming consists of performing operations on RDDs**

# Creating RDD (I)

Python

```
lines = sc.parallelize(["workshop", "spark"])
```

Scala

```
val lines = sc.parallelize(List("workshop", "spark"))
```

Java

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("workshop", "spark"))
```

# Creating RDD (II)

Python

```
lines = sc.textFile("/path/to/file.txt")
```

Scala

```
val lines = sc.textFile("/path/to/file.txt")
```

Java

```
JavaRDD<String> lines = sc.textFile("/path/to/file.txt")
```

# RDD persistence

```
MEMORY_ONLY
MEMORY_AND_DISK
MEMORY_ONLY_SER
MEMORY_AND_DISK_SER
DISK_ONLY
MEMORY_ONLY_2
MEMORY_AND_DISK_2
OFF_HEAP
```

# RDDs

**RDDs can hold any serializable type of element**

- Primitive types such as integers, characters, and booleans
- Sequence types such as strings, lists, arrays, tuples, and dicts (including nested data types)
- Scala/Java Objects (if serializable)
- Mixed types

**§ Some RDDs are specialized and have additional functionality**

- Pair RDDs
- RDDs consisting of key-value pairs
- Double RDDs
- RDDs consisting of numeric data

# Creating RDDs from Collections

You can create RDDs from collections instead of files

`-sc.parallelize(collection)`

```
myData = ["Alice","Carlos","Frank","Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2) ['Alice', 'Carlos']
```

# Creating RDDs from Text Files (1)

**For file-based RDDs, use `SparkContext.textFile`**

- Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files. Examples:

- `sc.textFile("myfile.txt")`

- `sc.textFile("mydata/")`

- `sc.textFile("mydata/*.log")`

- `sc.textFile("myfile1.txt,myfile2.txt")`

- Each line in each file is a separate record in the RDD

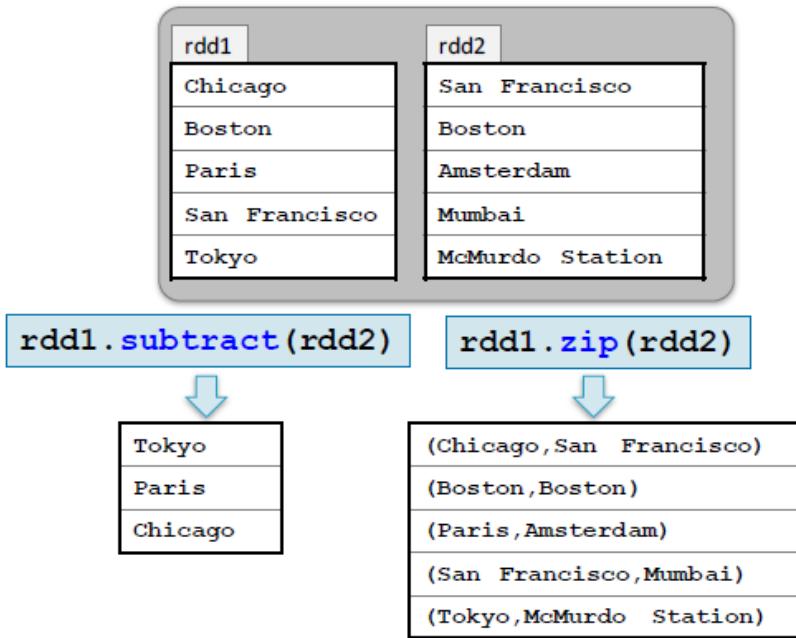
**Files are referenced by absolute or relative URI**

- Absolute URI:

- `file:/home/training/myfile.txt`

- `hdfs://nnhost/loudacre/myfile.txt`

# Examples: Multi-RDD Transformations (1)



# Examples: Multi-RDD Transformations (2)

rdd1	rdd2
Chicago	San Francisco
Boston	Boston
Paris	Amsterdam
San Francisco	Mumbai
Tokyo	McMurdo Station

`rdd1.union(rdd2)`



Chicago
Boston
Paris
San Francisco
Tokyo
San Francisco
Boston
Amsterdam
Mumbai
McMurdo Station

`rdd1.intersection(rdd2)`



Boston
San Francisco

# Some Other General RDD Operations

## Other RDD operations

~~-*first*~~ returns the first element of the RDD

~~-*foreach*~~ applies a function to each element in an RDD

~~-*top(n)*~~ returns the largest  $n$  elements using natural ordering

## Sampling operations

-*sample* creates a new RDD with a sampling of elements

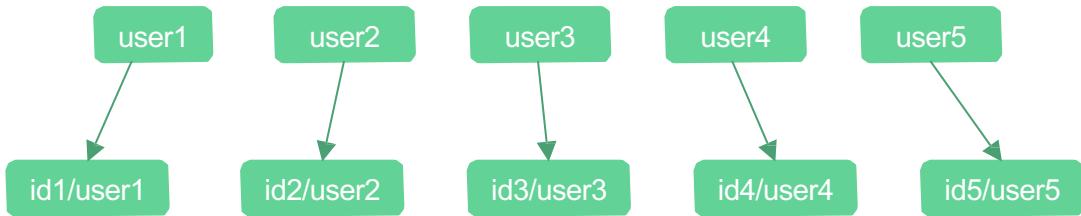
-*take* Sample returns an array of sampled elements

# Other data structures in Spark

- ★ Paired RDD
- ★ DataFrame
- ★ DataSet

# Paired RDD

Paired RDD = an RDD of key/value pairs



# Pair RDDs

## § Pair RDDs are a special form of RDD

- Each element must be a key-value pair (a two-element *tuple*)
- Keys and values can be any type

## § Why?

- Use with map-reduce algorithms
- Many additional functions are available for common data processing needs
- Such as sorting, joining, grouping, and counting

Pair RDD

(key1 , value1)
(key2 , value2)
(key3 , value3)
...

# Creating Pair RDDs

**The first step in most workflows is to get the data into key/value form**

- What should the RDD should be keyed on?
- What is the value?

**Commonly used functions to create pair RDDs**

- map**
- flatMap / flatMapValues**
- keyBy**

# Example: A Simple Pair RDD

Example: Create a pair RDD from a tab-separated file

```
> val users = sc.textFile(file).
 map(line => line.split('\t')).
 map(fields => (fields(0), fields(1)))
```

user001\tFred Flintstone  
user090\tBugs Bunny  
user111\tHarry Potter  
...

(user001, Fred Flintstone)  
(user090, Bugs Bunny)  
(user111, Harry Potter)  
...

# Example: Keying Web Logs by User ID

```
> sc.textFile(logfile).
 keyBy(line => line.split(' ') (2))
```

User ID
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" ...
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" ...
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" ...
...

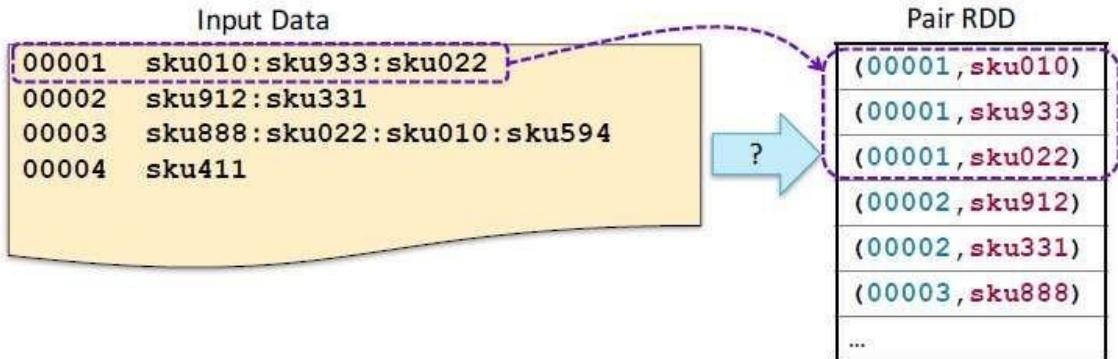


(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html...)
(99788,56.38.234.188 - 99788 "GET /theme.css...)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html...)
...

# Mapping Single Rows to Multiple Pairs

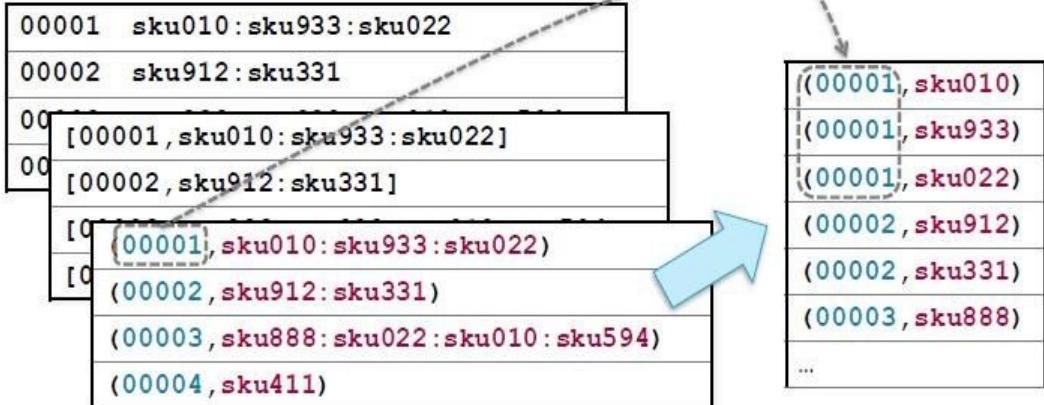
- How would you do this?

- Input: order numbers with a list of SKUs in the order
- Output: `order` (key) and `sku` (value)



# Answer : Mapping Single Rows to Multiple Pairs

```
> sc.textFile(file) \
 .map(lambda line: line.split('\t')) \
 .map(lambda fields: (fields[0], fields[1])) \
 .flatMapValues(lambda skus: skus.split(':'))
```



# Map-Reduce

## § Map-reduce is a common programming model

- Easily applicable to distributed processing of large data sets

## § Hadoop MapReduce is the major implementation

- Somewhat limited
- Each job has one map phase, one reduce phase
- Job output is saved to files

## § Spark implements map-reduce with much greater flexibility

- Map and reduce functions can be interspersed
- Results can be stored in memory
- Operations can easily be chained

# Map-Reduce in Spark

## § Map-reduce in Spark works on pair RDDs

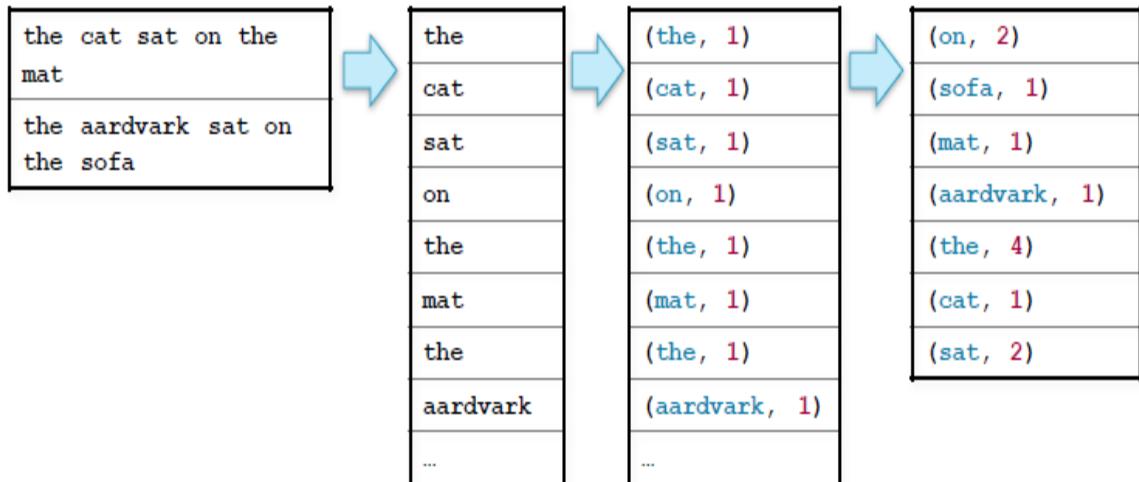
### § Map phase

- Operates on one record at a time
- “Maps” each record to zero or more new records
- Examples: `map`, `flatMap`, `filter`, `keyBy`

### § Reduce phase

- Works on map output
- Consolidates multiple records
- Examples: `reduceByKey`, `sortByKey`, `mean`

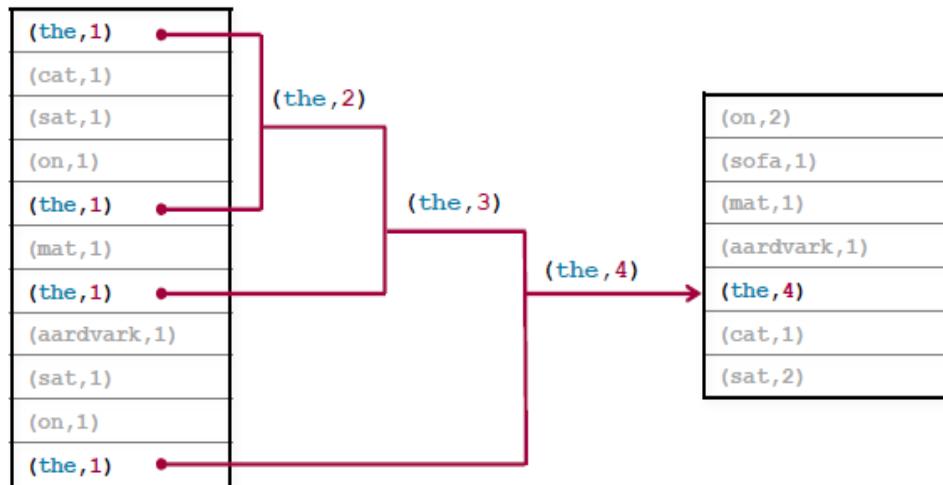
# Example: Word Count



# reduceByKey

The function passed to reduceByKey combines values from two keys

- Function must be binary



