# Intro

## OLTP & OLAP

OLTP: Online Transaction Processing (DBMSs)

hierarchical & relational database

support millions of small queries. Good for DB transactions (create, update, or retrieve data)

OLAP: Online Analytical Processing (Data Warehousing)

for analysis of current data

Good for complex but fewer queries.

Ex.

SELECT Quarter, Region, SUM(Sales\_revenue)

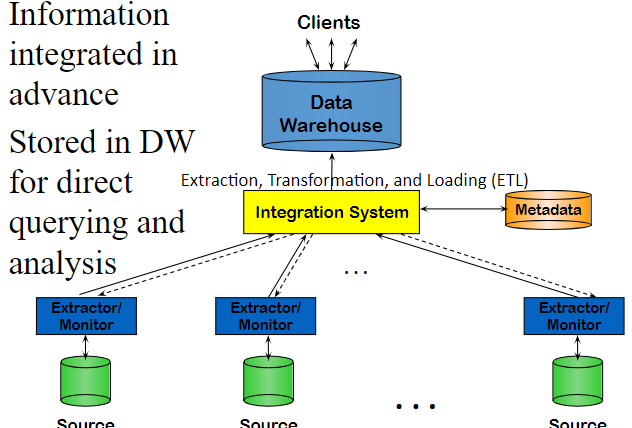
FROM Results

GROUP BY CUBE (Quarter, Region)

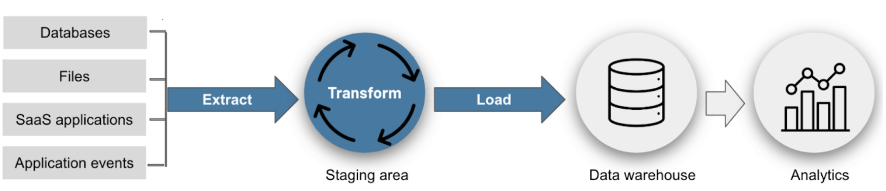
⇒ 2 systems

each store has their own small database

data warehouse for direct querying



ETL = Extraction + Transformation + Loading (preprocessing data)



transform (ex. translate relational data to JSON format)

| Terms | OLTP | OLAP |
| --- | --- | --- |
| User  Function  DB Design  Data  View  Usage  Unit of work  Access  Operations  # Records accessed  #Users  Db size  Metric | Clerk, IT Professional  Day to day operations  Application-oriented (E-R based)  Current, Isolated  Detailed, Flat relational  Structured, Repetitive  Short, Simple transaction  Read/write (INSERT, UPDATE,DELETE, SELECT)  Index/hash on primary key  Tens  Thousands  GB-TB  Trans. throughput | Knowledge worker  Decision support  Subject-oriented (Star, snowflake)  Historical, Consolidated  Summarized, Multidimensional  Ad hoc  Complex query  Read Mostly (SELECT)  Lots of Scans  Millions  Hundreds  >TB  Query throughput, response |
| Limitation | Not scale to millions of users/operations per sec  Rigid schema | Predefined schema  Few operations supported like drilling, rollup  Slow to adapt to new business needs  Don’t support streaming data (or support with delay) |
| Example | MySQL  Oracle  IBM DB2  SQL Server  Amazon Aurora  NoSQL:  MongoDB  Hbase  Cassandra  Couchbase  AsterixDB (UCI/UCR project) | Teradata  Amazon Redshift  IBM  Oracle  SAS  Tableau  SAP  Vertica  Snowflake |

## Big Data Mgm Sys

|  | NoSQL (address some OLTP limitations) | Distributed Job Execution Platforms (address some OLAP limitations) |
| --- | --- | --- |
| Limitations | fast fine grained operations | expensive to start the job |
| Examples | MongoDB  ElasticSearch for text…  good for query document   * given keyword, find matching doc | Hadoop/MapReduce  Spark  Good for index documents   * given doc, build inverted index |

RTAP: Real-Time Analytics Processing (Big Data Architecture & technology)

to improve business response

## Data Cleaning

Missing data, unit mismatch, **entity resolution** (different representation of the same thing), erroneous data (outlier)

Why data cleaning?

Distortion – some samples are corrupted by a process

Selection Bias - likelihood of a sample depends on its value

Left and right censorship - users come and go from our scrutiny

Dependence – samples are supposed to be independent, but are not (e.g. social networks)

Cloud Computing options

self-managed (one premise)

EC2 instances

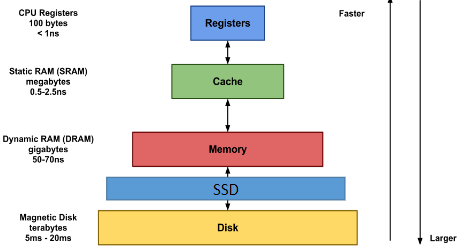
fully-managed (AWS) more costly

Shared Nothing Cluster

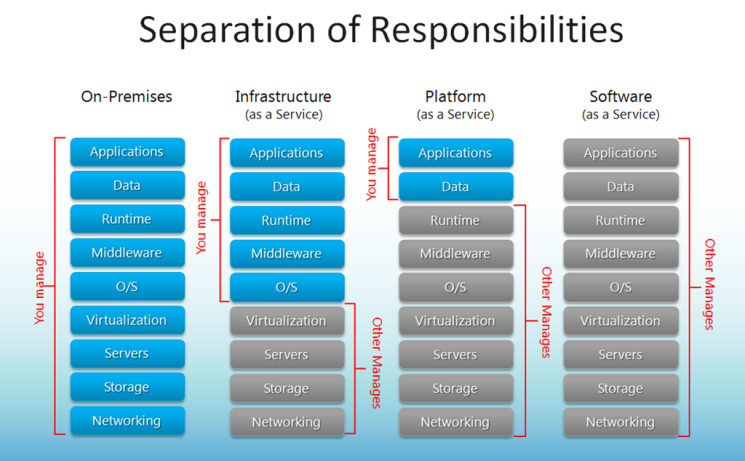
DB resources

increase available memory by using a cluster of computers connected through fast network

Non-volatile RAM (NVRAM)



IaaS vs. PaaS vs. SaaS



IaaS (no hardware, most affordable)

AWS EC2, Cisco Metapod, Microsoft Azure, Google Compute Engine (GCE)

PaaS (no hardware + low level code)

AWS Elastic Beanstalk, AWS Aurora, Windows Azure, Heroku, Force.com, Google App Engine, Apache Stratos

SaaS (application ready to use)

Google Apps (e.g., Gmail), Salesforce, Workday, Concur, Citrix GoToMeeting, Cisco WebEx, Microsoft Office 365

Ex. both can vertical scaling

| MySQL on EC2 (IaaS) | Aurora MySQL (SaaS) |
| --- | --- |
| cheaper  customizable | easier to scale horizontally  easier to backup - restore |

Ex. AWS Aurora DB Cluster (1 primary write + up to 15 replicas)

if many writes happen often ⇒ bottleneck

usually higher primary > replicas (be mindful that the smallest machine might be the primary, which can slow down the performance)

# Relational Databases

Database System Concepts

Relational database: a set of relations = tables

Row = tuple =record

Field = attribute = column

Relation: made up of 2 parts:

Instance: a table, with rows and columns.

