Search

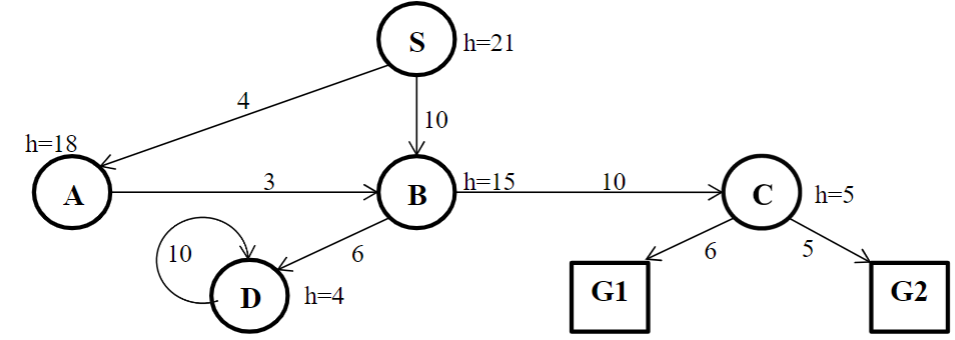
Tree Search - do NOT remember visited nodes (guaranteed to have NO cycle)

space efficient

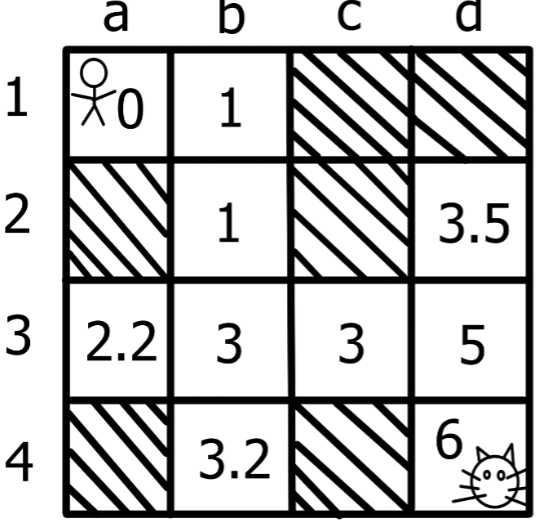
Tree size can be where b = constant branching factor and d = depth of goal state

Graph Search - DO remember visited nodes

time efficient



Order of Expansion path cost

DFS SABDDDD…

BFS SABDC**G1** SBCG1 26

IDS S SAB SAB**B**DC**G1** SBCG1 26

UCS SABDCG2 SABCG2 22

Best-First SBDDD…

A\* SABDCG2 SABCG2 22

Ex. cost = 1/move. h(n) is shown in the grid

A\* search sort by h(x) + g(x)

6 6

5 5.5

6 5 6

6

order of node = d4 d3 c3 d2 b3 b2 b1 a1

Best-First sort by h(x)

order of node = d4 d3 c3 b3 b2 b1 a1

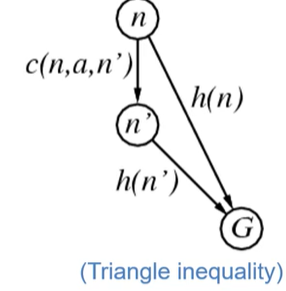
**Admissible** heuristic = for every node n, h(n) <= h\*(n)

**h(n)** = estimate cost from n to goal

h\*(n) = true cost from n to goal

**only underestimates but never overestimate the cost to get to the goal**

Theorem: if h(n) is admissible, A\* Tree search is optimal

Consistent heuristic

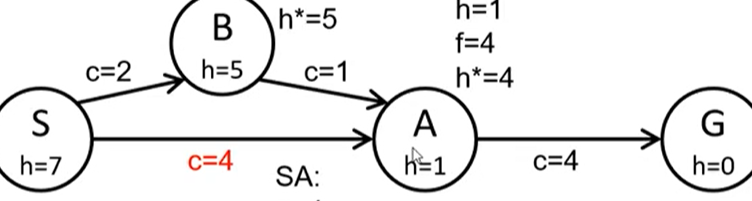
cost(start to end) >= – h = h(start) - h(end)

**⇒ consistency** ⇒ admissible

Theorem: if h(n) is consistent, A\* Graph search is optimal

consistency ⇒ A\* search expands in order of f value

Ex. admissible but not consistent



admissible ⇐ h(S) = 3 < 7 = h\*(S) h(B) = 2 < 5 = h\*(B) h(A) = 1 <= 1 = h\*(A)

cost is the upper bound of h

S → B: h = 2 <= 2 **B→A: h=4 > 1 not consistent**

CSP

MRV (minimum remaining values) or MCV (most constrained values)

a heuristic to pick next **variable** with fewest legal values (easy to detect failure)

→ reduce the branching factor

LCV (least constraining value) to pick a **value** for a variable

the one that rules out the fewest values in the remaining variables

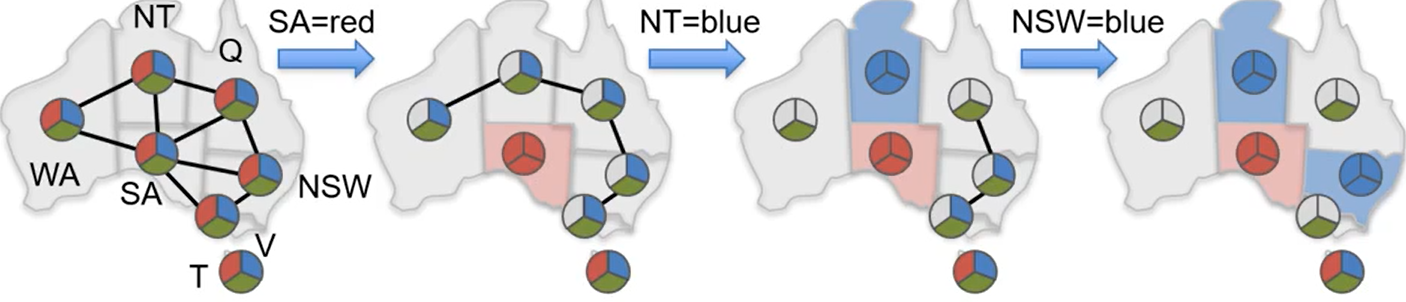
ex. pick red because it allows more values for SA



Ex. same problem using degree heuristic. We will start from SA because degree(SA) = 5

Remove edges after assignment

→ select NT or Q or NSW



MRV + degree will have the same expansion in this problem

forward checking - checks the unassigned neighbors in constraint graph

keep track of legal values of all variables

ONLY check neighbor of the most recently assigned variable

backtrack if neighbor has NO legal values

arc consistency - check consistency among all other neighbors

Logic

For , always use ⇒ and for always use ^

(P ⇒ Q) ↔ (¬P ∨ Q)

FOL to CNF

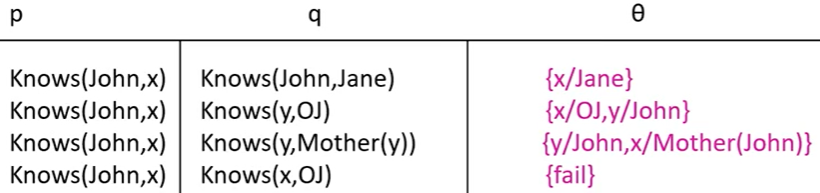
no ⇒ in the form

replace all y with F(x)

replace all y with F() or C

Unification

F(x) = G(z) cannot substitute







Game – MinMax & Pruning

= highest-value choice found at any choice point of path at MAX

default -inf

MAX node update if new max (with child’s value)

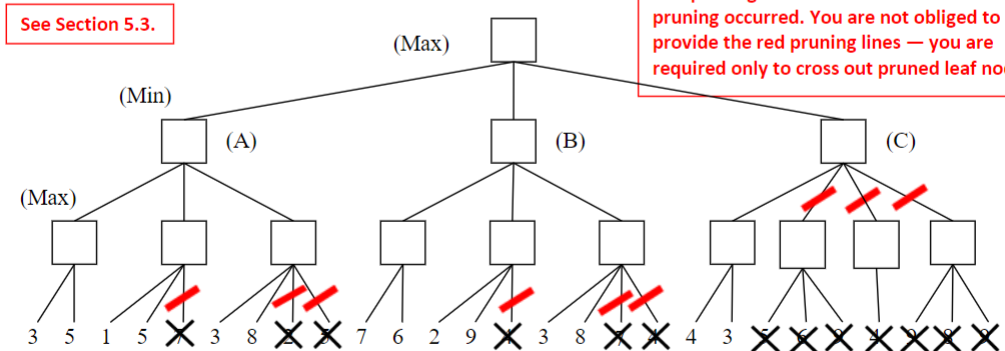
= lowest-value choice found at any choice point of path at MIN player

default inf

MIN node update if new min (with child’s value. NOT a or b)

prune whenever

MAX nodes, we want bigger value in children’s min



Probability

Posterior/Conditional Probability P(a | b) = a given b = P(a,b) / P(b)

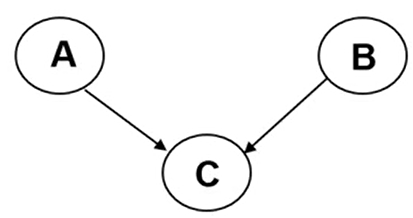
Question 6 <https://canvas.eee.uci.edu/courses/58458/files/folder/Past%20Tests/2020?preview=23910437>

For each node # probabilities = O(n\* 2^k) where k = # parents at maximum.

sum all nodes ⇒ number of parameters

Bayesian Network

Ex. p(a,b,c) = p(c|a,b) p(a) p(b) p(a,b,c) = p(c|b) p(b|a) p(a)

False There are 2 n decision trees with n Boolean attributes.

False For any Boolean function, there is a compact decision tree.

True Overfitting occurs when the model complexity is too high for the available data.

False Information gain from an attribute test is the expected gain in entropy.

**Graph - Neo4j (nodes + edges)**

index-free adjacency (edge instead of relation table)

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

good for complex/traversing relationships (ex. social networks)

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

bad for massive database / analytical purpose (slower than relational DB)

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. Good to maintain uniqueness constraints.

