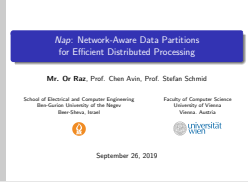
Nap: Network-Aware Data Partitions for Efficient Distributed Processing

Mr. Or Raz, Prof. Chen Avin, Prof. Stefan Schmid

School of Electrical and Computer Engineering
Ben-Gurion University of the Negev
Beer-Sheva, Israel

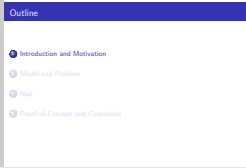
Faculty of Computer Science
University of Vienna
Vienna, Austria

September 26, 2019



Hello everyone, my name is Or Raz, I am a Master graduate from the school of Electrical and Computer Engineering in Ben-Gurion University of the Negev, Israel. This research has been done with the support of Professors Chen Avin and Stefan Schmid, and my Thesis is mainly about this work. Today, I will talk about Nap, a scheme that takes the network into consideration when partitioning the data, and therefore minimizes the completion time in distributed processing frameworks, such as Hadoop.

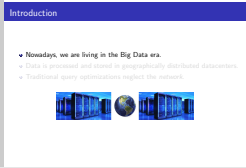
Outline



First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Introduction

- Nowadays, we are living in the Big Data era.
- Data is processed and stored in geographically distributed datacenters.
- Traditional query optimizations neglect the *network*.



- The amount of data queried and processed by emerging applications is growing explosively (in many fileds such as health, business, and science).
- Traditionally, data processing frameworks were designed to run in Homogeneous environments or within a single datacenter, but today it is les common with more Geographically distributed processing.
- Because the scale of data and the data itself is generated in a geographicall distributed fashion (IOT).
- Therefore, to maximize performance, we need to consider the available network resources which has been neglected in the optimization analysis, otherwise we could have a poor performance (wide-area analytics).

Introduction

- Nowadays, we are living in the Big Data era.
- Data is processed and stored in geographically distributed datacenters.
- Traditional query optimizations neglect the *network*.

- The amount of data queried and processed by emerging applications is growing explosively (in many fileds such as health, business, and science).
- Traditionally, data processing frameworks were designed to run in Homogeneous environments or within a single datacenter, but today it is les common with more Geographically distributed processing.
- Because the scale of data and the data itself is generated in a geographicall distributed fashion (IOT).
- Therefore, to maximize performance, we need to consider the available network resources which has been neglected in the optimization analysis, otherwise we could have a poor performance (wide-area analytics).

Introduction

- Nowadays, we are living in the Big Data era.
- Data is processed and stored in geographically distributed datacenters.
- Traditional query optimizations neglect the *network*.

Introduction
- Nowadays, we are living in the Big Data era.
- Data is processed and stored in geographically distributed datacenters.
- Traditional query optimizations neglect the network.

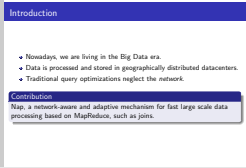
- The amount of data queried and processed by emerging applications is growing explosively (in many fileds such as health, business, and science).
- Traditionally, data processing frameworks were designed to run in Homogeneous environments or within a single datacenter, but today it is les common with more Geographically distributed processing.
- Because the scale of data and the data itself is generated in a geographicall distributed fashion (IOT).
- Therefore, to maximize performance, we need to consider the available network resources which has been neglected in the optimization analysis, otherwise we could have a poor performance (wide-area analytics).

Introduction

- Nowadays, we are living in the Big Data era.
- Data is processed and stored in geographically distributed datacenters.
- Traditional query optimizations neglect the *network*.

Contribution

Nap, a network-aware and adaptive mechanism for fast large scale data processing based on MapReduce, such as joins.



- Our contribution is Nap, a mechanism which minimizes the completion tim in a network-aware manner and is optimized to the current network conditions. In addition, it doesn't require any logic modifications where it only fools the application for a better partitioning of the data.
- We are particularly interested in workloads based on relational databases an consider the most fundamental operation in distributed data processing: joins.

Multiway Join

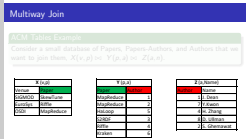
ACM Tables Example

Consider a small database of Papers, Papers-Authors, and Authors that we want to join them, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.