#rows = cardinality, #attributes = degree / arity.

Schema: specifies name of relation, plus name and type of each column.

E.G. Students(sid: string, name: string, login: string, age: integer, gpa: real).

Can think of a relation as a set of rows or tuples (i.e., all rows are distinct).

Key

Foreign key must be a unique/primary key of some table

## Database Management Systems

ACID transaction

atomicity = happen or not happen at all

consistency = consistent states always

isolation = lock the uncommitted/intermediate data (hidden from other trans)

dead lock ⇐ 2 phase locking (before r/w an object, a transaction request a lock on it and wait till given)

durability = when trans is complete, it’s persistent

Ex. transfer $100 from A to B

c → balance is always positive (for A and B)

i → concurrent execution of withdraw, deposit, transfers does not result in an incorrect balance of account.

d → After transfer terminates, the new balances reflect that $100 was transferred despite failures

Good practice of JDBC (Java database connection)

try{

connection.setAutoCommit(false);

…

connection.commit();

}catch(exception e){

connection.rollback();

}

too many transactions will slow system down → consider the tradeoffs

## Performance Metrics (Index)

index on file speeds up selections on the search key fields for the index

2 popular indexes

B+ tree index (better for disk storage) only leaf nodes have the data record

B tree’s internal nodes contains data or pointer

Because B+ trees don't have data associated with interior nodes, more keys can fit on a page of memory

What height is required for N records? [link](https://cs.stackexchange.com/questions/82015/maximum-depth-of-a-b-tree)

Good for range or equality queries

hash index

Good for equality but not range (h = n % m to insert into m buckets)

Ex. boats(bid, color, rating) with bid as PK

select \* boats where color = ‘red’

because we are selecting all records, a covering index ⇒ improve performance

Covering index provides all the data required for a query without having to access the actual table

* attributes in WHERE are candidates for index key (exact match → hash, range → tree)
* index-only strategies (with covering index) ⇒ clustering is not important in this case
* choose index that benefits as many queries as possible
* avoid index on columns of low selectivity (few unique values)
  + many records with the same value ⇒ spend more time jumping between files to find all the records
* avoid long string index (key will be bigger and need more disk space)

## Join Algorithms

Hash Join

both table are hashed based on the join key

Efficient for large tables, especially when one of the tables can fit into memory.

Nested Loop Join

each row from the first table is compared with every row from the second table

Efficient for small tables or when joining on non-indexed columns

Index Nested Loops

nested loop join but with index on inner table to speed up the matching

Efficient when there is an index on the join key in the inner table

Sort Merge

Both tables are sorted based on the join key, and then a merge operation is performed to identify matching rows.

Efficient for large tables when both tables are already sorted.

# NoSQL

Traditional RDBMSs can be either scaled:

Vertically (or Up) by hardware upgrades (SQL usually ⇐ ACID compliance)

(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) by adding more machines (NoSQL usually)

Requires database sharding and probably replication

Limited by the Read-to-Write ratio and communication overhead

Sharding data

Def = one large file are splitted into multiple chunks across multiple machines

to allow for concurrent/parallel accesses

to improve performance of complex query

Ways to shard data

partition-based (eg. MongoDB, Cassandra, HBase)

When data distribution is inherently clear (eg. region, time, categories)

Good for range or order queries ⇐ reduce time to scan needed data

Easy for management

hash-based

When Even Distribution of Data is preferred

Good for random access of query and loading balance

Good for avoiding hotspot

Replication data across servers (storing same data in diff servers)

Allow parallel access ⇒ improve performance

Parallel edit ⇒ slow queries

Avoiding single point of failures

Enhancing scalability and availability

Consistency is challenging

2PC (2-Phase Commit protocol) to ensure atomicity and consistency

ask if all machines are ready to update

lock the data if all are ready

unlock after making the change

* good for banking and security but bad for social media app
* usually use it on critical data/scenarios

CAP theorem (3 desired features of a distributed databases)

Consistency

(strict C + inefficient (eg. bank)) vs (loose C + efficient (eg. Twitter))

config of database (eg. isolation level in MySQL)

eventual consistency = all replicas will become consistent eventually

make sure DB has eventual consistency

RYOW (read-your-own-writes) protocol

Availability (quick access for social media posts)

Partition Tolerance

network between two servers are not working ⇒ choose C or A

NoSQL properties

no strict schema

no strict adherence to ACID (may not have transaction)

availability > consistency (use eventual consistency)

NoSQL Types

Document Stores - JSON, MongoDB (docs can be indexed to improve performance)

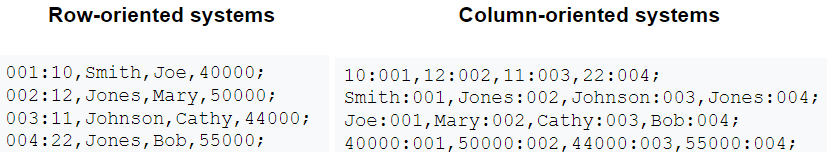
Graph DB - Neo4j, AWS Neptune

represented as vertices & edges

powerful for graph-like queries (shortest path)

Key-Value Stores - DynamoDB, Apache Cassandra, Google BigTable (big hash table)

Columnar - Amazon Redshift, Snowflake (good for analysis)



eg. avg(salary) without any index

row-oriented system ⇒ scan the whole file

column-oriented system ⇒ only scan salary column

## Cassandra (key-value)

Syntax

compaction strategy = how the data is compressed?