**DELETE** deletes nodes, relationships, or paths. # Can only be deleted without 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 node

CREATE (Keanu:Person {name:'Keanu Reeves', born:1964}) #create node, using merge to upsert

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

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:Person)-->(b:Movie) RETURN a,b #all edges between person and movie

MATCH(p:Person {name:’Keanu Reeves’) → (x) RETURN COUNT(\*) #nodes connect to the ‘Keanu Reeves’

MATCH(p:Person {name:’Keanu Reeves’) → (x) RETURN p,x #nodes connected to ‘Keanus Reeves’ node

MATCH(p:Person {name:’Keanu Reeves’) –[r]→ () RETURN type(r), COUNT(\*) #group edges by its type and return its count.

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)

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 #Id,Name,Year

CREATE (:Artist {name: line.Name, year: toInteger(line.Year)}) # 1,ABBA,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 NoQL**

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 edit files (merging will keep the system busy → how frequent)

\* we will search records from the nearest edit file to oldest

\* 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)

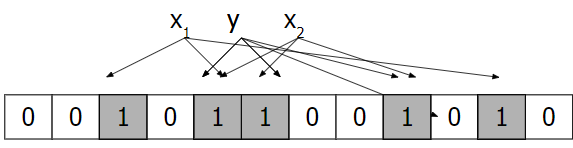
Compaction Strategy

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

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

small number ⇒ frequent merge （6 records to fill the buffer and ready to 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)

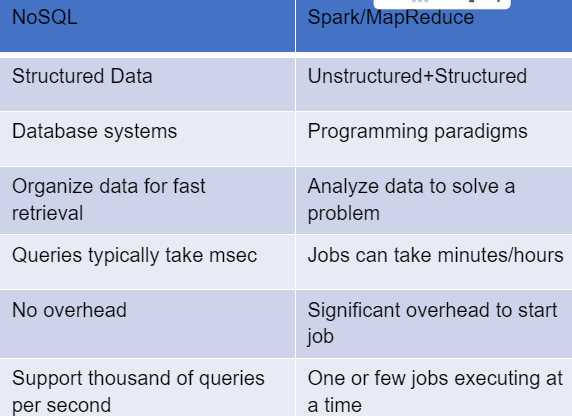
Cassandra: (Default is 0.1)

ALTER TABLE keyspace.table WITH

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

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**

A standard architecture for solving such problems is:

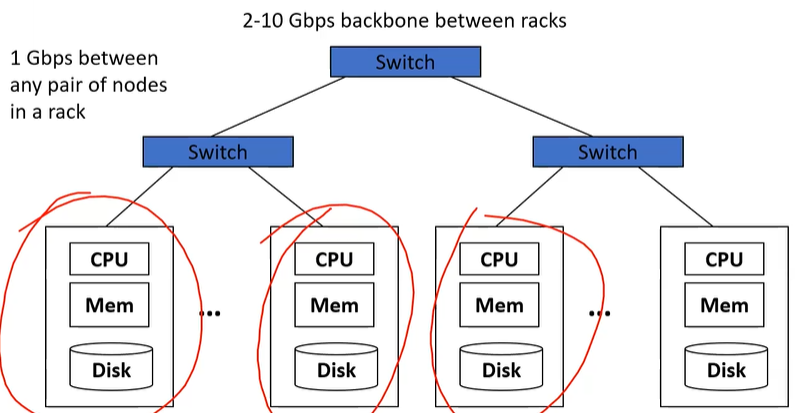
Clusters of commodity nodes

Commodity network (ethernet) to connect them

NoSQL (retrieve data) multiple purposes/tasks + similar DB

Spark/MapReduce (optimize job) 1 purpose/task + bigger DB

Cluster Architecture



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

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

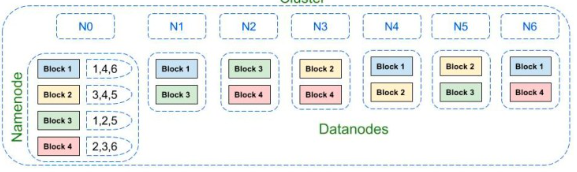
Storage Infrastructure – File system Google: Google File System (GFS). Hadoop: HDFS

Programming models MapReduce Spark

**Hadoop Distributed File System (HDFS)**

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 (metadata)

only responsible for file lookup

replication in case of failure

Data Node – Slave Node (actual data)

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

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. cheap to create new one

May be cached in memory (=> speed up) fallback to disk possible

May be reconstructed if failure

RDD operations

Transformations (lazy execution ← batch similar exe) build new RDDs (textFile, map, filter, join, group-by)

Actions （trigger execution) return value or export data (count, collect, save)

DAGScheduler

splits graph into stages of tasks (must complete previous stage to go to the next)

RDD object → DAGScheduler → task scheduler (launch task via cluster manager) → worker (execute)

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

Ex. create 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:

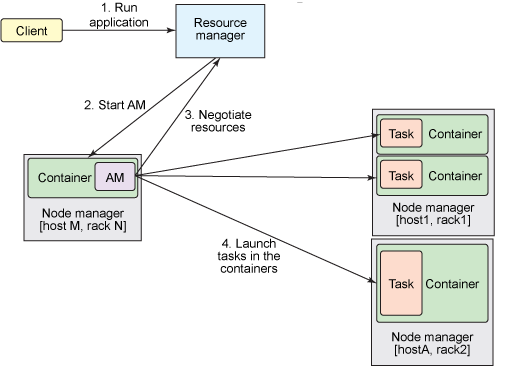
Process large amounts of data stored in files (via HDFS

ensures data is not lost through replication + allow parallel reading/writing)

Handle hardware failures

Distribute work to cluster machines

* Spark more operations than MapReduce + large memory to minimize disk operations

YARN (Yet Another Resource Management)

Manages resource allocation for processing data in HDFS

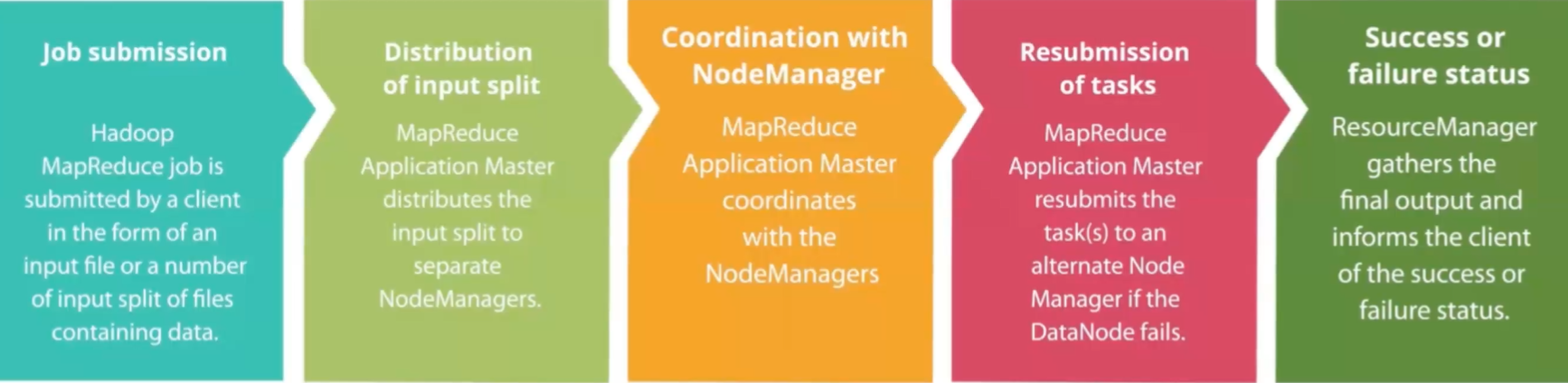
Limitation of Hadoop 1.x = Scalability + Not optimal utilization of the cluster

Supports only MapReduce jobs

resource manager (resource allocation in cluster)

(AM) application master = programs executed inside the containers

node managers create containers (monitoring their resource usage)

