DS 551 Midterm Notes.

Definitions:

- Lec1: Big Data
  - 3Vs:
    - Variety: data type variety
    - Volume: magnitude of stored data
    - Velocity: rate of flowing data in and out to users
  - Units of measurements:
    - Byte = 2^0
    - KB = 2^{10}
    - MB = $2^{10} \cdot 2^{10} = 2^{20}$
    - GB = 2^{30}

- Lec2: Firebase, ~~REST~~ and Web API.
  - CRUD: - Create. - Read - Update - Delete.
    - Create: Post / Put.
    - Read: Get
    - Update: Patch
    - Delete: Delete.
  - JSON: JavaScript Object Notation
    - value = str | num | obj | array | ~~true~~ true | false | null
    - objet = { string : value }.
    - array = [values]
  - CURL: - PATCH: Requires JSON obj; inserting
    - Put: ~~create~~ overwrite / create.
  - Filtering Data:
    - orderBy
      - L: equalTo / startAt / endAt.
      - L. limitToFirst / limitToLast
      - L. print
    - ordering:
      - b. null → false → true → num → str → obj

Examples:

- Valid Json?
  - [] ✓
  - {} ✓
  - {[]} ×
  - [{}] ✓
  - {"name": john} ×
  - {"name": "John"} ✓
  - {"name": 25} ✓
  - "name" ✓
  - 25 ✓
  - {25} ×
  - [25] ✓
  - True ×
  - true ✓
  - TRUE ×
  - Null ×
  - false ✓

No.

Date

Lec 3 Storage Systems

- Primary : Cache , Memory
- Secondary: Hard disks , SSD
- Tertiary : Tape, Optical Disk .

Speed ↑ Cost & Energy Consume. ↓

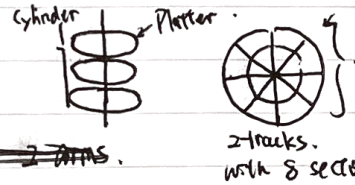
- Access Time : time taken before drive is ready to transfer data .
- Storage Device Characteristics:
  - Capacity
  - Cost : price per byte .
  - Bandwidth : how many bytes/sec can be transfred .
  - Latency : Time elapsed waiting for response .

- Completion Time = Latency + Size/Bandwith .
- Workload Completion depends on :
  - Workload Size (how much work)
  - Technology type
  - Operation type (read / write)
  - Access pattern (random/ sequential)
- Access pattern :
  - Sequential : Data to be accessed are located next to each other .
    ↳ faster for Big size .
  - Random : located randomly on device .
    ↳ faster for amount of small pieces .

- Magnetic Recording .
  - Write head : Applies Electric Current to change direction of mag field .
  - Read head : Senses direction

- Tape.
  - Linear Tape: Data recorded on parallel tracks
  - Performance Characteristics: high-latency, low-cost
    - Suitable for low frequency ~~& High~~ access.
    - " for high volume data.
    - " for sequential access.
  - File system:
    - 2 partitions { 1. Metadata, fast read. 2. Blocks of data.
    - Directory in XML

- Hard Disk Drive.
  - Disk System:



- 1/more magnetic disks
- each with 2 surfaces. ~~and 2 arms.~~
- each surface ~~head~~ has 1 head read/write.
- Each surface has number of tracks.
- each with number of sectors (blocks)
  - written all / nothing.

- Completion time: ~~Seek time~~
  + Seek time: $\frac{1}{3}$ max seek time.
  + Rotation time: $\frac{1}{2}$ full rotation time.
  + Transfer time: Data Size / Transmition Bandwidth.

- Actual Bandwith = $\frac{|W|}{t}$
  - where |W| size of data.
  - t completion time for W.

- SSD Solid State Drive.
  - Performance Characteristic:
    - Lower Power Consumption than HDD, but more expensive and less capacity
    - Limited Lifetime, can only write a limited number of times.
    - Better Latency with no Ts and Tr.
    - Better on Random access.
    - Better on Reading than writing.

HDD size:
- 256 cylinders.
- Each cyl has 16 heads.
- ~~Each head/platter has 64 tracks~~
- Each track has 64 sections.
- Sector size: 4kb.

$256 \times 16 \times 64 \times 4kb = 1GB$.