ALLOW FILTERING to filter on un-indexed columns

PK

primary key((taxi, origin), timestamp, trip)) → taxi, origin is partition value

CREATE TABLE test1(... primary key(taxi, timestamp) )

select \* from test1 where taxi > ‘4’

//not allowed because range queries might return large data

select \* from test1 where taxi = ‘4’

//allowed with/out timestamp equality

select \* from test1 where taxi = ‘4’ and origin = ‘x’

//not allowed unless creating an index on test1(origin)

Upsert

when inserting new record with duplicate PK → update the previous record

in MySQL, you can upsert by including ON DUPLICATE KEY UPDATE

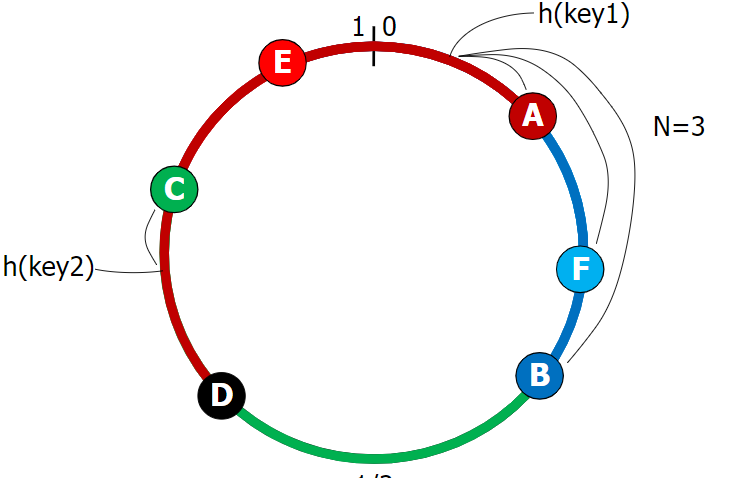
in NoSQL, update is cheaper than checking if the key exist

in SQL, checking is cheaper than updating on duplicate

| Cassandra | Traditional DB |
| --- | --- |
| Keyspace | Database |
| Column Family | Table |
| Flexible schema | Fixed schema |
| Column Family contains list of “nested key-value pairs”. (ROW x COLUMN key x COLUMN value) | Table contains array of arrays. (ROW x COLUMN) |

Partitioning/Sharding - Ring Topology

without replication = 3 → only A to E’s data belongs to A



Diff Replication Policies

Rack Unaware – replicate data at N-1 successive nodes after its coordinator

Rack Aware – avoid storing replicas in same rack

Datacenter Aware – avoid storing replicas in same data center

Cluster Membership

How nodes are added, deleted to the cluster

Default is ONE - Write

Must be written to the commit log and memtable of at least one replica node.

Default is ONE - Read

client submit query to Cassandra Cluster (CC) → choose closest replica to execute the query → return to CC → return to client

read and repair if digests diff in another replica (number of replicas ⇐ consistency level)

QUORUM - Write

Must be written to the commit log and memtable on a quorum of replica nodes across all datacenters.

QUORUM - Read

Returns the record after a quorum of replicas from all datacenters has responded.

## MongoDB

Basics

Document-based NoSQL. Data saved as BSON(binary JSON) key-value documents

\_id = primary key (auto-generated)

collection = table

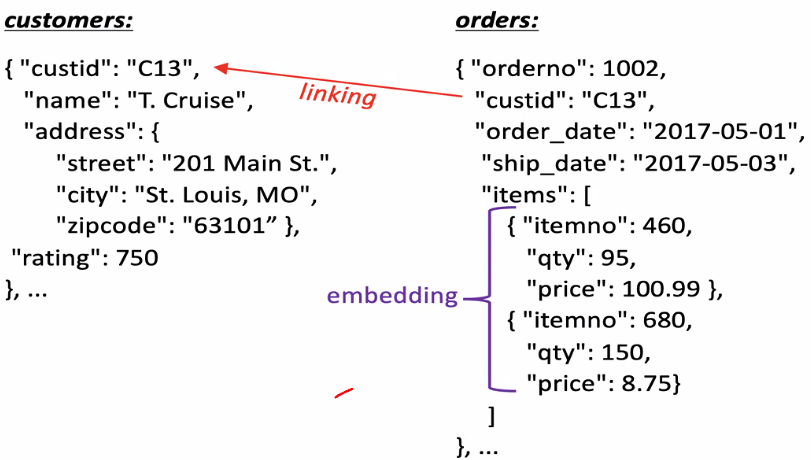
document = row

fields = attributes

linking = join

ObjectID(foreign\_id) to refer the id of another collection

we can di-normalize data by embedding all foreign data to current collection



Syntax

use db

show tables

db.createCollection

db.table.insertOne({...})

db.table.find({}, {\_id: 0, name:1})

among all data in table, exclude \_id and include name

db.table.find( {name : $in: [“John”, “Doe”, “Alice”]}

find the user whose name matches any of the values → John Alice is included but Joe Alice is not

db.table.find( {rating: {$gt: 5} } )

find sailor whose rating is greater than 5

db.table.updateOne( {name: ‘John’}, $set: {rating: 20} )

update rating to 20 if a sailor’s name is John

db.table.count({})

count all data (empty filtering query)

db.table.find({ $text: { $search: "job" } })

find documents where the text index includes the word "job."

db.table.createIndex({ “data.text”: "text" }) to create text index

db.table.createIndex({ name: 1 }, { name: "name\_idx" })

creates an index name\_idx on the "name" field

ascending order (1 represents ascending order)

db.table.aggregate( [

{ “$group” : {\_id: “$data.lang”, count : {$sum: 1}} }

] )

group by data.lang and count each group

db.orders.aggregate([

{ $match: { status: "A" } },

{ $group: {

\_id: "$cust\_id",

total: { $sum: "$amount" },

count: { $sum: 1 }

}

},

{ $sort: { total: -1 } }

])

find users with status of A, group by cust\_id, count each group and total of each group. Sort by total in descending order

SELECT sum(amount) as total, count(\*) as count

FROM orders WHERE status = ‘A’

GROUP BY cust\_id

ORDER BY total DESC

Midterm 1

SQL

ORDER BY … DESC

B+ tree is at least half full

split and push the key up when full after inserting

borrow and rotate key from parent when less than half-full

MongoDB

db.orders.aggregate([

{$match : {size:”medium”}},

{$group : {\_id:”name”, totalQuantity: {$sum : “$quantity”}}}

]) to get total order quantity of medium size pizzas

$project is the same as SELECT …

$match, $group, $skip -

$unwind: deconstructs the items

$sort: -1 desc

Ex.

db.collection.aggregate([

{ $skip: 5 }, // Skips the first 5 documents

{ $limit: 10 } // Returns up to 10 documents after skipping

])