```
> val counts = sc.textFile (file) . flatMap
 (line => line.split (' ') . map (word => (word
, 1)) . reduceByKey (v1 ,v2) => v1+v2)
```

OR

```
> val counts = sc.textFile (file) . flatMap
 (_.split (' ') .
 map ((_, 1)) .
 reduceByKey (_+_)
```

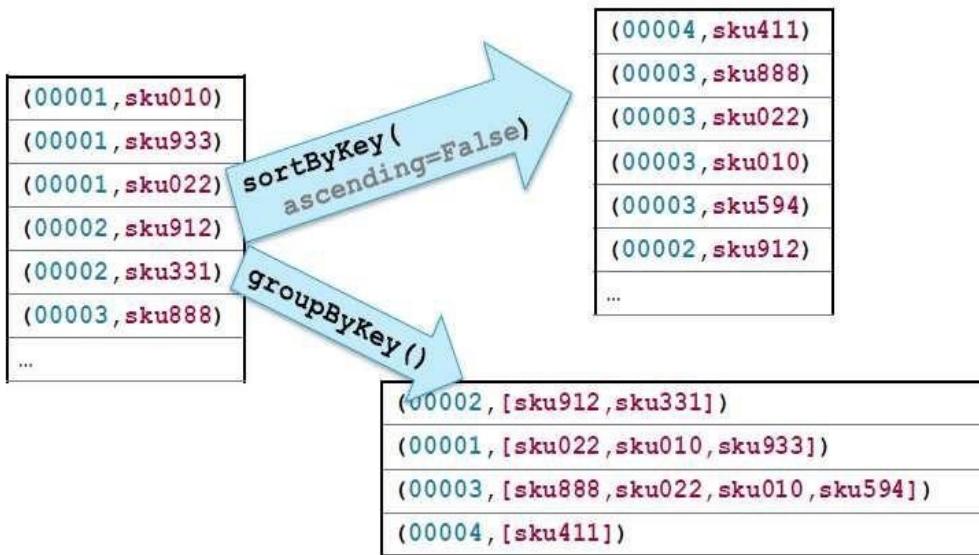
# Pair RDD Operations

§ In addition to map and reduceByKey operations, Spark has several operations specific to pair RDDs

## § Examples

- countByKey** returns a map with the count of occurrences of each key
- groupByKey** groups all the values for each key in an RDD
- sortByKey** sorts in ascending or descending order
- join** returns an RDD containing all pairs with matching keys from two RDD

# Example: Pair RDD Operations



# Example: Joining by Key

```
> movies = moviegross.join(movieyear)
```

RDD:moviegross
(Casablanca, \$3.7M)
(Star Wars, \$775M)
(Annie Hall, \$38M)
(Argo, \$232M)
...

RDD:movieyear
(Casablanca, 1942)
(Star Wars, 1977)
(Annie Hall, 1977)
(Argo, 2012)
...

(Casablanca, (\$3.7M, 1942))
(Star Wars, (\$775M, 1977))
(Annie Hall, (\$38M, 1977))
(Argo, (\$232M, 2012))
...

# Other Pair Operations

## § Some other pair operations

- keys** returns an RDD of just the keys, without the values
- values** returns an RDD of just the values, without keys
- lookup(key)** returns the value(s) for a key
- leftOuterJoin**, **rightOuterJoin** , **fullOuterJoin** join two RDDs, including keys defined in the left, right or either RDD respectively
- mapValues**, **flatMapValues** execute a function on just the values, keeping the key the same

# What is Spark SQL?

## § What is Spark SQL?

- Spark module for structured data processing
- Replaces Shark (a prior Spark module, now deprecated)
- Built on top of core Spark

## § What does Spark SQL provide?

- The DataFrame API—a library for working with data as tables
- Defines DataFrames containing rows and columns
- DataFrames are the focus of this chapter!
- Catalyst Optimizer—an extensible optimization framework
- A SQL engine and command line interface

# SQL Context

## § The main Spark SQL entry point is a SQL context object

- Requires a **SparkContext** object
- The SQL context in Spark SQL is similar to Spark context in core Spark

## § There are two implementations

- SQLContext**
- Basic implementation
- HiveContext**
- Reads and writes Hive/HCatalog tables directly
- Supports full HiveQL language
- Requires the Spark application be linked with Hive libraries
- Cloudera recommends using **HiveContext**

# Creating a SQL Context

## § The Spark shell creates a HiveContext instance automatically

- Call `sqlContext`
- You will need to create one when writing a Spark application
- Having multiple SQL context objects *is* allowed

## § A SQL context object is created based on the Spark context

Language: Scala

```
import org.apache.spark.sql.hive.HiveContext
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
```

# DataFrames

## § **DataFrames are the main abstraction in Spark SQL**

- Analogous to RDDs in core Spark
- A distributed collection of structured data organized into Named columns
- Built on a *base RDD* containing **Row** objects

# Creating a DataFrame from a Data Source

§ `sqlContext.read` returns a `DataFrameReader` object

§ `DataFrameReader` provides the functionality to load data into a `DataFrame`

§ Convenience functions

- `json(filename)`
- `parquet(filename)`
- `orc(filename)`
- `table(hive-tablename)`
- `jdbc(url,table,options)`

# Example: Creating a DataFrame from a JSON File

```
Language: Scala
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val peopleDF = sqlContext.read.json("people.json")
```

File: people.json

```
{"name": "Alice", "pcode": "94304"}
{"name": "Brayden", "age": 30, "pcode": "94304"}
{"name": "Carla", "age": 19, "pcode": "10036"}
{"name": "Diana", "age": 46}
{"name": "Étienne", "pcode": "94104"}
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

# Example: Creating a DataFrame from a Hive/Impala Table

Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val customerDF = sqlContext.read.table("customers")
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
...	...	...



cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
...	...	...

# Loading from a Data Source Manually

§ You can specify settings for the DataFrameReader

**-format**: Specify a data source type

**-option**: A key/value setting for the underlying data source

**-schema**: Specify a schema instead of inferring from the data source

§ Then call the generic base function load

```
sqlContext.read.
 format("com.databricks.spark.avro") .
 load("/loudacre/accounts_avro")
```

```
sqlContext.read.
 format("jdbc") .
 option("url", "jdbc:mysql://localhost/loudacre") .
 option("dbtable", "accounts") .
 option("user", "training") .
 option("password", "training") .
 load()
```

# Data Sources

## § Spark SQL 1.6 built-in data source types

- table
- json
- parquet
- jdbc
- orc

## § You can also use third party data source libraries, such as

- Avro (included in CDH)
- HBase
- CSV
- MySQL
- and more being added all the time

# DataFrame Basic Operations

- § Basic operations deal with DataFrame metadata (rather than its data)
- § Some examples
  - `-schema` returns a schema object describing the data
  - `-printSchema` displays the schema as a visual tree
  - `-cache / persist` persists the DataFrame to disk or memory
  - `-columns` returns an array containing the names of the columns
  - `-dtypes` returns an array of (column name,type) pairs
  - `-explain` prints debug information about the DataFrame to the console

# DataFrame Basic Operations

Language: Scala

```
> val peopleDF = sqlContext.read.json("people.json")
> peopleDF.dtypes.foreach(println)
(age, LongType)
(name, StringType)
(pcode, StringType)
```

# DataFrame Actions

§ Some DataFrame actions

- collect returns all rows as an array of Row objects
- take( $n$ ) returns the first  $n$  rows as an array of Row objects
- count returns the number of rows
- show( $n$ ) displays the first  $n$  rows  
(default=20)

Language: Scala

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age name pcode
null Alice 94304
30 Brayden 94304
19 Carla 10036
```

# DataFrame Queries

## § DataFrame query methods return new DataFrames

- Queries can be chained like transformations

## § Some query methods

- distinct** returns a new DataFrame with distinct elements of this DF
- join** joins this DataFrame with a second DataFrame
  - Variants for inside, outside, left, and right joins
- limit** returns a new DataFrame with the first **n** rows of this DF
- select** returns a new DataFrame with data from one or more columns of the base DataFrame
- where** returns a new DataFrame with rows meeting specified query criteria (alias for **filter**)

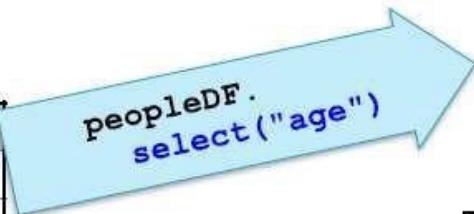
# DataFrame Query Strings

- Some query operations take strings containing simple query expressions

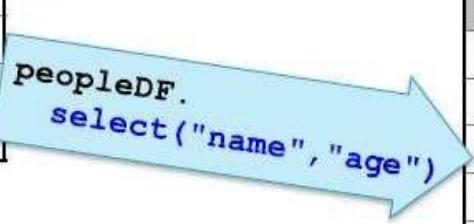
- Such as `select` and `where`

- Example: `select`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age
null
30
19
46
null



name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

# Querying DataFrames using Columns

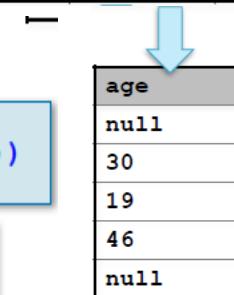
## § Columns can be referenced in multiple ways

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

### ▪ Scala

```
val ageDF = peopleDF.select(peopleDF("age"))
```

```
val ageDF = peopleDF.select($"age")
```



age
null
30
19
46
null

# Joining DataFrames

## § A basic inner join when join column is in both DataFrames

age	name	PCODE
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

```
language: Python/Scala
```

```
peopleDF.join(pcodesDF, "PCODE")
```

PCODE	CITY	STATE
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

PCODE	AGE	NAME	CITY	STATE
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
94104	null	Étienne	San Francisco	CA

# Joining DataFrames

- Specify type of join as inner (default), outer, left\_outer, right\_outer, or leftsemi

age	name	PCODE
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Etienne	94104

```
language: Python
peopleDF.join(pcodesDF, "PCODE",
 "left_outer")
```

```
language: Scala
peopleDF.join(pcodesDF,
 Array("PCODE"), "left_outer")
```

PCODE	CITY	STATE
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

PCODE	AGE	NAME	CITY	STATE
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
null	46	Diana	null	null
94104	null	Etienne	San Francisco	CA

# SQL Queries

§ When using HiveContext, you can query Hive/Impala tables using HiveQL

- Returns a DataFrame

Language: Python/Scala

```
sqlContext.
 sql("""SELECT * FROM customers WHERE name LIKE "A%" """)
```

Table: customers

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
...	...	...



cust_id	name	country
001	Ani	us

# Saving DataFrames

- § Data in DataFrames can be saved to a data source
- § Use DataFrame.write to create a DataFrameWriter
- § DataFrameWriter provides convenience functions to externally save the data represented by a DataFrame
  - jdbc inserts into a new or existing table in a database
  - json saves as a JSON file
  - parquet saves as a Parquet file
  - orc saves as an ORC file
  - text saves as a text file (string data in a single column only)
  - saveAsTable** saves as a Hive/Impala table (**HiveContext** only)

Language: Python/Scala

