Mining Massive Datasets

Robert T. Bauer

Why Massive Datasets?

- Having the data creates opportunities to improve the world
- Having the data creates entry barriers.
- Having the data allows you to make an inference others can't – this makes entry nearly impossible.

At a very fundamental level, we advance knowledge in two ways:

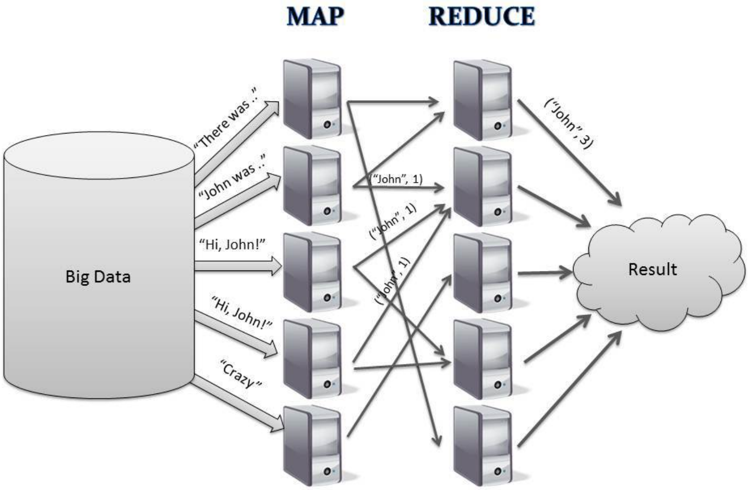
1. Systems of axioms as sources of new theories (called autonomous systems of axioms)
2. Given a set B, usually large and possibility infinite, of preassigned propositions, find a system of axioms from which the propositions follow. This gives the "theory" a physical (more generally, an empirical) basis.

Massive datasets provide an empirical foundation for a heternomous system of axioms.