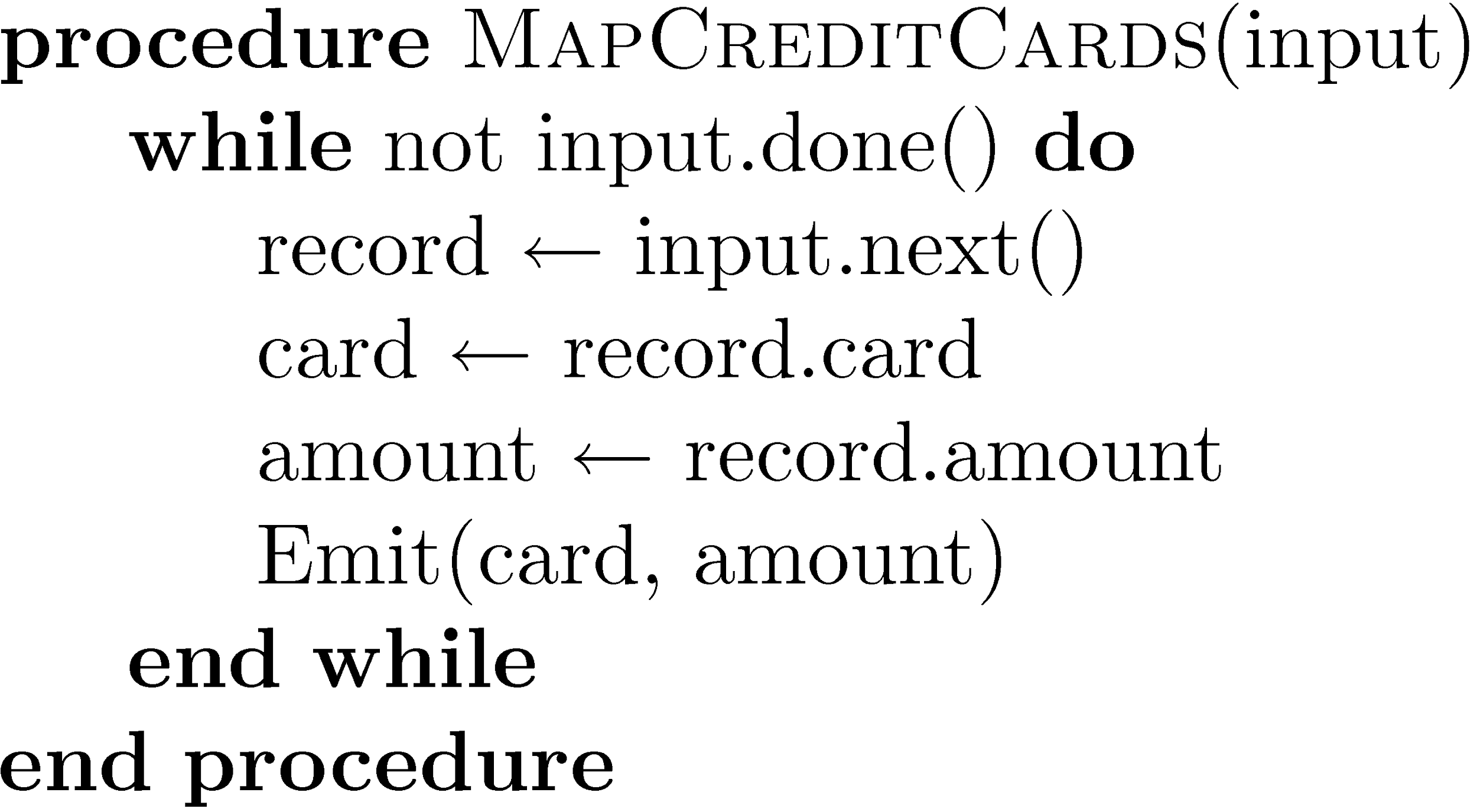
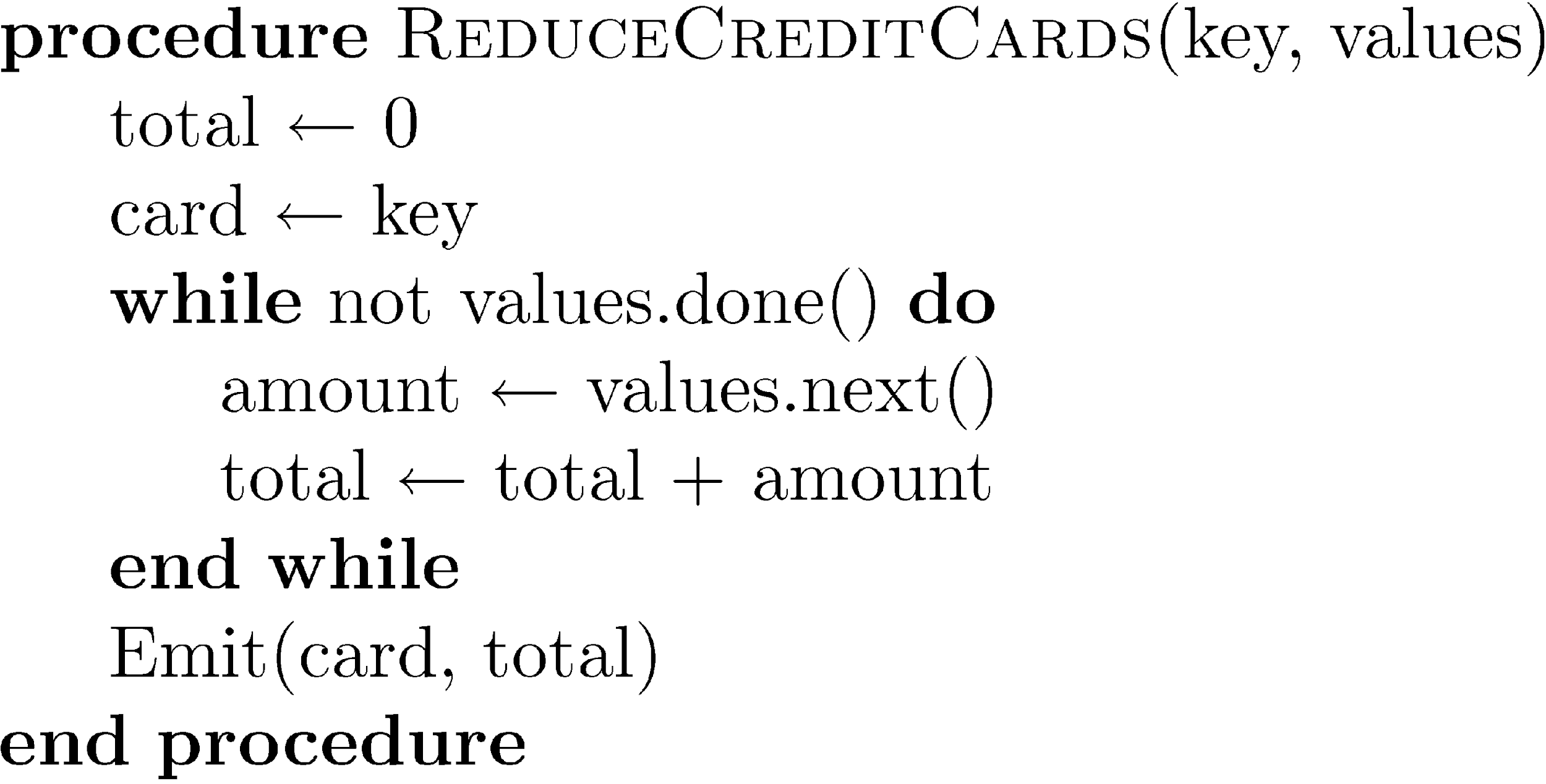


Combiner can be same as reducer (preprocess records)

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

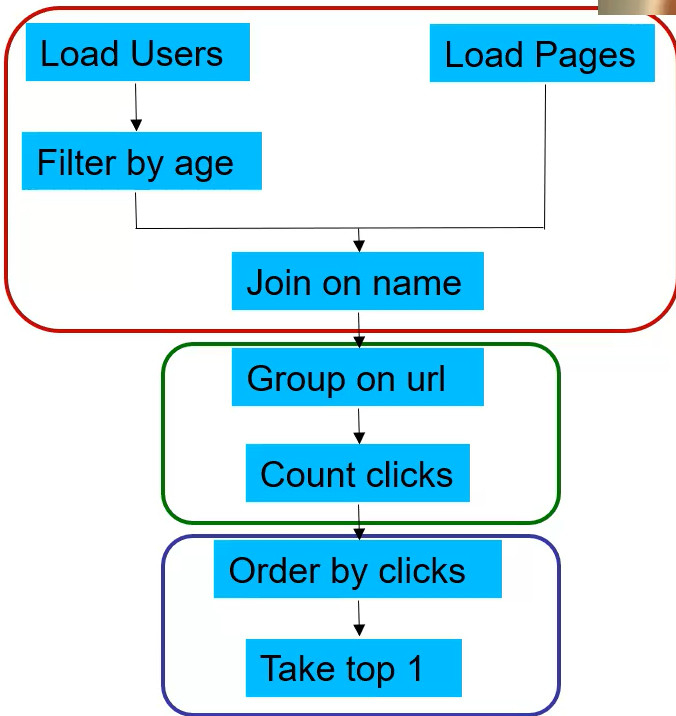
Ex. word count. It’s quicker to transfer (xy, 4) than transfer (xy, 1) 4 times

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;

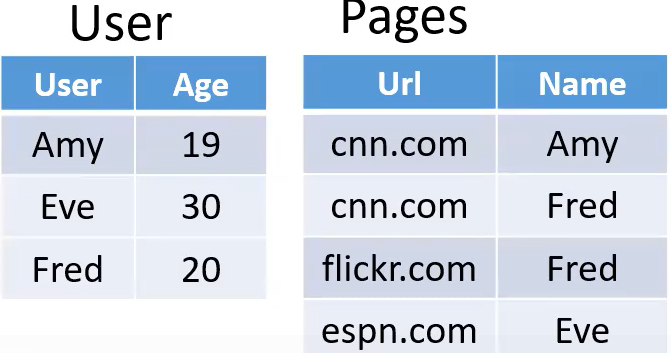
Pages = load ‘pages’ as (user, url);

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’;

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

* sc.parallelize([1, 2, 3]) # Turn a Python collection into an RDD

# 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 \* 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 execute lazy executions. 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])
* nums.collect() # => [1, 2, 3] # Retrieve RDD contents as a local collection
* nums.take(2) # => [1, 2] # Return first K elements
* nums.count() # => 3
* nums.reduce(lambda x, y: x + y) # => 6 Merge elements with an associative function
* nums.saveAsTextFile(“hdfs://hostname:8020/file.txt”) #Write elements to a text file

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

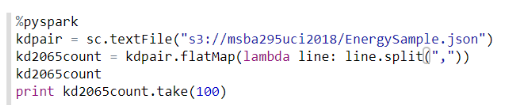
reduceByKey = combiner in MapReduce

* 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

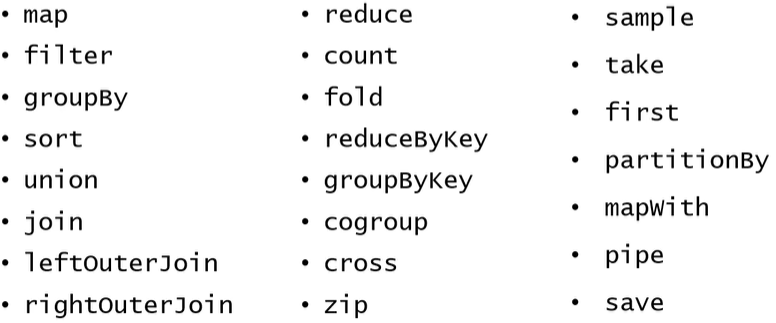
* lines = sc.textFile(“hamlet.txt”)
* counts = lines.flatMap(lambda line: line.split(“ ”))  
   .map(lambda word => (word, 1))  
   .reduceByKey(lambda x, y: x + y)

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 + DataFrames

Distributed collection of data organized into named columns

Look like regular RDDs; internally store data in a more efficient manner via using schema.