```
peopleDF.write.saveAsTable("people")
```

# Options for Saving DataFrames

## § DataFrameWriter option methods

- format specifies a data source type
- mode determines the behavior if file or table already exists:  
overwrite, append, ignore or error (default is error)
- partitionBy stores data in partitioned directories in the form  
*column=value* (as with Hive/Impala partitioning)
- options specifies properties for the target data source
- save is the generic base function to write the data

Language: Python/Scala

```
peopleDF.write.
 format("parquet") .
 mode ("append") .
 partitionBy ("age") .
 saveAsTable ("people")
```

# DataFrames and RDDs

## § DataFrames are built on RDDs

- Base RDDs contain **Row** objects
- Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null, Alice, 94304]
Row[30, Brayden, 94304]
Row[19, Carla, 10036]
Row[46, Diana, null]
Row[null, Étienne, 94104]

# DataFrames and RDDs

§ Row RDDs have all the standard Spark actions and transformations

–Actions: `collect`, `take`, `count`, and so on

–Transformations: `map`, `flatMap`, `filter`, and so on

§ Row RDDs can be transformed into pair RDDs to use map-reduce methods

§ DataFrames also provide convenience methods (such as `map`, `flatMap`, and `foreach`) for converting to RDDs

# Working with Row Objects

- Use **Array**-like syntax to return values with type **Any**
- **row(n)** returns element in the *n*th column
- **row.fieldIndex("age")** returns index of the **age** column
- Use methods to get correctly typed values
- **row.getAs[Long]("age")**
- Use type-specific **get** methods to return typed values
- **row.getString(n)** returns *n*th column as a string
- **row.getInt(n)** returns *n*th column as an integer
- And so on

# Example: Extracting Data from Row Objects

## Extract data from Row objects

Language: Python

```
peopleRDD = peopleDF \
 .map(lambda row: (row.pcode, row.name))
peopleByPCode = peopleRDD \
 .groupByKey()
```

Row[null, Alice, 94304]
Row[30, Brayden, 94304]
Row[19, Carla, 10036]
Row[46, Diana, null]
Row[null, Étienne, 94104]

Language: Scala

```
val peopleRDD = peopleDF.
 map(row =>
 (row(row.fieldIndex("pcode")),
 row(row.fieldIndex("name"))))
val peopleByPCode = peopleRDD.
 groupByKey()
```

(94304, Alice)
(94304, Brayden)
(10036, Carla)
(null, Diana)
(94104, Étienne)

(null, [Diana])
(94304, [Alice, Brayden])
(10036, [Carla])
(94104, [Étienne])

# Converting RDDs to DataFrames

§ You can also create a DF from an RDD using `createDataFrame`

```
import org.apache.spark.sql.types._
import org.apache.spark.sql.Row
val schema = StructType(Array(
 StructField("age", IntegerType, true),
 StructField("name", StringType, true),
 StructField("PCODE", StringType, true)))
val rowrdd = sc.parallelize(Array(Row(40, "Abram", "01601"),
 Row(16, "Lucia", "87501")))
val mydf = sqlContext.createDataFrame(rowrdd, schema)
```

Language: Scala

# RDD Operations

## Transformations

map()  
flatMap()  
filter()  
union()  
intersection()  
distinct()  
groupByKey()  
reduceByKey()  
sortByKey()  
join()  
  
...

## Actions

count()  
collect()  
first(), top(n)  
take(n), takeOrdered(n)  
countByValue()  
reduce()  
foreach()  
  
...

# Lambda Expression

PySpark WordCount example:

```
input_file = sc.textFile("/path/to/text/file")
map = input_file.flatMap(lambda line: line.split(" ")) \
 .map(lambda word: (word, 1))
counts = map.reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("/path/to/output/")
```

lambda arguments: expression

# PySpark RDD API

<https://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD>

`map(f, preservesPartitioning=False)`

[\[source\]](#)

Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

`flatMap(f, preservesPartitioning=False)`

[\[source\]](#)

Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

# Practice with flight data (1)

Data: **airports.dat** (<https://openflights.org/data.html>)

[*Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source*]

**Try to do somethings:**

- Create RDD from textfile
- Count the number of airports
- Filter by country
- Group by country
- Count the number of airports in each country

# Practice with flight data (2)

- Data: **airports.dat** (<https://openflights.org/data.html>)  
*[Airport ID, Name, City, Country, IATA, ICAO, Latitude, Longitude, Altitude, Timezone, DST, Tz database, Type, Source]*
- Data: **routes.dat**  
*[Airline, Airline ID, Source airport, Source airport ID, Destination airport, Destination airport ID, Codeshare, Stops, Equipment]*

Try to do somethings:

- Join 2 RDD
- Count the number of flights arriving in each country

# Creating a DataFrame(1)

```
%pyspark
from pyspark.sql import *

Employee = Row("firstName", "lastName", "email", "salary")

employee1 = Employee('Basher', 'armbrust', 'bash@edureka.co', 100000)
employee2 = Employee('Daniel', 'meng', 'daniel@stanford.edu', 120000)
employee3 = Employee('Muriel', None, 'muriel@waterloo.edu', 140000)
employee4 = Employee('Rachel', 'wendell', 'rach_3@edureka.co', 160000)
employee5 = Employee('Zach', 'galifianakis', 'zach_g@edureka.co', 160000)

employees = [employee1,employee2,employee3,employee4,employee5]

print(Employee[0])

print(employees)

dframe = spark.createDataFrame(employees)
dframe.show()
```

# Creating a DataFrame

## From CSV file:

```
%pyspark
flightData2015 = spark\
 .read\
 .option("inferSchema", "true")\
 .option("header", "true")\
 .csv("/usr/zeppelin/module9/2015-summary.csv")

flightData2015.show()
```

## From RDD:

```
%pyspark
from pyspark.sql import *
list = [('Ankit',25),('Jalfaizy',22),('saurabh',20),('Bala',26)]
rdd = sc.parallelize(list)
people = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
df = spark.createDataFrame(people)

df.show()
```

# DataFrame APIs

- **DataFrame**: show(), collect(), createOrReplaceTempView(), distinct(), filter(), select(), count(), groupBy(), join()...
- **Column**: like()
- **Row**: row.key, row[key]
- **GroupedData**: count(), max(), min(), sum(), ...

<https://spark.apache.org/docs/latest/api/python/pyspark.sql.html>

# Spark SQL

- Create a temporary view
- Query using SQL syntax

```
%pyspark
flightData2015.createOrReplaceTempView("flight_data_2015")

maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")

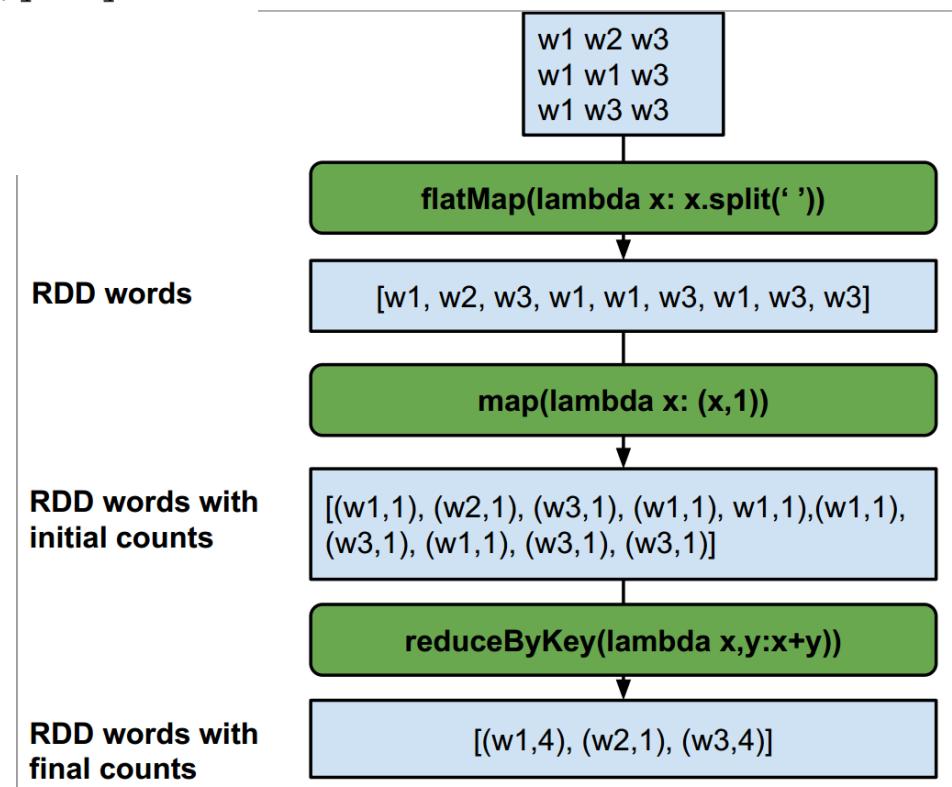
maxSql.show()
```

# **Spark running mode**

- Local
- Clustered
  - Spark Standalone
  - Spark on Apache Mesos
  - Spark on Hadoop YARN

# Hello World: Word-Count

```
1 import sys
2 from pyspark import SparkContext
3 sc = SparkContext(appName="WordCountExample")
4 lines = sc.textFile(sys.argv[1])
5 counts = lines.flatMap(lambda x: x.split(' ')) \
6 .map(lambda x: (x, 1)) \
7 .reduceByKey(lambda x,y:x+y)
8 output = counts.collect()
9 for (word, count) in output:
10 print "%s: %i" % (word, count)
11 sc.stop()
```



# Run using command line

- Turn on docker bash
- spark-submit wordcount.py README.md
- Result will be shown as follows