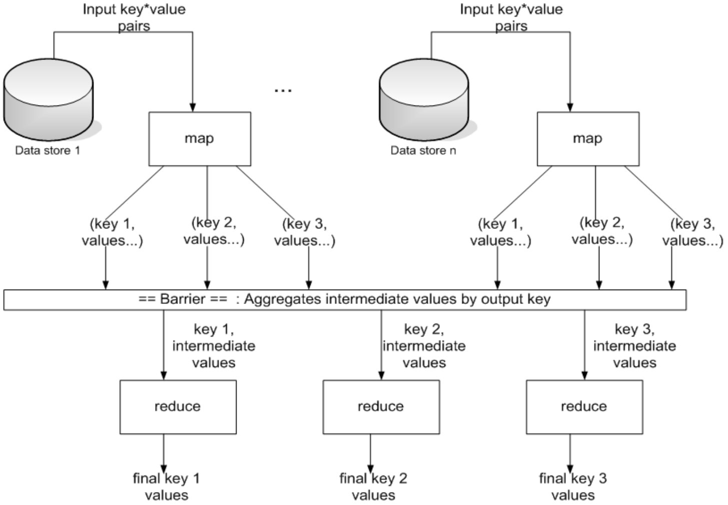
Outline

$$\Longrightarrow$$

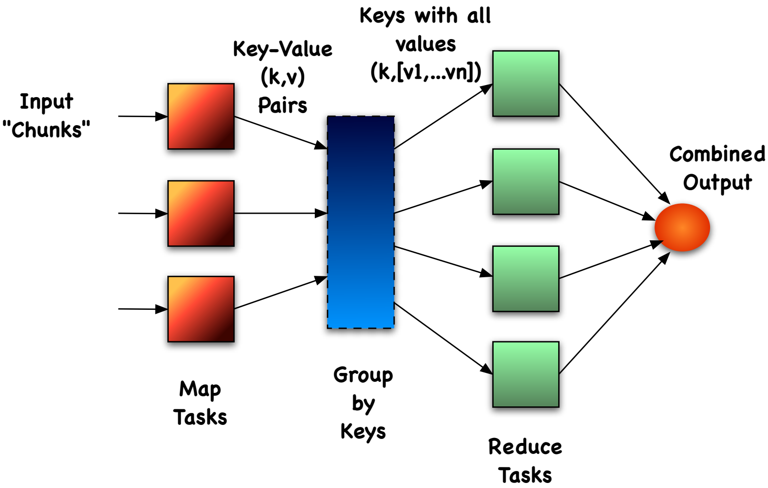
Map-Reduce Computation Schematic I



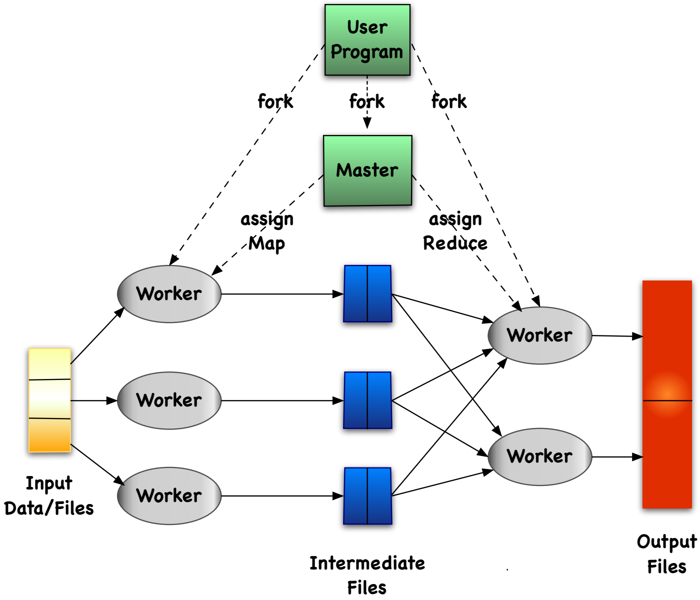
Map-Reduce Computation Schematic II



Map-Reduce Key-Value Pair



Map-Reduce Execution



Algorithms Using Map-Reduce (MR)

MR isn't "the" solution - nor is it the solution for every problem requiring parallel compute nodes. MR works well when we have

very large files that are rarely updated in place.

For example,

- No: Responding to product requests
- Yes: Large matrix-vector multiplication
- No: Recording sales
- Yes: Finding users whose buying patterns are similar
- Yes: Counting

MR Example: Counting Words I

```
text_files=glob.glob('MyFiles/*.txt')

def getFileContents(fname):
 return open(fname).read()

source = dict((fn, getFileContents(fn))
 for fn in text_files)

# tell what do with the MR result
fout = open('MyFiles/outCount','w')
def final(k,v):
 print k,v
 f.write(str((k,v)))
```

MR Example: Counting Words II

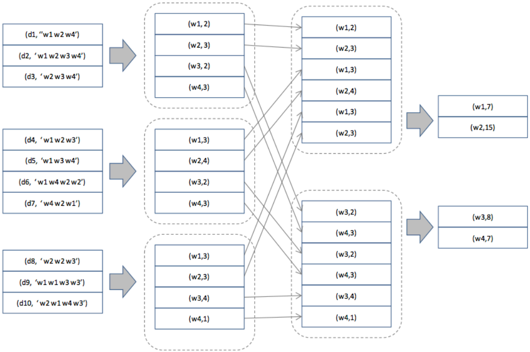
```
# the map function
#  ‘‘return’’ pair (word,1), for each word
def mapfn(k,v):
 for l in v.splitlines():
   for w in l.split():
    yield w.lower(), 1


# the reduce function
#  compute the ‘‘count’’
def reducefn(k,v)
 return k, len(v)
```

MR Example: Counting Words III

Map: $(d_k, `w_1 \ldots w_n') \rightarrow [(w_i, c_i)]$ **document -> word-count pairs**

Reduce: $(w_i, [c_i]) \rightarrow (w_i, \sum_i c_i)$ **word, count-list -> word-count-total**



It's a math problem to determine the optimal number of mappers and reducers; not a tuning problem. Why?

Page Rank

Say we have an "internet" with 3 websites (s_0, s_1 & s_2) with the following probabilities of leaving one site for another:

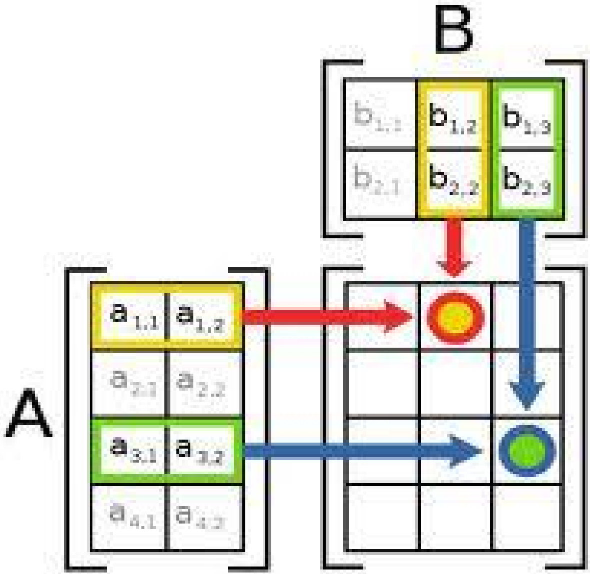
$$P = \begin{bmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{bmatrix}$$

The ergodic behaviour, usually called the *stationary* distribution ("visitation" ratio in performance modelling):

$$\lim_{n\to\infty} P^n = \begin{bmatrix} 0.625 & 0.3125 & 0.0625 \\ 0.625 & 0.3125 & 0.0625 \\ 0.625 & 0.3125 & 0.0625 \end{bmatrix}$$

tells us that 62.5% of the time, site s_0 is "visited", 31.25% it is site s_1 and 6.25% of the time it is site s_2 — site s_0 is the most *popular* and site s_2, the least.

