Algorithms for Massive Datasets

Course held by Prof. Dario Malchiodi

Federico Cristiano Bruzzone
Id. Number: 27427A

MSc in Computer Science

UNIVERSITY OF MILAN
Computer Science Department
ADAPT-Lab

Academic Year 2022–2023

COMPUTER SCIENCE DEPARTMENT • ADAPT-LAB

Contents

1 Data Mining

1.1 Things Useful to Know

1. The **TF.IDF** (*Term Frequency times Inverse Document Frequency*) measure of word importance.
2. Hash functions and their use.
3. Secondary storage (disk) its effect on running time of algorithms.
4. The base e of natural logarithm and identities involving that constant.
5. Power laws.

1.1.1 Importance of Words in Documents

Classification often starts by looking at documents, and finding the significant words in those documents. Our first guess might be that the words appearing most frequently in a document are the most significant. However, that intuition is exactly opposite of the truth. The formal measure of how concentrated into relatively few documents are the occurrences of a given word is called **TF.IDF**. It is normally computed as follows. Suppose we have a collection of N documents. Define f_{ij} to be the *frequency* of term (word) i in document j. Then, define the *term frequency* $\boldsymbol{TF}_{ij}$ to be:

$$\boldsymbol{TF}_{ij} = \frac{f_{ij}}{\max_k f_{kj}}$$

That is, the term frequency of term i in document j is f_{ij} normalized by dividing it the maximum number of occurrences of any term (perhaps excluding stop words) in the same document.

The IDF for a term is defined as follows. Suppose term i appears in n_i of the N documents in the collection. Then $\boldsymbol{IDF}_i = \log_2(N/n_i)$. The **TF.IDF** score for term i in document j is then $\boldsymbol{TF}_{ij} \times \boldsymbol{IDF}_i$.

1.1.2 Hash Functions

A hash function h takes a *hash-key* value as an argument and produces a *bucket number* as a result. The bucket number is an integer, normally in the range 0 to $B-1$, where B is the number of buckets. Hash-keys can be of any type. There is an intuitive property of hash functions that they "randomize" hash-keys.

1.1.3 Indexes

An *index* is a data structure that makes it efficient to retrieve objects given the value of one or more elements of those objects. The most common situation is one where the objects are records, and the index is on one of the fields of that record. Given a value v for that field, the index lets us retrieve all the records with value v in that field.

1.1.4 Secondary Storage

Disks are organized into *blocks*, which are the minimum units that the operating system uses to move data between main memory and disk. It takes approximately ten milliseconds to *access* and read a disk block. That delay is at least five orders of magnitude (a factor of 10^5) slower than the time taken to read a word from main memory. You can assume that a disk cannot transfer data to main memory at more than a hundred million bytes per second (100MB), no matter how that data is organized. That is not a problem when your dataset is a megabyte. But a dataset of a hundred gigabytes or a terabyte presents problems just accessing it, let alone doing anything useful with it.

1.1.5 The Base of Natural Logarithms

The constant $e = 2.7182818...$ has a number of useful special properties. In particular:

$$\lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n = e$$

Some algebra lets us obtain approximations to many complex expression. Condiser $(1+\alpha)^\beta$, where α a small. We can rewirte the expression as $(1+\alpha)^{(1/\alpha)(\alpha\beta)}$. Then subsitute $\alpha = 1/n$ and $1/\alpha = n$, so we have that $(1+\frac{1}{n})^{n(\alpha\beta)}$, which is

$$\left(\left(1+\frac{1}{n}\right)^n\right)^{\alpha\beta}$$

Since α is assumed small, n is large, so the subexpression $\left(1+\frac{1}{n}\right)^n$ will be cose to the limiting value of e.

Some other usefull approximations follow from the Taylor expansion of e^x. That is,

$$\begin{aligned} e^x &= \sum_{i=0}^{\infty}\frac{x^i}{i!} \\ &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots \end{aligned}$$

When x is large, the above series converges slowly, although it does converge because $n!$ grows faster than x^n for any constant x.