X (v,p)	
Venue	Paper
SIGMOD	SkewTune
EuroSys	Riffle
OSDI	MapReduce

Y (p,a)	
Paper	Author
MapReduce	1
MapReduce	2
HaLoop	5
S2RDF	3
Riffle	4
Kraken	6

Z (a,Name)	
Author	Name
1	J. Dean
7	Y.Kwon
4	H. Zhang
8	D. Ullman
2	S. Ghemawat



First, lets take a look on these three tables that has two joint attributes, p and a.

Multiway Join

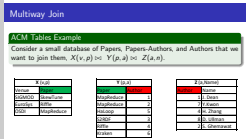
ACM Tables Example

Consider a small database of Papers, Papers-Authors, and Authors that we want to join them, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.

X (v,p)	
Venue	Paper
SIGMOD	SkewTune
EuroSys	Riffle
OSDI	MapReduce

Y (p,a)	
Paper	Author
MapReduce	1
MapReduce	2
HaLoop	5
S2RDF	3
Riffle	4
Kraken	6

Z (a,Name)	
Author	Name
1	J. Dean
7	Y.Kwon
4	H. Zhang
8	D. Ullman
2	S. Ghemawat

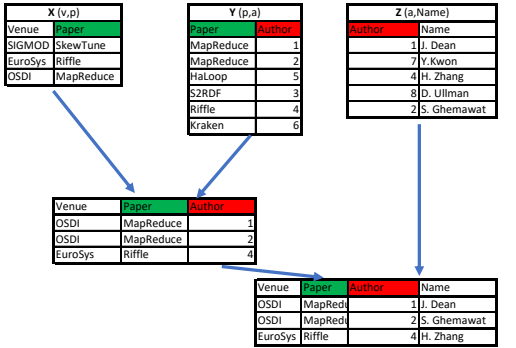


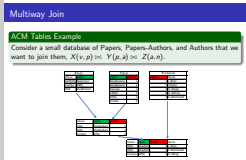
We consider an operation which joins all of these tables, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$ where $\bowtie$ denotes the join operator.
Attributes: v - the Venue, p - the Paper ID, a - the Author ID, and n - the Author name.

Multiway Join

ACM Tables Example

Consider a small database of Papers, Papers-Authors, and Authors that we want to join them, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.





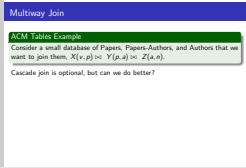
The traditional way to join these tables is by a cascade join or a sequential join, which takes few phases or only two for this example. This fundamental operation and others are done in a distributed manner, when the tables are too large for storing in one computer.

Multiway Join

ACM Tables Example

Consider a small database of Papers, Papers-Authors, and Authors that we want to join them, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.

Cascade join is optional, but can we do better?



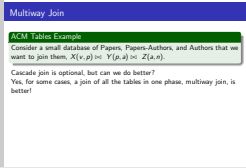
This raise the question of can we do better?
Afrati and Ullman in "Optimizing multiway joins" have shown back in 2011 a way to join all the tables in one phase with MapReduce paradigm, a multiway join. This method involves replication of the tables. I am going to focus about this example today, due it's clear efficiency and because the join operation has been used in many "Big Data" frameworks, including implementations in MapReduce.

Multiway Join

ACM Tables Example

Consider a small database of Papers, Papers-Authors, and Authors that we want to join them, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.

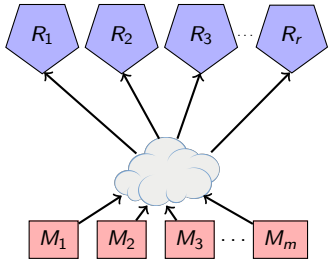
Cascade join is optional, but can we do better?
Yes, for some cases, a join of all the tables in one phase, multiway join, is better!



This raise the question of can we do better?
Afrati and Ullman in "Optimizing multiway joins" have shown back in 2011 a way to join all the tables in one phase with MapReduce paradigm, a multiway join. This method involves replication of the tables. I am going to focus about this example today, due it's clear efficiency and because the join operation has been used in many "Big Data" frameworks, including implementations in MapReduce.

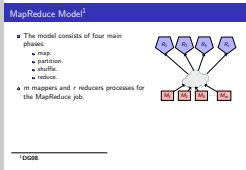
MapReduce Model[1]

- The model consists of four main phases:
  - map.
  - partition.
  - shuffle.
  - reduce.
- m mappers and r reducers processes for the MapReduce job.