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

Ex.

df = spark.read.format("csv") #load from CSV file

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

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

df = spark.read.json("s3://msba295uci2018/EnergySample3.json") #from JSON file

df.groupBy(“room”).count().show() #group by

s.createOrReplaceView(“students”) #creates view on dataframe to run SQL

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

spark.sql(“select avg(age) as average\_age FROM students”).show() #avg

tweets10k = spark.read.json("s3://vagelis-testbucket1/10Ktweets.json");

tweets10k.createOrReplaceTempView("tweets10kview");

# find authors that contain "Trump" in their username

Trumpusers = spark.sql("select includes.users[0].username from tweets10kview where

includes.users[0].username like '%Trump%'");

#compute frequency of each hashtag

from pyspark.sql import SparkSession

import pyspark.sql.functions as F;

singlehashtags = hashtags.withColumn('tag', F.explode(hashtags['tag'])); //split array column into \* rows:

singlehashtags.createOrReplaceTempView("singlehashtagsview");

groupedhashtags = spark.sql("select tag, count(\*) as c from singlehashtagsview group by tag order by c desc");

groupedhashtags.show(10);

logfile = sc.textFile("s3://vagelis-testbucket1/logfile.txt");

messages = logfile.map(lambda s: s.split(': ')[1]); #extract text after :

rev = messages.filter(lambda s: s.startswith('reverse')); #find rows starting with “reverse”

passw = messages.filter(lambda s: 'password' in s); #find rows containing “password”

Spark Programming (11/21)

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

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

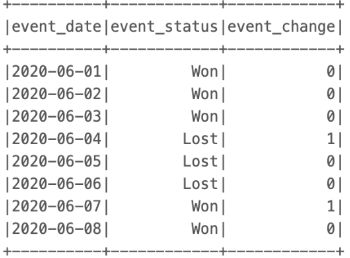
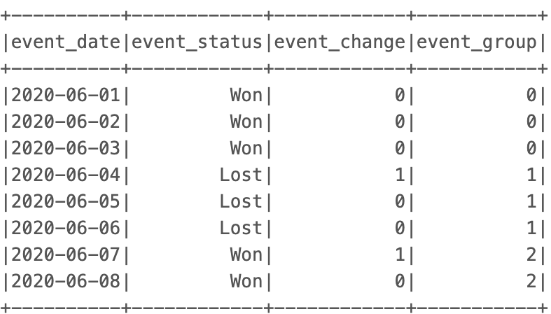
even = squares.filter(lambda x: x % 2 == 0)

Review

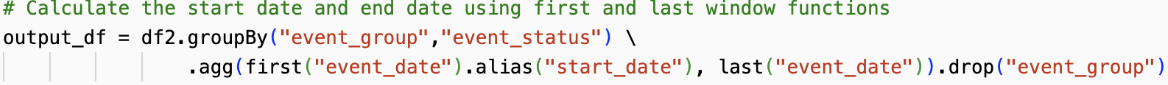
Ex. group the **continuous** event and display the start and end date

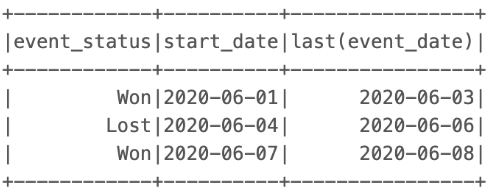




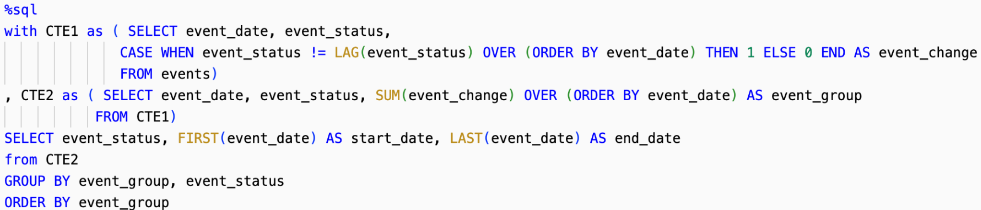
 







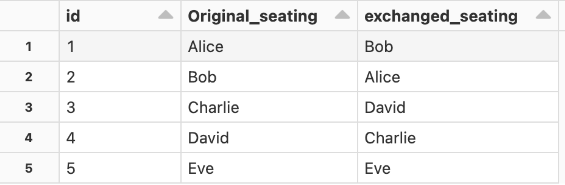
df.createOrReplaceTempView(“events”)

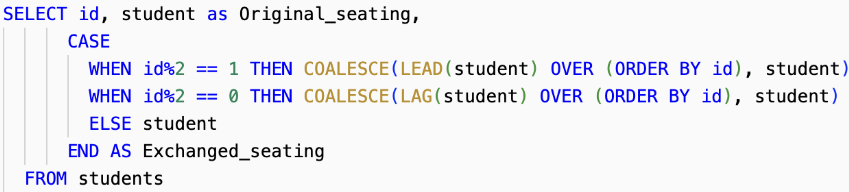


Ex. exchange pairs of students

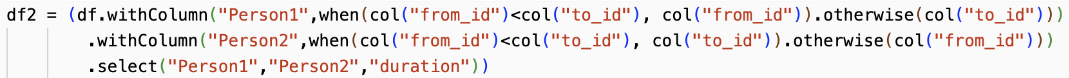








Ex. calls between two persons



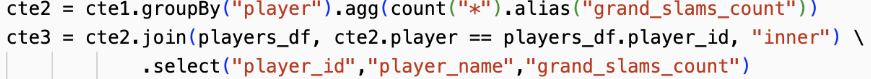
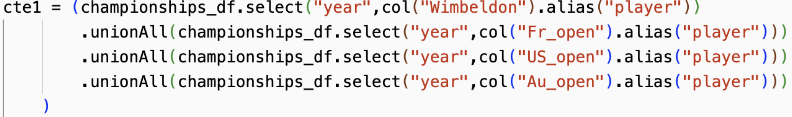


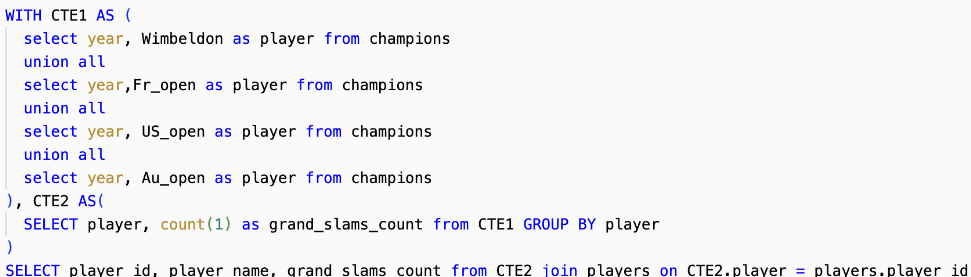
SELECT MIN(from\_id, to\_id) as person1, MAX(from\_id, to\_id) as person2

count(\*) as call\_count, sum(duration) as total\_duration

FROM calls GROUP BY 1, 2

Ex. count grand slam titles





Ex. employee who earns more than manager

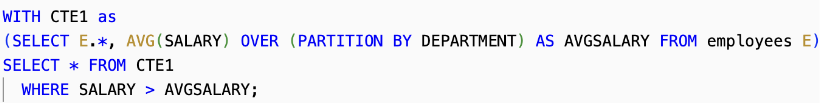




Ex. employee who earns more than department average

window\_spec = Window.partitionBy(“department”)

df2. df.withColumn(“avg\_salary”, avg(col(“salary”)).over(window\_spec))



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

OLTP: Online Transaction Processing (DBMSs) for small queries + transactions (create, update, or retrieve data)

OLAP: Online Analytical Processing (Data Warehousing) for analysis + complex but fewer queries.

ETL = Extraction + Transformation + Loading (preprocessing data)

| 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 |
| Example | MySQL, Oracle IBM DB2, SQL Server, Amazon Aurora, NoSQL, MongoDB, Hbase  Cassandra, Couchbase, AsterixDB | 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) |
| --- | --- | --- |
| Limitation | 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)

IaaS vs. PaaS vs. SaaS

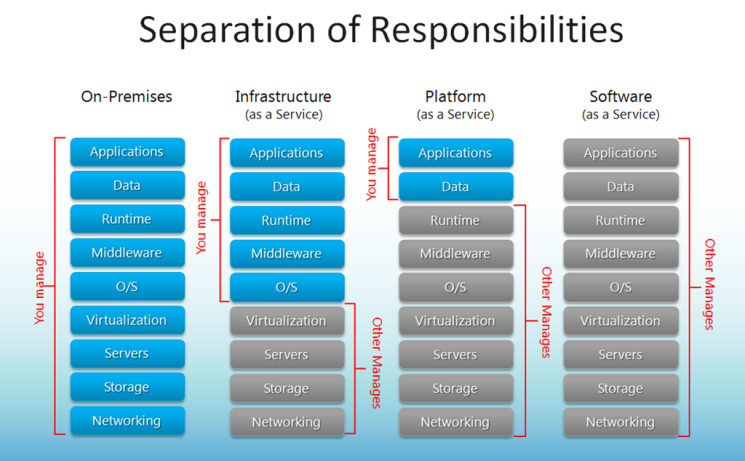
IaaS (no hardware, most affordable) AWS EC2, Cisco Metapod, Microsoft Azure, Google Compute Engine

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



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

both can vertical scaling

**Relational Databases**

Relational database: a set of relations = tables

Row = tuple =record, #rows = **cardinality**

Field = attribute = column, , #attributes = **degree** / arity.