```
19/05/19 07:56:51 INFO scheduler.TaskSchedulerImpl: Removed
pool
19/05/19 07:56:51 INFO scheduler.DAGScheduler: ResultStage 1
0 finished in 0.150 s
19/05/19 07:56:51 INFO scheduler.DAGScheduler: Job 0 finishe
0, took 2.050066 s
Turks: 1
States,294: 1
Algeria,United: 1
States,2025: 1
States,955: 1
States,Czech: 1
Colombia,United: 1
States,588: 1
States,Dominican: 1
```

# **Lab: Word-Count**

- Lab on the Zeppelin notebook
- Github source code
  - <https://github.com/bk-blockchain/big-data-class>

## **Flight data:**

- Analyzing flight data from the United States Bureau of Transportation statistics
- Lab on the Zeppelin notebook
- Github source code
  - <https://github.com/bk-blockchain/big-data-class>

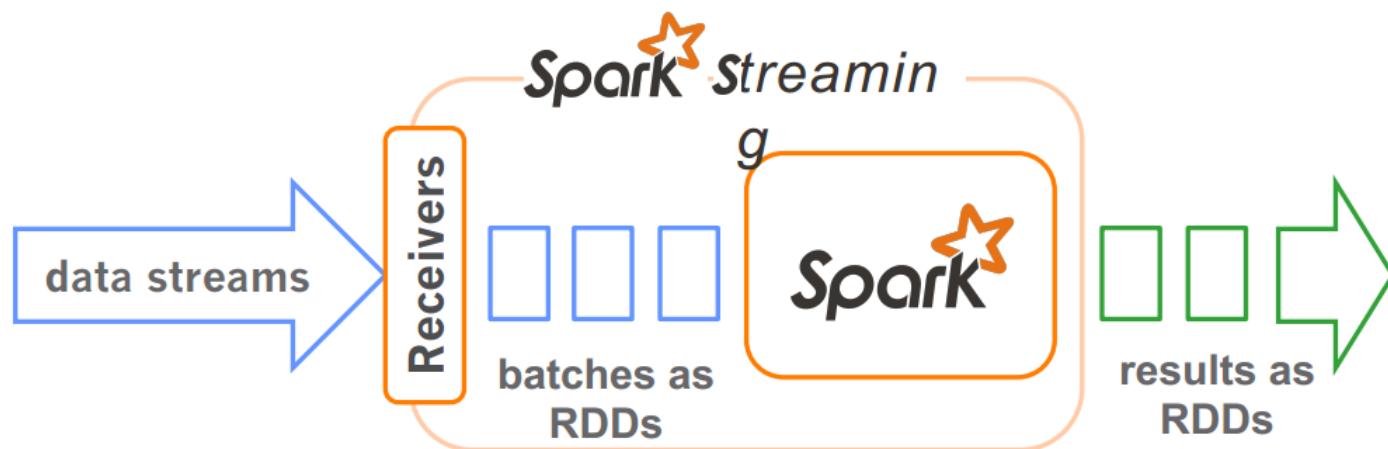
# Spark Streaming

- Scalable, fault-tolerant stream processing system
- Receive data streams from input sources, process them in a cluster, push out to databases/dashboards



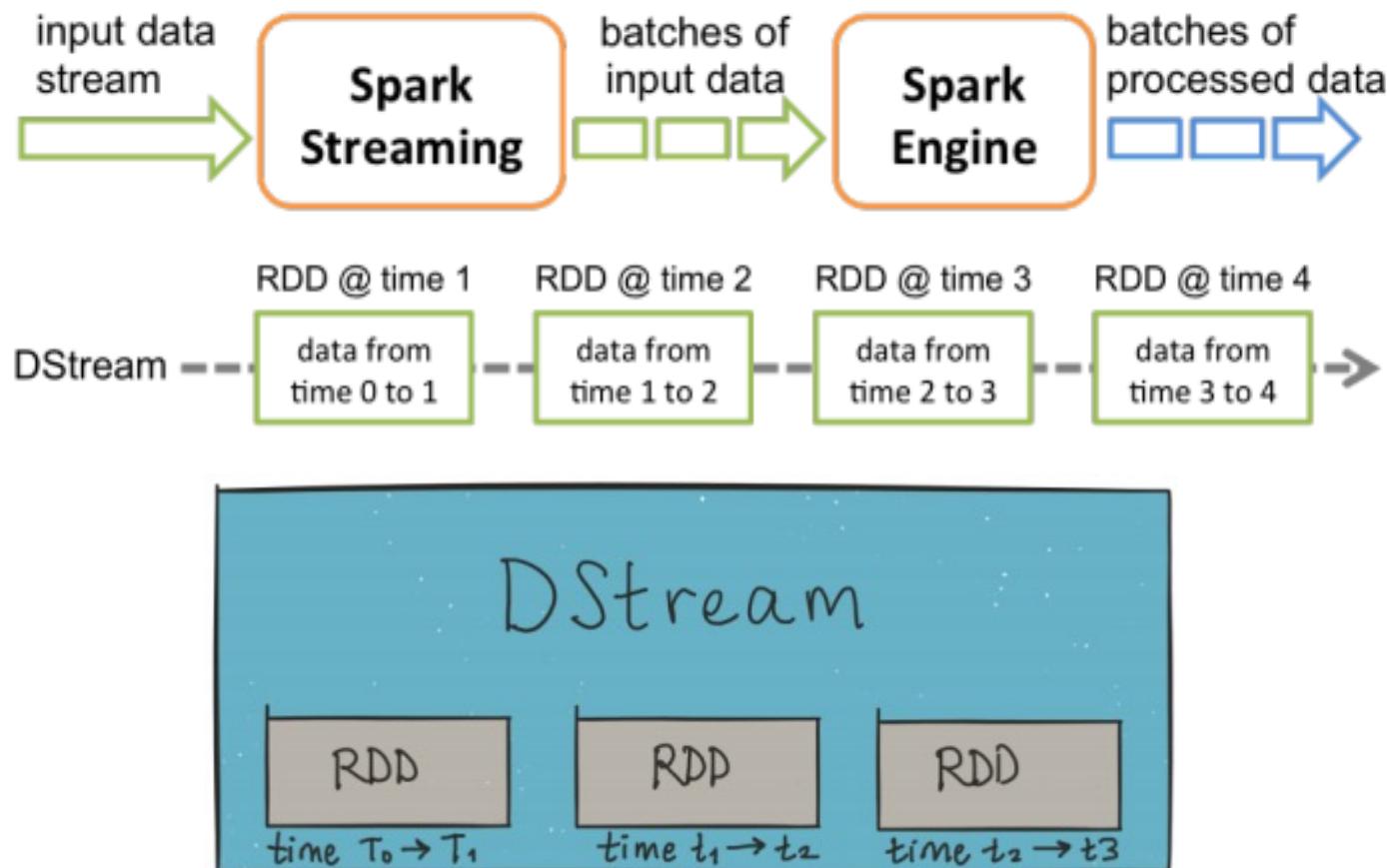
# How does it work?

- The stream is treated as a **series** of very **small**, **deterministic batches** of data
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Processed results are pushed out in batches



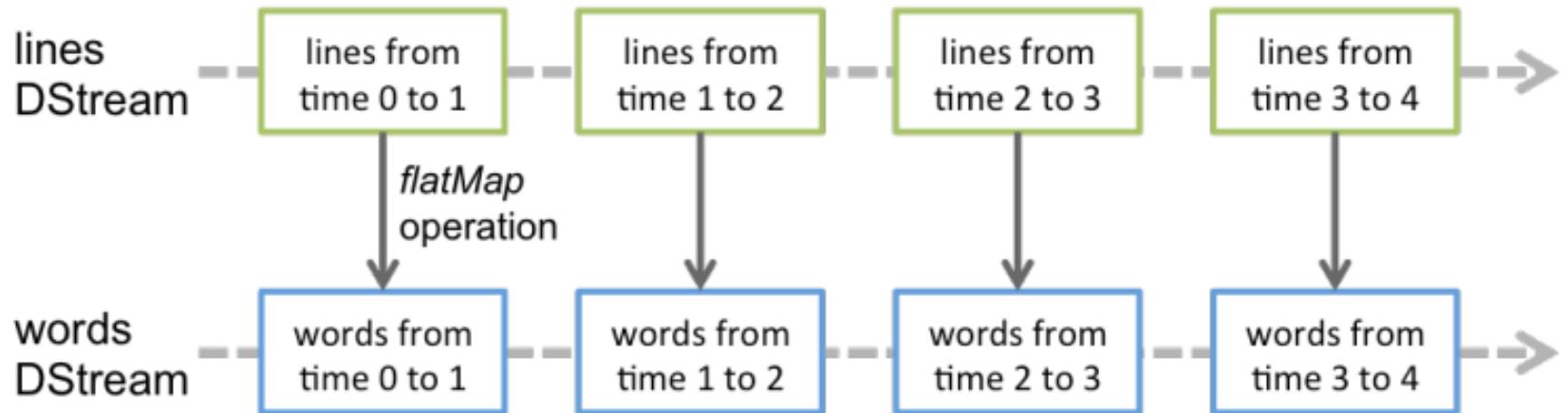
# Discretized Stream (DStream)

- Sequence of RDDs representing a stream of data



# Discretized Stream (DStream)

- Any operation applied on a DStream translates to operations on the underlying RDDs



# StreamingContext

- The **main entry** point of all Spark Streaming functionality

```
val conf = new
SparkConf().setAppName(appName).setMaster(master)
val ssc = new StreamingContext(conf, batchinterval)
```

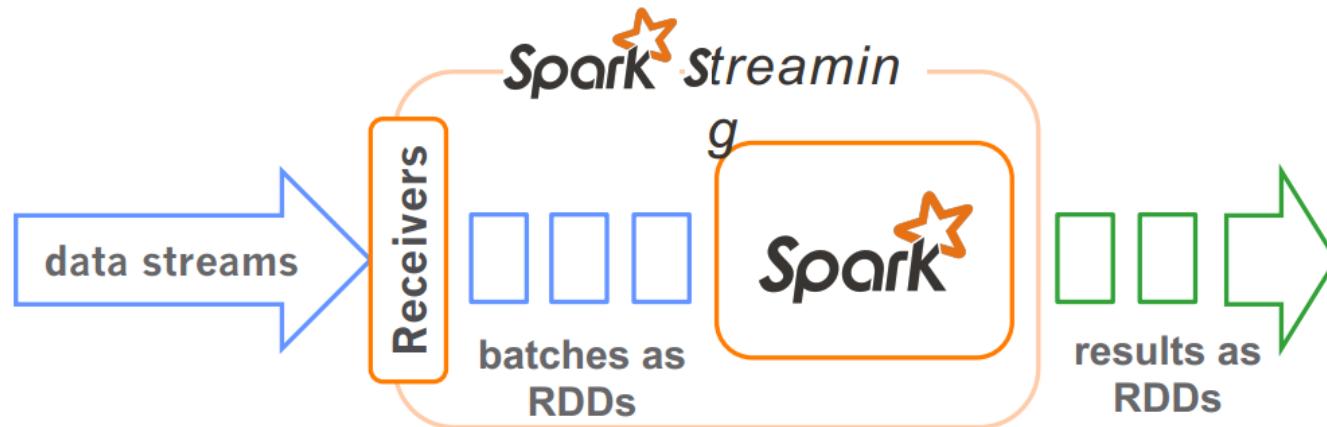
- **appname**: name of the application
- **master**: a Spark, Mesos, or YARN cluster URL
- **batchinterval**: time interval (in second) of each batch

# **Operation on DStreams**

- Three categories
  - Input operation
  - Transformation operation
  - Output operation

# Input Operations

- Every **input DStream** is associated with a **Receiver** object
- Two built-in categories of streaming sources:
  - Basic sources, e.g., file systems, socket connection
  - Advanced sources, e.g., Twitter, Kafka



# Input Operations

- Basic sources
  - Socket connection

```
// Create a DStream that will connect to hostname:port
ssc.socketTextStream("localhost", 999)
```

- File stream

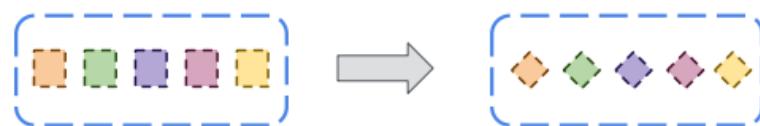
```
streamingContext.fileStream[...] (dataDirectory)
```

- Advanced sources

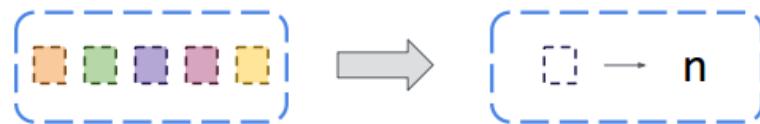
```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

# Transformation

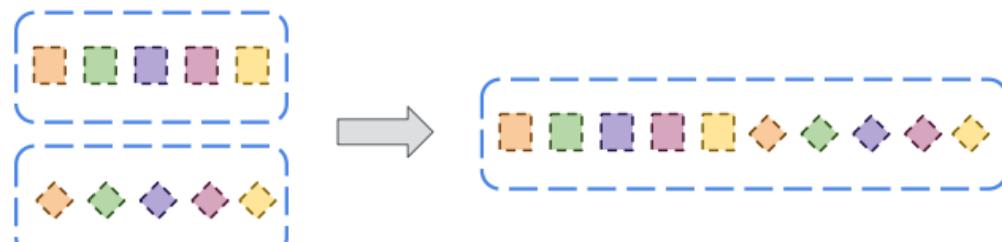
map,  
flatmap,  
filter



count,  
reduce,  
countByValue,  
reduceByKey



union,  
join  
cogroup



# Transformation

Transformation	Meaning
map (func)	Return a new DStream by passing each element of the source DStream through a function func
flatmap(func)	Similar to map, but each input item can be mapped to 0 or more output items
filter(func)	Return a new DStream by selecting only the records of the source DStream on which func returns true

# Transformation

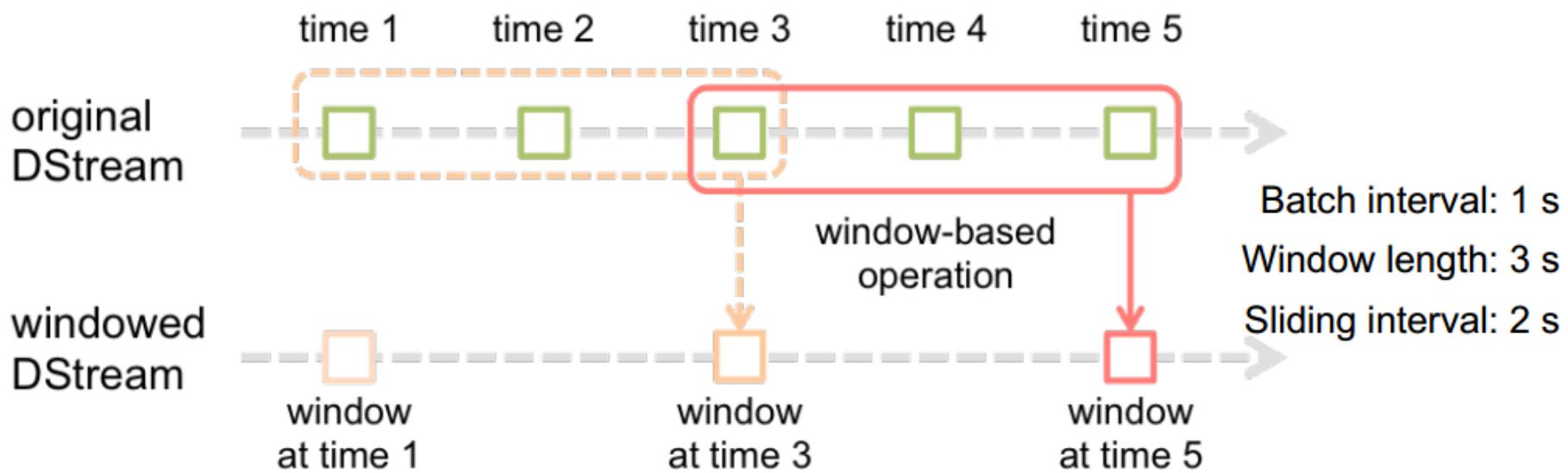
Transformation	Meaning
count	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream
countbyValue	Returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each RDD of the source DStream.
reduce(func)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function func (which takes two arguments and returns one).
reduceByKey(func)	When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function

# Transformation

Transformation	Meaning
union(otherStream)	Return a new DStream that contains the union of the elements in the source DStream and otherDStream.
join(otherStream)	When called on two DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key.

