Acknowledgements

I am indebted to Prof. Dr.-Ing. Sebastian Michel, Prof. Johan Gamper, and Dr. Holubova for providing me slides.
N-Grams

• Statistics about **variable-length word sequences** (contiguous)
  (e.g., frequency of bigrams such as aa, ab, ac, ..., xx) have **many applications** in fields including
  – Information Retrieval
  – Natural Language Processing
  – Digital Humanities

• E.g., [http://books.google.com/ngrams/](http://books.google.com/ngrams/)

• A n-gram dataset is also available from there

n-gram slides based on a talk by Klaus Berberich
Google Books Ngrams

Graph these case-sensitive comma-separated phrases:

- xml
- p2p
- distributed database

between 1995 and 2002 from the corpus English with smoothing of 3.

Search lots of books

The graph shows the trend of the usage of these phrases from 1995 to 2002.
n-grams Example

• **Document:** a x b b a y

• **Possible n-grams:**
  – length 1: (a), (x), (b), (y)
  – length 2: (ax), (xb), (bb), ...
  – length 3: (axb), (xbb), ...
  – length 4: (axbb), (xbb), (bbay)
  – length 5: (axbba), (xbbay)
  – length 6: (axbbay)
Computing n-grams in MR

• Given a set of documents.

• How can we efficiently compute n-grams, that
  – occur at least $\tau$ times
  – and consist of at most $\sigma$ words

using MapReduce?

Klaus Berberich, Srikanta J. Bedathur: Computing n-gram statistics in MapReduce. EDBT 2013:101-112
Example

• Let us consider the following three documents:

  \[ d_1 = [ a \times b \times x ] \]
  \[ d_2 = [ b \times a \times b \times x ] \]
  \[ d_3 = [ x \times b \times a \times b ] \]

• With parameters \( \tau=3 \) and \( \sigma=3 \) we expect as output:

  \[
  \begin{align*}
  <a> & : 3 & <b> & : 5 & <x> & : 7 \\
  <ax> & : 3 & <xb> & : 4 \\
  <axb> & : 3
  \end{align*}
  \]
Naïve Solution: Simple Counting

**map** (did, content):

```python
for k in <1 ... σ>:
    for all k-grams in content:
        emit(k-gram, did)
```

**reduce** (n-gram, list<did>):

```python
if length(list<did>) >= τ:
    emit(n-gram, length(list<did>))
```

Note: if a k-gram appears multiple times in the document, it is also emitted multiple times.
A Priori Based

• (Famous) **A priori Principle**: $k$-gram can occur more than $\tau$ times only if its constituent $(k-1)$-grams occur at least $\tau$ times

(a,b,c) qualified **only**

if (b,c), (a,b) and (a), (b), (c)

How to implement?

*) Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami: Mining Association Rules between Sets of Items in Large Databases. SIGMOD Conference 1993: 207-216
A Priori Based (Cont’d)

• **Iterative Implementation:**
  – First 1-grams that occur τ times
  – Then 2-grams that occur τ times
  – ...

• Needs **multiple MapReduce rounds** (of full data scans)

• **Already determined k-grams are kept**
Suffix Based*

• **Emit only suffixes** in map phase
• Each of them **represents multiple n-grams** corresponding to its prefixes
  
  – For instance, **AXBXX** represents
    
    • a, ax, axb, axbx, and axbxx

```python
map(did, content): for all suffixes in content:
    emit(suffix, did)
```

*) Klaus Berberich, Srikanta J. Bedathur: Computing n-Gram Statistics in MapReduce. EDBT 2013: 101-112
Suffix Based: Partitioning

• Partition the suffixes **by first word**
  – to ensure all n-grams end up properly for counting, that is:
    • all occurrences of *ax* have to end up at same reducer
    • suffix property: *ax* is only generated from suffixes that start with *ax*..

```python
partition(suffix, did):
    return suffix[0] % m
```

Analogously with custom grouper.
Suffix Based: Sorting

- Reducer has to generate n-grams based on suffixes
  - read prefixes
  - count for each observed prefix its frequency
  - optimization: sort suffixes in reverse lexicographic order
  - then: simple counting using stack

```c
compare(suffix0, suffix1):
  return -strcmp(suffix0, suffix1)
```
Discussion

• Let’s assess aforementioned algorithms with respect to properties like:
  – multiple MapReduce jobs vs. single job
  – amount of network traffic
  – ease of implementation
Literature

• Jeffrey Dean und Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters“. Google Labs.