1.1.6 Power Laws

There are many phenomena that relate two variables by a *power law*, that is, a linear relationship between the logarithms of the variables. Figure 1.1 suggest such a relationship that is: $\log_{10} y = 6 - 2\log_{10} x$

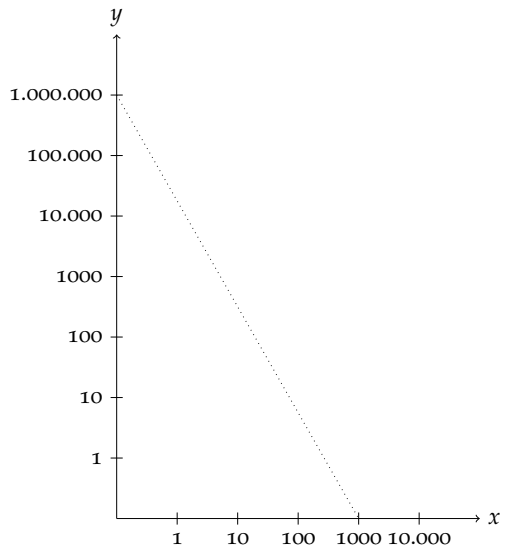

Figure 1.1. *A power law with a slope of* -2.

The Matthew Effect

Often, the existence of power laws with values of the exponent higher than 1 are explained by the *Matthew effect*. In the biblical *Book of Matthew*, there is a verse about "the rich get richer." Many phenomena exhibit this behavior, where getting a high value of some property causes that very property to increase.

2
MapReduce and Cost Model

2.1 Algorithms Using MapReduce

MapReduce is not a solution to every problem, not even every problem that profitably can use many compute nodes operating in parallel. The original purpose for which the Google implementation of MapReduce was created was executed very large matrix-verctor multiplication as are needed in the calculation of PageRank (See Chapter 3). We shall see that matrix-vector and matrix-matrix calculations fit nicely into the MapReduce style of computing. Another imortant class of operations that can use MapReduce effectively are the relational-algebra operations.

2.1.1 Matrix-Vector Multiplication by MapReduce

Suppose we have an $n \times n$ matrix M, whose element in row i and column j will be denoted m_{ij}. Suppose we also have a vector $\mathbf{v}$ of length n, whose j-th element is $\mathbf{v}_j$. Then the matrix-vector product is the vector $\mathbf{x}$ of length n, whose i-th element $\mathbf{v}_i$ is given by

$$\mathbf{x}_i = \sum_{j=1}^{n} m_{ij}\mathbf{v}_j$$

Let n be large, but not so large that vector $\mathbf{v}$ cannot fit in memory and thus be avaiable to every Map task.The matrix M and vector $\mathbf{v}$ are stored in the distributed file system (DFS). We assume that the row-column coordinates of each matrix element will be discoverable from from its position in the file, or because it is stored explicitly as a triple (i, j, m_{ij}).We also assume the position of the element $\mathbf{v}_j$ in the vector $\mathbf{v}$ is discoverable in the same way.

The Map Function: The Map function is written to apply to one element of m. Each Map task will operate on a chunk of the matrix M. From each matrix element m_{ij} it produce a key-value pair $(i, m_{ij}\mathbf{v}_j)$

The Reduce Function: The Reduce function simply sumus all the values associated with a given key i. The result will be a pair $(i, \mathbf{x}_i)$.

2.1.2 If the Vector v Cannot Fit in Memory

However, it is possible that the vector $\mathbf{v}$ is so large that it will not fit in its entirety in main memory. In this case, we can divide the matrix into vertical *stripes* of equal width

and divide the vector into an equal number of horizontal stripes, of the same height.

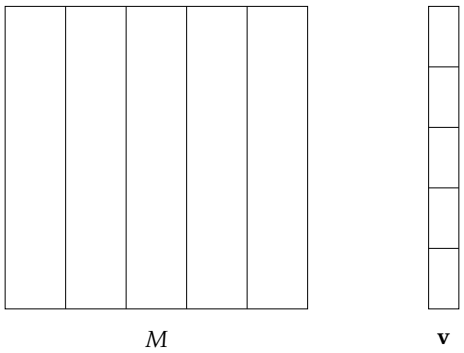

Figure 2.1. *Division of a matrix and vector into stripes.*

Each Map task is assigned a chunk from one of the stripes of the matrix and gets the entire corresponding stripe of the vector. The Map and Reduce tasks can then act exactly as was described above for the case where Map tasks get the entire vector. We shall take up matrix-vector multiplication using MapReduce again in Chapter 3.

2.1.3 Relational-Algebra Operations

A good starting point for exploring applications of MapReduce is by considering the standard operations on relations. A *relation* is a table with column headers called *attributes*, rows of the relation are called *tuples* and the sets of attributes of a relation is called its *schema*.