# Window Operations

- Spark provides a set of transformations that apply to a sliding window of data
- A window is defined by: **window length** and **sliding interval**



# Window Operations

- `window(windowLength, slideInterval)`
  - Returns a new DStream which is computed based on windowed batches
- `countByWindow(windowLength, slideInterval)`
  - Returns a sliding window count of elements in the stream.
- `reduceByWindow(func, windowLength, slideInterval)`
  - Returns a new single-element DStream, created by aggregating elements in the stream over a sliding interval using func.

# Output Operation

- Push out DStream's data to external systems, e.g., a database or a file system

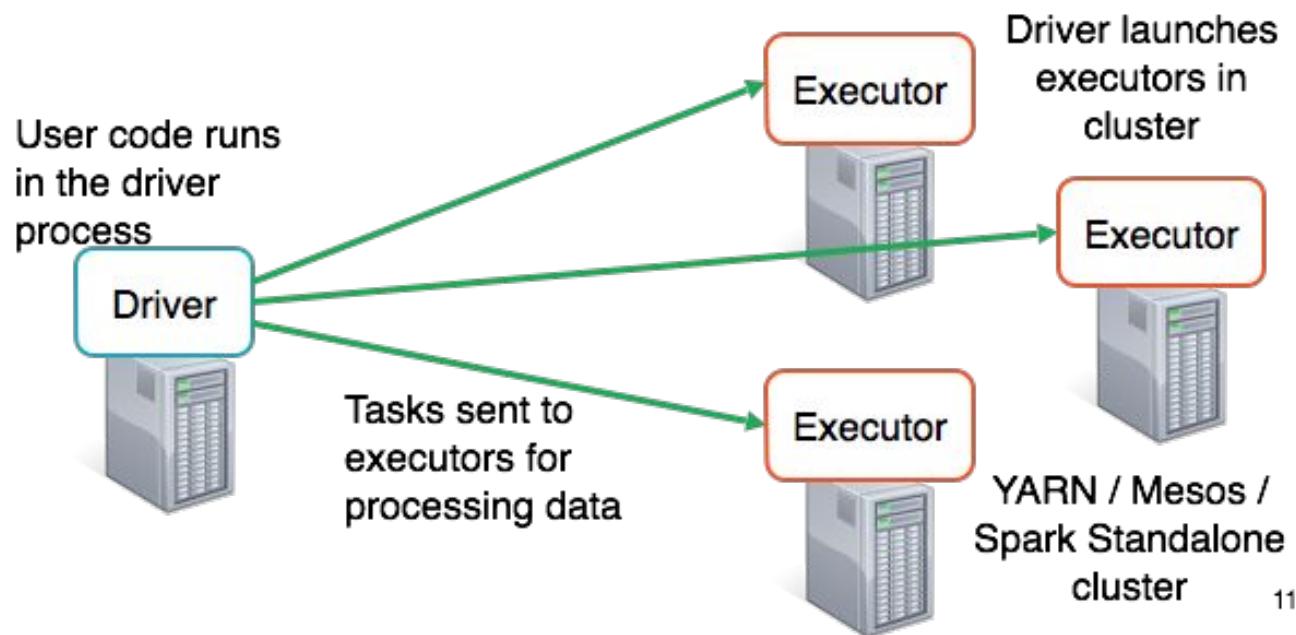
Operation	Meaning
print	Prints the first ten elements of every batch of data in a DStream on the driver node running the application
saveAsTextFiles	Save this DStream's contents as text files
saveAsHadoopFiles	Save this DStream's contents as Hadoop files.
foreachRDD(func)	Applies a function, func, to each RDD generated from the stream

# Example

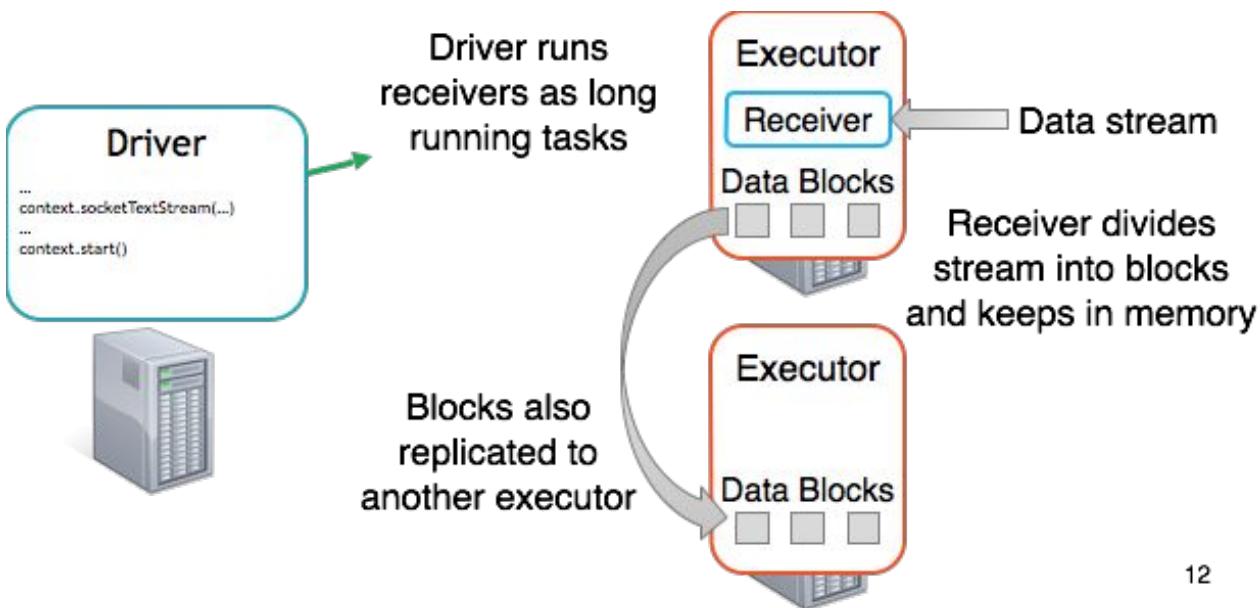
## Word Count

```
val context = new StreamingContext(conf, Seconds(1))
val lines = context.socketTextStream(...)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_+_)
wordCounts.print() ↗ Print the DStream contents on screen
context.start() ↗ Start the streaming job
```