Matrix Multiplication



Matrix × Vector

Multiplying (large) matrices is fundamental to big data.

$$x_i = \sum_j^n m_{ij} v_j$$

When n is on the order of 10^2 map-reduce or even a distributed file system (DFS) isn't the right choice. But what to do when n is on the order of 10^{10}?

Let's tackle the case where n is large, but everything fits into memory.

Map-Reduce (fits in memory): Matrix $\times$ Vector

We store M and v in memory.

We want the reduce function to get key-value pairs:

$$(i, m_{ij} v_j)$$

The reduce function sums these pairs, giving (i, x_i).

Each map task is designed to handle one "chunk" of M. For example, given k map tasks, let

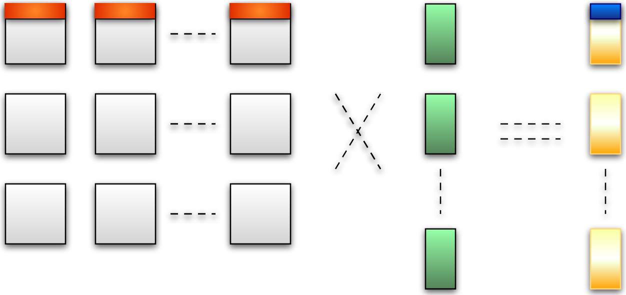
$$\mathtt{mti} \; = \; ((i-1)n + j) \bmod k$$

be the map task id for processing matrix element m_{ij}. The "source" is then the key-value pair $(\mathtt{mti}, (i, j))$.

M and v are (in memory and) available to each map task; the map function, as mentioned above, emits $(i, m_{ij} v_j)$.

Map-Reduce (mem too small): Matrix $\times$ Vector

The technique is to break M into square chunks and to partition v into compatible regions.



Relational Algebra Operations

A (relational) *links* table:

From	To
ulr1	url2
url1	url3
url2	url3
url2	url4
...	...

(Some) Relational Operations:

Selection: "where" clause; $\sigma_C(R)$

Projection: "select" attributes; $\pi_S(R)$

Join: "natural" - match common attributes; $T \bowtie U$

Map-Reduce - Selection

Selection, $\sigma_C(R)$, is very simple and can be done either in the map or reduce function.

Map: For each tuple $t \in R$, if t satisfies C emit key-value pair (t, t).

Reduce: Emit each key-value pair; reduce is the "identity" function.

Map-Reduce - Projection

Projection, $\pi_S(R)$, filters out attributes not in S - this can cause duplicate "rows" in the output.

We use the reduce function to eliminate duplicates.

Map: For each tuple $t \in R$, construct tuple t' by removing components not in the projection.

Emit the key-value pair (t', t').

Reduce: The map-reduce aggegration

$$[(t', t')] \rightarrow (t', [t', t', \ldots, t'])$$

passes $(t', [t', t', \ldots, t'])$ to the reduce function which emits (t', t').

Map-Reduce - Natural Join (Map)

We illustrate a natural join by finding all 2 hop links in the "from-to" table. For example, if we have $\delta(u,t)$ and $\delta(t,v)$, then we can determine that v is "reachable" from u in two hops (i.e. $\delta(u,t) \wedge \delta(t,v) \Rightarrow \delta(u,v)$).

Map: For each tuple $(u,t) \in \mathrm{from\text{-}to}$ emit two key-value pairs:

$$(t, (\mathbf{ONE}, u)) \quad (u, (\mathbf{TWO}, t))$$

Consider the *links*:

$$\delta(u,t) \quad \delta(w,t) \quad \delta(t,x) \quad \delta(t,v)$$

Map emits:

$$\begin{array}{ll} (t, (\mathbf{ONE}, u)) & (u, (\mathbf{TWO}, t)) \\ (t, (\mathbf{ONE}, w)) & (w, (\mathbf{TWO}, t)) \\ (x, (\mathbf{ONE}, t)) & (t, (\mathbf{TWO}, x)) \\ (v, (\mathbf{ONE}, t)) & (t, (\mathbf{TWO}, v)) \end{array}$$

