Computing Scientometrics in Large-Scale Academic Search Engines with MapReduce

Leonidas Akritidis
Panayiotis Bozanis

Department of Computer & Communication Engineering,
University of Thessaly, Greece

13th International Conference on Web Information System Engineering
WISE 2012, November 28-30, Paphos, Cyprus

Scientometrics

- Metrics evaluating the research work of a scientist by assigning impact scores to his/her articles.
- Usually expressed as definitions of the form:
  - A scientist a is of value V if at least V of his articles receive a score $S \geq V$.
- A researcher must author numerous qualitative articles.

h-index

- The first and most popular of these metrics is h-index (Hirsch, 2005).
- A researcher a has h-index h, if h of his/her P^a articles have received at least h citations.
- This metric indicates the impact of an author.
- An author must not only produce numerous papers;
- His/her papers must also be cited frequently.

Time-Aware h-index Variants

- The contemporary and trend h-index (Sidiropoulos, 2006) introduced temporal aspects in the evaluation of a scientist's work.
- They assign to each article of an author time-decaying scores:
  - Contemporary Score: $$S_c^{p_i} = \gamma \frac{\left| P_c^{p_i} \right|}{\left(\Delta Y_i \right)^{\delta}}$$
  - Trend Score: $$S_t^{p_i} = \gamma \sum_{n=1}^{\left| P_c^{p_i} \right|} \frac{1}{\left(\Delta Y_n \right)^{\delta}}$$

Contemporary and Trend h-indices

- $(\Delta Y)_i$: The time (in years) elapsed since the publication of the article i.
- $\gamma = 4, \delta = 1, \left| P_c^{p_i} \right|$: number of papers citing p_i
- Contemporary score: The value of an article decays over time.
- Trend Score: An article is important if it continues to be cited in the present.
- A scientist a has contemporary h-index h_c if at least h_c of his articles receive a score $S_c \geq h_c$

Scientometrics Computation

- Easy for small datasets
- For h-index we just need to identify the articles of each researcher and enumerate all their incoming citations.
- However, for large datasets the computation becomes more complex:
  - The data (authors, citations, and metadata) do not fit in main memory.
  - Tens of millions of articles and authors

Academic Search Engines

John Smith

Information retrieval - Web search - Web mining - similarity search - algorithms and data structures

Google scholar

Search Authors

Citation indices

	All	Since 2007
Citations	??	??
h-index	??	??
i10-index	??	??

Citations to my articles

Microsoft Academic Search Beta

All Domains

Explore 38,855,528 publications and 19,018,573 authors, 3,369 updated last week.

Advanced Search

Academic > Authors > **John Smith**

Embed Subscribe

John Smith

Publications: ?? | Citations: ?? | G-Index: ?? | H-Index: ??

Interests: Hardware & Architecture, Software Engineering, Information Retrie

Collaborated with ?? co-authors from ?? to ?? ; Cited by ?? authors

Homepage | Bing

Microsoft Academic Search Beta

MapReduce (1)

- MapReduce: A fault tolerant framework for distributing problems to large clusters.
- The input data is split in chunks; each chunk is processed by a single Worker process (Map).
- Mapper outputs are written in intermediate files.
- In the sequel, the Reducers process and merge the Mappers' outputs.
- The data is formatted in key-value pairs.

MapReduce (2)

- The MapReduce algorithms are expressed by writing only two functions:
  - Map: $map(k_1, v_1) \rightarrow list(k_2, v_2)$
  - Reduce: $reduce(k_2, list(v_2)) \rightarrow list(k_3, v_3)$
- The MapReduce Jobs are deployed on top of a distributed file system which serves files, replicates data, and transparently addresses hardware failures.

Combiners

- An optional component standing between the Mappers and the Reducers.
- It serves as a mini-Reducer; it merges the values of the same keys for each Map Job.
- It reduces the data exchanged among system nodes, thus saving bandwidth.
- Implementations
  - Explicit: Declare a `Combine` function
  - In-Mapper: merge the values of the same key within `map`.
    - Guaranteed execution!