# Execution in any Spark Application

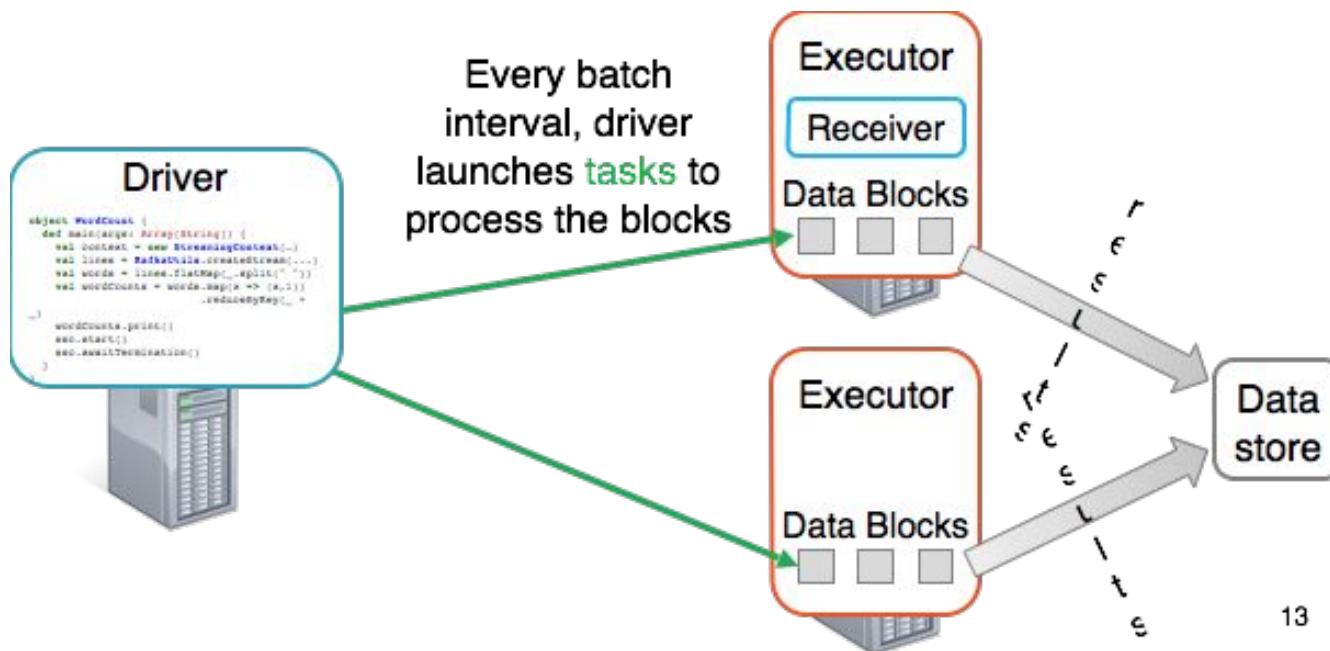


# Execution in Spark Streaming: Receiving data

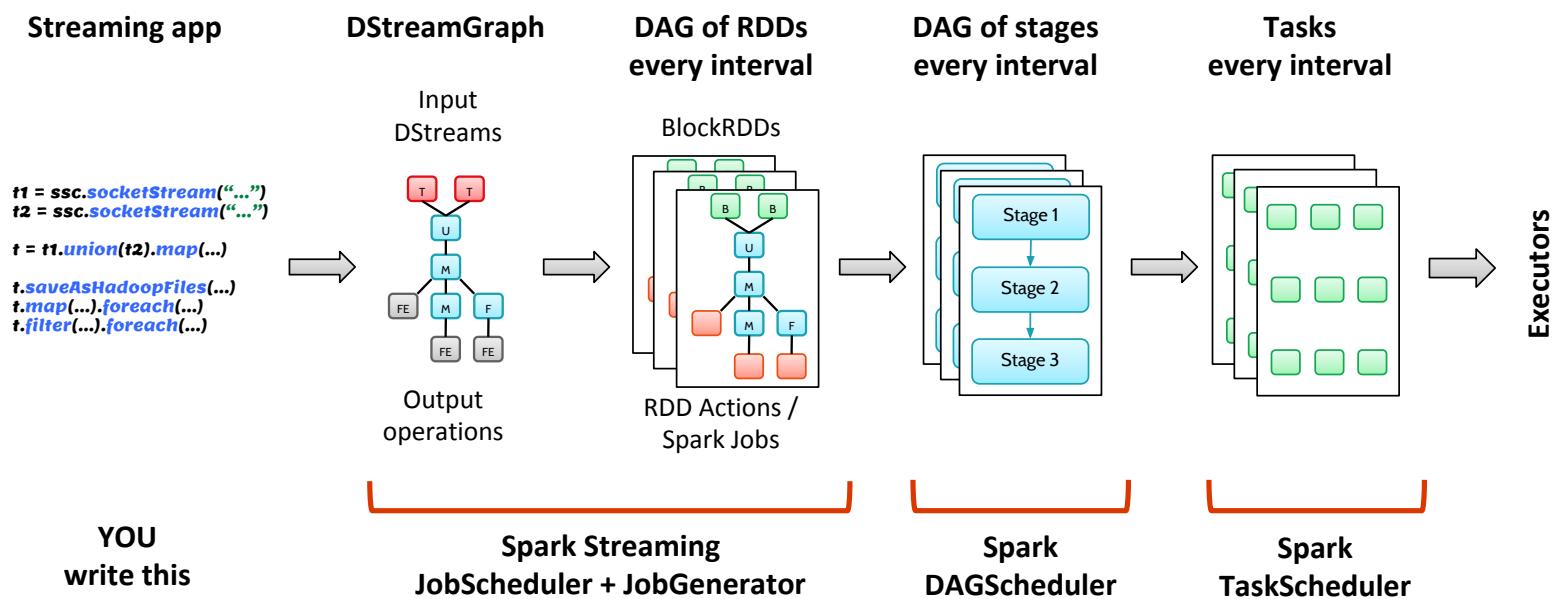


12

# Execution in Spark Streaming: Processing data



# End-to-end view



# Dynamic Load Balancing

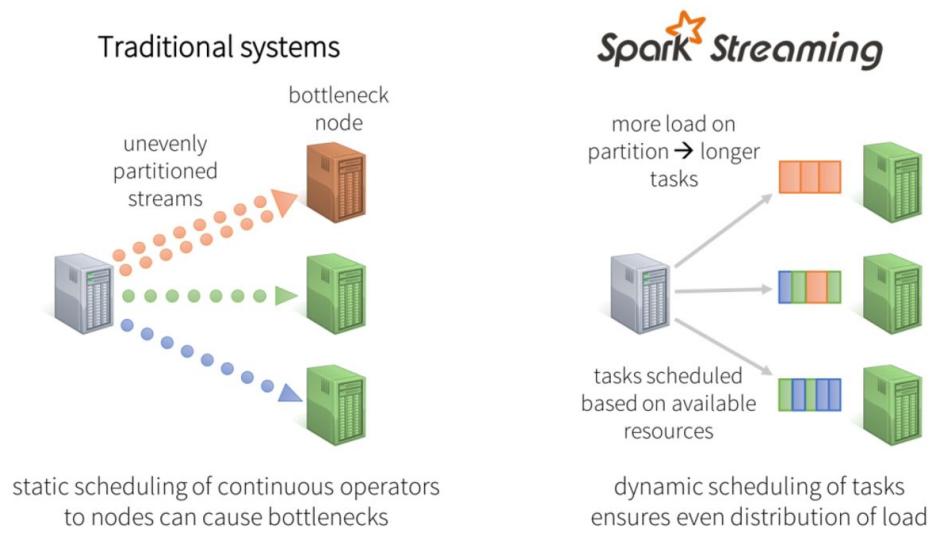


Figure 3: Dynamic load balancing

# Fast failure and Straggler recovery

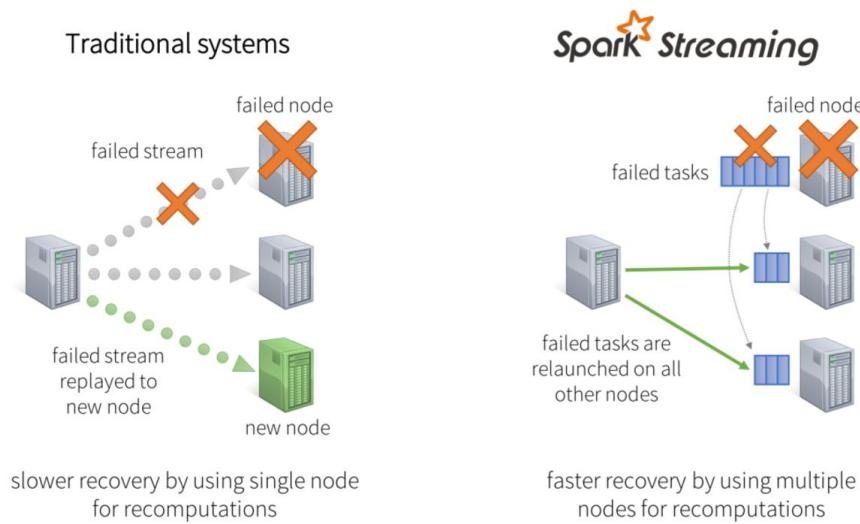


Figure 4: Faster failure recovery with redistribution of computation

## Acknowledgement and References

### Books:

- Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia. Learning Spark. Oreilly
- James A. Scott. Getting started with Apache Spark. MapR Technologies

### Slides:

- Amir H. Payberah. Scalable Stream Processing – Spark Streaming and Flink
- Matteo Nardelli. Spark Streaming: Hands on Session
- DataBricks. Spark Streaming
- DataBricks: Spark Streaming: Best Practices

# Machine learning

## ARTIFICIAL INTELLIGENCE

IS NOT NEW

### ARTIFICIAL INTELLIGENCE

Any technique which enables computers to mimic human behavior



1950's

1960's

1970's

1980's

1990's

2000's

2010s

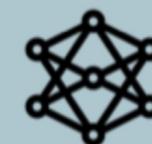
### MACHINE LEARNING

AI techniques that give computers the ability to learn without being explicitly programmed to do so



### DEEP LEARNING

A subset of ML which make the computation of multi-layer neural networks feasible



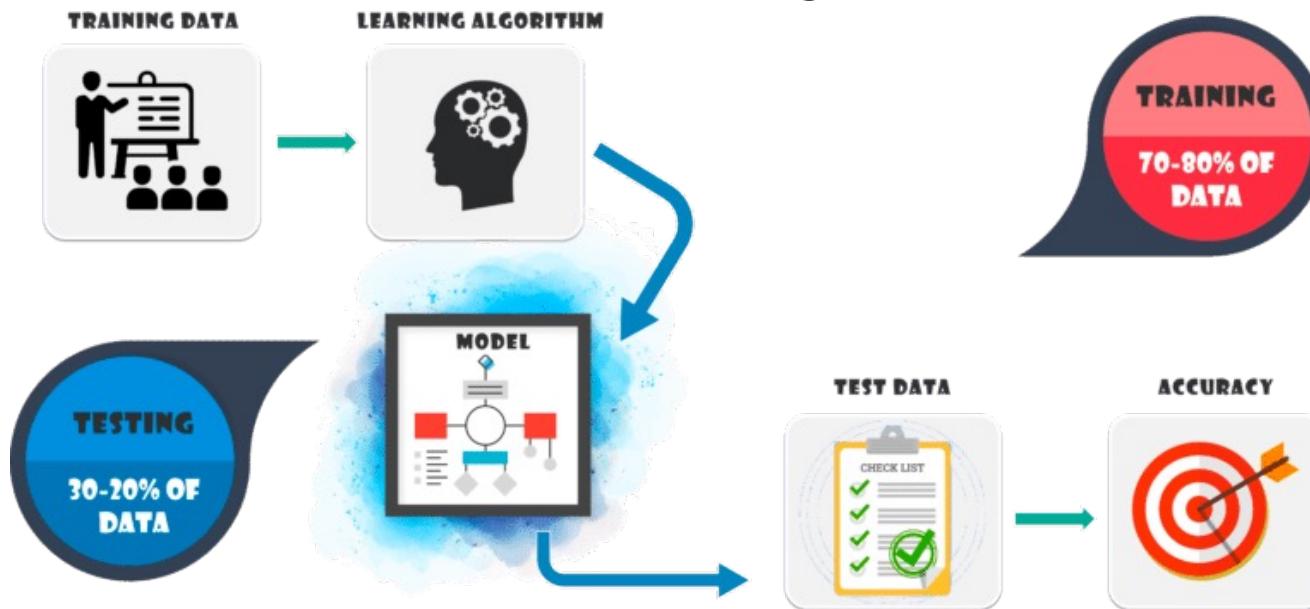
ORACLE®

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. |

From [1]

# Machine Learning Lifecycle

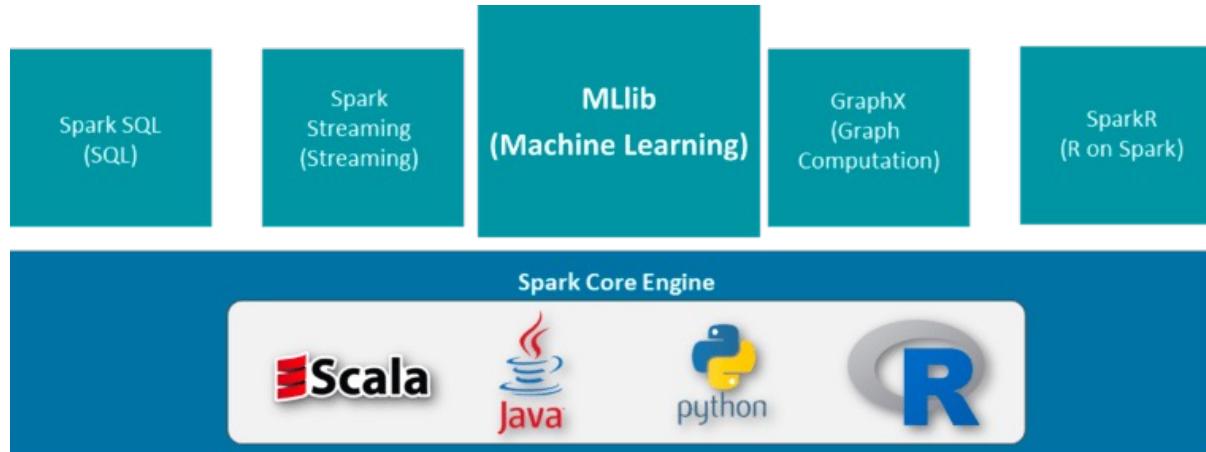
- Two major phases
  - **Training Set**
    - You have the complete training dataset
    - You can extract features and train to fit a model.
  - **Testing Set**
    - Once the model is obtained, you can predict using the model obtained on the training set



From [2]

# Spark ML and PySpark

- Spark ML is a machine-learning library
  - Classification: logistic regression, naive Bayes
  - Regression: generalized linear regression, survival regression
  - Decision trees, random forests, and gradient-boosted trees
  - Recommendation: alternating least squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs)
  - Topic modeling: latent Dirichlet allocation (LDA)
  - Frequent item sets, association rules, and sequential pattern mining
- PySpark is an interface for using Python



# Binary Classification Example [3]

- **Binary Classification** is the task of predicting a binary label
  - Is an email spam or not spam?
  - Should I show this ad to this user or not?
  - Will it rain tomorrow or not?
- The Adult dataset
  - <https://archive.ics.uci.edu/ml/datasets/Adult>
  - 48842 individuals and their annual income
  - We will use this information to predict if an individual earns **<=50K or >50k** a year

# Dataset Information

- Attribute Information:
  - age: continuous
  - workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
  - fnlwgt: continuous
  - education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc...
  - education-num: continuous
  - marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent...
  - occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners...
  - relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried
  - race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
  - sex: Female, Male
  - capital-gain: continuous
  - capital-loss: continuous
  - hours-per-week: continuous
  - native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany...
- Target/Label: - <=50K, >50K

# Analyzing Flow

- Load data
- Preprocess Data
- Fit and Evaluate Models
  - Logistic Regression
  - Decision Trees
  - Random Forest
- Make Classification

# **Lab: Running Binary Classification on Zeppelin**

- Get the prepared notebook
- Run and try to understand algorithms

# Spark ML

- Spark ML is a machine-learning library
  - Classification: logistic regression, naive Bayes
  - Regression: generalized linear regression, survival regression
  - Decision trees, random forests, and gradient-boosted trees
  - Recommendation: alternating least squares (ALS)
  - Clustering: K-means, Gaussian mixtures (GMMs)
  - Topic modeling: latent Dirichlet allocation (LDA)
  - Frequent item sets, association rules, and sequential pattern mining

