Two Computational Paradigms for Big Data

Ravi Kumar

Google
Outline of the lecture

Motivation

- **Data Stream** model
- **MapReduce** model

For each model

- Description
- Canonical examples
- Sample algorithms

**Goal**: To give an idea of the algorithmic techniques in massive data

Will not cover the full list
Motivation

- Big data (Tb++)
  - Data grows faster than CPU speed
  - Web, social, satellite, genome, sensor, ...

- Efficient algorithms
  - Linear time is the only feasible option
  - Data can be distributed across machines/racks

- Small memory footprint
  - Cannot hold the data or even portions of it in main memory

- Approximate answers suffice
  - An exact answer is not worth going after

How to model efficient computation on massive data?
Sample questions

• How many unique web search queries were issued yesterday?

• How many shopping sites were clicked together in a user web search session?

• What were the top queries (by volume) last week?

• Which is the densest community in a social network?

• Which web sites have similar usage patterns?

• How many “friend-of-a-friend is a friend” instances are in a social network?
## CPU vs Memory

<table>
<thead>
<tr>
<th></th>
<th>Polynomial</th>
<th>Sublinear</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Single</strong></td>
<td>RAM</td>
<td>Data streams Sketches</td>
</tr>
<tr>
<td><strong>Multiple</strong></td>
<td>PRAM</td>
<td>MapReduce</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distributed streams Sketches</td>
</tr>
<tr>
<td></td>
<td>Batch</td>
<td>Online</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Single</td>
<td>RAM</td>
<td>Data streams</td>
</tr>
<tr>
<td>Multiple</td>
<td>MapReduce</td>
<td>Distributed streams</td>
</tr>
</tbody>
</table>
I. Data Stream Model
Data Streams: Outline

- Model description and characteristics
- Statistical problems
  - Distinct elements
  - Frequency moments
  - Frequent elements
- Graph applications
  - Triangle counting
  - Densest subgraph
DS: Characteristics

- Data arrives in a **stream**
  - No random access to data
  - Can be new data or updates to existing data

- Typically single CPU

- Very **limited** amount of memory
  - Some cases, only Mb even for Tb+ data
  - Cannot store any non-trivial portion of the data in memory
  - Data size may be infinite/unknown in advance

- Ideally, make a **single pass** over the data
  - In some cases, can make multiple passes
DS: Practical applications

- **Web mining**
  - How many unique queries were processed by the search engine in the last two days?
  - What are the most frequent queries in the past hour?
  - Find dense communities in a social network

- **Databases**
  - Query selectivity, join size estimation
  - Query planning and execution

- **Networks**
  - Sensor/satellite data
  - Traffic and packet monitoring
Basic DS model

Compute a function of inputs $X = x_1, \ldots, x_n$

- Ideally use $O(\log^c n)$ memory
- Small processing time per element
- Approximate solutions are sufficient
- Can use randomness (many cases, needed)
- Make one or few passes over the data
Find **how to represent** the input
- Sufficient for the function to be approximated
- Small enough to be space-efficient

**Update** the stored representation upon seeing another input
- Efficient and incremental updates

**Tools** at disposal
- Randomization
- Hashing
- Sketching
Given an arbitrary permutation of \([n]\) with one number missing, find the missing number

**Input.** \(\pi_1, \ldots, \pi_{n-1}\) (Eg, \(n = 5\), input = 3 4 1 5)

**Algorithm.** \(S = \sum_{i=1,n-1} \pi_i\)

**Output** \(n(n+1)/2 - S\)

Memory used = \(O(\log n)\) bits

Can extend this to two numbers missing and so on
DS Warmup: Reservoir sampling

Given an infinite stream, maintain a uniform random sample from the stream (seen so far)

**Input.** $x_1, x_2, \ldots$

**Algorithm.** $S = x_1$

for each $i$:

$S = x_i$ with probability $1/i$

**Lemma.** $S$ is a uniform random sample at any point

**Proof.** Fix $i$. At any time $t \geq i$,

$
\Pr [x_i = S] = (1/i) \cdot i/(i+1) \cdot (i+1)/(i+2) \cdots = 1/t
$
**F₀: Counting distinct elements**

Given $X = x_1, \ldots, x_n$ compute $F_0(X)$, the **number of distinct elements** in $X$

**Motivation.** Number of unique web queries

**Example.** $X = 3 \ 3 \ 2 \ 4 \ 5 \ 2 \ 7 \ 2 \ 3$; $F_0(X) = 5$

**$(\epsilon, \delta)$-approximation.** Output $F_0'(X)$ such that with probability $1 - \delta$, $F_0'(X) = (1 \pm \epsilon) F_0(X)$