Map-Reduce - Natural Join (Reduce)

The map-reduce aggregration combines:

$$\begin{array}{ll} (t, (\mathbf{ONE}, u)) & (u, (\mathbf{TWO}, t)) \\ (t, (\mathbf{ONE}, w)) & (w, (\mathbf{TWO}, t)) \\ (x, (\mathbf{ONE}, t)) & (t, (\mathbf{TWO}, x)) \\ (v, (\mathbf{ONE}, t)) & (t, (\mathbf{TWO}, v)) \end{array}$$

To get:

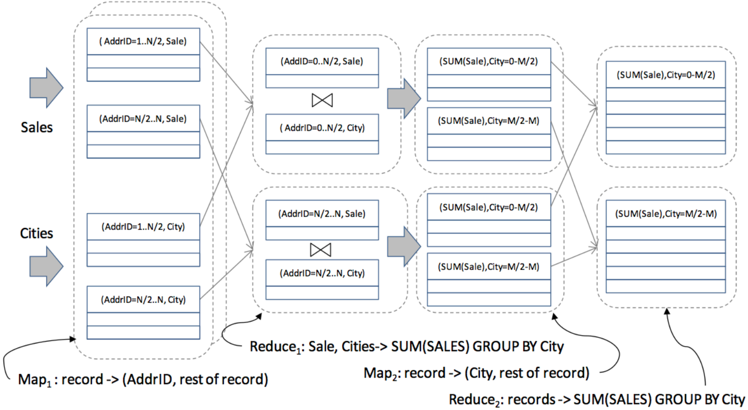
$$\begin{array}{ll} (x, [(\mathbf{ONE}, t)]) & (v, [(\mathbf{ONE}, t)]) \\ (u, [(\mathbf{TWO}, t)]) & (w, [(\mathbf{TWO}, t)]) \end{array}$$

$$(t, [(\mathbf{ONE}, u), (\mathbf{ONE}, w), (\mathbf{TWO}, x), (\mathbf{TWO}, v)]$$

Reduce: Cartesian product $(\mathbf{ONE} \times \mathbf{TWO})$ and filter out the intermediate (t) hop, emitting:

$$(u, x), \quad (u, v), \quad (w, x), \quad (w, v)$$

Map-Reduce SQL Query I

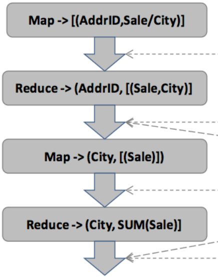


SQL: SELECT SUM(Sale), City FROM Sales, Cities WHERE Sales.AddrID=Cities.AddrID GROUP BY City

Map-Reduce SQL Query II

1. $\mathbf{MAP}_1$ emits $(\text{addrId}, \text{sale})$ or $(\text{addrId}, \text{city})$ depending on "type"
2. $\mathbf{AGGREGATION}_1$ gives $(\text{addrId}, [\text{city}, \text{sale}_1, \ldots])$
3. $\mathbf{REDUCE}_1$ emits $(\text{city}, \sum \text{sale}_i)$
4. $\mathbf{AGGREGATION}_2$ gives $(\text{city}, [\sum \text{sale}, \sum \text{sale}, \ldots])$
5. $\mathbf{REDUCE}_2$ emits $(\text{city}, \sum \sum \text{sale})$

Map-Reduce SQL Query III



Pig Latin:

```
tmp = COGROUP Sales BY AddrID, Cities by AddrID
ioin = FOREACH tmp GENERATE FLATTEN(Sales), FLATTEN(Cities)
grp = GROUP join BY City
res = FOREACH grp GENERATE SUM(Sale)
```

HiveQL:

```
INSERT OVERWRITE TABLE join
SELECT s.Sale, c.City FROM Sales s
       JOIN Cities c ON s.AddrID=c.AddrID;

INSERT OVERWRITE TABLE res
SELECT SUM(join.Sale) FROM join GROUP BY join.City
```

SQL: SELECT SUM(Sale), City from Sales, Cities WHERE Sales.AddrID=Cities.AddrID GROUP BY City

Closing Thoughts

Map-Reduce is fairly straight forward. Understanding the aggregation step is key; there are lots of patterns.

In January, I will cover computational complexity of map-reduce. This will allow us to determine optimal layout of data, how many mappers, reducers, and best way to aggregate data. Working with the computational models also allows us to determine whether map-reduce even makes sense for the problem (there are other parallel approaches).

I hope you have a better understanding of map-reduce and the basic patterns. You can download octo or mincemeat and play with map-reduce on your desktop.

Next month, I will go over recommender systems.

Discussion

Q & A