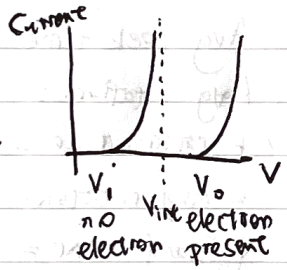
HDD transmition:
- 4TB with 4 platters, 8 heads.
- 4096 b for each sector.
- Transmition Bandwidth: 100 mb/sec.
- Max seek time: 12 ms.
- RPM: 10,000

- file size: 100 sectors. x 4 kb = 400 kb.
- Avg seek time: $\frac{1}{3} \times 12 = 4$ ms.
- Avg rotation:
  - rotation time = $\frac{60 \times 1000}{10000} = 6$ ms
  - Avg rotation = $\frac{1}{2} \times 6ms = 3$ ms
- ~~Tr~~ Transfer time = $400kb / 100 mb/sec = 2^2 \times 2^{-10}$ sec

Sequential vs. Random:
- Sequential access time of 10 MB.
  $t = ~~avg~~ sec\ T_s + T_r + T_t$
- Random access time of 10 mb
  $t = 2500 \times (T_s + T_r + T_t$

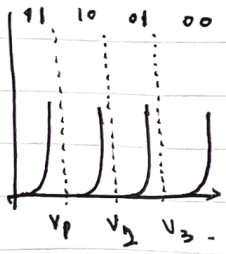
- ~~Comp~~ File System:
  - Chip → dies → planes → blocks → pages → cells.
  - cells are made of floating gate transistors.
  - Page is smallest to transfer, read, write.
  - Block is smallest to be erased.

- How floating gate works. (SLC)
  - Floating gate "floates" inside oxide isolated from other components.
  - Electrons are attracted or repelled from FG by applying voltage.
  - States of data:
    1 : if electron is present.
    0 : if not.
  - Reading:
    - By apply V_{ins}, see if current is detected.
      1 if detected
      0 if not.



  - Write & Erase.
    - Write 1 → 0 : Applying positive charge will attract electrons.
    - Erase 0 → 1 : Applying much higher negative charge will repel
      ↳ which may affect surroundings.

- MLC:



SLC vs. MLC.

SLC:
- Less complex
- Faster.
- More reliable
- Less storage.
- More costly.

Lec4 File System.

- Files & Directories.
  - Files are stored in blocks.
  - Files organized into directories.
- CRUD:
  - Create: open(), write()
  - Read: open(), read(), lseek()
  - ~~write~~ Update: write(), lseek(),
  - Delete: Unlink()
- Inode:
  - Created when creating files
  - Recording meta-data.
  - Store locations of blocks & replicas.
- Output of Stat():
  - Access Permissions: Owner Group Else.
    rwx rwx r--
    0 111 111 100
    0 7 7 4
  - ~~Sector Inode~~ Sector # = (INode Address + inumber × inode size) / sector size.

Inode: INumber → location of inode.

~~offset~~ INumber = 32.

offset Address = 3×4k + 32×256

= 20k.

Sector # = 20k / 512 = 40.

Lec 5. HDFS

- Hadoop: Large scale distributed & parallel batch-processing infrastructure.
  - Large scale: Handle large data and computations.
  - Distributed: data distributed among multiple machines.
  - Batch processing: series of jobs without human intervention.
- Distribution.
  - A single namenode storing meta-data
    - Directory hierarchy
    - Attributes eg. permission, time, size
    - Mapping to data nodes.

- Datanodes:
  - Data are distributed among multiple data nodes.
  - 2 to 3 replicas.
  - Each replica can recieve the request.
- Secondary NameNode.
  - Maintain checkpoints of namenode.
  - For recovery

- Block Size: 128 MB.
  - Good for fast streaming read of data by sequential pattern.
  - Reduce metadata required.

- Client to HDFS.
  1. Ask NameNode:
     - Reading: getBlockLocations()
     - Writing: addBlock()
  2. Ask DataNode:
     - Reading via readBlock()
     - Writing via writeBlock().
- ~~One~~ getBlockLocation():
  - Input: File Name,
    offset (where to start read)
    Length (how long)
  - Output: Located Blocks.
    (including offset and data node)

- ~~Write Block~~:
- Procedure of writing
  1. To NameNode:
     ~~or addBlock(str: src, str: clientName)~~
     1) create(path, permission, client, flag, replicas, size)
     2) addBlock(↓ , ↓)
        ↳ LocatedBlock × 1
        ↳ Block, offset, datanodes with replicas
  2. To DataNode: block to be written
     1) writeBlock(block, rest of data nodes, current datanode).
        - Block break down to packets (64kb)
        - Client send the packets to list of datanodes (Not replicas)
        - Datanodes send packet to Next datanode
        - Acknowledgement → packet removed from ack que (packets)
     2) Acknowledge NameNode to update.