- Assume $x_i \in [m]$ and log $m = O(\log n)$; else hash input
- Sampling needs lots of space
- Without randomization/approximation, this is hard
F₀: Intuition

Suppose \( h: [m] \rightarrow (0, 1) \) is truly random

Then \( \min \{ h(x_i) \} \) is roughly \( \sim 1/F₀(X) \)

Reciprocal of this value is \( F₀(X) \)

More robust: Keep the \( t \)-th smallest value \( v_t \)

\( v_t \) is roughly \( \sim t/F₀ \)

A good estimator of \( F₀ \) is \( t/v_t \)
F₀: Algorithm

\[ t = \frac{1}{\epsilon^2}; \quad h: [m] \rightarrow [m^3], \]

pairwise independent

\[ T = \emptyset \]

for \( i = 1, \ldots, n: \)

\[ T \leftarrow t \text{ smallest values in } T \cup h(x_i) \]

\[ v_t = t\text{-th smallest value in } T \]

Output \( tm^3/v_t \)

- **Space:** \( O(\log m) \) for \( h \) and \( O(1/\epsilon^2 \cdot \log m) \) for \( T \)

- **Time:** Balanced binary search tree for \( T \)

Compute many estimators independently and output their median

- To achieve confidence \( 1-\delta \)

**Theorem.** \( F_0'(X) \) is an \((\epsilon, \delta)\)-approximation
F₂: Second frequency moment

- fₗ = frequency of j = #{ j occurs in X}

Given X = x₁, …, xₙ compute F₂(X) = ∑ₗ₌₁..ₘ fₗ²

Motivation. Self-join size estimation

Example. X = 3 3 2 4 5 2 7 2 3; f₂ = f₃ = 3, f₄ = f₅ = f₇ = 1

F₂(X) = 3² + 3² + 1² + 1² + 1² = 21

(ε, δ)-approximation. W.p. 1 – δ, F₂′(X) = (1 ± ε) F₂(X)

- Sampling needs lots of space
- Without randomization/approximation, this is hard
**F₂: Algorithm**

\[ h: [m] \rightarrow \{ +1, -1 \} \]

\[ Z = \sum_{i=1..n} h(x_i) \]

Output \( Z^2 \)

Random projection of the frequency vector

\[ Z = \sum_{i=1..n} h(x_i) = \sum_{j=1..m} f_j h(j) \]

\[
\begin{pmatrix}
-1 & +1 & -1 & -1
\end{pmatrix}
\begin{pmatrix}
f_1 \\
f_2 \\
f_3 \\
f_4
\end{pmatrix}
\]
F₂: Analysis

- \( h: [m] \to \{ +1, -1 \} \); \( E[h(j)] = 0 \)

**Lemma.** \( E[Z^2] = F_2 \)

**Proof.**
\[
Z = \sum_{i=1..n} h(x_i) = \sum_{j=1..m} f_j \cdot h(j)
\]

\[
E[Z^2] = E[ \sum_{j, j'=1..m} f_j \cdot h(j) \cdot f_{j'} \cdot h(j') ]
\]
\[
= E[ \sum_{j \neq j'} f_j \cdot h(j) \cdot f_{j'} \cdot h(j') ] + \sum_{j=1..m} f_j^2
\]
\[
= 0 + \sum_{j=1..m} f_j^2 = F_2
\]

**Lemma.** \( \text{Var}[Z^2] \leq 2F_2^2 \)
Compute estimators $Z_1, \ldots, Z_k$ independently and output average $Y = \sum Z_i^2 / k$ where $k = 1/\epsilon^2$
• This reduces the variance

Compute estimators $Y_1, \ldots, Y_l$ independently and output median $\{ Y_i \}$
• To achieve confidence $1-\delta$

• **Space:** $O(1/\epsilon^2 \cdot \log 1/\delta \cdot \log m)$; one pass

**Theorem.** $F_2'(X)$ is an $(\epsilon, \delta)$-approximation
Application: Triangle counting

Given an undirected graph, count the number of triangles

Motivation. Social network analysis
Triangle counting (contd)

For edge \((u, v)\), float virtual triples \(\langle u, v, w \rangle\) for \(w \neq u, v\)

Compute \(F_0, F_1, F_2\) of this virtual stream

\[ T_i = \# \text{ triples with exactly } i \text{ edges among them} \]

\[ T_0 + T_1 + T_2 + T_3 = \binom{n}{3} \]

\[ F_0 = T_1 + T_2 + T_3 \]

\[ F_1 = \sum_{u,v,w} i \cdot T_i = m(n - 2) \]

\[ F_2 = \sum_{u,v,w} i \cdot T_i^2 = T_1 + 4T_2 + 9T_3 \]