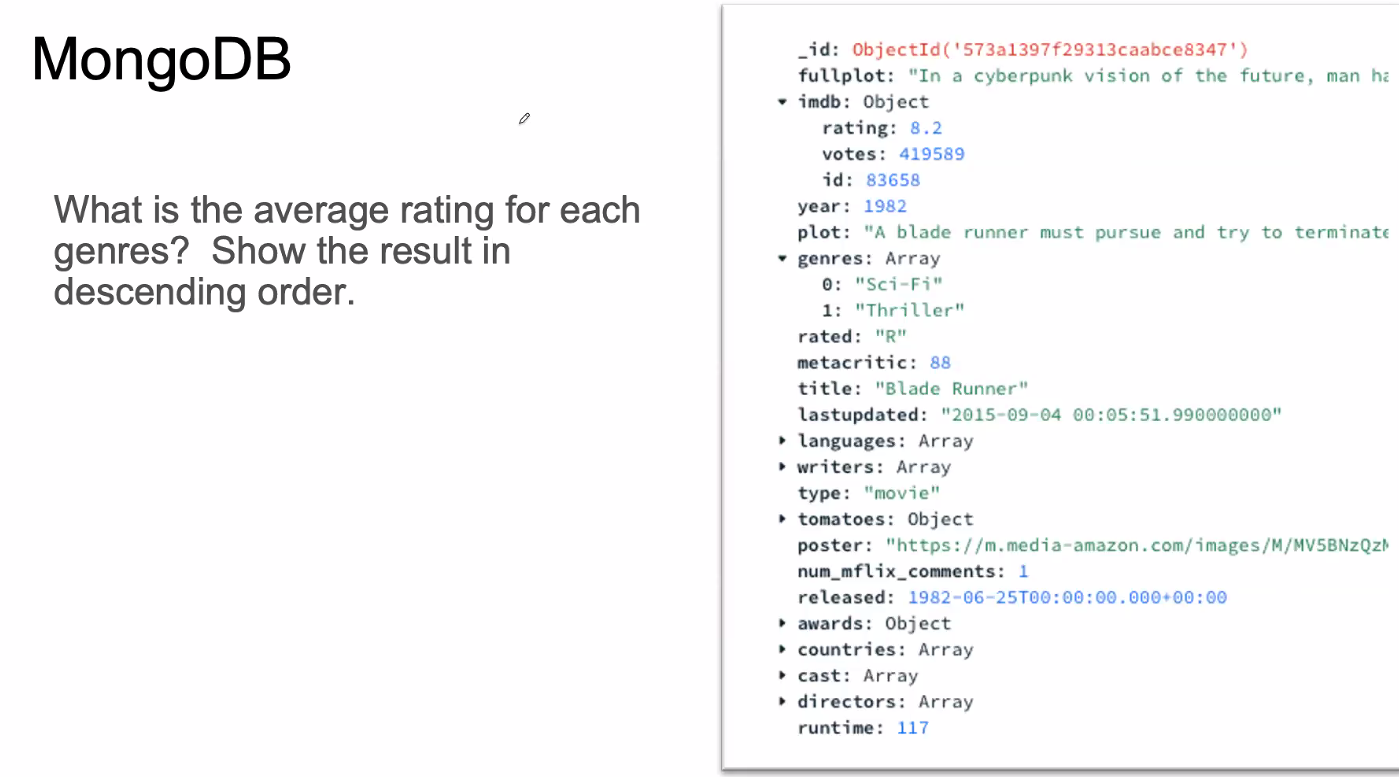
SELECT \*

FROM items

ORDER BY some\_column

OFFSET 5 ROWS -- Skips the first 5 rows

FETCH NEXT 10 ROWS ONLY;



db.movies.aggregate([

{$project : {\_id:0, genres: 1, rating:1}},

{$unwind: genres},

{$group : {\_id:”genres”, rating: {$avg : “$imb.rating”}}},

{$sort : {rating : -1}}

])

## Graph - Neo4j

Graph Database (nodes + edges)

index-free adjacency = use edge instead of relation table

schema-free = no need to specify the type of nodes during creation

Advantages of Graph Databases

good for complex relationships

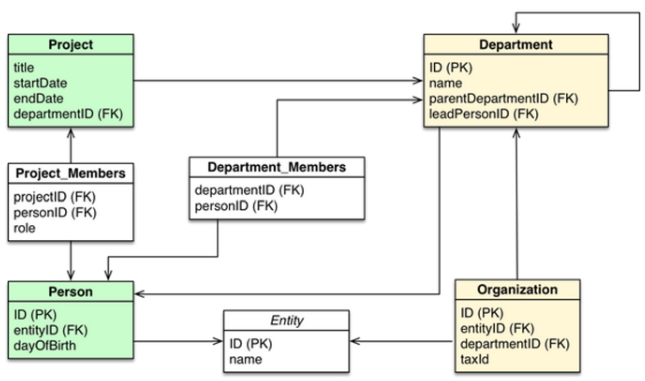
good for traversing relationships (ex. social networks)

bad for managing and modeling data (especially for less interconnected data)

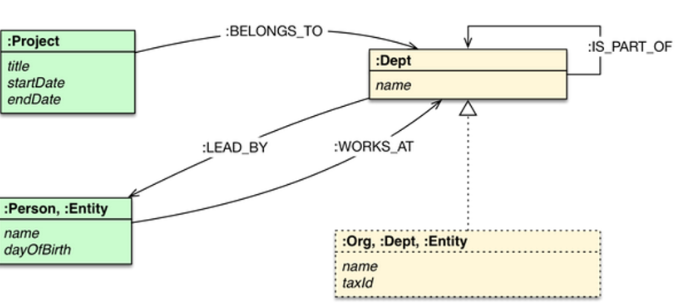
bad for dealing with massive database

bad for analytical purpose (slower than relational DB)

Ex. Relational



Graph



Syntax

CREATE( tmp\_var : type\_of\_var {properties: ‘prop\_val’, …} ) to create node

CREATE (tmp\_var) - [: edge\_name : {prop : [‘prop\_val?’]}] --> (tmp\_var) to create edge

**MERGE** matches existing or creates new nodes and patterns. This is especially useful together with uniqueness constraints.

**DELETE** deletes nodes, relationships, or paths. Nodes can only be deleted when they have no other relationships still existing

**DETACH DELETE** deletes nodes and all their relationships

**SET** sets values to properties and add labels on nodes

**REMOVE** removes properties and labels on nodes

**ORDER BY** is a sub-clause that specifies that the output should be sorted and how

MATCH (n) RETURN n to see all nodes & edges

MATCH (n) DETACH DELETE n to delete all nodes & edges

MATCH (tmp1 : type\_of\_tmp1) --> (tmp2: type\_of\_tmp2)

RETURN tmp1, tmp2 return all edges?? from tmp1\_type to tmp2\_type

* if it’s -- instead of → then it will return all edges between nodes

Ex.

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

CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})

CREATE (Keanu)-[:ACTED\_IN {roles:['Neo']}]->(TheMatrix)

MATCH (a:Person)-->(b:Movie) RETURN a,b

#all edges between person and movie

MATCH(p:Person {name:’Keanu Reeves’) → (x) RETURN COUNT(\*)

#how many nodes connect to the Person node whose name is ‘Keanu Reeves’

MATCH(p:Person {name:’Keanu Reeves’) → (x) RETURN p,x

#find all nodes connected to ‘Keanus Reeves’ node

MATCH(p:Person {name:’Keanu Reeves’) –[r]→ () RETURN type(r), COUNT(\*)

#from the same node, group edges by its type and return its count.