Dec 6 XML & XPath.

Definition: XML a syntax for data.

Benefits: • Can Translate Any Data to XML
• Over Web
• Into App.

Terminologies:
- Tags: < start / End >
- Elements: <start> </end>
- Single Root element
- Attributes: <start att="" >
  ☆ Must be quoted " or '

DTD: Document Type Definitions.
Definition: Markup for describing schema
well-formed XML: tags are correctly closed
valid: if XML has a DTD and conforms

XPATH:
- // : finding descendants eg. //employee (find all elements with tag)
- [int]: select child by index. eg. /root/child[1] (start from 1)
- text(): All text nodes
- * : All element nodes
- node(): All nodes.
- @: attributes eg. @price
- contains(@element, 'text')
- and, or, not()
- | : alternatives

DTD:

```
<!DOCTYPE company [        (root name)
  <!ELEMENT element_name (CONTENT)>
  ]      :
]>
```

CONTENT:
- RE over other elements
- Text only: #PCDATA / #CDATA.
- EMPTY
- ANY
- Mixed

RE:
- Optional eg. (firstname?)
- Multiple eg (phone*)
- Alternation eg (phone|email)

Lec 7: ER Models

Definitions:

- ER: Entity Relationship
- Entity: real-world objects described using attributes
- Attribute: each has an atomic domain
- Entity set: a collection of similar ~~objects~~ entities
- Relationship: subset of $A \times B$.

  A: {a, b, c}

  B: {1, 2, 3}

  R: {(a, 1), (b, 2), (c, 3)}.

Relationships:

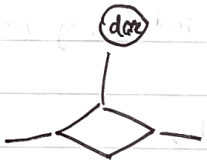
- One-to-one: both tables can have only one record on each side of the relationships
- one-to-many / many-to-one:

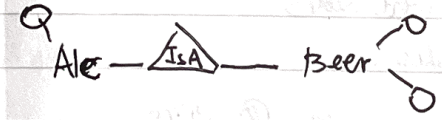
Only one record in Table A may related to any number of records in Table B.

- Many-to-many: multiple records in both tables may be related to any number of records in the other table.
- Binary vs. Multiway: Can convert between.
- Attribute: Attribute of relationship

What does Arrow means?

- End of the Arrow is determines the head of the Arrow.





Subclass:

Definition: Special case, fewer cases, more properties.

ER.	Object-Oriented
- Only one class	- Multiple classes
- All attributes together	- stored seperate.

Constraints:

Definitions: Rule that applies to ~~a table~~ all rows of a table at all times

- Keys: ~~one~~ unique attribute ~~(s)~~ identifier. (can consist on multiple att)
- Single-value constraint: An entity can have at most 1 for a given att or rel
- Referential Integrity constraints: refered entity must exist in database exactly once.
- Domain constraints: domain of att.
- General constraints.

Weak Entity Set:

Definition: the key att of the set comes from other entity sets.

- Causation:
  - ~~(s)~~ Converting Multiway to Binary
  - Part-of relationship

Lec 8 Relational Model

Definition: Different from ER, everything is represented with tables / relationships

	Modification cost	Stability
• Domain: Dtype.		
• Schema: Relation (att_1, att_2, att_n)	to high	high
• Instance: Collection of data stored at a moment.	Low	low.

• Tuple	element of string × int × string eg: t = ("gizmo", 19, "gadgets")	function t: A → att domains eg: t(name) = "gizmo"
• Relation	subset of string × int × string – Order is important!	A set of tuples / functions. – Att name is important!
Example	Insert t = (123, 'john', 35)	Insert Employee(123, 'john') = (35).

Transfering ER to Relational:

- Entity sets: – Set name as relation name.
  – att as att
- Relationships: – Relationship name as relation name.
  – entity att as atts, watch for conflicts
- Many to One: – "Many" Entity as table.
  – "One" as att.
- Many to Many: Not a good idea due to Redundancy
- Weak Entity sets: No need for additional table for relationship.
- Subclass:
  1. OO Approach: Each tree creates a new table. Names in Tables are distinct. eg. Product (name, cat)
     EduProduct (name, cat, ageGroup)
  2. E/R Approach: • A superclass table eg. Product (name, cat)
     • Several Subclass table. EduProduct (name, ageGroup)
     • Intersects.
  3. Null Val Approach: • One table eg. Product (name, cat, ageGroup).
     • class as att, class att as att.
     • Null for no class.