We often write an expression like $R(A_1, A_2, \ldots, A_n)$ to say that a relation R has the attributes $(A_1, A_2, \ldots, A_n)$

From	*To*
url1	url2
url1	url3
url2	url3
url3	url4
...	...

Figure 2.2. *Relation Links consists of the set of pairs of URL's, such that the first has one or more links to the second.*

A relation can be stored as a file in a distributed file system. There are several standard operations on relations, often referred to as *reletional algebra*, that used to implement queries. The relational-algebra operations are:

Relation (represents a table): $R(A, B) \subseteq A \times B$

1. *Selection*: Apply a condition C to each tuple in the relation and produce as output

only those tuple that satisfy C. The result of this selection is denoted $\sigma_C(R)$.

$$\forall t \in R \xrightarrow{MAP} (t,t) \text{ if } c(t)$$
$$(t,(t)) \xrightarrow{REDUCE} (t,t)$$

Map: For each tuple t in R, test if it satisfies C. If so, produce the key-value pair (t,t).
Reduce: is the identity.

2. *Projection*: For some subset S of the attributes of the relation, produce from each tuple only the components for the attributes in S. The result of this projection is denoted $\pi_S(R)$.

$$\forall t \in R \xrightarrow{MAP} (t',t')$$
$$(t',(t',\ldots,t')) \xrightarrow{REDUCE} (t',t')$$

Map: create a tuple t' that contains only the attributes in S.
Reduce: for each key t' produced by the Map, there will be ore or more key-value pairs (t',t'), so the Reduce function will remove the duplicates.

3. *Union*, *Intersection*, and *Difference*: These operations are defined in the usual way.

$$\text{Union:} \quad \begin{cases} \forall t \in R \xrightarrow{MAP} (t,t)\,, \quad \forall t \in S \xrightarrow{MAP} (t,t) \\ (t,(t)) \xrightarrow{REDUCE} (t,t) \\ (t,(t,t)) \xrightarrow{REDUCE} (t,t) \end{cases}$$

$$\text{Intersection:} \quad \begin{cases} \forall t \in R \xrightarrow{MAP} (t,t)\,, \quad \forall t \in S \xrightarrow{MAP} (t,t) \\ (t,(t)) \xrightarrow{REDUCE} \emptyset \\ (t,(t,t)) \xrightarrow{REDUCE} (t,t) \end{cases}$$

$$\text{Difference:} \quad \begin{cases} \forall t \in R \xrightarrow{MAP} (t,'R')\,, \quad \forall t \in S \xrightarrow{MAP} (t,'S') \\ (t,R) \xrightarrow{REDUCE} (t,t) \\ (t,S) \xrightarrow{REDUCE} \emptyset \\ (t,(R,S)) \xrightarrow{REDUCE} \emptyset \end{cases}$$

Considering the Union:

- **Map**: Turn each input t ubti a key-value pair (t,t)
- **Reduce**: Associated with each key t there will be either one or two values. Produce output (t,t) ub either case.

Considering the Intersection:

- **Map**: Turn each input t ubti a key-value pair (t, t)
- **Reduce**: If key t has value list (t, t), then produce (t, t), otherwise produce nothing.

Considering the Intersection:

- **Map**: For a tuple t in R, produce the key-value pair (t', R'). For a tuple t in S, produce the key-value pair $(t,' S')$.
- **Reduce**: For each key t, if the associated value list is (t', R'), then produce (t, t), otherwise produce nothing.

4. *Natural Join*: Given two realtions, compare each pair of tuples, one from each relation. If the tuples agree on all the attributes that are common to the two schemas, then produce a tuple that has components for each of the attributes in either schema and agrees with the two tuples on each attribute. If the tuples disagree on one or more shared attributes, then produce nothing from this pair of tuples. All *equijoins* can be executed in the same manner. The natural join of $R(A, B)$ and $S(B, C)$ is denoted $R \bowtie S$.

$$\forall (a, b) \in R \xrightarrow{MAP_R} (b, (a,' R'))$$
$$\forall (b, c) \in S \xrightarrow{MAP_S} (b, (c,' S'))$$
$$(b, ((a_1, R), (c_1, S), (c_3, S), (a_2, R), \dots)) \xrightarrow{REDUCE} \begin{cases} 1.\ \text{Sort } \ell \text{ using 2nd element} \\ 2.\ a \text{ list of elements from } R \\ 3.\ b \text{ list of elements from } S \\ 4.\ \forall (a, c) \in a \times b \\ 5.\ \text{Output}(a, b, c) \end{cases}$$