Relation: instance (table with row+col) + schema (relation name + column name/type)

ACID transaction – too many transactions will slow system down → consider the tradeoffs

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

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) log\_(c/2) N where c = cap of a node

Good for range or equality queries

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

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, index on color will NOT help

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

to allow for concurrent/parallel accesses + to improve performance of complex query

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

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

Easy for management

hash-based sharding

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 by 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 (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)**

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** (SQL prefer checking, NoSQL prefer updating)

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

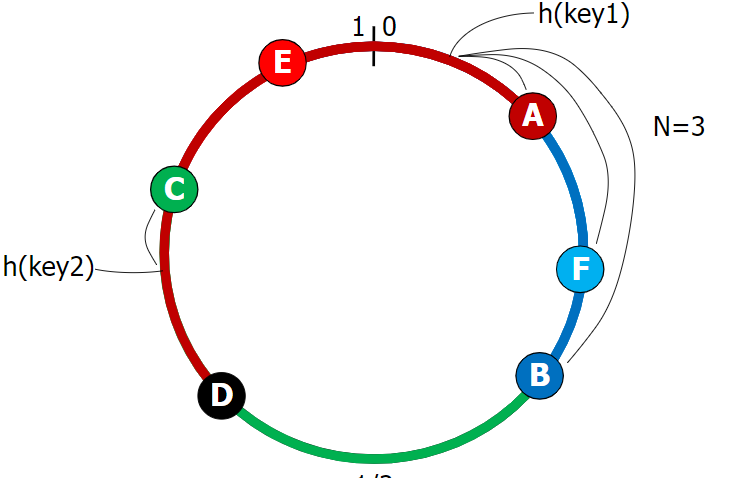
in MySQL, you can upsert by including ON DUPLICATE KEY UPDATE

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

| 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.)

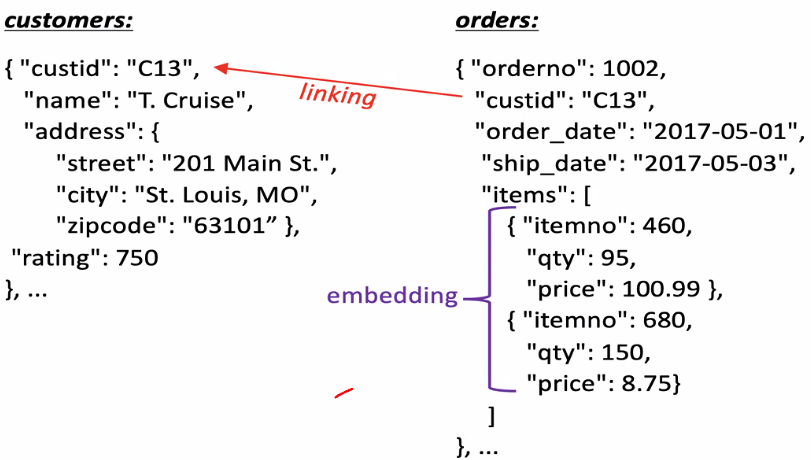
Default is ONE - Read (read and repair if digests diff in another replica (number of replicas ⇐ consistency level))

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

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 (Document-based) BSON 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

$project is the same as SELECT …

$match (HAVING?), $group, $skip (OFFSET)

$unwind: deconstructs the items (array becomes elements)

$sort: -1 desc

Ex.

use db → show tables → db.createCollection

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

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

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" }) create text index on data.text

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

db.orders.aggregate([

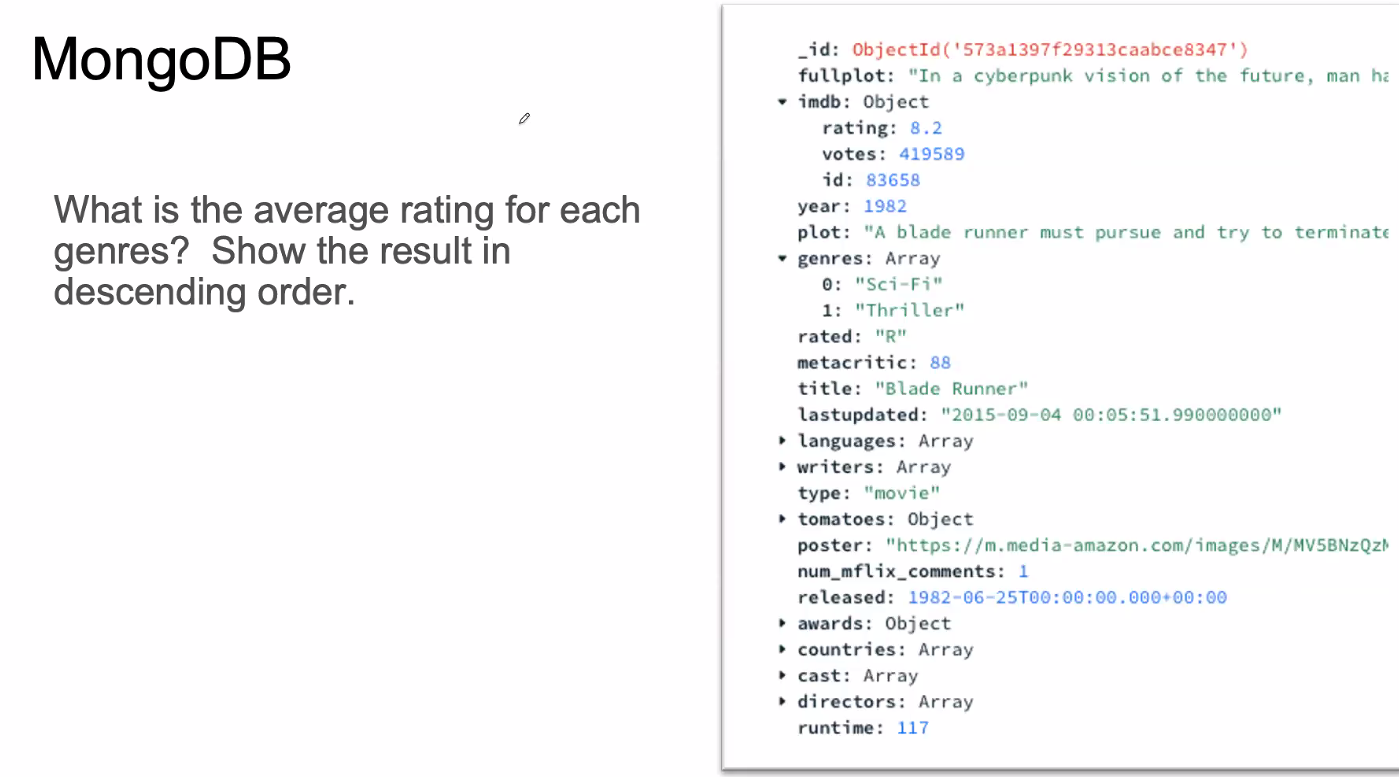
{$match : {size:”medium”}},

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

]) to get total order quantity of medium size pizzas

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}}

]) to get avg rating for each genre (result in desc order)

SELECT genres, AVG(rating) AS rating

FROM (

SELECT genres, rating

FROM movies, UNNEST(genres) AS genres

)

GROUP BY genres

ORDER BY rating DESC;

**Quiz**

OLTP queries are simpler than OLAP queries.

MapReduce can be used to create the text index (e.g. inverted index) of a Web search engine.

Cloud database systems are NOT always cheaper than on-premise ones.

The transfer of funds between two bank accounts is an example of an **OLTP** query.

Computer instances participating in database clusters use shared memory. (FALSE)

An index-only execution is only possible using clustered indexes. (FALSE)

Replication can help to increase the read queries but slows down the write queries ⇒ throughput of a database is undetermined

The ring topology in Cassandra is used to support sharding but not replication

fastest possible key-value retrieval → choose Cassandra

store and query a collection of complex (i.e. with nesting) JSONs → choose MongoDB

Ex. Supplies(s\_no, s\_name, s\_city, p\_no, p\_name, p\_color, quantity, price)

SELECT \* FROM Supplies

WHERE s\_no = 10 AND p\_no = 1001 AND color = 'Yellow' AND quantity > 50

PRIMARY KEY ((s\_no, p\_no), (quantity, color)) BUT NOT PRIMARY KEY ((s\_no, p\_no, quantity, color))

because quantity is a range query.

SQL Basics

* vertical partition with interested columns only
* horizontal partition with partial rows(B+ tree OR multiple tables with 1 total view OR 1 table with multiple views)

ORDER BY field\_name DESC/ASC [NULLS FIRST] IN or EXISTS: *find name of sailors who’ve reserved boat #103*

LIMIT # [OFFSET #] used for paginating LIKE. ‘\_’ = any 1 char. ‘%’ = 0 or more arbitrary chars

LIMIT 5 OFFSET 2 (get row 3-8 inclusive) SET search\_path TO [schema\_name]

UNION removes the duplicates. UNION ALL keeps duplicates (UNION tables must have same columns)

EXCEPT = set difference (sailor who reserved a red boat but never reserved a green boat)

ANY, ALL: select \* from Sailors S where S.rating > ANY (select S2.rating from Sailors S2 where S2.sname = ‘Horatio’)

NULL = unknown or inapplicable (Eliminated on where and aggregate). (T & null = null, false & null = false) IS NULL.

CRUD: insert into table() values/select delete from table where… update table set column = value where…

Grouping - select clause <= grouping list. having clause must be aggregate op or in grouping list

Inner Join (default) eliminates NULL on both sides. Left outer join keeps left rows even if they don’t have a match on the other table

Division: *find sailors who’ve reserved all boats*

select S.sname from Sailors S where **NOT EXISTS** (

(select B.bid from Boats B) **EXCEPT** (select R.bid from Reserves R where R.sid = S.sid))

select S.sname from Sailors S where **NOT EXISTS**((select B.bid from Boats B where **NOT EXISTS**

(select R.bid from Reserves R where R.bid = B.bid and R.sid = S.sid)))

select S.sname from Sailors S where

(select count(distinct R.bid) from Reserves R where R.sid = S.sid) = (select count(B.bid) from Boats B)

Disk and Files

* HDD seek time (move arm to position disk head) I/O dominates, SSD transfer time (move data to/from disk) dominates

Index I(k) = data record with key k OR RID of the data with key OR list of RIDs. (index might slow down update and insert)

Primary based on key. Secondary otherwise. Unique index: candidate key.

Clustered index: range selection on clustered index will find its start point and go to the next leaf node until end point.