[1] **DG08**.

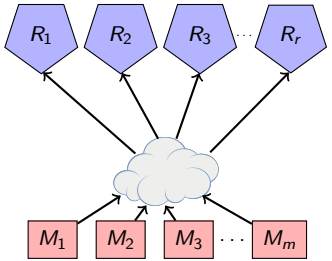
[a] **DG08**.

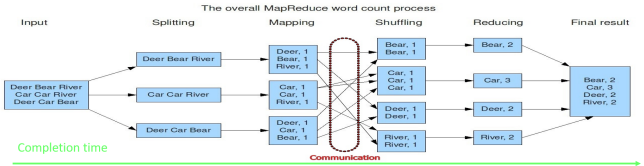


- This is a state-of-the-art programming model for processing large data sets using parallelism and decentralization concepts.
- It has a Master-Slave architecture.
- All the phases are performed locally except the shuffle, which transfer data between the processes, and thus it can be a bottleneck for the whole execution.
- It is highly useful, and efficient tool for large-scale fault tolerant data analysis with crash recovery mechanism.
- Apache Hadoop (batch processing with massive amounts of data) and Apache Spark (overcome latency and the inability to stream data by exploiting the memory) are software frameworks that use this paradigm.

MapReduce Model[2]

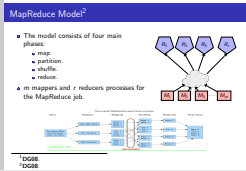
- The model consists of four main phases:
  - map.
  - partition.
  - shuffle.
  - reduce.
- *m* mappers and *r* reducers processes for the MapReduce job.





[1] **DG08**.

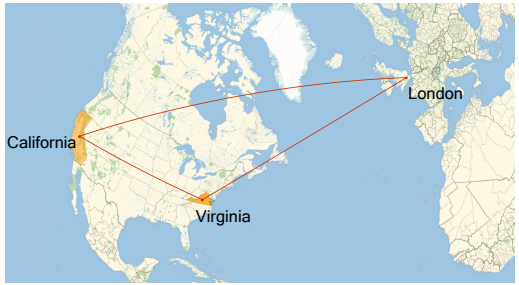
[a] **DG08**.



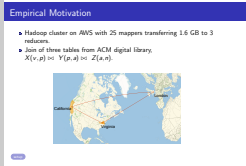
- This is a state-of-the-art programming model for processing large data sets using parallelism and decentralization concepts.
- It has a Master-Slave architecture.
- All the phases are performed locally except the shuffle, which transfer data between the processes, and thus it can be a bottleneck for the whole execution.
- It is highly useful, and efficient tool for large-scale fault tolerant data analysis with crash recovery mechanism.
- Apache Hadoop (batch processing with massive amounts of data) and Apache Spark (overcome latency and the inability to stream data by exploiting the memory) are software frameworks that use this paradigm.

Empirical Motivation

- Hadoop cluster on AWS with 25 mappers transferring 1.6 GB to 3 reducers.
- Join of three tables from ACM digital library, $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$.



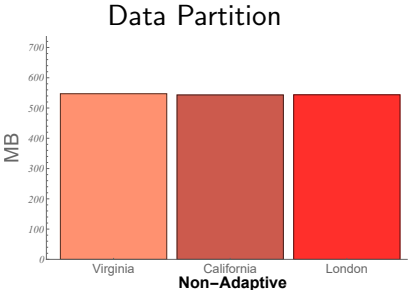
setup

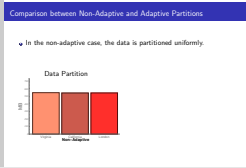


Now we can present the essence of the network in wide area processing with the following example. We preform a multiway join of three tables, using one master and three workers that transfer 1.6 GB between them. The workers share the same compute capabilities while they are spread geographically in different places and California's downlink is limited to 0.5 Gbps.

Comparison between Non-Adaptive and Adaptive Partitions

- In the non-adaptive case, the data is partitioned uniformly.