MATCH (p:Person {name:’Keanu Reeves’’}), (m:Movie {title:'The Matrix'})

CREATE (p) -[:ACTED\_IN]->m

#finds person and movies with given fields and create an edge in between

MATCH (p:Person), (m:Movie {title:'The Matrix'})

WHERE p.name = ‘Keanu Reeves’

CREATE (p) -[:ACTED\_IN]->m

#equivalent way to create an edge in existing two nodes

MATCH(a {name:’Keanu’}) – (b:Movie) RETURN a, b

#we can match without specifying var\_type

MATCH (p: Person {name…}), (l:Person {name…})

WHERE (p) -[\*]- (l) RETURN p, l

#check if two nodes are connected in the graph (directly or not)

# \* means any type of edges + any length of edges

MATCH (p: Person {name…}), (x)

WHERE (p) -[\*1..2]- (x) RETURN p, x

#return any nodes connected to p in length of 1 or 2 edges

LOAD CSV WITH HEADERS FROM 'file:///artists-with-headers.csv' AS line

CREATE (:Artist {name: line.Name, year: toInteger(line.Year)})

artists-with-headers.csv

Id,Name,Year

1,ABBA,1992

2,Roxette,1986

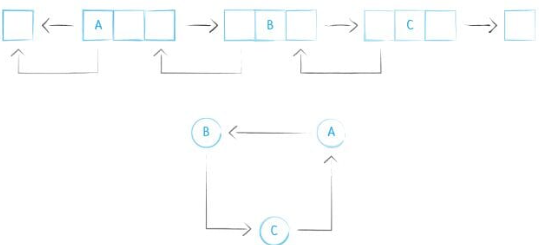
3,Europe,1979

4,The Cardigans,1992

Implementation

index-free adjacency (connect two entities with pointers (direct) while relational DB uses indirect connections)

edges always doubly linked list



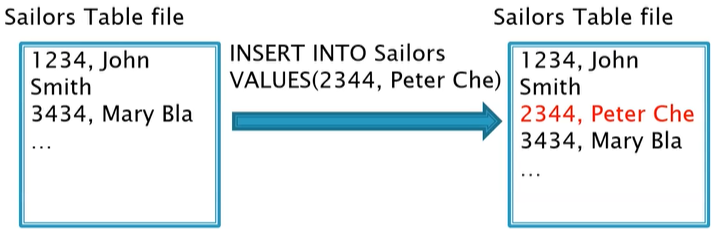
data stored on disk, cached in memory

## Data Storage in NoSQL

In-Place Storage for traditional DB

expensive: random disk access for each edit

make sure every edit is on the right place (slow edit but quick read)



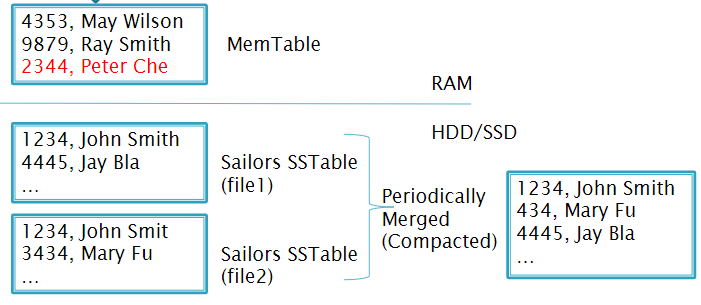
Log Structured Merge Tree for NoSQL

batch insertions in a buffer → insert full buffer to disk

eventual consistency (complicates read)

compaction/merge → clean up file1 & file2 to obtain the newest version

merging will keep the system busy → how frequent



\* select 1234 record will only search from mem table and file1. If we find it in file1, we don’t go to file2 (dead files)

\* insertion of new 2344 record will upsert (relational DB will check for duplicate before edit because it’s easier)

eg. LevelDB = a single node LSM DB (Cassandra runs on a cluster)

Compaction Strategy

ALTER TABLE users WITH compaction = { 'class' : 'LeveledCompactionStrategy’ }

ALTER TABLE users WITH compaction = {'class' : 'SizeTieredCompactionStrategy', 'min\_threshold' : 6 }

6 records to fill the buffer and ready to merge

small number ⇒ frequent merge

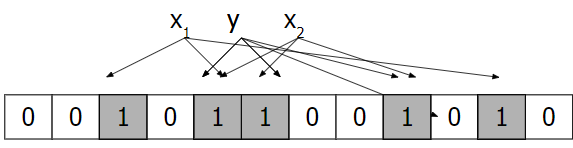
high write throughput → infrequent compaction

high read throughput → frequent compaction

Bloom Filter

quick check if key is in the SSTable (log files)

allow false positive (expect yes, but actually not)



Cassandra:

ALTER TABLE keyspace.table WITH

bloom\_filter\_fp\_chance=0.01 #rate of false positive chance

Default is 0.1

too low false positive FP rate ⇒ large bloom filter (performance vs space)

* More RAM ⇒ lower FP rate is affordable
* Few reads ⇒ No need for low FP rate

# Big Data Frameworks

Motivation

20+ billion web pages x 20KB = 400+ TB

1 computer reads 30-35 MB/sec from disk

~4 months to read the web

A standard architecture for solving such problems is:

Clusters of commodity nodes

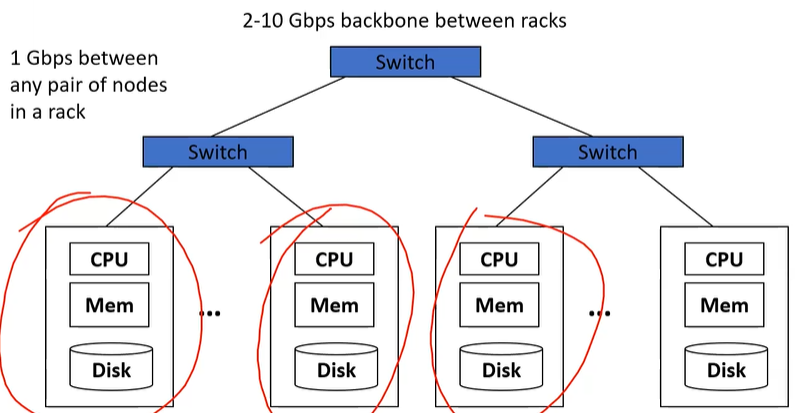
Commodity network (ethernet) to connect them

| NoSQL | Spark/MapReduce |
| --- | --- |
| Structured Data | Unstructured+Structured |
| Database systems | Programming paradigms |
| Organize data for fast retrieval | Analyze data to solve a problem |
| Queries typically take msec | Jobs can take minutes/hours |
| No overhead | Significant overhead to start job |
| Support thousand of queries per second | One or few jobs executing at a time |

NoSQL is to retrieve the data faster. For multiple purposes/tasks + smaller DB

Spark/MapReduce is for optimizing the job overall. For single purpose/task + bigger DB

Cluster Architecture



Good at handling failures (recover a job failure if one server dies)