Parallelizing the Problem

- Goal: Compute Scientometrics in parallel
- Input: $\left(p_i, C^{p_i}\right) \rightarrow \left(paperID, paperContent\right)$
- Output: $\left(a, M_x^a\right) \rightarrow \left(author, metric\right)$
- To reach our goal, we have to construct for each author, a list of his/her articles sorted by decreasing score:

$$\left(a, SortedList\left[\left(p_1, S_x^{p_1}\right), \left(p_2, S_x^{p_2}\right), ..., \left(p_N, S_x^{p_N}\right)\right]\right)$$

- Then, we just iterate through the list and we compute the desired metric value.

Methods 1,2 – `Map` Phase

```
1: class Mapper
2:     method map (integer p_i; string C^{p_i})
3:         P ← ExtractReferences(C^{p_i})
4:         for all references p ∈ P
5:             S^p ← ComputeScore(p)
6:             A^p ← ExtractAuthors(p)
7:             for all authors a ∈ A^p
8a:                emit (a, pair[p, S^p])
8b:                emit (pair[a, p], S^p)
```

(author, <paper,score>)

(<author,paper>, score)

- Notice: We emit the *references' authors, not* the paper's authors

Method 1, `Reduce` Phase

- Reducer Input: (`author`, *`pair`* `<paper,score>`)
- Create an associative array which stores for each author, a list of `<paper,score>` pairs.
- Sum partial paper scores
- Sort the array by descending `score`.
- Compute metric.

Algorithm 2 Method 1, Reducer class

```
1: class Reducer
2:     method reduce (string a; pairs[integer p, float S^p])
3:         H ← new AssociativeArray
4:         for all pair v ∈ pairs[integer p, float S^p]
5:             if v.p ∈ H
6:                 H^p.S ← H^p.S + v.S^p
7:             else
8:                 H.add(v)
9:         sort H in descending S order
10:        integer papers ← 0, metric ← 0
11:        for all pairs ∈ H
12:            papers ← papers + 1
13:            if H^p.S ≥ papers
14:                metric ← metric + 1
15:            else
16:                stop iteration
17:        emit (a, metric)
```

Method 2, `Reduce` Phase

- Reducer Input: (*pair* <author,paper>, score)
- Keys sorted by `author` and `paper` (secondary sort).
- We create an associative array which stores for each `author`, a list of <paper,score> pairs
- Compute metric

Algorithm 3 Method 2, Reducer class

```
1: class Reducer
2:     method initialize
3:         string a_prev ← ""
4:         integer p_prev ← 0
5:         integer n ← 0
6:         H ← new Array
7:     method reduce (pair[string a, integer p]; float S^p)
8:         if a = a_prev
9:             if p = p_prev
10:                H(n) ← H(n) + S^p
11:            else
12:                H.add(S^p)
13:                n ← n + 1
14:        else
15:            Perform steps 9–17 of Algorithm 2
16:            H.reset()
17:            a_prev ← a
18:            p_prev ← p
19:            n ← 0
20:    method close
21:        emit (a, metric)
```

Method 1-C, `Map-Reduce`

Algorithm 4 Method 1-C: Improved version of method 1 with Combiners

```
1: class Mapper
2:     method initialize
3:         H ← new AssociativeArray
4:     method map (integer p_i; string C^{p_i})
5:         P ← ExtractReferences(C^{p_i})
6:         for all references p ∈ P
7:             S^p ← ComputeScore(p)
8:             A^p ← ExtractAuthors(p)
9:             for all authors a ∈ A^p
10:                if a ∉ H
11:                    L^a ← new Array
12:                    L^a.add(p, S^p)
13:                    H.add(a, L^a)
14:                else
15:                    if p ∉ H.L^a
16:                        H.L^a.add(p, S^p)
17:                    else
18:                        H.L^a.update(p, +S^p)
19:    method close
20:        for all authors a ∈ H
21:            emit (a, list(p, S^p))
```

Merge unique authors

```
22: class Reducer
23:     method reduce (string a; list[integer p, float S^p])
24:         H ← new AssociativeArray
25:         for all pair v ∈ list[integer p, float S^p]
26:             if v.p ∈ H
27:                 H^p.S ← H^p.S + v.S^p
28:             else
29:                 H.add(v)
30:         Perform steps 9–17 of Algorithm 2
```