Estimate \(F_0, F_2\) and use that to estimate \(T_3\)
CountMin: Frequent elements

- Given $X = x_1, \ldots, x_n$, additively approximate $f_j$
  - We focus on high-frequency counts

- Motivation: Most frequent web queries

- Main idea: use a hash function to count the number of occurrences of an element
  - Use multiple hash functions to mitigate collisions
CM: Algorithm

\( h_i : [m] \rightarrow [w], \ w = 1/\epsilon \)

for \( i = 1, \ldots, n: \)

\( \text{for } d = 1, \ldots, \log 1/\delta: \)

\( \text{count}[d, h_d(x_i)]++ \)

Output

\( f_j' = \min_d \{ \text{count}[d, h_d(j)] \} \)
Lemma. \( f_j' \geq f_j \) and \( f_j' \leq f_j + \epsilon |f| \) with probability \( 1 - \delta \)

Proof.

- Count is always overestimated
  - Hashes can collide
- Chance the minimum count is way off is low
  - From Markov’s inequality
Densest subgraph

- Find the **densest subgraph** in a undirected graph
  - **Density** of a subgraph is the ratio of the number of edges to the number of nodes
  - Motivation. **Community** finding
  - **c-approximation** = when density is at most \( c \) times worse than the best density

\[
\text{Density}(\bullet) = \frac{5}{4} = 1.2
\]
DSG: Algorithm

A simple iterative algorithm

- Compute the average degree
- Delete all nodes whose degree is below the average
- Keep track of the density at each step

Output the densest graph seen during the iteration
DSG: Example

density = 16/11 = 1.45; average degree = 2*density = 2.90
Best density = 1.45
DSG: Example (contd)

density = 16/11 = 1.45; average degree = 2*density = 2.90
Best density = 1.45
DSG: Example (contd)

density = $9/5 = 1.8$; average degree = $2 \times \text{density} = 3.6$

Best density = 1.8
DSG: Example (contd)

density = 9/5 = 1.8; average degree = 2*density = 3.6
Best density = 1.8
DSG: Example (contd)

density = 3/3 = 1; average degree = 2*density = 2
Best density = 1.8
DSG: Example (contd)

Best density = 1.8
DSG: Quality

**Theorem.** The output is a $2+\epsilon$-approximation

**Proof.**

- $V^* =$ optimal subgraph, $\rho^* =$ optimal density

- Each degree in $V^*$ is at least $\rho^*$
  - Otherwise can improve the optimum

- Assume you only remove the lowest degree node and consider the first subgraph $V'$ when you are about to remove a node in $V^*$
  - In reality, remove all nodes with degree $\leq (1+\epsilon)$ average degree

- Density of $V' \geq |\rho^* V'/2| / |V'| = \rho^*/2$
**Theorem.** The algorithm makes $O(\log_{1+\epsilon} n)$ passes and uses $O(n)$ memory

**Proof.**

- One *cannot* have too many nodes above average
- At most most $1/(1+\epsilon)$ fraction of nodes *survive* each pass
- Hence the algorithm terminates after $\log_{1+\epsilon} n$ passes
DSG: Performance

IM: Remaining graph vs passes

Remaining nodes

passes

\( \varepsilon = 0 \)
\( \varepsilon = 1 \)
\( \varepsilon = 2 \)
DS: Some references

- N. Alon, Y. Matias, M. Szegedy. The space complexity of approximating the frequency moments, JCSS 1999
- Z. Bar-Yossef, R. Kumar, D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs, SODA 2002
- B. Bahmani, R. Kumar, S. Vassilvitskii. Densest subgraphs in streaming and MapReduce. VLDB 2012
II. MapReduce Model
MapReduce: Outline

- Model description and characteristics
- Graph applications
  - Connected components
  - Counting triangles
  - Maximal matching
- Clustering application
  - K-means
MR: Characteristics

- **Multiple** CPUs
  - 10s to 10,000s processors

- **Sub-linear** memory
  - Few Gb of memory per machine, even for Tb+ sized data
  - Memory is *not shared* (unlike PRAM)

- **Batch processing**
  - Data processed in *multiple rounds*
  - Extensions used for incremental updates, online algorithms
MR: Practical applications

- MR used very widely for large data analysis
  - Industry: Google, Yahoo, Amazon, Facebook, Netflix, LinkedIn, New York Times, eHarmony, ...
  - Adopted by the academic community as well

- Open source versions available
  - Hadoop and its extensions

- Many abstractions on top of MR
  - Pregel, Hive, Pig, ...
  - Same computational model underneath
Basic MR model