# **Classification vs Prediction**

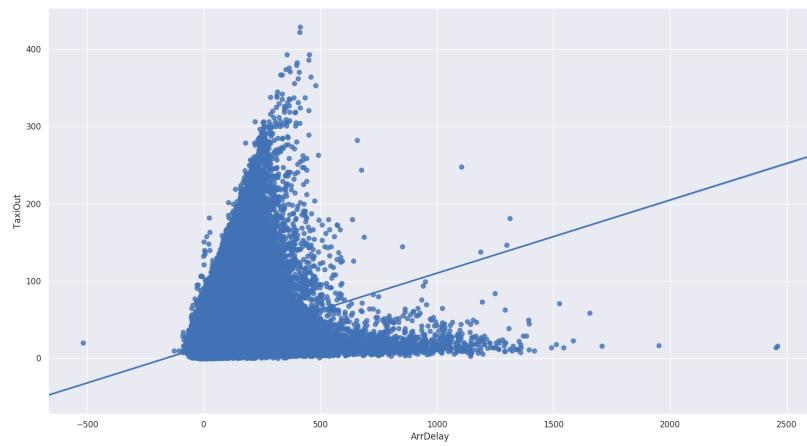
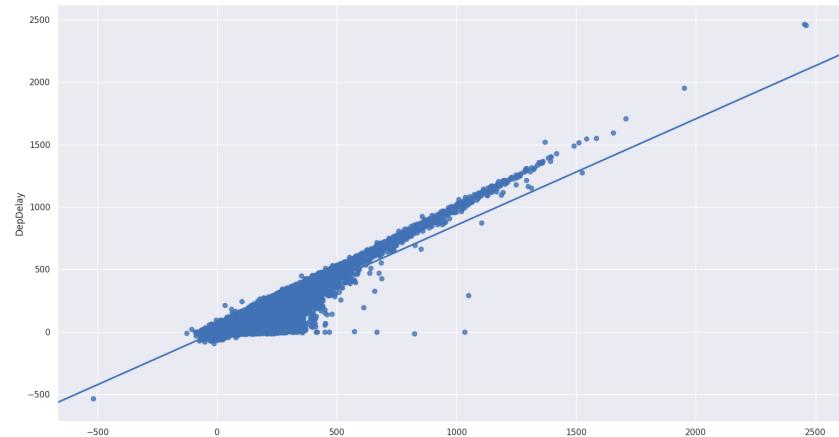
- Classification models predict categorical class labels [2]
  - Binary classification
- Prediction models predict continuous valued functions
  - Regression analysis is a statistical methodology that is most often used for numeric prediction

# Predicting the arrival delay of commercial flights [1]

- Problem
  - We want to be able to predict, based on historical data
    - The arrival delay of a flight using only information available before the flight takes off
- Dataset
  - <http://stat-computing.org/dataexpo/2009/the-data.html>
  - The data used was published by the US Department of Transportation
  - It compromises almost 23 years worth of data
- Approach
  - Using a regression algorithm

# Dataset Information

	Name	Description
1	Year	1987-2008
2	Month	1-12
3	DayofMonth	1-31
4	DayOfWeek	1 (Monday) - 7 (Sunday)
5	DepTime	actual departure time (local, hhmm)
6	CRSDepTime	scheduled departure time (local, hhmm)
7	ArrTime	actual arrival time (local, hhmm)
8	CRSArrTime	scheduled arrival time (local, hhmm)
9	UniqueCarrier	<a href="#">unique carrier code</a>
10	FlightNum	flight number
11	TailNum	plane tail number
12	ActualElapsedTime	in minutes
13	CRSElapsedTime	in minutes
14	AirTime	in minutes
15	ArrDelay	arrival delay, in minutes
16	DepDelay	departure delay, in minutes



# Analyzing Flow

- Load data
- Preprocess Data
- Train the data and obtain a model
- Evaluate the resulting model
- Make Predictions

# **Lab: Running Prediction of Flight Delay on Zeppelin**

- Write code using PySpark
  - Get the prepared notebook
- Run and try to understand algorithms
- The original source code (in Scala)
  - <https://github.com/pedroduartecosta/Spark-PredictFlightDelay>

# References

- [1] <https://medium.com/@pedrodc/building-a-big-data-machine-learning-spark-application-for-flight-delay-prediction-4f9507cdb010>
- [2] [https://www.tutorialspoint.com/data\\_mining/dm\\_classification\\_prediction.htm](https://www.tutorialspoint.com/data_mining/dm_classification_prediction.htm)

# GraphX

- Apache Spark's API for graphs and graph-parallel computation
- GraphX unifies ETL (Extract, Transform & Load) process
- Exploratory analysis and iterative graph computation within a single system

# Use cases

- Facebook's friends, LinkedIn's connections
- Internet's routers
- Relationships between galaxies and stars in astrophysics and Google's Maps
- Disaster detection, banking, stock market

# RDD on GraphX

- GraphX extends the Spark RDD with a Resilient Distributed Property Graph
- The property graph is a directed multigraph which can have multiple edges in parallel
- The parallel edges allow multiple relationships between the same vertices

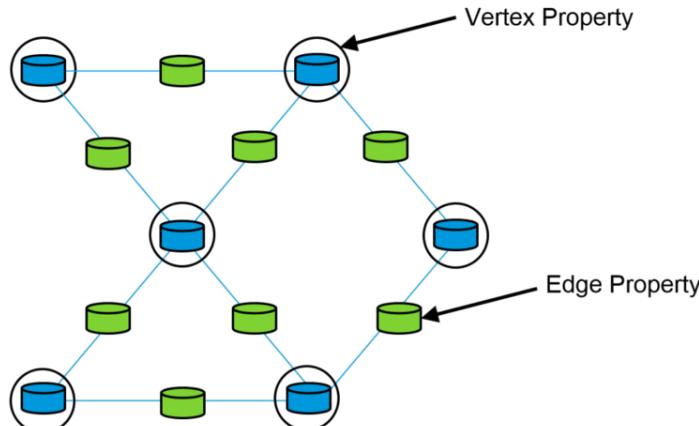


Figure: Property Graph

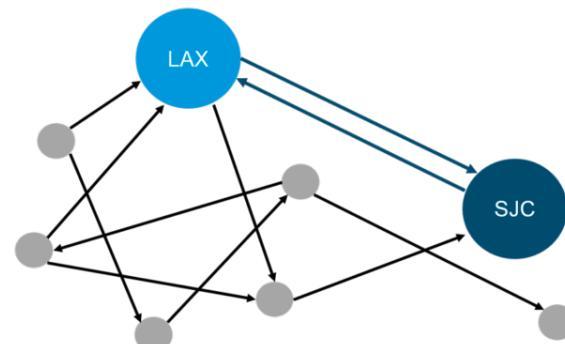


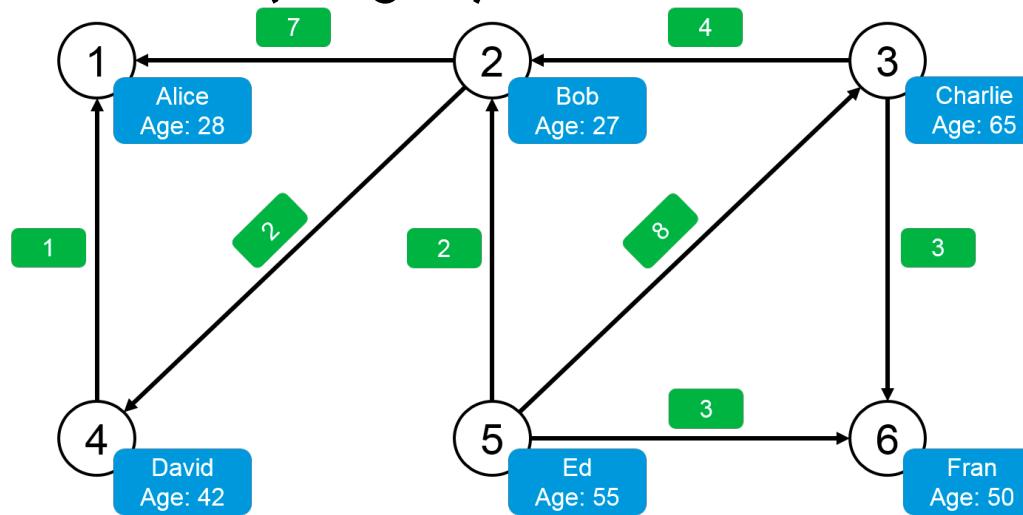
Figure: An example of property graph

# Spark GraphX Features

- **Flexibility**
  - Spark GraphX works with both graphs and computations
  - GraphX unifies ETL (Extract, Transform & Load), exploratory analysis and iterative graph computation
- **Speed**
  - The fastest specialized graph processing systems
- **Growing Algorithm Library**
  - Page rank, connected components, label propagation, SVD++, strongly connected components and triangle count

# GraphX with Examples

- The graph here represents the Twitter users and whom they follow on Twitter. For e.g. Bob follows Davide and Alice on Twitter
- Looking at the graph, we can extract information about the people (vertices) and the relations between them (edges)



# Source code

```
1 //Importing the necessary classes
2 import org.apache.spark._
3 import org.apache.spark.rdd.RDD
4 import org.apache.spark.util.IntParam
5 import org.apache.spark.graphx._
6 import org.apache.spark.graphx.util.GraphGenerators
```

**Displaying Vertices:** Further, we will now display all the names and ages of the users (vertices).

```
1 val vertexRDD: RDD[(Long, (String, Int))] = sc.parallelize(vertexArray)
2 val edgeRDD: RDD[Edge[Int]] = sc.parallelize(edgeArray)
3 val graph: Graph[(String, Int), Int] = Graph(vertexRDD, edgeRDD)
4 graph.vertices.filter { case (id, (name, age)) => age > 30 }
5 .collect.foreach { case (id, (name, age)) => println(s"$name is $age") }
```

The output for the above code is as below:

David is 42

Fran is 50

Ed is 55

Charlie is 65

# More source code

**Displaying Edges:** Let us look at which person likes whom on Twitter.

```
1 | for (triplet <- graph.triplets.collect)
2 |
3 | println(s"${triplet.srcAttr._1} likes ${triplet.dstAttr._1}")
4 | }
```

The output for the above code is as below:

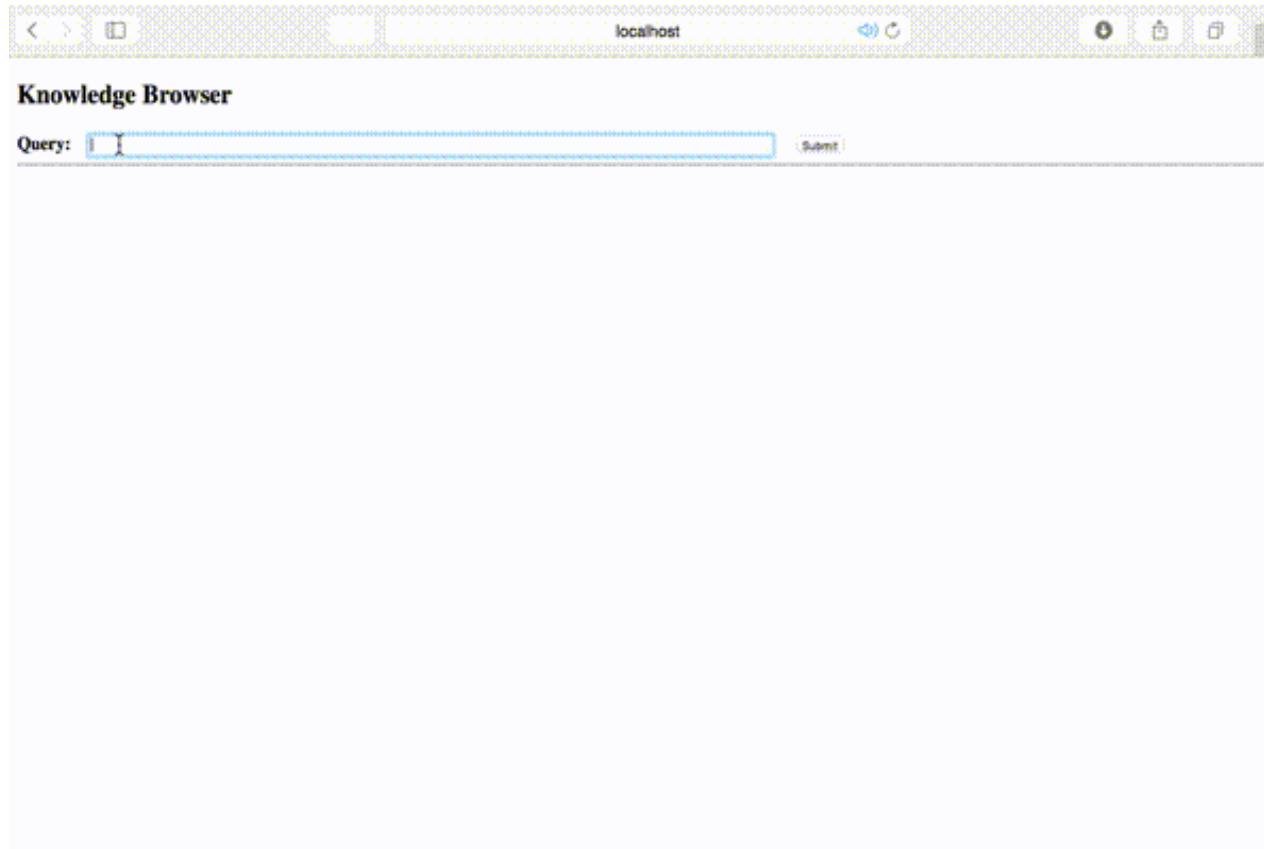
```
Bob likes Alice
Bob likes David
Charlie likes Bob
Charlie likes Fran
David likes Alice
Ed likes Bob
Ed likes Charlie
Ed likes Fran
```