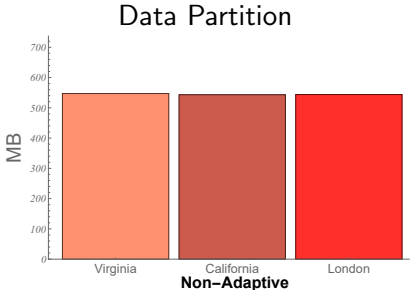


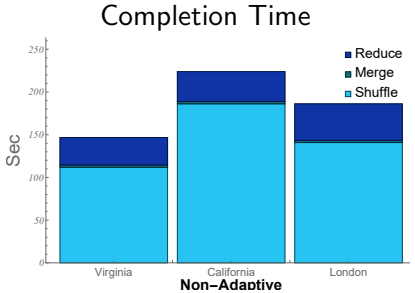


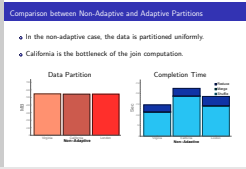
Originally Hadoop paritition the data equally around it's computers (reducers) but in this cluster we should consider the network for partitioning the data, we call this partition non-adaptive as it doesn't adapt to the network.

Comparison between Non-Adaptive and Adaptive Partitions

- In the non-adaptive case, the data is partitioned uniformly.
- California is the bottleneck of the join computation.



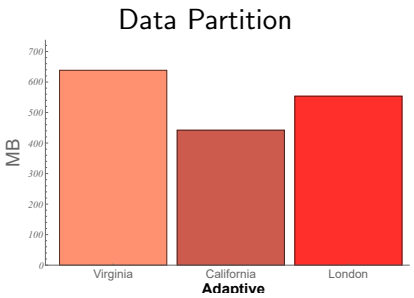


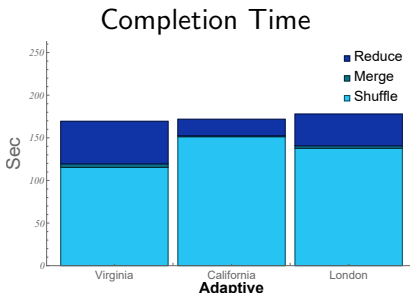


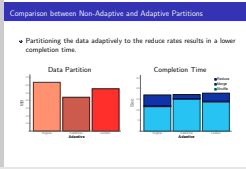
We show the average results for **ten** runs of Hadoop jobs. (Expalin the axes carefully) 1.6 Gb is distributed equally which leads to California's slow completion time, that act as a bottleneck, and a very fast Virginia's time. The main difference between the three, layes on the California's slow downlink which can clearly be seen from the dominance of the shuffle time in the completion time.
Reduce and merge functions are performed locally while the shuffle involves communication, Virginia's shuffle and completion time are 111 and 147 seconds, whereas California's shuffle and completion time are 186 and 225 seconds.

Comparison between Non-Adaptive and Adaptive Partitions

- Partitioning the data adaptively to the reduce rates results in a lower completion time.







We show the average results for **ten** runs of Hadoop jobs. (Expalin the axes carefully) 1.6 Gb is distributed equally which leads to California's slow completion time, that act as a bottleneck, and a very fast Virginia's time. The main difference between the three, layes on the California's slow downlink which can clearly be seen from the dominance of the shuffle time in the completion time.

Reduce and merge functions are performed locally while the shuffle involves communication, Virginia's shuffle and completion time are 111 and 147 seconds, whereas California's shuffle and completion time are 186 and 225 seconds.

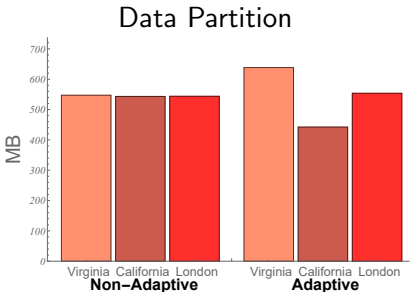
Thus, we modify the partitioning of data, more wisely, to be aware of the reducer's downlinks.

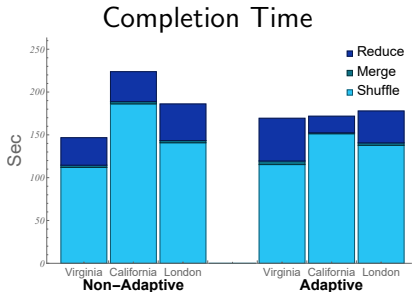
Reduce does matter- vary from 20 to 40 seconds which is close to 14% of the C.