Idea:

Bring computation close to the data (save time transferring)

Store files multiple times for reliability

MapReduce and Spark address these problems

Computational/data manipulation model

Elegant way to work with big data

Storage Infrastructure – File system

Google: Google File System (GFS). Hadoop: HDFS

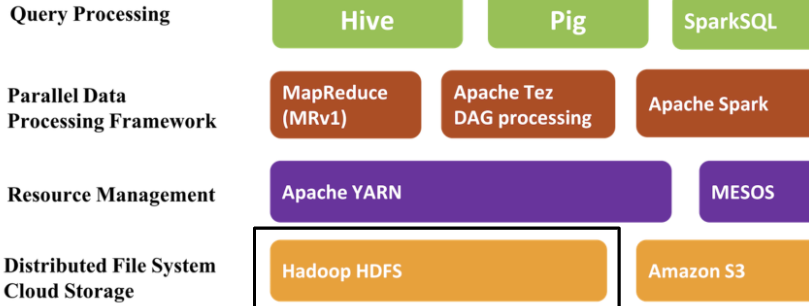
Programming models

MapReduce

Spark

## Hadoop Distributed File System (HDFS)

Big Data Stack



Partitioning and Replication

File is split into contiguous blocks. Each block is big

Each block is replicated (usually 3x or more)

Master – Slave architecture

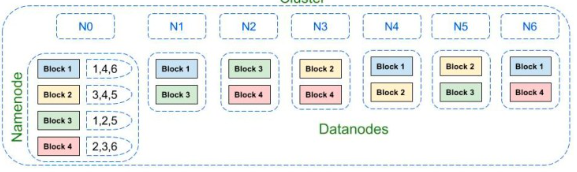
Name Node – Master node (stores metadata - where data is stored)

replication in case of failure

only responsible for file lookup

Data Node – Slave Node (actual data stored here)

responsible for data transferring



HDFS is inefficient for storing many small files

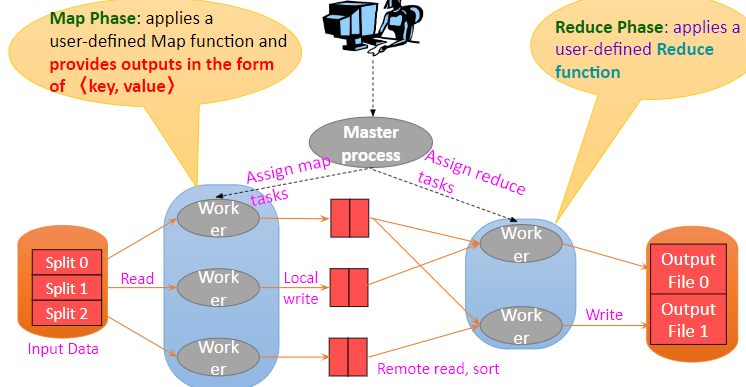
## MapReduce

Stages

Map to extract data and output in the form of 〈key (k), value(d)〉

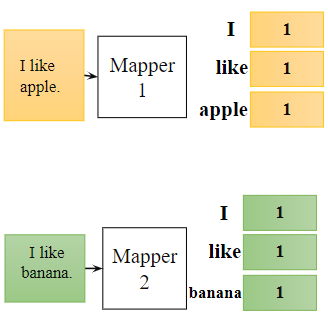
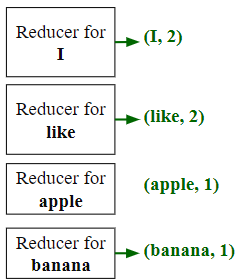
Shuffle to group by key and send to Reduce

Reduce to Aggregate, summarize, filter or transform



Map (read from disk) Shuffle (write to disk) Reduce (r/i to disk)

Ex. Word Count Map Reduce

Limitation of MapReduce (can be handled by Spark)

too many disk accesses during Shuffling (big memory in Spark is needed)

no stream processing (only batch data)

hard to translate code to Map-Reduce format

## Spark

RDD (Resilient Distributed Dataset)

Partitioned collection of records

Spread across the cluster (sharding)

Read-only

Expensive to update the data. Not expensive to create new one

May be cached in memory (=> speed up)

fallback to disk possible

May be reconstructed if failure

RDD operations

Transformations build new RDDs through operations on other RDDs

Examples: map, filter, join, group-by

Lazy execution (optimize by combining some similar operations)

Actions return value or export data

Examples: count, collect, save

Trigger execution

Ex. the first two are transformations

lines = sc.textFile(“hdfs://...”)

errors = lines.filter(lambda s: s.startswith(“ERROR”))

messages = errors.map(lambda s: s.split(“\t”)[2])

messages.cache()

SparkSQL - DataFrame

def = distributed collection of data divided into named columns = schema of its row

looks like RDD, but internally store data in efficient manner

able to run SQL queries

Ex. creat and show dataframe (from json

energyDF = spark.read.json("s3://msba295uci2018/EnergySample3.json")

energyDF.show()

energyDF.createOrReplaceView(“energy”)

//two ways to query and show

queryDF = spark.sql(“select humidity from energy”)

queryDF.show()

energyDF.select(energyDF["humidity"]).show()

Hive & Pig

Run on top of MapReduce

Provide higher-level operations to query data

Hive is based on SQL

Summary

MapReduce and Spark:

Process large amounts of data stored in files

Handle hardware failures

Distribute work to cluster machines

Files stored in HDFS which:

Ensures data is not lost through replication

Allows parallel reading/writing

Spark:

has more operations than MapReduce

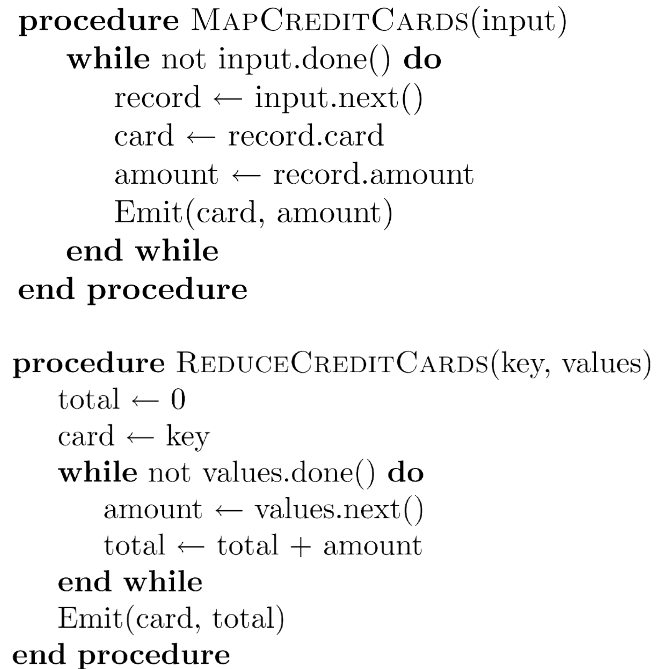
leverages large memory to minimize disk operations