* secondary index on a 1ͨ000ͨ000 page primary data file. Each page holds 50 recordsͥ. The secondary index has RID lists with average key = 5. LookupByKey ⇒ unclustered secondary index, # data page I/Os = 5. clustered index, # data page I/Os = 1

ISAM (indexed sequential access method):

create leaf, index then overflow page. delete ⇒ remove entry form leaf/overflow page

B+ tree: insert/delete log\_f(N) height balanced. f = fanout. d<=m<=2d for each node. d=order of the B+ tree. (equality + range)

insert: add new K to leaf. If leaf is full ⇒ split it into two and **copy up the index**. If the parent is full, **push up the index**.

delete: remove K in L. If L < d, borrow sibling S and **update** index. Otherwise, merge L + S and **delete** index

Hash-based index: (equality)

# primary pages fixed. h(k) mod N = page to store k (N = # of pages) not balanced.

Index Choice on 1 table: equality ⇒ hash range ⇒ B+ tree

* WHERE 1: select \*, range, equality with duplicate value ⇒ clustered if not PK
* WHERE 2+: 2 equ=composite or 2 index; 2 range=clustered composite; 1 equ + 1 range = clustered <equ, range>
* index-only query. Index on <E.dno> select E.dno, count(\*). B+ index on <E.dno, E.sal> select E.dno, MIN(E.sal)

Index Choice on Joins

* Index Nested Loop Join (cluster inner table like E.dno **if not primary key**)
* Sort Merge Join clustered B+ tree on both.
* Hash Join: hash one table, no index needed
* select E.ename, D.mgr from Emp E, Dept D where **D.dname(index)** = “Toy” and **E.dno(clustered index)** = D.dno.

NoSQL

Use VALUE to skip the wrapper. Ex. name of customer with highest rating.

select value c1.name from customers as c1 where c1.rating = (select value max(c2.rating) from customers c2)**[0]**

Nested select: select value {“name”: c.name, “orders”: (select VALUE o.num from orders o where c.cid = o.cid)}. unnest

Ex. SELECT DISTINCT VALUE o.custid FROM orders AS o WHERE **SOME/EVERY** i IN o.items SATISFIES i.price >= 25

* exclude items without any values. some and every = every and array\_count() > 0

Grouping & Aggregation: group by field GROUP AS g array\_max(select value rating from customers)

Missing: MISSING (no header); NULL(header with null); UNKNOWN (missing or null)

CASE WHEN o.ship\_date IS MISSING THEN “TBD” ELSE o.ship\_date END

Rank: with gap SELECT … **rank**() over (order by field) as rank\_name FROM … order by rank\_name. Without gap: **dense\_rank**()

rank() over (partition ) will display rank inside the partition group. Ex. group by dept\_name and rank by GPA for each

SELECT id, dept\_name, rank() over (partition by dept\_name order by GPA desc [NULLS last]) as dept\_rank

From dept\_grades order by dept\_name, dept\_rank

Ex. top-N result: select \* from (select ID, rank() over (order by GPA desc) as s\_rank) where s\_rank >= 3

Ex. select ID, row\_number() over(order by GPA desc) as s\_row from student\_grades order by s\_row

* rank over is done after applying grouping/aggregation. multiple over clauses can be in one select
* Other rank-like functions: row\_number, cume\_dist = cumulative distribution, percent\_rank, ntile = divide into n buckets

Window:

Ex. daily\_sales(date, value) (bold text = current row +/- 1 row)

select date, avg(value) over (order by date rows between **1 preceding and 1 following**) as daily

from daily\_sales

get data from beginning to current ⇒ **rows unbounded preceding** OR **rows between unbounded preceding and current**

range between 10 preceding and current row ⇒ **rows with values [current row - 10, current row]**

Ex. select account\_number, date\_time, sum(value) over (

partition by account\_number order by date\_time rows unbounded preceding) as balance

from transaction order by account\_number, date\_time

Cross-Tab:

Ex. 8 groupings: 3+3+1+1: select **coalesce**(item\_name, “all”), color, clothes\_size, sum(quantity) from sales

group by **cube**(item\_name, color, clothes\_size) order by item\_name, color, clothes\_size

Ex. 4 groupings = (item\_name, color, clothes\_size) + (item\_name, color) + (item\_name) + ()

select item\_name, color, clothes\_size, sum(quantity) from sales group by **rollup**(item\_name, color, clothes\_size)

Grouping to get cell value (1 = null value = all. 0 = other cases)

SELECT item\_name, color, clothes\_size, sum(quantity), **grouping**(item\_name) as item\_name\_flag, **grouping**(color) as color\_flag, **grouping**(clothes\_size) as size\_flag FROM sales group by cube(item\_name, color, clothes\_size)

ACID (Atomicity = All or Nothing, Durability persist on failure; Consistency; Isolation multiple transactions)

Ex. reading uncommitted data (dirty reads)

T1 R(A) SL,**W(A)** XL R(B),W(B), **Abort/ROLLBACK**

T2 **~~R(A)~~**~~,W(A),C~~[fail to read A because there is a XL (exclusive lock)]

Ex. unrepeatable reads: both cannot W(A) because 2 transactions have SL in A ⇒ dead lock

T1 **R(A)** SL **R(A)**,~~W(A),C~~ [fail to write A because SL by both]

T2 R(A) SL,**~~W(A)~~**~~,C~~[fail to write A because SL by both]

Ex. overwriting uncommitted data: XL ⇒ cannot access A

T1 W(A) XL **W(B)**, C

T2 **~~W(A)~~**, W(B), C [fail to write A because XL in A]

Isolation level: Serializable > repeatable read (re-read gets the same result) > read committed > read uncommitted

Recovery between checkpoint C1 and C2.

T before C1 ⇒ ignore. Completed T after C1 ⇒ redo. uncompleted after C1 ⇒ undo

Abort a transaction

abort Ti ⇒ all actions of Ti must be undone

if Tj reads a value written by Ti ⇒ Tj must also be aborted

avoid cascading aborts by releasing transaction’s locks only at commit time (strict 2PL protocol)

if Ti writes an object, Tj can read it only after Ti commits

to undo the actions, DBMS keeps a log where every writes are recorded

Dataframes

Ex. sailors = pandas.read\_csv(“path/of/csv”)

sailors.dtype ⇒ data type. if NaNN in row, data object is returned

sailors.describe() ⇒ for each numeric column, display statistics like mean, min…

selections .head(3)/tail(3) sailors[sailors.age < 10]

projections sailors[[“sname”, “ratings”]] sailors.assign(dogage = sailors.age / 7)

joins pandas.merge(reserves, boats, left\_on=”bid”, right\_on=”bid”, how=”left”)

unions boats = pandas.concat([boats, moreboats], ignore\_index = True)

grouping bymajor = sailors.groupby(“major”) ⇒ returns group object

bymajor.size() ⇒ return size of each group

bymajor.count() ⇒ return size of each group for the other columns (sid, rating…)

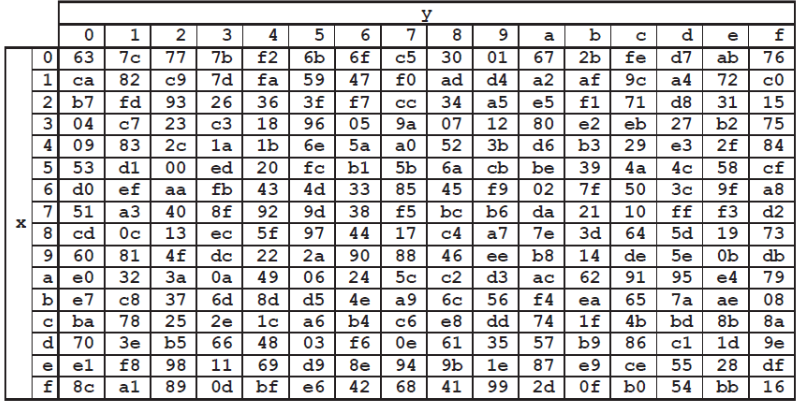
bymajor.rating.max() ⇒ return max rating for each group

bymajor.max()[[“rating”, “age”]] ⇒ return multiple max for each group

bymajor.agg({‘rating’:[‘min’, ‘max’], ‘age’: [‘count’, ‘mean’, ‘std’]})

For rating, get min and max. For age, get count, mean and std.

bymajor.rating.max().sort\_values(ascending = False) [:3] ⇒ get top rated group



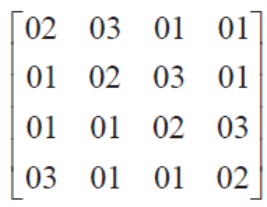
SubByte with s-box.

ShiftRows: 1st unchanged, 2nd left shift by 1 byte, 3rd left shift by 2 byte, 4th left shift by 3 byte

MixColumn: cell = row in fixed A \* column in current B

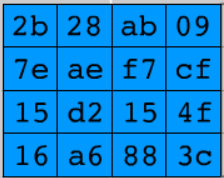
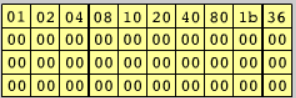
a \* 0x02 = shift left by one. If leftmost bit = 1 before shift, result applies XOR (0001 1011)

a \* 0x03 = (a \* 0x02) XOR (a)



AddRoundKey. Rcon(8) = 02 00 00 00 as column

Ex.

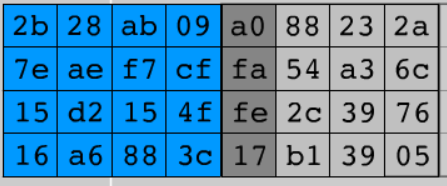
with Rcon

Round 1

1st column: 09 cf 4f 3c → up shift by 1 → Substitute → XOR 2b 7e 15 16 XOR **01 00 00 00** → a0 fa fe 17

2nd column: a0 fa fe 17 XOR 28 ae d2 a6 = 88 54 2c b1

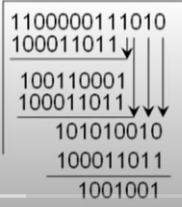
…



Round 2

1st column: 2a 6c 76 05 → up shift by 1 → substitute → XOR a0 fa fe 17 XOR **20 00 00 00** → f2 c2 95 f2’

polynomial multiplication: 0x36 \* 0x93 = 1 1000 0011 1010 XOR gf(2^8) 1 0001 1011 →



RSA key generation: get p,q(both prime, p!=q) ⇒ n = p \* q ⇒ Φ(n)=(p-1)(q-1) ⇒ select e where gcd(Φ(n),e)=1 AND 1<e<Φ(n) ⇒ select d where d\*e mod Φ(n) = 1 AND d < Φ(n).