Lec 9 SQL.

- DB Modifications
  1. Insert:
     - Insert into R Values('a', 'b');
     - Insert into R(A) Values('a');
     - Insert into R (SUBQ);
  2. Delete:
     - Delete from R where A='a';
     - Delete from R;
     - Delete from R where exists (SUBQ)
  3. Update.
     - Update R. Set A='a' where B='b';

- DDL
  1. Create:
     - Create Table R(A, B);
  2. Drop:
     - Drop Table R;
  3. Constraints.
     - Create Table R(
       A CHAR(20),
       B VARCHAR(20),
       Real / float
       INT / TINYINT / SMALLINT..
  4. Key
     - Primary
     - Unique.
  5. Default:
     - Default 'a'
  6. Alter:
     - Alter Table R ADD. A CHAR(20);
     - Alter Table R DROP A;

Lec 10 Constraints

- Keys:
  - Unique and Primary:
    - Referenced att must be either
    - Null can exist in Unique, not Primary.
  - Foreign Key: Referencing relation

- Actions taken for Foreign Key Update/Delete:
  1. Default: Reject modification.
  2. Cascade: Same changes made in R.
     - Delete R tuple.
     - Update value in R.
  3. Set NULL

- Policies:
  - Cascade
  - ~~Update~~ Set Null

  > ON < Delete / Update.

- ~~Value~~ att-based Constraint: Check
- Tuple-base constraint

/Unique

create table R(a int primary key);

OR create table R(a int, primary key(a));

- With Attribute:

  create table R(a int references B(b));

- As Element

  create table R(a int, ~~refere~~

  foreign key(a) references B(b));

(a) foreign key(a) references B(b)

ON Delete. set null

ON Update ~~se~~ cascade);

Check(p <= 50),

(at the end) Check(bar = 'Joe' OR price <= 5.00)

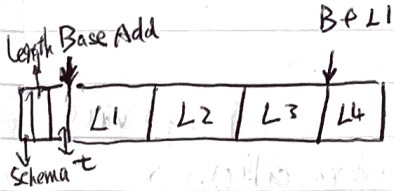
Lec 10 Views

- Virtual Views: Computed on-time.
  - Up to date.
  - Slow
- Materialized Views: Precomputed.
  - May have ~~stale~~ stale data.
  - Fast.

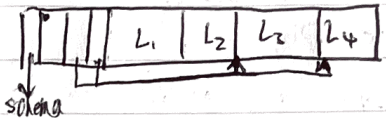
Lec 11

Representation of Data

- Fixed Length



- ~~Header~~ Variable Length
  - Place Fixed Length ~~blocks~~ records first



- Storing Records:
  - ~~Store~~ Records from right

External Sorting.

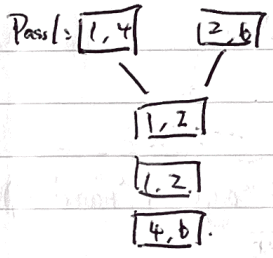
- Buffer Page: Page use to store I/O and intermediate data.
- 2-way Merge Sort

  Pass 0: sort into pairs. (1 buffer)

  Pass 1-N: [1,4] [2,6] (3 buffers)

  = Cost = $2N(\lceil \log_2 N \rceil + 1)$

  $k = \lceil \log_2 N \rceil + 1$



- M-1 way Merge Sort.

  Cost = $2B * k$

  $k = \lceil \log_{M-1}(B/M) \rceil + 1$

~~Cost I/O~~

eg: M=5, B(R)=108.

P0: $N = \lceil 108/5 \rceil = 22$, B=5.

P1: $N = \lceil 22/4 \rceil = 6$, B=20.

P2: $N = \lceil 6/4 \rceil = 2$, B=80.

P3: $N = \lceil 2/4 \rceil = 1$

Lec 12. MongoDB.

Features:

1. Document Database. {key: val} Document ≈ JSON Obj.
2. Documents stored in collections have different types.
3. NoSQL
4. Allow hierachy & array & more.
5. Scalable.