SparkSQL allows SQL operations on top of files

YARN for resource management

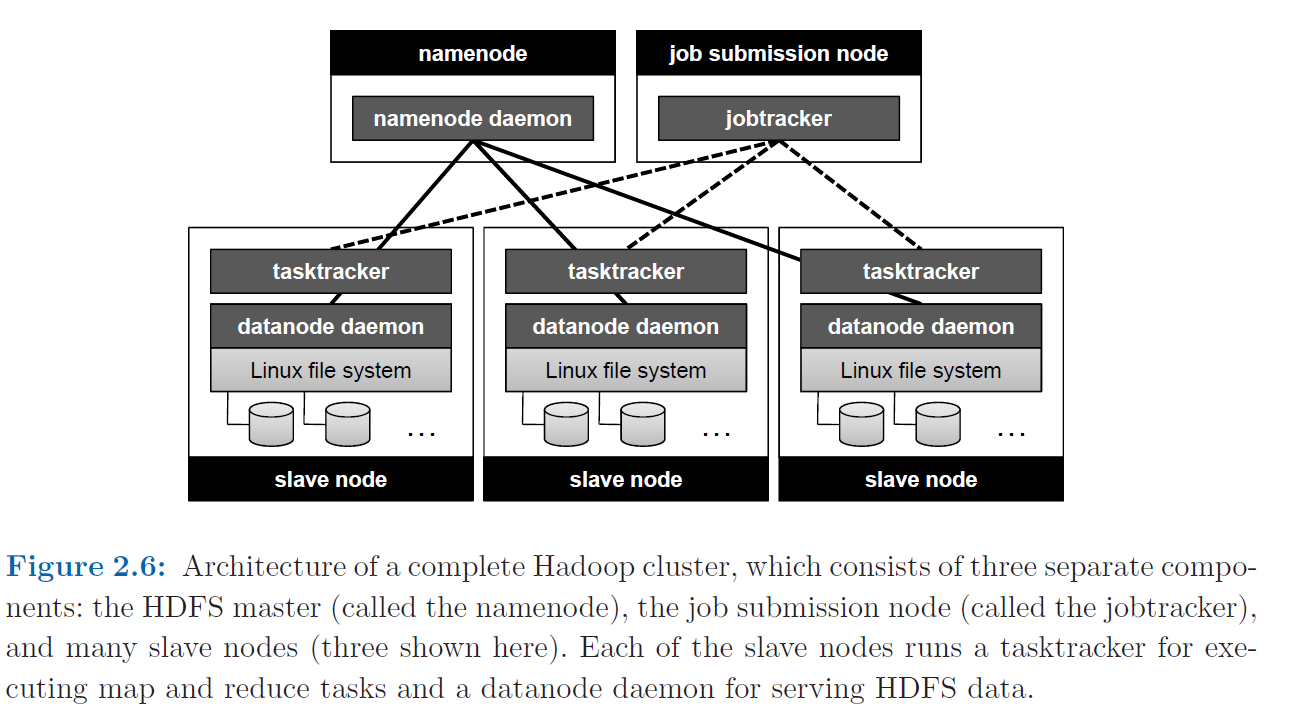
<https://docs.google.com/presentation/d/1_LP813s0KOJ8u62gy1WRBOMFjlNEPEBr/edit#slide=id.p19>

combiner can be same as reducer (preprocess records)



Combiner after mapper

can be used to reduce the network traffic when sending to reducer

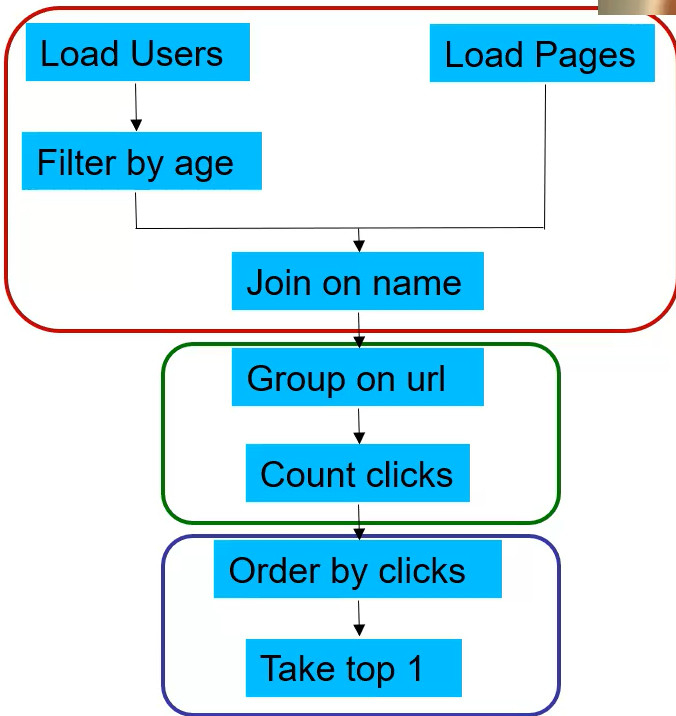
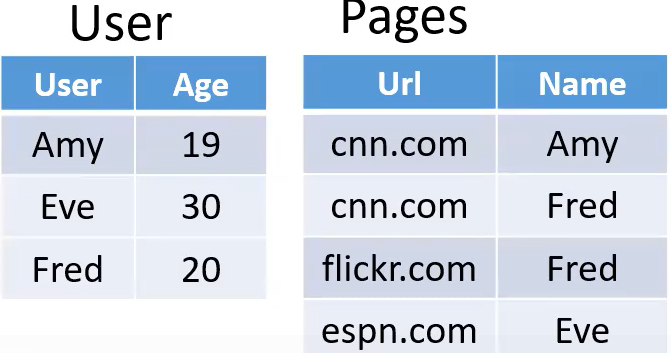


moving program to data instead of the opposite

Pig-Latin (spark-like) and HiveSQL (sql-like)

input data → define objects for input → provides relational operators (select, join…)

Ex. find top 1 visited pages by user aged 18-25



Users = load ‘users’ as (name, age);

Filtered = filter Users by age >= 18 and age <= 25;

Pages = load ‘pages’ as (user, url);

Joined = join Filtered by name, Pages by user;

Grouped = group Joined by url;

Summed = foreach Grouped generate group count(Joined) as clicks;

Sorted = order Summed by clicks desc;

Top1 = limit Sorted 1;

store Top1 into ‘top1site’;