- In-Mapper Combiner (unique keys).
- `map` emits `author` as key, and a list of `<paper,score>` pairs as value.
- The Reducer merges the lists associated with the same key.
- The list is sorted by `score`

Method 2-C, `Map-Reduce`

Algorithm 5 Method 2-C: Improved version of method 2 with the introduction of in-Mapper Combiners. The Reducer is identical to the one of Algorithm 3.

```
1: class Mapper
2:     method initialize
3:         H ← new AssociativeArray
4:     method map (integer p_i; string C^{p_i})
5:         P ← ExtractReferences(C^{p_i})
6:         for all references p ∈ P
7:             S^p ← ComputeScore(p)
8:             A^p ← ExtractAuthors(p)
9:             for all authors a ∈ A^p
10:                if pair(a, p) ∉ H
11:                    H.add(pair(a, p), S^p)
12:                else
13:                    H.update(pair(a, p), +S^p)
14:    method close
15:        for all pairs (a, p) ∈ H
16:            emit (pair(a, p), S^p)
```

```
22: class Reducer
23:     method reduce (string a; list[integer p, float S^p])
24:         H ← new AssociativeArray
25:         for all pair v ∈ list[integer p, float S^p]
26:             if v.p ∈ H
27:                 H^p.S ← H^p.S + v.S^p
28:             else
29:                 H.add(v)
30:         Perform steps 9–17 of Algorithm 2
```

- In-Mapper Combiner (unique keys).
- `map` emits `<author,paper>` pairs as key, and `<score>` as value.
- The Reducer merges the lists associated with the same key.
- The list is sorted by `score`

Experiments

- We applied our algorithms at the CiteSeerX dataset, an open repository comprised of 1,8 million research articles.
- We used the XML version of the dataset.
  - Total input size: ~28 GB.
  - Small, but the largest publicly available.

Statistic	Value
Input Records	1,844,272
Input Size	27.6 GB
Output Records	2,865,282
Output Size	39.9 MB

Table 3. Problem input-output statistics

MapReduce I/O Sizes

- The methods which employ Combiners perform reasonably better
- Method 1-C: The Mappers produce 21.7 million key-value records (gain ~41%). Total output size = 600MB (gain ~13% less bandwidth).
- Method 2-C: 34.2 million records and 643 MB (~7%).

Method	Mapper Output		Reducer
	Records	Size (MB)	Input Groups
method 1	36,687,999	688.4	2,865,282
method 2	36,687,999	688.4	12,260,311
method 1-C	21,736,395	600.8	2,865,282
method 2-C	34,251,437	643.2	12,260,311

Table 4. Record counts and data sizes for the four examined methods

Running Times

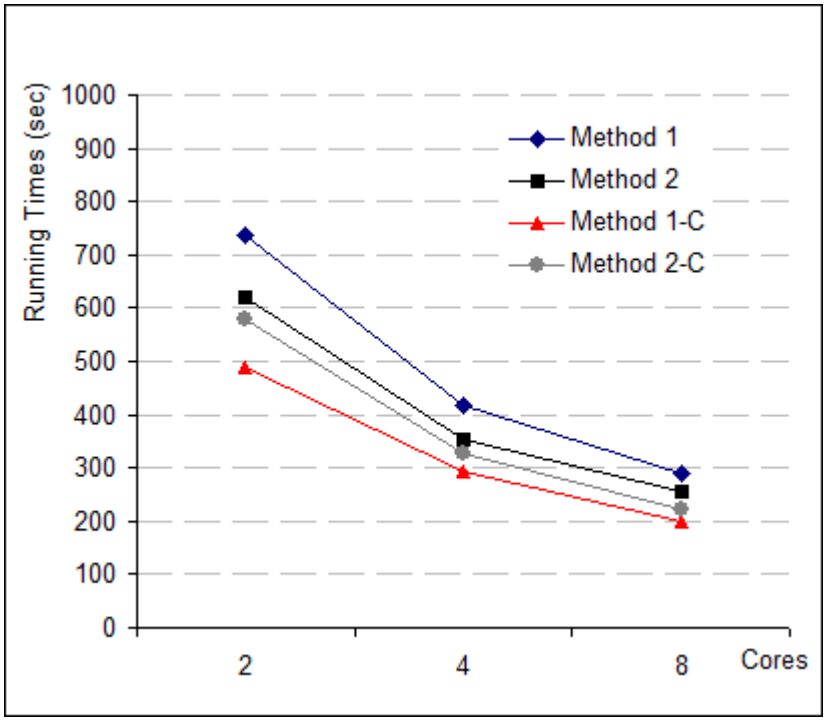
- We also measured the running times of the four methods on two clusters:
  - A local, small cluster comprised of 8 CoreI7 processing cores.
  - A commercial Web cloud infrastructure with up to 40 processing cores.
- On the first cluster, we replicated the input data across all nodes. On the second case, we are not aware of the data physical location.