Public key = {e, **n**}, Private Key = {d, **n**} ⇒ C = M^e (mod n) where M < n ⇒ M = C^d (mod n)

Euler Totient:

ø(p) = len([i if relativelyPrime(i, N) for i in range(0,N+1)])

N is prime, ø(p)=p-1.

ø(p\*q)=ø(p) \* ø(q) if p, q are relatively prime

gcd(a,b) = ax+by=1. MI(multiplicative inverse) of x mod y = a mod x (to get positive). MI of y mod x = b mod y

1 = 138\*12 + (-5)\*331. gcd(a,b) = ax+by = 1, MI of 12 mod 331 = 138. MI of 331 mod 12 = -5 mod 12 = 7

(show if (a^ Φ (n)) mod n =1 when a is relatively prime to n. for each n. compute Φ (n) and for each element a in Φ (n), show)

Confidentiality symmetric. C= E[M] with S\_ab, M = D[C] with S\_ab

asymmetric. C=E[M] with public B. D[C] with private B for A sends to B

Message symmetric: A: H =hash[M], C=E[H] with S\_ab → B: **H’**=hash[M], **D’**=D[C] with S\_ab.

authentication asymmetric. A **sign**  → B decrypt

User authentication exchange digital certificates. A’s DC = [A + A’s public key] is signed by private key of CA

Secure communication:

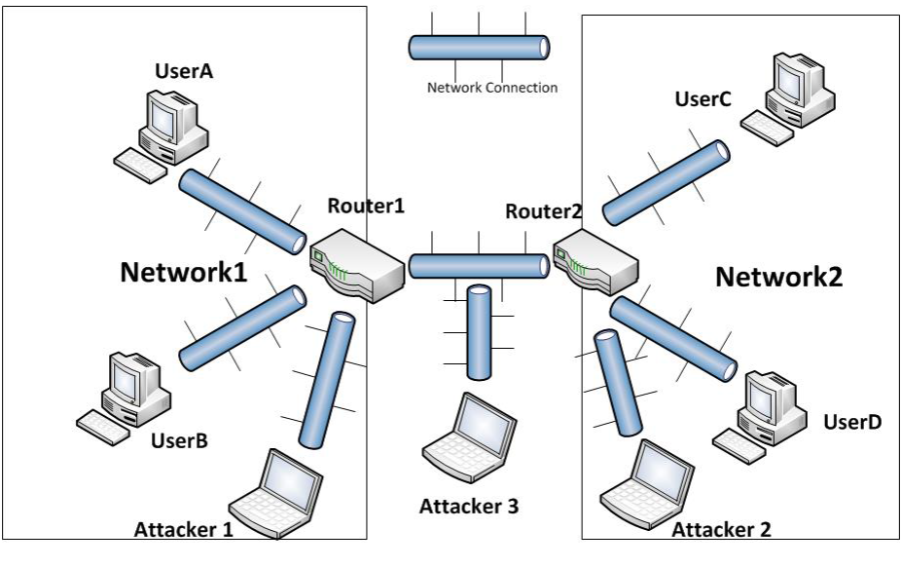
Alice — IP/TCP handshake → B

verify certificate with CA’s public key ← digital certificate (public key signed by CA’s private key) of B —

extract B’s public key from certificate — session key encrypted with B’s public key →

* end-to-end confidentiality ⇐ use public key to exchange symmetric session key
* forward secrecy ⇐ use different session key

201



Attacker1 on UserA and UserB in Network1?

packet sniffing and spoofing. Attacker 1 is on the same local network as UserA and UserB. If UserA or UserB is broadcasting to get an ip address or mac address, Attacker 1 can respond with a fake ip address and pretend to be someone else.

ARP poisoning. Attacker 1 changes the configuration file of UserA or UserB.

Denial of Service. Attack 1 can create tons of requests and make Router1 unavailable.

Application-level security can UserA use to secure (confidential) its communication to UserD?

HTTPs (secure communication between A and D…) with tcp or https protocols preinstalled

SSH for remote access (Private/Public key pairs makes sniffing virtually useless)

Secure (confidential) their communication between Router1 and Router2 against Attacker3

IPSec to encrypt data packet between Router1 and Router2 (wrapping the whole data packet with a new ip header)

Only can see the ip address of the destination IPSec ⇒ no one can view the data

The user has no control over the IPSec.

VPN tunnel

physical NIC (network interface card) ⇒ create virtual NIC by authentication in an encrypted process with UCI public key ⇒ traffic through UCI will use virtual NIC and other ones will use physical NIC (because UCI don’t allow access outside of the UCI to limit traffic?)

VPN client website needs to check HTTPS to ensure the website is not putting a malicious public key, and directing the user to send data to a malicious computer

Encrypted public key with VPN (UCI.edu)

Encrypt symmetric session key using VPN public key

VPN server decrypts using VPN private key

With IPSec and VPN between R1 and R2, is it secure between User A and D?

No, it’s not secure before/after the IPSec transaction. Like between User A and R1 and between User D and R2 (spoofing, ARP poisoning, DoS)

User C and B can form secure communication with HTTPs, which will be confidential to Attacker 1 and 3.

UserA and UserD

use VPN tunnel (network level protocol) to secure (confidential) their communications from Attacker 1-3. UserA sends UserD the certificate. User D verifies the certificate

Attacker 1 is trying to launch a ping flooding attack by generating the ICMP Echo Request spoofing the IP address of UserC. Which router should have acted first to stop this attack?

Router 1 should use packet filtering to filter out and block packets that contain conflicting source address information.

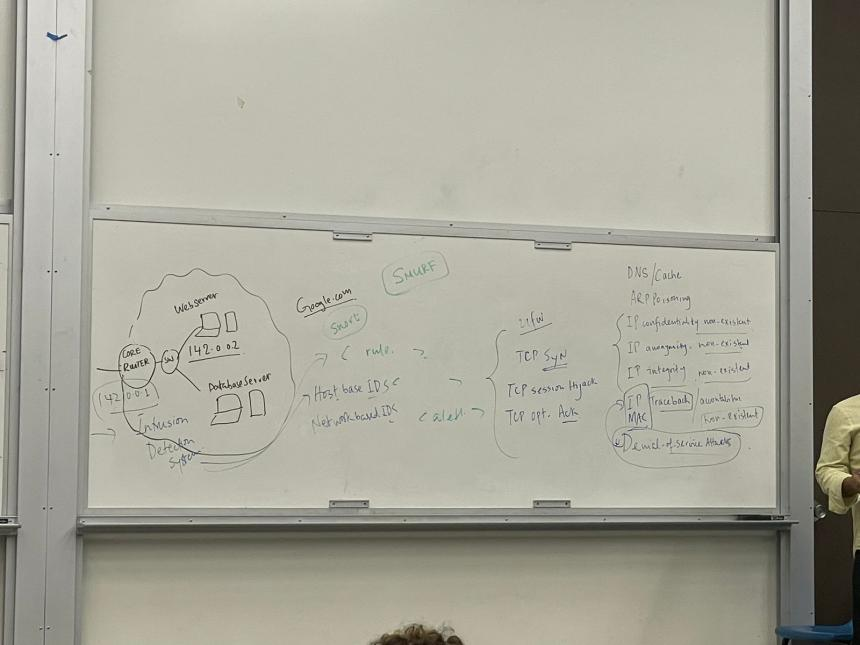
Attacker 2 has got control over the web server on UserC. Elaborate how the attacker can use it to propagate malware to UserA and UserB.

Image files for the user to download…

* Once the endpoint is compromised, everything is possible for attackers.

TEXTBOOK NETWORK





Browser:

check if you have access to the website you typed in by the hash/cache?

get the IP address and dns at Default Gateway

**DNS/Cache Attack**

1. modify local cache to connect to malicious website/computer
2. modify dns to go to a malicious computer

fix

IP address must maintain its integrity

ARP to translate IP addresses into MAC addresses.

**ARP Poisoning Attack**

Over a Local Area Network (LAN)

send malicious ARP packets to a default gateway⇒ change the pairings in its IP to MAC address table ⇒ spoof other machine

Problem with IP

**IP Confidentiality**

IP packets can be looked at in plaintext when passing ip packet without SSL

**IP anonymity**??

**IP MAC Integrity**??

**IP Traceback** not existing

send a fake ip packet and no way to traceback where it was sent from

Smurf Attack (Fraggle attack uses UDP protocol instead)

sending internet control message protocol (ICMP) packets and cause denial of service (**DDoS**)

TCP Session Hijacking

an authorized user gains access to a legitimate connection of another client in the network. The attacker can read and modify transmitted data packets, as well as send their own requests to the addressee.

TCP OPT. attack

ddos ⇐ increasing the number of requests ⇐ sending fake packets

Routering

ISP1 (interface …)

ISP2

collect data from customer packets

Line Router

gets the address of the website

Application Layer HTTP, NNTP, Telnet, FTP, and so on

Transport Layer Security TLS

Transmission Control Protocol TCP

Internet Layer IP

Packet-filtering firewalls in ISP

filters IP packets based on source and destination IP address, and source and destination port (if the source port doesn’t match IP address, drop the packet)

* ISP has the lookup table to get IP address by source port
* network based firewall and host based firewall

(Proxy firewalls

acts as a gateway between internal users and the internet. Proxies = gateway applications used to route internet and web access from within a firewall)

IDS = Intrusion detection system

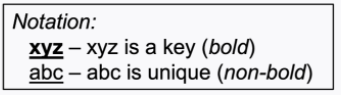
detect suspicious activities → generates alerts for a security operations center (SOC) for further investigation/actions/remediation

also has host-based and network-based system

Relational Database: #row = cardinality, #fields = degree or arity

Schema = name of relation + name + type of column. EX. Students (sid:string, name:string)

Primary Key Constraints (unique identifier): superkey (if it contains a candidate key), primary key, candidate keys

Foreign Key (by default no action) on delete cascade, on delete set null/default 

Total Participation NOT NULL + Foreign key

Weak Entities: NOT NULL + Foreign key + on delete cascade

FOREIGN KEY (eid) REFERENCES Employees (eid) ON DELETE CASCADE

IsA Hierarchies: Hourly Emps & contract\_Emps –IsA (covering not disjoint) — Employees