Map: For each tuple (a, b) of R, produce the key-value pair $(b, (a', R'))$. For each tuple (b, c) of S, produce the key-value pair $(b, (c', S'))$.
Reduce: Each key value b will be associated with a list of pairs that are either of the form (R, a) or (S, c). Construct all pairs consisting of one with first component R and the other with first component S. The output from this key and value list is a sequence of key-value pairs. Each value is one of the triple (a, b, c) such that (R, a) and (S, c) are in the value list.

5. *Grouping* and *Aggregation*: Given a relation R, partition its tuples according to their values in one set of attributes G, called the *grouping attributes*. The permitted aggregations are SUM, COUNT, AVG, MIN, and MAX, with the obvious meanings. We denote a grouping-and-aggregation operation on a relation R by $\gamma_X(R)$, where X is a list of element that are either.
   (a) A grouping attribute, or
   (b) An expression $\theta(A)$, where A is an attribute and θ is an aggregation function.

The result of this operation is one tuple for each group, with components for each grouping attribute and for each aggregation.

$$\forall (a,b) \in R \xrightarrow{MAP} (a,b)$$
$$(a,(b_1,\ldots,b_n)) \xrightarrow{REDUCE} (a,\theta(b_1,\ldots,b_n))$$

Map: For each tuple (a,b,c) produce the key-value pair (a,b).
Reduce: Each key a represent a group. Apply the aggregation operator θ to the list $(b_1,\ldots,b_n)$ of values associated with a. The output is the pair (a,x), where x is the result of the aggregation.

2.1.4 Matrix Multiplication

If M is a matrix with element m_{ij} in row i and column j, and N is a matrix with element n_{jk} in row j and column k, then the product $P = MN$ is the matrix P with element p_{ik} in row i and column k, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

Obviously, the number of columns of M must be equal to the number of rows of N, so the sum over j makes sense. We could view matrix M and N as relations, $M(I,J,V)$ with tuples (i,j,m_{ij}) and $N(J,K,V)$ with tuples (j,k,n_{jk}). The product MN is a natural join, $M(I,J,V) \bowtie N(J,K,V)$, having only attribute J in common and produce tuples (i,j,k,v,w) for each tuple (i,j,v) in M and each tuple (j,k,w) in N. This five-component tuple represents the pair of matrix element (m_{ij}, n_{ik}) but we prefer to have a four-component tuple $(i,j,k,v \times w)$ because represents the product between $m_{ij}n_{jk}$.

We can perform grouping using I and K as attributes and aggregation as the sum of $V \times W$, and we can implement matrix multiplication as two MapReduce operation. Let N_c the number of columns of N and M_r the number of rows of M.

$$M = [m_{ij}]_{M_r \times X} \qquad N = [n_{jk}]_{X \times N_c} \qquad P = MN = [p_{ik}]_{M_r \times N_c} \quad p_{ik} = \sum_{j=1}^{X} m_{ij} n_{jk}$$

$$(i,j,m_{ij}) \in M(I,J,V)$$
$$(j,k,n_{jk}) \in N(J,K,W)$$
$$M \bowtie N(i,j,k,m_{ij},n_{jk})$$

First implementation:

$$\forall (i,j,m_{ij}) \in M \xrightarrow{Map_M} (j,(M,i,m_{ik}))$$
$$\forall (j,k,n_{jk}) \in N \xrightarrow{Map_N} (j,(N,k,n_{jk}))$$
$$(j,((M,1,m_{1j}),\ldots,(M,M_r,m_{M_rj})),(N,1,n_{j1}),\ldots,(N,N_c,n_{jN_c})))$$

Map: For each matrix element m_{ij}, produce the key value pair $(j, (M, i, m_{ij}))$ and n_{jk} produce the key value pair $(j, (N, k, n_{jk}))$. M and N in the values are not the matrices but just labels to distinguish the two sources of tuples.

Reduce: For each key j, look at the list of values. For each value that comes from M and N, produce the key-value pair with key equal to (i, k) and value equal to the product of these elements $m_{ij}n_{jk}$.