Running Times

- All four methods have the same computational complexity.
- We expect timings proportional to the size of the data exchanged among the Mappers and the Reducers.
- This favors the Methods 1-C and 2-C which limit data transfers via the Combiners.

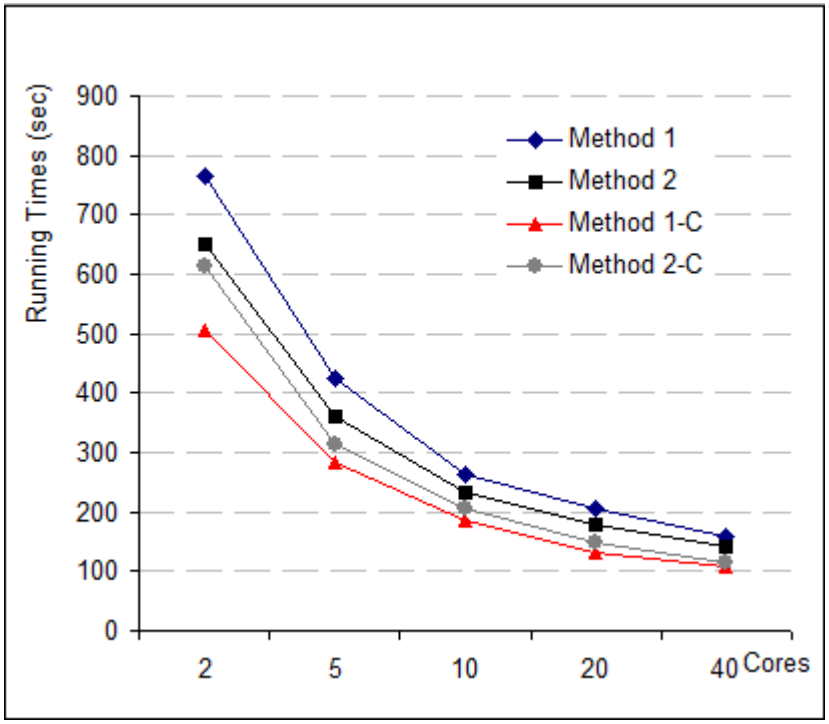
Running Times – Local Cluster

- All methods scale well to the size of the cluster
- Method 1C is the fastest
  - It outperforms method 2-C by ~18%
  - It outperforms method 1 by 30-35%



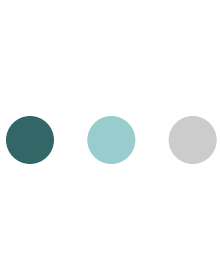
Running Times – Web Cluster

- We repeated the experiment on a Web cloud infrastructure.
- Running times between the two clusters are not comparable.
  - Different hardware and architecture
- Method 1-C is still the fastest



Conclusions

- We studied the problem of computing author evaluation metrics (scientometrics) in large academic search engines with MapReduce.
- We introduced four approaches.
- We showed that the most efficient strategy is to create one list of `<paper,score>` pairs for each unique author during the `map` phase.
- In this way we achieve at least 20% reduced running times and we gain ~13% bandwidth.



Thank you!

Any Questions?