• Klaus Berberich, Srikanta J. Bedathur: Computing n-gram statistics in MapReduce. EDBT 2013: 101-112

• Rakesh Agrawal, Tomasz Imielinski, Arun N. Swami: Mining Association Rules between Sets of Items in Large Databases. SIGMOD Conference 1993: 207-216


GRAPH PROCESSING IN MAPREDUCE
Graph Processing in MapReduce

- General: Graph Representation
  - usually: Adjacency list
    - v1 -> v2, v4, v5
    - v2 -> v4
    - v3 -> v5
    - ...

![Graph Diagram](image)
Graph algorithms and MapReduce

A centralized algorithm typically traverses a tree or a graph one item at a time (there’s only one “cursor”)

Most algorithms that are based on graphs make use of multiple map/reduce stages processing one “wave” at a time

Sometimes iterative MapReduce, other times chains of map/reduce
Iterative MapReduce

The basic model:

```plaintext
copy files from input dir → staging dir 1
(optional: do some preprocessing)

while (!terminating condition) {
    map from staging dir 1
    reduce into staging dir 2
    move files from staging dir 2 → staging dir1
}

(optional: postprocessing)
move files from staging dir 2 → output dir
```

Note that reduce output must be compatible with the map input!

What can happen if we filter out some information in the mapper or in the reducer?
MapReduce on Graphs

Suppose we want to:
compute a function for each vertex in a graph...
... using data from vertices at most k hops away

We can do this as follows:
"Push" information along the edges
  "Think like a vertex"
Finally, perform the computation at each vertex

May need more than one MapReduce phase
Iterative MapReduce: Outputs of stage $i \rightarrow$ inputs of stage $i+1$
Path-Based Algorithms

Sometimes our goal is to compute information about the paths (sets of paths) between nodes.

Edges may be annotated with cost, distance, or similarity.

Examples of such problems:

- Shortest path from one node to another Minimum spanning tree (minimal-cost tree connecting all vertices in a graph)

- Steiner tree (minimal-cost tree connecting certain nodes)

- Topological sort (node in a DAG comes before all nodes it points to)
Single-Source Shortest Path

Given a directed graph $G = (V, E)$ in which each edge $e$ has a cost $c(e)$:

- Compute the cost of reaching each node from the source node $s$ in the **most efficient way** (potentially after multiple 'hops')
SSSP: Intuition

We can formulate the problem using induction

The shortest path follows the principle of optimality: the last step \((u,v)\) makes use of the shortest path to \(u\)

We can express this as follows:

```cpp
bestDistanceAndPath(v) { 
    if (v == source) then { 
        return <distance 0, path [v]>
    } else { 
        find argmin_u (bestDistanceAndPath[u] + dist[u,v])
        return <bestDistanceAndPath[u] + dist[u,v], path[u] + v>
    }
}
```
SSSP

Traditional approach: Dijkstra's algorithm

V: vertices, E: edges, S: start node

foreach v in V
    dist_S_to[v] := infinity
    predecessor[v] = nil
spSet = {}
Q := V
while (Q not empty) do
    u := Q.removeNodeClosestTo(S)
    spSet := spSet + {u}
    foreach v in V where (u,v) in E
        if (dist_S_To[v] > dist_S_To[u]+cost(u,v)) then
            dist_S_To[v] = dist_S_To[u] + cost(u,v)
            predecessor[v] = u

Initialize length and last step of path to default values

Update length and path based on edges radiating from u
SSSP: Dijkstra in Action

Q = {s,a,b,c,d}  
spSet = {}

dist_S_To: {(a,∞), (b,∞), (c,∞), (d,∞)}

predecessor: {(a,nil), (b,nil), (c,nil), (d,nil)}
SSSP: Dijkstra in Action

Q = \{a,b,c,d\}  \quad \text{spSet} = \{s\}

dist_S_To: \{(a, 10), (b, \infty), (c, 5), (d, \infty)\}

predecessor: \{(a, s), (b, \text{nil}), (c, s), (d, \text{nil})\}
SSSP: Dijkstra in Action