Hive example…

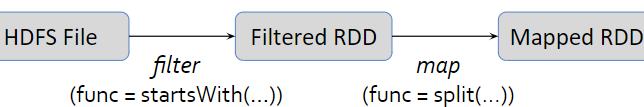
Spark + Python

spark requires more memory to speed up the program

fault recovery

RDDs are only created if you want to edit it

Ex.mapped RDD dies → starts from HDFS and rebuild



Ex. parallelize to enter input by typing instead reading from file

# Turn a Python collection into an RDD

* sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3

* sc.textFile(“file.txt”)
* sc.textFile(“directory/\*.txt”)
* sc.textFile(“hdfs://namenode:9000/path/file”)
* sc.textFile("s3://sparkdemo/inputfile")

Ex. map to match one input to one output. flatMap to match one input to multiple output

* nums = sc.parallelize([1, 2, 3])

# Pass each element through a function

* squares = nums.map(lambda x: x\*x) // {1, 4, 9}

# Keep elements passing a predicate

* even = squares.filter(lambda x: x % 2 == 0) // {4}

# flatMap can output multiple elements per input element

> x = sc.parallelize(["spark rdd example", "sample example"])

> y = x.flatMap(lambda x: x.split(' ')) // {'spark', 'rdd', 'example', 'sample', 'example'}

Ex. collect to print. reduce to use multiple input and generate the output

* reduce must be associative function (order of the input does NOT matter)
* nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection

* nums.collect() # => [1, 2, 3]

# Return first K elements

* nums.take(2) # => [1, 2]

# Count number of elements

* nums.count() # => 3

# Merge elements with an associative function

* nums.reduce(lambda x, y: x + y) # => 6

# Write elements to a text file

* nums.saveAsTextFile(“hdfs://hostname:8020/file.txt”)

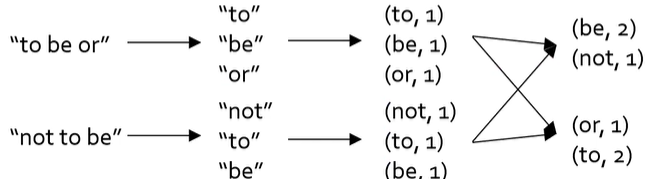
Ex. key-value operator to perform mapreduce-like tasks

reduceByKey = combiner in MapReduce

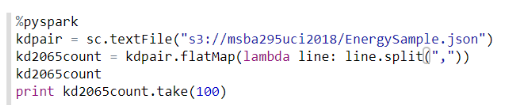
* pets = sc.parallelize(  
   [(“cat”, 1), (“dog”, 1), (“cat”, 2)])
* pets.reduceByKey(lambda x, y: x + y)  
   # => {(cat, 3), (dog, 1)}
* pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
* pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}

Ex. word count

* lines = sc.textFile(“hamlet.txt”)
* counts = lines.flatMap(lambda line: line.split(“ ”))  
   .map(lambda word => (word, 1))  
   .reduceByKey(lambda x, y: x + y)



Ex. json input file



Ex.

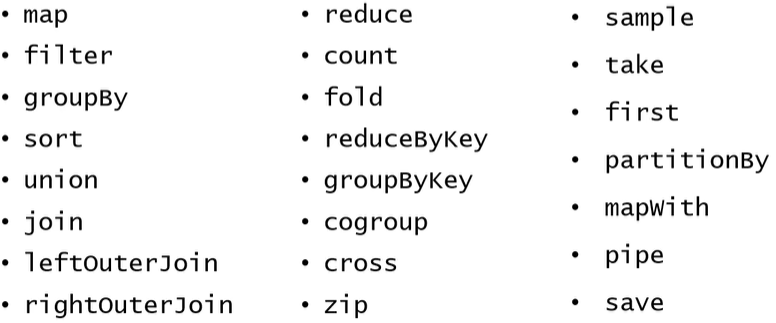
* visits = sc.parallelize([ (“index.html”, “1.2.3.4”),  
   (“about.html”, “3.4.5.6”),  
   (“index.html”, “1.3.3.1”) ])
* pageNames = sc.parallelize([ (“index.html”, “Home”),  
   (“about.html”, “About”) ])
* visits.join(pageNames)   
  # (“index.html”, (“1.2.3.4”, “Home”))  
  # (“index.html”, (“1.3.3.1”, “Home”))  
  # (“about.html”, (“3.4.5.6”, “About”))
* visits.cogroup(pageNames)   
  # (“index.html”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))  
  # (“about.html”, ([“3.4.5.6”], [“About”]))

All the pair RDD operations take an optional second parameter for number of tasks

Ex. parallelization number (highest number will be number of your cores)

* + words.reduceByKey(lambda x, y: x + y, 5)
  + words.groupByKey(5)
  + visits.join(pageViews, 5)

Spark operators



Spark + DataFrames

Distributed collection of data organized into named columns

Conceptually equivalent to table in relational database.

Look like regular RDDs; internally they store data in a more efficient manner, taking advantage of their schema.

Provide new operations not available on RDDs, such as the ability to run SQL queries.

from a JSON file

df = spark.read.json("s3://msba295uci2018/EnergySample3.json")

From csv

df = spark.read.format("csv")

.option("header", "true") //first line in file has headers

.load("s3://msba295uci2018/EnergySample3.csv")

use view instead of table

s.createOrReplaceView(“students”)

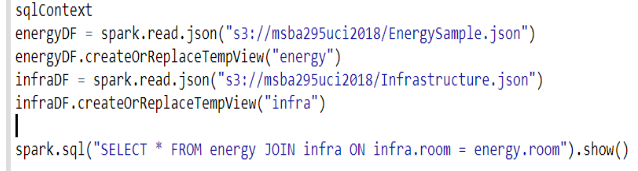
//create view s1 as select \* from students where age > 18

select \* from s1

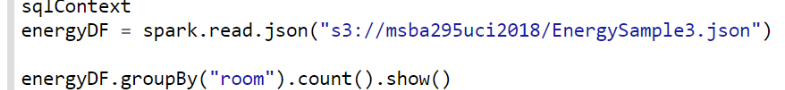
Ex. avg



Ex. join



Ex. group by



Spark Programming (11/21)

nums = sc.parallelize([1,2,3])

squares = nums.map(lambda x: x\*x)

even = squares.filter(lambda x: x % 2 == 0)

Midterm review (11/28)

group the **continuous** event and display the start and end date



df\_change = df.withColumn(“event\_change”, when(col(“event\_status”) != lag(“event\_status”).over(window.orderBy(“event\_date”)),1).otherwise(0))

//show for df and collect for rdd

tweet = spark.read.json(...)

spark.read.format("csv")

.option("header", "true") //first line in file has headers

.load("s3://msba295uci2018/EnergySample3.csv")

tweet.show() only shows the top level

tweet.printSchema() shows the whole tree

tweet.show(5, truncate = 100) shows the first 5 rows, each row contains 100 char

MongoDB is better for small queries (query again and again)