Computation consists of one or more rounds of Map and Reduce steps

The input is partitioned across different machines

- **Map step (mapper)**
  Each mapper processes its input and emits \( \langle \text{key}; \text{value} \rangle \)

- **Reduce step (reducer)**
  Each reducer processes all pairs with the same key

Internal **shuffle** phase ensures this
MR model (contd)

Input

Mappers

Shuffle phase

Output

<User Input>

Reducers

\langle \text{key; value} \rangle
MR model: Example

Input

Mappers

Reducers

Output
Multi-round MR
Modeling MR

- **Input** size = $n$

- **Memory**
  - Cannot store the data in memory
  - Use *sublinear memory* per machine: $n^{1-\epsilon}$ for some $\epsilon > 0$

- **Machines**
  - Machines do not share memory
  - *Sublinear number of machines*: $n^{1-\epsilon}$ for some $\epsilon > 0$

- **Synchronization**
  - Computation proceeds in *rounds*
  - Goal is to run in a constant number of rounds
Communication cost

- Very **important**, makes a big difference
- Huge improvements possible by
  - Moving code to data (and not data to code)
  - Working with graphs: save graph structure locally between rounds
  - Job scheduling (eg, same vs different racks)
- **Minimizing communication** always a goal
Given a sparse term-document matrix \( A = [A_{ij}] \) in a row-major order, output \( A \) in column-major order.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

1; 1:a, 2:b
2; 2:c
3; 2:d
3; 2:e
1; 1:a
2; 3:e
1; 1:a
MR Algorithmic philosophy

- Find the core of the problem
- **Partition** the input across mappers
- Solve the problem on partitioned inputs on each mapper and emit the **partial solutions**
- **Tools** at disposal
  - Sampling
  - Filtering redundant information
  - Sketching
  - Careful partitioning to mitigate skew
- Reducer **combines** the individual solutions
Connected components

Given an undirected graph, find the connected components.
Motivation: Basic question.
CC: Algorithm

Partition the edges randomly

Mapper 1

Mapper 2
CC Algorithm (contd)

Each mapper finds CC on its input
Reducer combines edges and computes CC
CC Analysis

\( n = \text{number of nodes}, \ m = \text{number of edges} \)

If we have \( k \) mappers

- Total time = \( T_{cc}(m/k + n) + T_{cc}(nk) \)
- Total memory per machine = \( O(n) \)
  - Since we can stream through the edges
- Number of rounds = 2

- Works well for graphs where the number of edges is super-linear in the number of nodes
Triangle counting

Given an undirected graph, count the number of triangles

Motivation: Social network analysis
Naïve algorithm

Sequential algorithm

for $u \in V$:

for $(v, w) \in N(u)$:

if $(v, w) \in E$:

triangles[$u$]++

Running time: $\sum_v \deg(v)^2$
Naïve algorithm in MR

Generate all length-two paths and check if there is an edge linking the end nodes

Map1

Input: \((u, v) \in E\)
Emit: \(\langle u; v \rangle\)

Reduce1

Input: \(\langle u; N(u) \rangle\)
Output: \(\langle (v, w); u \rangle\)
for \((v, w) \in N(u)\)

A path from \(v\) to \(w\) goes through \(u\)
Naïve algorithm (contd)

Map2

Input: \((u, v) \in E\) and \(\langle(v, w); u\rangle\)

Emit: \(\langle(u,v); \$angle\) \(\langle(v, w); u\rangle\)

Reduce2

Input: \(\langle(v, w); u_1, u_2, \ldots, u_k, \$\rangle\)

Output: If \(\$\) is present, then \(\text{triangle}[u_i] += 1/3\)

\(\$\) confirms the existence of \((v, w)\) edge
Naïve algorithm: Scalability

Does this **scale**?

- Best parallel running time: \( \max_v \deg(v)^2 \)
- If maximum degree is high (e.g., Twitter follower graph), then the algorithm is **slow** even with maximum parallelism
Two ideas:

• Count each triangle once from the point of view of the lowest degree node in the triangle

• Divide the graph into overlapping subgraphs across mappers

for $u \in V$:

for $(v, w) \in N(u)$:

if $\text{deg}(u) \leq \min(\text{deg}(v), \text{deg}(w))$

if $(v, w) \in E$:

triangles[$u$]++
Improved algorithm: Analysis

**Theorem.** Total work $= \mathcal{O}(m^{3/2})$

**Proof.** Let $L = \{ u : \deg(u) \leq \sqrt{m} \}$ and $H = \{ u : \deg(u) > \sqrt{m} \}; \ |H| \leq \sqrt{m}$.

Nodes in $H$ produce paths for at most $|H|^2$ neighbor pairs $\Rightarrow$ $\mathcal{O}(m^{3/2})$ paths.