_id: unique_key.

ObjectId(): 12 byte. hex val

- 4-byte: sec since 1970/1/1
- ~~machine~~ 3-byte: machine.
- 2-byte: process id.
- 3-byte: counter.

- Operations:

use <db>

show <db> = db

db.dropDataBase()

db.createCollection('a')

db.a.drop()

- Adding Documents.

db.person.insert({"_id": 1, "name": "a"})

db.person.insertMany([{..}, {..}])

- Sorting: find().sort({"age": 1, "name": -1})

1 for asc, -1 for dec.

- limit & skip: find().skip(1).limit(1)
- distinct: NO find().distinct() ✗

distinct take over find.

- length: .length ⇐
- count: find().count() / .count()
- Projections: find({}, {"name": 1, "age": 1})
- Renaming: find({}, {"home": "$address", "_id": 0})
- Update:

update({}, {$set: {}}, {multi: true})

- $set: update instead of overwrite.
- multi: multiple

updateMany()

- Query

db.person.find()

- Operators:
  - $lt, $gt, $lte, $gte, $eq, $ne, $in, $all
  - $and, $or, $not
  - * $and/$or requries [..]
  - * $not can not be top level
  - eg: ① find({$or: [{}, {}]})
    ② find({"a": {$gt: 25}})
- Pattern matching: /kevin/i.

contains ↓ case insensitive ↓

find({name: {$not: /kevin/i}})

need: {name: {$exists: true}}

⇓

{name: {$not: /kevin/i, $exists: true}}

- On array $elemMatch.

-Aggregation:

aggregate ([{ $group : { _id: "$ colname1",
total: { $sum: "$colname2" } } })

Select category_id, sum(col2) total
From ..
GB. colname1

• Having

aggregate (.. , { $match : { total : { $gt : 50 } } })

Operator:
$sum,
$ avg
$ min
$ max.

• Count

aggregate ({ $group: { ... , total: { $sum: 1 } } }).

• Multi GB.

aggregate({ $group: { _id: { a: "A", b: "B" } ... } }).

• Pipeline:

$match → $ group, → $match → $sort → $limit.

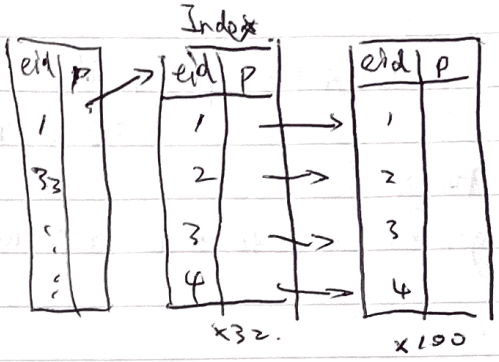
• Join

aggregate({ $lookup: { from: "A", localfield: "a",
"foreign-fied: "b",
as: " c " } })

Lec 13 Indexing

Index is a data structure that speeds up selections on the search key field(s)

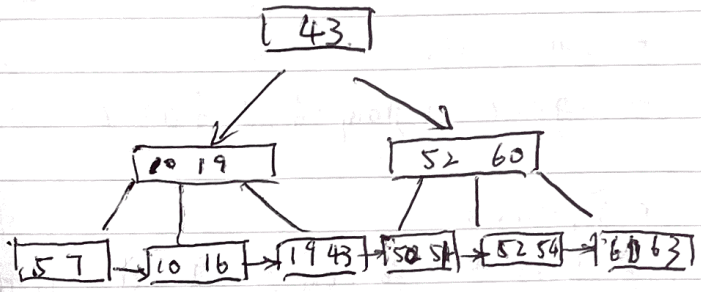
- Classification:
  - Clustered: records stored in order of search key
  - Unclustered: records stored not in order.
- Search key: subset of fields of a rel.



B+ Trees

A search tree that efficiently supports range queries by making leaves into a linked list.

- Rules:
  - Order d.
  - Except root, every node has $\geq d$, $\leq 2d$ keys.
  - ~~Each leaf has~~



$d = ?$

Key = 4 b
Pointer = 8 b
Block = 4096 b.

$2d \times 4 + (2d+1) \times 8 \leq 4096$ b.
$d = 170$

- Searching

1- Start from root, search left boundary vertically
2- locate right boundary while reading all blocks until right boundary (exists or passed).