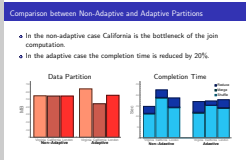
Nap might distribute the keys in a non-uniform way, which results in reducers receiving a non-equal share of the shuffled data to process, and

Comparison between Non-Adaptive and Adaptive Partitions

- In the non-adaptive case California is the bottleneck of the join computation.
- In the adaptive case the completion time is reduced by 20%.

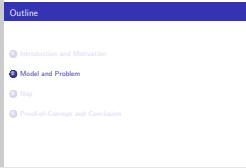






We show the average results for **ten** runs of Hadoop jobs. (Expalin the axes carefully) 1.6 Gb is distributed equally which leads to California's slow completion time, that act as a bottleneck, and a very fast Virginia's time. The main difference between the three, layes on the California's slow downlink which can clearly be seen from the dominance of the shuffle time in the completion time.
Reduce and merge functions are performed locally while the shuffle involves communication, Virginia's shuffle and completion time are 111 and 147 seconds, whereas California's shuffle and completion time are 186 and 225 seconds.
Thus, we modify the partitioning of data, more wisely, to be aware of the reducer's downlinks.
Reduce does matter- vary from 20 to 40 seconds which is close to 14% of the C.
Nap might distribute the keys in a non-uniform way, which results in reducers receiving a non-equal share of the shuffled data to process, and

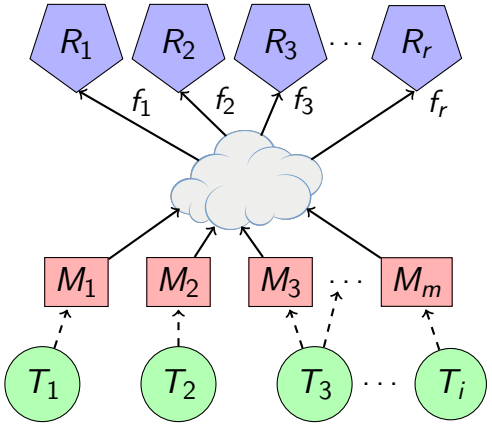
Outline



First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

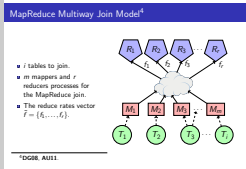
MapReduce Multiway Join Model[4]

- i tables to join.
- m mappers and r reducers processes for the MapReduce join.
- The reduce rates vector $\bar{f} = \{f_1, \ldots, f_r\}$.



[4] **DG08**, **AU11**.

[a] **DG08**. **AU11**.



The following model is for the multiway join case study and it is based on the MapReduce architecture. We assume that the reducers are the bottlenecks, and in particular their reduce rates, which can be the downlink or the processing rates. Every reducer i has a positive reduce rate, f_i, where $\bar{f}$ is sorted in decreasing order ($f_r = 1$).
Both mappers and reducers are processes and considered as workers in the model.
The dashed lines emphasize that tables are split among mappers, where each table can be located in one machine or split into many machines.
The cloud (and its links) represents the communication network, which is needed for routing the data from the mappers to the reducers.
This model is an extension to Afrati and Ullman model, when all the reducers had the same rate, $\forall i,\ f_i = 1$, and in the paper it has been presented for a more general use case.

Problem Definition

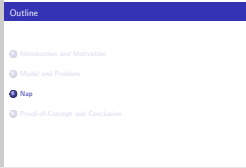
Problem

Consider a MapReduce job (multiway join), J, with r reducers, and $\bar{f}$ reduce rates vector. Our goal is to partition the data according to $\bar{f}$, and thus minimize the (job) completion time C.

Problem Definition
Problem
Consider a MapReduce job (multiway join), J, with r reducers, and $\bar{f}$ reduce rates vector. Our goal is to partition the data according to $\bar{f}$, and thus minimize the (job) completion time C.

While traditionally, the optimizations has been done for the computational power, the amount of data, the structure and load of the network have been ignored, where C is as our primary metric of interest.

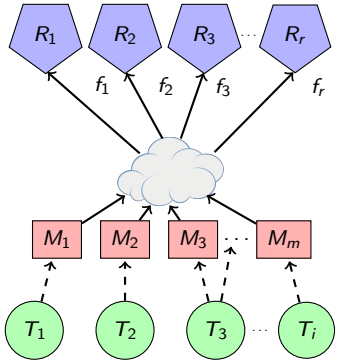
Outline



First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Network Aware and Adaptive Multiway Join

- Smartly assign virtual reducers (chunks of data) on the reducers.
- Reducer i hosts v_i virtual reducers.
- B - total communication cost.
- W - sum of the reduce rates.
- C - straggler's finish time.