Now, we perform a grouping and aggregation by another MapReduce operation.

$$((i,j),(a_{ik}b_{kj})) \xrightarrow{Map} ((i,j),(a_{ik}b_{kj}))$$
$$((i,j),(a_{i1}b_{1j})) \xrightarrow{Reduce} ((i,j),p_{ij}) \text{ where } p_{ij} \in P$$

Map: This function is just the identity.

Reduce: For each key (i, k), producethe sum of the list of values associated with this key. The result is the key-value pair with key (i, k) and value equal to the product p_{ik} where p_{ik} is the value of the matrix $P = MN$ in row i and column k.

Second implementation:

We can perform matrix multiplication $P = MN$ in a **single MapReduce operation**. Let N_c the number of columns of N and M_r the number of rows of M.

$$\forall m_{ij} \in M \xrightarrow{Map_M} ((i,k),(M,j,m_{ij})) \quad \forall\, k = 1,2,\ldots,N_c$$
$$\forall n_{jk} \in N \xrightarrow{Map_N} ((i,k),(N,j,n_{jk})) \quad \forall\, i = 1,2,\ldots,M_r$$
$$((i,k),((M,1,m_{i1}),\ldots,(M,X,m_{iX}),(N,1,n_{1j}),\ldots,(N,X,n_{Xk})) \xrightarrow{Reduce} ((i,k),p_{ik})$$

Map: For each matrix element m_{ij}, produce the key value pair $((i, k), (M, j, m_{ij}))$ and for each matrix element n_{jk}, produce the key value pair $((i, k), (N, j, n_{jk}))$.

Reduce: Each key (i, k) have an associated list of values $((M, j, m_{ij}), (N, j, n_{jk}))\ \forall\ j$. The reduce function needs to connect the two values with the same j, an easy way to do this step is sort by j in two separate lists and the j-th element will have the third componenet m_{ij} and n_{jk}, then we can compute the product $m_{ij}n_{jk}$ and sum over j.

2.2 The communication cost model

The communication cost model is a model that allows us to estimate the communication cost of a MapReduce algorithm. The communication cost is the amount of data that is transferred between the mappers and the reducers. The communication cost is the most important factor in the performance of a MapReduce algorithm.

Imagine that an algorithm is implemented by an acyclic network of tasks. The *communication cost* of a task is the size of the input to the task and the task can be measured in bytes. The *communication cost of an algorithm* is the sum of the communication cost of all the tasks implementing that algorithm.

We can explain and jusity the **importance of communication** cost as follows:

- The algorithm execution by each task tends to be very simple.
- The typical interconnect speed for a computing cluster is 1 Gbps, and it is slow compared to the speed of the CPU.
- Even if a task executes at a compute node that has a copy of the chunk(s) on which the task operates, that chunk normally will be stored on disk, and the time taken to move the data into main memory may exceed the time needed to operate on the data once it is available in memory.

We might still ask why we **count only input size**, and not output size. The answare to this question involves two points:

1. If the output of the one task τ is the input of another task, then the size of τ's output will be accounted for when measuring the input size for the receiving task.
2. The output is rarely large compared with the output with the input or the intermediate data produced by the algorithm.

An example of join operation:

$$
\begin{aligned}
R(A,B) \bowtie S(B,C) \\
\forall (a,b) \in R \xrightarrow{M} (b,(a,R)) \\
\forall (b,c) \in S \xrightarrow{M} (b,(c,S)) \\
|R| = r, \qquad |S| = s \\
\text{Cost: } O(r+s)
\end{aligned}
$$

2.2.1 Wall-Clock Time

While communication cost often influences our choice of algorithm to use in a cluster-computing environment, we must also be aware of the importance of *wall-clock time*, the time it takes a parallel algorithm to finish.

2.2.2 Multiway Joins

To see how analyzing the communication cost can help us choose an algorithm in the cluster-computing environment, we can consider the problem of computing a multiway join. There is a general theory in which we:

1. Select certain attributes of the realtions involved in the natural *multi*-join to have their value hashed, each to some number of buckets.
2. Select the number of buckets for each of these attributes, subejct to the contraint that the product of the number for each attribute is k.
3. Idetify each of the k reducers with a vector of bucket numbers. These vectors have one component for each of the attributes selected in step (1).

4. Send tuples of each realtion to all those reducers where it might find tuples to join with. The given tuple t will have values for some of the attributes selected at step (1), in this way we can apply the hash function(s) to those values to determine certain components of the vector that identifies the reducers.

3
Link Analysis

Bibliography