- Insertion

- search vertically for the node that needs to be inserted
- If overlow ($> 2d+1$):
  - split in half, middle to parent.

  else
  - ↳ stop.

- Splitting

- Parent nodes (internal) don't need to keep node

- Deletion

- Search vertically
- If ($< d-1$)
  - ↳ merge with neighbor.

    or
  - ↳ borrow from neighbor.

Lec 14 Query Execution

- Logical Operators : what they do
- Physical Operators : how they do.
  - scan
  - hash
  - sort
  - index

- Cost Model
  - M : Memory available ~~as buffer~~ in main mem.
  - B(R) : Num of blocks holding R.
  - T(R) : Num of tuples in R.
  - V(R,a) = Num of distinct values of att "a" in R
  - Selectivity : diversity of an att in R.
  - clustered : records stored in block ∈ R.
  - unclustered : mixed in blocks.

Operator : Scan.

Cost : clustered : B(R)
unclustered : T(R).

One Pass Algorith

- Selection & Projection
  Cost : B(R)
- Duplicate Elim.
  Cost : B(R)
- Grouping
  cost : B(R)
- Binary Opr.
  Cost : B(R)+B(S)

Nested Loop.

- For every (M-2) size R, join with whole table S.
- cost : B(S) · B(R)/(M-2) + B(R).
- ~~S<R.~~

Two-Pass. (& write)

- Sorting : DE | 1. Sort : 2B(R).
  2. Merge : B(R).

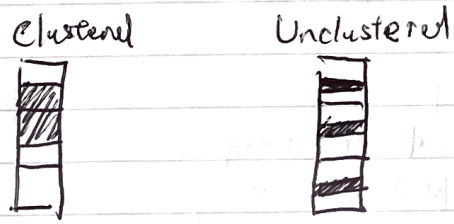
Grp: same.

Binary Opr. : 3B(R) + 3B(S)

Join : Sort-Merge Join : 3B(R)+3B(S)

*Merge phase cost increase due to extra merge passes.

simple-sort join : ~~5~~B(R)+5B(S).



~~Sort-Merge~~ Join

Hashing Based:

DE : Hash into buckets : 2B(R)
Check Duplicate. : B(R)

Partitioned Hash Join :

Cost : 3(B(R)+3B(S)) Read Si, hash it, load one block from R. Join

Index Based :

Selection : B(R)/V(R,a)
T(R)/V(R,a)

Join : B(R) + T(R) B(S)/V
R(R)+T(R) T(S)/V

Sort-Merge Join : B(R) + B(S)

Lec15. NoSQL.

Relational Database

* Mature and stable
* Feature-rich versatile: SQL
* ACID:
  - Atomicity: All or None opr. executed
  - Consistency: DB is consistent with opr.
  - Isolation: concurrent opr. don't interfere.
  - Durability: can recover from failure.
* Challenges:
  - Internet scale Apps.
  - Big data, unstructured.
  - New workloads

NoSQL Database: Not only SQL

* Flexible (non-relational)
* Scalable. horizontally
* Data replicated.
* Weaker consistency
* High availability

eg:

- Firebase
- MongoDB
- DynamoDB.

Consistency:

- Eventual Consistency:
  - Weak consistency
  - Update will eventually show up.
- Inconsistency Window:
  Time between update acknowledged to user/all replicas
  Determined by: Communication Delay, Load on sys., Number of replicas.

DynamoDB:

- Schema-less (other than primary key)
- DB structure.
  DB
  → Table
  → Items/rows
  → Attributes
- Partition key: Wide range of value.
- Sort key: searching within a partition
- Datatypes:
  - String, Binary, Number, Stringset, Numberset, Binaryset, Map, List, Boolean, Null

CAP: Can have at-most 2.

- Consistency: Every read receives most recent write/error.
- Availability: Every request receives a response
- Tolerance to Net Partitions: Sys continue to operate when messages are dropped.

Lec 16. MapReduce

MapReduce: A computational paradigm
- Parallel
- Distributed

A MapReduce Job consists of a num of:
- Map tasks: Data Transmation
- Reduce tasks: Combines results
- Internal Shuffle tasks: distribute tasks.

Mapper:
1. Invokes Map func for each input key value pair.