# Other example in PySpark

```
2 ## pyspark --packages graphframes:graphframes:0.6.0-spark2.2-s_2.11
3 from graphframes import *
4 from pyspark import *
5 from pyspark.sql import *
6 spark = SparkSession.builder.appName('fun').getOrCreate()
7 vertices = spark.createDataFrame([('1', 'Carter', 'Derrick', 50),
8 ('2', 'May', 'Derrick', 26),
9 ('3', 'Mills', 'Jeff', 80),
10 ('4', 'Hood', 'Robert', 65),
11 ('5', 'Banks', 'Mike', 93),
12 ('98', 'Berg', 'Tim', 28),
13 ('99', 'Page', 'Allan', 16)],
14 ['id', 'name', 'firstname', 'age'])
15 edges = spark.createDataFrame([('1', '2', 'friend'),
16 ('2', '1', 'friend'),
17 ('3', '1', 'friend'),
18 ('1', '3', 'friend'),
19 ('2', '3', 'follows'),
20 ('3', '4', 'friend'),
21 ('4', '3', 'friend'),
22 ('5', '3', 'friend'),
23 ('3', '5', 'friend'),
24 ('4', '5', 'follows'),
25 ('98', '99', 'friend'),
26 ('99', '98', 'friend')],
27 ['src', 'dst', 'type'])
28 g = GraphFrame(vertices, edges)
29 ## Take a look at the DataFrames
30 g.vertices.show()
31 g.edges.show()
32 ## Check the number of edges of each vertex
33 g.degrees.show()
```

# Spark Knowledge Graph

- Example: <https://github.com/spoddutur/graph-knowledge-browser>



## Acknowledgement and References

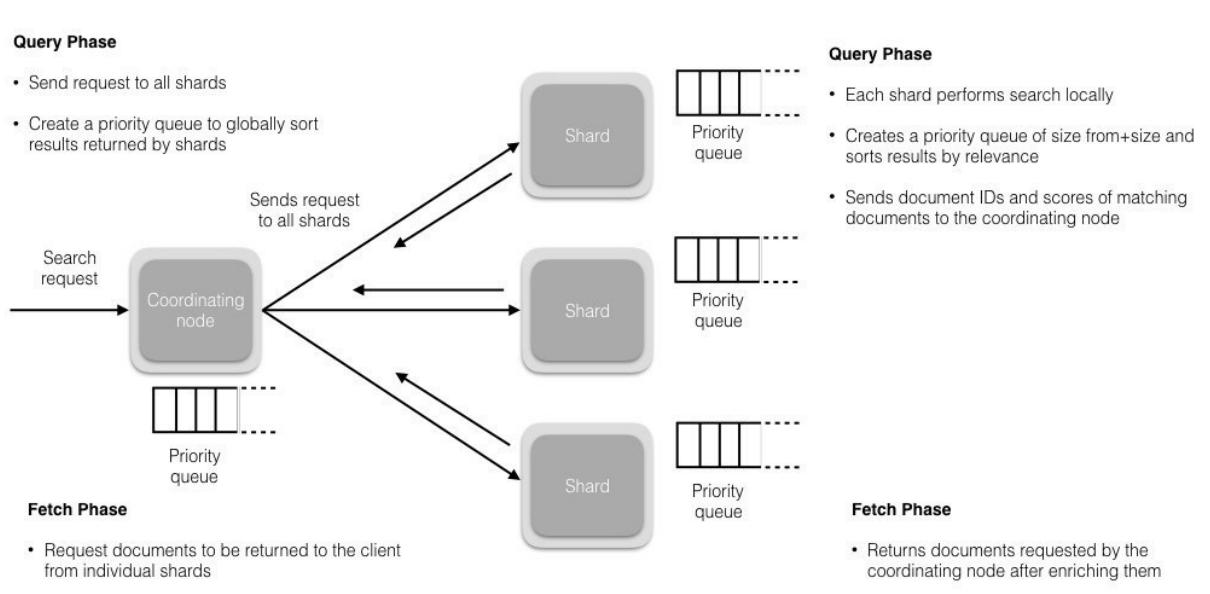
Books:

Slides:

- <https://www.edureka.co/blog/spark-graphx/>

# Elasticsearch

- Full-text search engine
- Based on the Lucene library
- HTTP web interface and schema-free JSON documents



# Lab: Elasticsearch and Kibana

- Set up Elasticsearch to store data
- Write/Read data to Elasticsearch
- Install and run Kibana

# Installation

- Install Docker and login
  - <https://docs.docker.com/docker-for-windows/install/>
  - <https://docs.docker.com/docker-for-mac/install/>
- Login to @chung-pi gitlab to pull images
  - docker login registry.gitlab.com -u bi-class -p bqp\_cSsCJ2kaNjMu1U4A
- Pull images
  - docker pull registry.gitlab.com/chung-pi/bi-docker/elasticsearch
  - docker pull registry.gitlab.com/chung-pi/bi-docker/kibana:latest
- If Internet is not available
  - docker load --input elasticsearch.tar
  - docker load --input kibana.tar

# Start Elasticsearch and Kibana

- Clone bi-class git project
  - <https://gitlab.com/chung-pi/bi-class>
- Start containers using docker-compose
  - docker-compose up -d --build elasticsearch
  - docker-compose up -d --build kibana

- Elasticsearch
  - <http://localhost:9200>
- Kibana
  - <http://localhost:5601>

# Load data to Elasticsearch

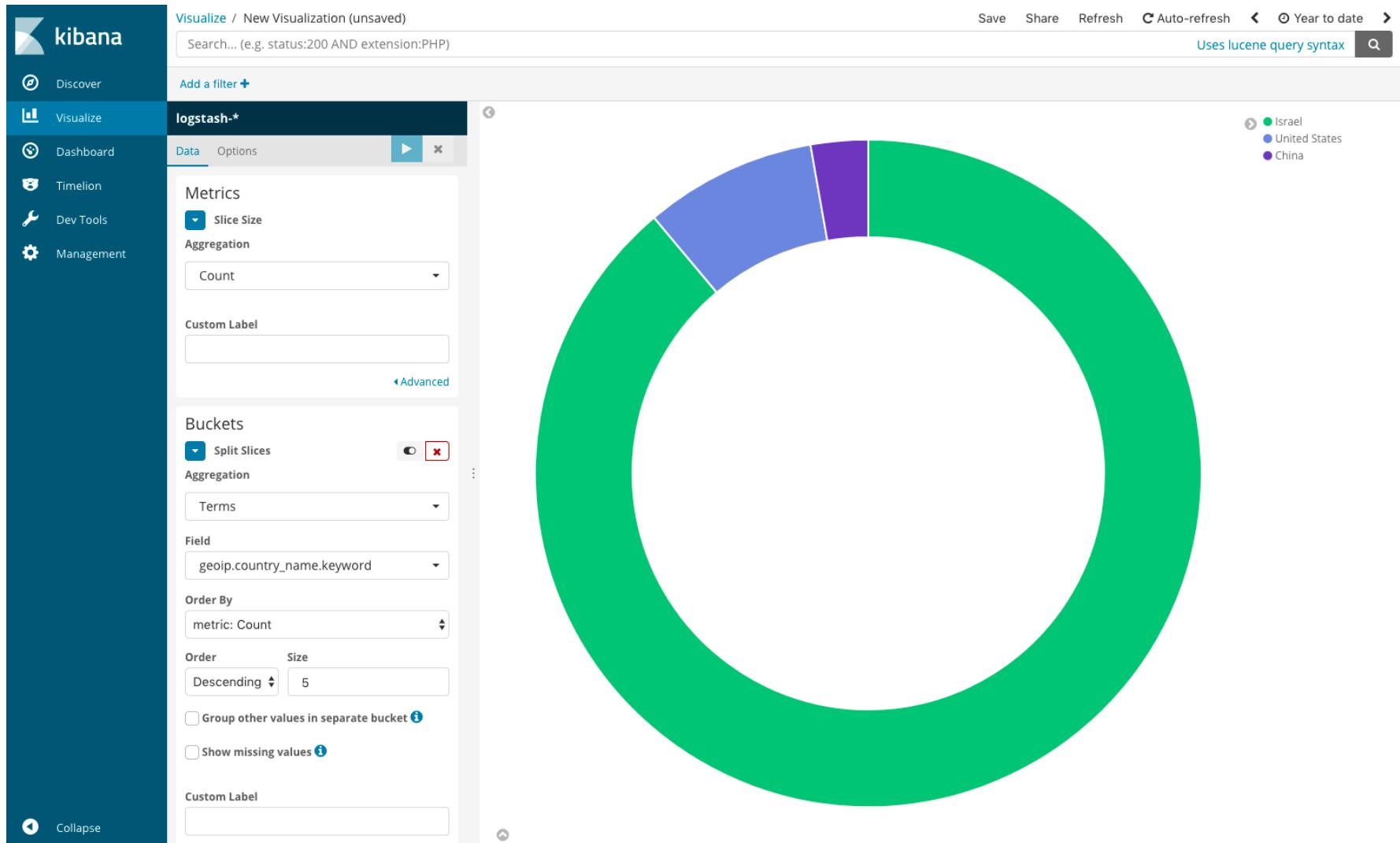
- Using CURL
  - curl -O  
<https://download.elastic.co/demos/kibana/gettingstarted/7.x/accounts.zip>
  - unzip accounts.zip
  - curl -H 'Content-Type: application/x-ndjson' -XPOST 'localhost:9200/bank/account/\_bulk?pretty' --data-binary @accounts.json
- Checking data using Kibana
  - Open Kibana on browser

# Understanding Kibana aggregations

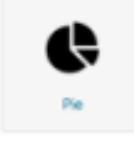
- There are two types of aggregations
  - **Bucket** aggregations groups documents together in one bucket according to your logic and requirements
  - **Metric** aggregations are used to calculate a value for each bucket based on the documents inside the

Metric aggregations	Bucket aggregations
<ul style="list-style-type: none"><li>• Count</li><li>• Sum</li><li>• Average</li><li>• Media</li><li>• Min</li><li>• Max</li><li>• Unique Count</li><li>• Standard Deviation</li><li>• Percentiles</li><li>• Percentile Ranks</li></ul>	<ul style="list-style-type: none"><li>• Date Histogram</li><li>• Date Range</li><li>• Filters</li><li>• Histogram</li><li>• IPv4 Range</li><li>• Range</li><li>• Terms</li><li>• Significant Terms</li><li>• Geohash</li></ul>

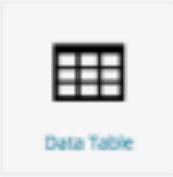
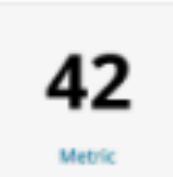
# Example



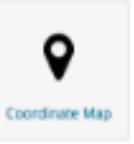
# Basic Charts

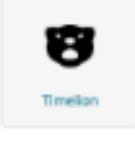
 Area	For visualizing time series data and for splitting lines on fields	Users over time
 Heat Map	For showing statistical outliers and are often used for latency values	Latency and outliers
 Horizontal Bar	Good for showing relationships between two fields	URL and referrer
 Line	are a simple way to show time series and are good for splitting lines to show anomalies	Average CPU over time by host
 Pie	Useful for displaying parts of a whole	Top 5 memory consuming system procs
 Vertical Bar	Great for time series data and for splitting lines across fields	URLs over time

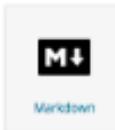
# Data

 Data Table	Best way to split across multiple fields in a custom way	Top user, host, pod, container by usage
 Gauge	A way to show the status of a specific metric using thresholds you define	Memory consumption limits
 Goal	Similar to a Gauge, useful for monitoring a specific metric defined as a goal	No. of errors per service
 Metric	Useful visualization for displaying a calculation as a single number	No. of Docker containers run.

# Map, Time series, and Others

		Help add a geographical dimension to IP-based logs	Geographic origin of web server requests.
-----------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	----------------------------------------------------	-------------------------------------------

		Allows you to create more advanced queries based on time series data	Percentage of 500 errors over time
-----------------------------------------------------------------------------------	-----------------------------------------------------------------------------------	----------------------------------------------------------------------	------------------------------------

	<b>Experimental</b> - Allows you to create selectors or sliders for alternating between options.	Switch between
	A great way to add a customized text or image based visualization to your dashboard based on markdown syntax	Company logo or a description of a dashboard
	Helps display groups of words sized by their importance	Countries sending requests to a web server
	<b>Experimental</b> - allows you to add custom visualizations based on Vega and VegaLite	-