Q = \{a, b, d\}   \quad spSet = \{c, s\}

\text{dist}_S_{ \text{To}} = \{(a, 8), (b, 14), (c, 5), (d, 7)\}

\text{predecessor: } \{(a, c), (b, c), (c, s), (d, c)\}
SSSP: Dijkstra in Action

Q = {a,b}  spSet = {c,d,s}
dist_S_To: {(a,8), (b,13), (c,5), (d,7)}
predecessor: {(a,c), (b,d), (c,s), (d,c)}
Q = \{b\} 
spSet = \{a,c,d,s\} 
dist_S_To: \{(a,8), (b,9), (c,5), (d,7)\} 
predecessor: \{(a,c), (b,a), (c,s), (d,c)\}
SSSP: Dijkstra in Action

Q = {}
spSet = {a, b, c, d, s}
dist_S_To: {(a, 8), (b, 9), (c, 5), (d, 7)}
predecessor: {(a, c), (b, a), (c, s), (d, c)}
SSSP: How to parallelize?

Dijkstra traverses the graph along a single route at a time, prioritizing its traversal to the next step based on total path length (and avoiding cycles)

No real parallelism!

Intuitively, we want something that “radiates” from the origin, one “edge hop distance” at a time

Each step outwards can be done in parallel, before another iteration occurs - or we are done

Recall our earlier discussion: Scalability depends on the algorithm, not (just) on the problem!
Dijkstra’s algorithm carefully considered each u in a way that allowed us to prune certain points. Instead we can look at all potential u’s for each v compute iteratively, by keeping a “frontier set” of u nodes i edge-hops from the source.
SSSP: MapReduce formulation

init:
For each node, node ID $\rightarrow \langle \infty, -, \{<\text{succ-node-ID}, \text{edge-cost}>\} \rangle$

map:
take node ID $\rightarrow \langle \text{dist}, \text{next}, \{<\text{succ-node-ID}, \text{edge-cost}>\} \rangle$
For each succ-node-ID:
emit succ-node ID $\rightarrow \{<\text{node ID}, \text{distance}+\text{edge-cost}>\}$
emit node ID $\rightarrow \text{distance},\{<\text{succ-node-ID}, \text{edge-cost}>\}$

reduce:
distance := min cost from a predecessor; next := that predec.
emit node ID $\rightarrow \langle \text{distance}, \text{next}, \{<\text{succ-node-ID}, \text{edge-cost}>\} \rangle$

Repeat until no changes
Postprocessing: Remove adjacency lists
Iteration 0: Base case

mapper:  (a,<s,10>)  (c,<s,5>)  edges

reducer:  (a,<10, ...>)  (c,<5, ...>)

"Wave"
Iteration 1

mapper:  (a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) (b,<a,11>)
         (b,<c,14>) (d,<c,7>)

edges

reducer: (a,<8, ...>) (c,<5, ...>) (b,<11, ...>) (d,<7, ...>)
Iteration 2

mapper: \((a,<s,10>) (c,<s,5>) (a,<c,8>) (c,<a,12>) (b,<a,11>)\)
\((b,<c,14>) (d,<c,7>)\) (b,\(<d,13>) (d,\(<b,15>)\) edges

reducer: \((a,<8>) (c,<5>) (b,<11>) (d,<7>)\)
Question: If a vertex's path cost is the same in two consecutive rounds, can we be sure that this vertex has converged?
Summary: SSSP

Path-based algorithms typically involve iterative map/reduce. They are typically formulated in a way that traverses in “waves” or “stages”, like breadth-first search. This allows for parallelism. They need a way to test for convergence.

Example: Single-source shortest path (SSSP)

Original Dijkstra formulation is hard to parallelize. But we can make it work with the "wave" approach.
BFS in MapReduce

• How to implement Breadth First Search in MapReduce?

• Hint: Need to pass on structure (as seen) before. Augment nodes with additional information: visited, distance.