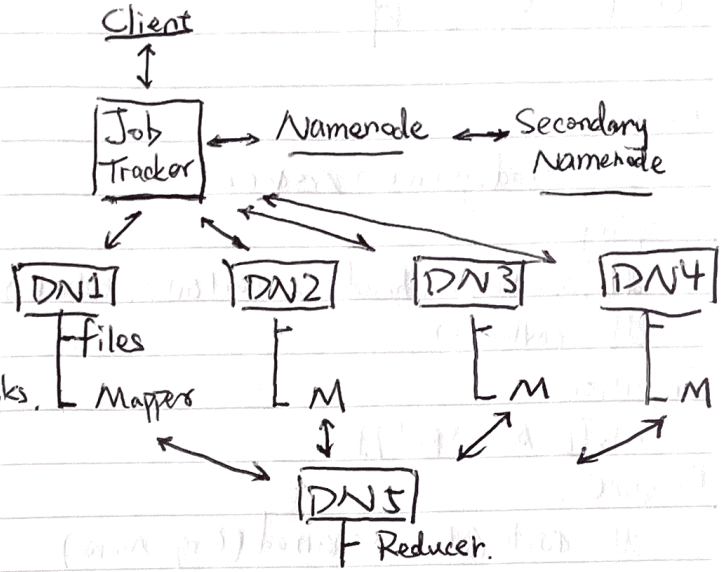
Reducer:
1. Invokes Reduce func for each different intermediate key.

Shuffler:
1. All intermediate key-value pairs are sent to the same Reduce Task.
   - Partitioner: Hashing /..

FileInputFormat:
- Take paths to files
- Read all files in paths
- Divide each files into Input Splits.
  - One Map Task for each Split.
- Subclass: InputFormat
  - Subclass: RecordReader.

FileOutputFormat:
- Output from Reducer.
- Each write in separate part-r-nnnnn.
  - r: Reducer out
  - nnnnn: partition #

Combiner:
- Require: Operation be Associative & Commutative.
- Input & Output Format = Map Out & Reduce In.



1. Client request to Job Tracker.
2. Job tracker:
   1) Ask Namenode for loc.
   2) Assign task to Task Tracker at Data Node.
   3) Reassign if fails.
3. Task tracker:
   1) Signal back to Job Tracker.
   2) Monitor status of task.
4. Mapper: <k, v> → <k', v'>
5. Shuffler:
   1) Partition Map output in buffer into R (#Reducer Tasks).
   2). Sort Partitions and write disk
   3). Merge on disk.
   4) Notify Job Tracker
   5). Copy data from all Mapper
   6) Merge data
   7.

Lec 7 Spark Df

Creation:
spark.read.json() / csv() spark.createDataFrame([(), ()], ['colA', 'colB'])

Display:
db.show(n) / .head(n) / .tail(n) / .take(n)
db.collect()

Projection
db[['A', 'B']]

Rename:
db.withColumnRenamed(Org, New)

Add Column:
db.withColumn(colName, val)

Filter:
db[(A > a) & (B > b)]

Distinct:
db.dropDuplicates() / db.distinct()

Aggregate ~~w/o GB~~:
db.agg({'ColA': 'max'})
db.agg(fc.max('colA').alias('maxColA'))

GroupBy:
db.groupBy(['colA', 'colB']).max(0)

OrderBy:
db.orderBy(['colA', 'colB'], [ascading=[T, F])

Agg Func:
count, max, min, avg/mean, sum

Limit: .limit(n)

Join:
db1.join(db2, (db1.a==db2.a) & (db1.b==db2.b))
db1.join(db2, 'colA')

Intersect: db.intersectAll() / Intersect() # removes dublicates

Subtract: db1.subtract(db2)

Lec 18 Spark RDD

Motivation:

- Hadoop ill-suited for
  - ↳ Reuse looping data
  - ↳ Interactive exploration.

RDD: Resilient Distributed ~~DB~~ Dataset

- Read only, partitioned collection
- Parallel Operate on partitions
- Fault Tolerant

Create Spark context

```
sc = SparkContext()
```

Create:

```
sc.textFile()
sc.parallelize(list, num)
```

Reduce: ⇒ val

```
data.reduce(func) #func (commutative & associative)
```

Reduce By key

```
data.reduceByKey(func) => RDD
# data = [(k, v)]
```

Map:

```
data.map(func)
```

Filter:

```
data.filter(func) # boolean func
```

Groupby Key

```
data.groupByKey() => [(k, [v, v...]), ...]
```

map Value:

```
data.mapValue(func) => [(k, f(v)) ...]
```