$m_i =$ total number of edges adjacent to nodes of degree $d_i$; $\sum m_i = 2m; \quad m_i \leq m/i$

Nodes in $L$ produce $\sum_{u \in L} \deg(u)^2 p = \sum_i i^2 \frac{m_i}{i} = \mathcal{O}(m^{3/2})$ paths.
Improved algorithm: tradeoff

Time vs memory \textit{tradeoff}

\textbf{Partition} nodes into equal-sized $V_1, \ldots, V_p$

Consider all possible \textit{triples} $\langle V_i, V_j, V_k \rangle$ and the induced subgraph $G_{ijk}$

Count triangles in each $G_{ijk}$ in \textit{parallel}

Carefully track \textit{overcounting}
Can count exactly how many subgraphs each triangle will be in

Tradeoff (contd)
Improved algorithm: Scalability

- **Balanced** running times
- Parameter $p$ controls memory per machine
- Number of edges per partition is $\sim \frac{m}{p^2}$
- Total work
  $$p^3 \Omega\left(\frac{m}{p^2}\right)^{3/2} = O(m^{3/2})$$
- **Lesson:** Treat data skew carefully
Maximal matching

Given an undirected graph, finding a maximal matching

Motivation: Matching advertiser and content
Max. matching (contd)

- First idea
  - Randomly partition the nodes across machines
  - Find a matching on each partition
  - Compute a matching on the matchings
  - Does not work: can make very slow progress

- A better idea
  - Find a seed matching on a sample
  - Prune all dead edges
    - Edge dead if an endpoint already matched
  - Recurse on remaining edges
MM: Algorithm

Sample edges uniformly at random
MM Algorithm (contd)

Sample edges uniformly at random

Find a maximal matching on the sample
Sample edges uniformly at random

Find a maximal matching on the sample

Look at the original graph, prune dead edges
MM Algorithm (contd)

Sample edges uniformly at random

Find a maximal matching on the sample

Look at the original graph, drop dead edges

Find maximal matching on the remaining edges
**Lemma.** Let the sampling rate \( p = n^{1+c}/m \) for some \( c > 0 \). Then, whp, the number of *edges left after the prune* step is at most \( 2n/p = 2m/n^c \).

**Proof.** Suppose \( J \) is unmatched after the prune step. \( \Rightarrow \) \( J \) was an independent in the sampled graph. If \( |E[J]| > O(n/p) \), then it is an independent set with probability at most \( \exp(-n) \). Take a union bound over all subsets \( J \).

**Corollary.** With \( O(n^{1+c}) \) memory, the algorithm runs in \( O(1) \) rounds; with \( O(n \log n) \) memory, the algorithms runs in \( O(\log n) \) rounds.
K-means

Given a set of points and $k > 0$, partition the points into $k$ clusters to minimize sum of squares error

Motivation: Fundamental clustering primitive
K-means: Algorithm

Lloyd’s method

- Randomly pick k centers
- Repeat until clustering does not change
  - Assign each point to its nearest center
  - Recompute centers of each cluster
Lloyd’s method: Example

- $k = 3$; pick 3 random centers
Lloyd’s method (contd)

- Assign each point to its nearest center
Lloyd’s method (contd)

- Recompute centers
K-means++: Instead of random initial centers, choose more carefully

- \( D(p) = \) distance of \( p \) to a closest center

Choose next cluster center \( \propto D^2(p) \)
- Give preference to points that are far away from existing centers
- Inherently sequential algorithm

**Theorem.** K-means++ is an \( O(\log k) \) approximation
How to adapt K-means++ to MapReduce?

Use **sample and prune** method

- Independently (over)sample $k \times L$ points in each round
- Prune back to $k$ points by reclustering

**Theorem.** After $\log_L$ rounds, we get $O(1)$-approximation
K-means||: Performance

Random Initialization

KDD Dataset, k=65

log # Rounds
MR: Some references

- J. Dean, S. Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI 2004
- H. Karloff, S. Suri, S. Vassilvitskii. A model of computation for MapReduce. SODA 2010
- S. Suri, S. Vassilvitskii. Counting triangles and the curse of the last reducer. WWW 2011
- S. Lattanzi, B. Moseley, S. Suri, S. Vassilvitskii. Filtering: A method for solving graph problems in MapReduce. SPAA 2011
- B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii. Scalable k-means++. VLDB 2012
Xíe Xíe! Xíe Xíe!

Questions/Comments

ravi.k53@gmail.com