1. Recommended Delta 3 relations: Employees (all employees partially reside), Hourly\_Emps, Contract\_Emps

Costly if frequently access the children with parent attributes. Good for maintaining PK uniqueness, FK references

1. Union of table: Employees (only Employees), Hourly\_Emps & Contract\_Emps

Redundant data of overlapping instances, okay for disjoint tablesFor all employees queries, it needs to union all three tables. Bad for PK uniqueness (it needs to check all three tables), FK references. For covering constraints, the only-employee should not exist

1. Mashup table: Emps(kind, common\_attibutes, children\_attributes)

SQL Views used for derived attributes, simplifying/eliminating join paths, beautifying “Mashup table”

* CHAR padded with extra spaces and VARCHAR padded without extra spaces
* For 1 to N relationship, the relation table can be integrated to one of the entity (N end)

Functional Dependency: X depends Y (X → Y) holds R for all instances (one instance can only violate FD not imply)

K = candidate key for R means K → R and K is minimal superkey for R

Closure of

Armstrong’s Axioms

| Reflexivity:  Augmentation:  Transitivity: | Union:  Decomposition:  Partial: |
| --- | --- |

FDs & Redundancy. for R(ABC), no FDs hold => no redundancy

Normal forms

1NF = flat (if all attributes are atomic). X = prime attribute (X is part of (candidate) key)

2NF = 1NF + NO non-key attribute is partially dependent on a candidate key

3NF = 2NF + no transitive dependencies to non-prime attributes

BCNF (no non-trivial FDs). NOT BCNF if multiple candidate keys overlap

. R is BCNF if the only non-trivial FDs are key constraints

Ex. Supply2(sno, sname, pno) with sno→sname, sname→sno. candidate key: sno+pno and sname+pno

FIX: Supplier2(sno, sname) + Supplies2(sno, pno)

* not always possible to get BCNF. No non-trivial FDs means no dependencies between existing attributes and it’s BCNF

Dependency-Preserving: every previous FDs has a home (hold on one of the decomposed tables)

* Dependency preserving doesn’t imply losslessness. Ex. ABC with A→B is decomposed into AB and C
* Some are not dependency preserving into BCNF. Ex. R(CSZ) with CS → Z and Z → C

Lossless Join Decomposition: Decompose R into X and Y is lossless-join if joining X and Y = R

* lossy join usually gives more data than original if X, Y doesn’t overlap on a key

Relational Algebra (operational)

**Selection** : selects rows that satisfy the condition (filter row). Ex,

**Projection**  removes attributes not in the projection list (filter column). Ex.

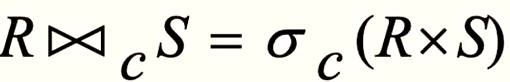
**Set-difference** in relation1, but not in relation2

**Union**  in relation1 and/or in relation2: same column + remove duplicates

**Cross-product** combines two relations. Ex. S1 x R1

Renaming operator:

Ex. is the same as

Condition Joins 

Theta-join: , same fields as the cross-product

Equi-join , only one field of equality + other fields like cross product

Natural join an equijoin on all commonly named fields

Division. Table A(x,y) and B(y), A/B contains the x such that for every y is in B Ex. find sailors who have reserved all boats

Optional

Relational Calculus (declarative) relation instances = schemas of input are fixed + schema for result is also fixed

Tuple Relational Calculus includes all tuples t with attrlist that cause formula P(t) to be true

t on the left of | must be the only free variables of formula P(...)

Atomic formula:

unsafe queries have an infinite number of answers. Ex.

relational algebra ⇐⇒ a safe query in DRC/TRC

Relational Completeness = every query is expressible in relational algebra/safe calculus

Free and bound variables: quantifier such as is bound variables, others are free

Ex. Sailors(sid, snake, rating, age), Reserves(sid, bid, date), Boats(bid, bname, color)

Find sailors with a rating above 7

Find ids of sailors who are older than 30 or who have a rating under 8 and named ‘Horatio’

| Find names of sailors who’ve reserved a red boat (with joins)    Find sailors who’ve reserved a red or/and a green boat: Cannot use s,r, b1,b2 because 1 reservation only matches 1 boat    Find names of sailors who’ve reserved all boats (universal quantification) | Find the names of sailors who’ve reserved all Interlake boats.      Print the names and ages of sailors who have reserved all of the non-red boats  {t(sname, age) | SOME s IN Sailors (s.sname = t.sname ^ s.age = t.age ^ ALL b IN Boats (b.color != 'red' ⇒ SOME r IN Reserves ( r.bid = b.bid ^ r.sid = s.sid)))} |
| --- | --- |

Expressions and Strings

AS for renaming fields in result. LIKE for string matching. ‘\_’ = any 1 char. ‘%’ = 0 or more arbitrary chars

SELECT DISTINCT s.sid AS ID

FROM Sailors S, Reserves R

WHERE S.sid = R.rid AND s.sname LIKE ‘\_r%’

ER Diagram Sum

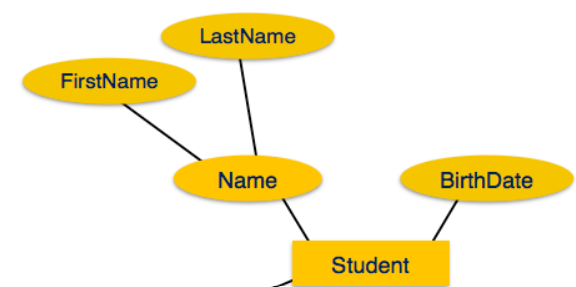
| Required attribute — Multivalued attribute  Key attribute  Derived attribute |  |
| --- | --- |
| Weak entity + total participation |  |

* All multivalued attributes is a separate table
* One-to-many can be reduced from 3 tables to 2 tables. Many-to-many cannot be reduced. One-to-One + 1 total participation can be reduced to 1 or 2 tables. One-to-One without total participation must have 2 tables at least

ER Diagram Representation Sum

Attributes

Composite attribute made of multiple attributes

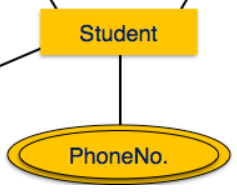


Derived attribute = values are derived from other attributes present in the database.

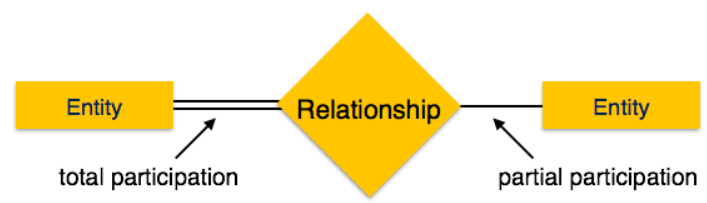
ex. age can be derived from data\_of\_birth.

Multi-value attribute contains more than one value for one attribute.

ex. a person can have more than one phone number, email\_address, etc.

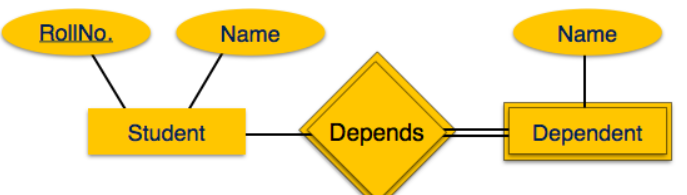


Total Participation = Weak



* N to N relationship with total participation cannot be represented in a table creation

Weak Entity and weak relationship



[ Student ] —1— <<Depends>> **—N—** [[ Dependents ]] means at least 1 is mandatory.

Dependens is double-lined because it’s the identifying relationship of the weak entity

Cardinality constraints (1-N relationship) M:N is default if omitted

**boldness** means the it’s mandatory

Ex. Professor —1—<head> —N— Department means department has exactly one professor head but professor can be the head of many departments

Create Table head(

pid CHAR(11),

did INTEGER,

Primary Key (did),

Foreign key (pid) References Professor,

Foreign key (did) References Department

)

Create Table Department2(

did INTEGER,...,

head\_pid CHAR(11)

Primary Key (did),

Foreign key (pid) References Professor

)

Ex. between Employees and Reports\_to

—supervisor 1 —

—subordinate N—

Ex binary RS: student — <enroll\_in> — courses

Weak Entity = identified uniquely only by considering the primary key of some other entity

Owner entity and weak entity must participate in a one-to-many relationship

Weak entity must have total participation in this identifying relationship set (delete owner entity will delete weak entity)

Dependent identifier is unique only within the owner context. Primary key is (owner\_key, weak\_key)

Ex Employee —1—<< Policy>> —N—Dependents

Ternary Relationship: relationship key <= entity keys

Patient —M— Prescribe —1—Doctor

|

N

|

Drug

A given patient + drug will be given by only one doctor(1)

A given patient + doctor may associated with several drugs(N)

A given doctor + drug may be associated with several patients(M)

IsA hierarchy

ER attributes, including key, are inherited

Covering constraints: B and C inherits A with covering => every A is either B or C

Overlap constraints: … with overlap => A can be B or C or both (B (not disjoint from) C)

Disjoint = if or not the subtypes are disjoint from one another (no overlap between subclass)

Aggregation

allows us to treat a relationship as an entity to participate in (other) relationships

Aggregation: the aggregated relationship are independent from the outside entity

Ternary relationship: the “aggregated” relationship are dependent on outside entity

More advanced

Multi-valued attributes: doubled circle

ex. [person] — ((phones))

Create Table Person\_phones(ssn, phone) with multiple rows for the same ssn

Composite attributes: attributes has multiple attributes (sub attributes can be mandatory)

ex. a person has an address which consists of street, city, zip…

Create Person(ssn, address\_street, address\_city, address\_zip)

Mandatory attributes: node between entity to attribute at the attribute end

ex. every person must have a name [person] —**\*** (name)

Create Table Person (name varchar(20) NOT NULL)

Derived (vs. stored) attributes: dashed circle

ex. employee’s age will change, so we store their date of birth and update accordingly

Create View age(ssn) As Select (TodayDate() - e.BirthDate()).ToYear() From Employee e Where e.ssn = ssn

Ternary Relationship can be translated to an entity with three relationships to the other three entities