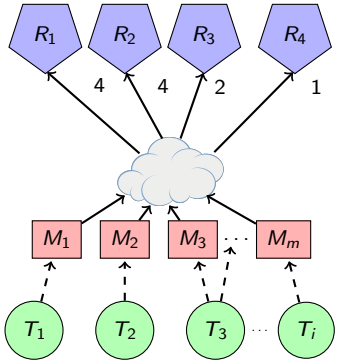


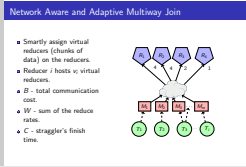
Network Aware and Adaptive Multiway Join
- Smartly assign virtual reducers (chunks of data) on the reducers.
- Reducer i hosts v_i virtual reducers.
- B - total communication cost.
- W - sum of the reduce rates.
- C - straggler's finish time.

The basic idea of Nap is simple, exploit the reduce rate of each reducer, and by that minimize the completion time.
This time is defined as the completion time of the whole process, and it is determined by the last reducer to complete the job, i.e., the straggler.
We achieve it by fooling Hadoop with the introduction of virtual reducers as our "new" workers in the MapReduce operation, where they are located inside the "physical" reduce processes.
Instead of partitioning the data uniformly between the reducers we uniformly partition it between the virtual reducers, and only decides which small chunk, amount of virtual reducers, should be placed in each reducer. Using more virtual reducers than reducers results in dividing the transferred data into smaller pieces, which is easier for tuning the partition of data and reducing the completion time.

Network Aware and Adaptive Multiway Join

- Smartly assign virtual reducers (chunks of data) on the reducers.
- Reducer i hosts v_i virtual reducers.
- B - total communication cost.
- W - sum of the reduce rates.
- C - straggler's finish time.

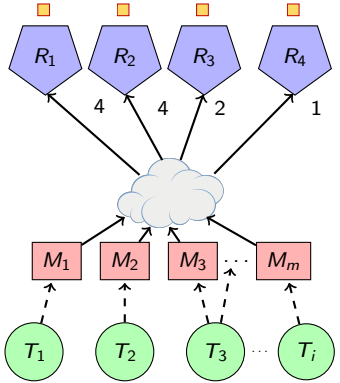


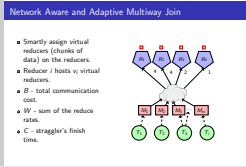


The basic idea of Nap is simple, exploit the reduce rate of each reducer, and by that minimize the completion time.
This time is defined as the completion time of the whole process, and it is determined by the last reducer to complete the job, i.e., the straggler.
We achieve it by fooling Hadoop with the introduction of virtual reducers as our "new" workers in the MapReduce operation, where they are located inside the "physical" reduce processes.
Instead of partitioning the data uniformly between the reducers we uniformly partition it between the virtual reducers, and only decides which small chunk, amount of virtual reducers, should be placed in each reducer. Using more virtual reducers than reducers results in dividing the transferred data into smaller pieces, which is easier for tuning the partition of data and reducing the completion time.

Network Aware and Adaptive Multiway Join

- Smartly assign virtual reducers (chunks of data) on the reducers.
- Reducer i hosts v_i virtual reducers.
- B - total communication cost.
- W - sum of the reduce rates.
- C - straggler's finish time.

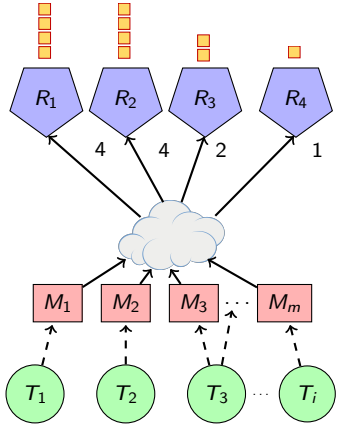


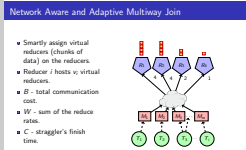


The basic idea of Nap is simple, exploit the reduce rate of each reducer, and by that minimize the completion time.
This time is defined as the completion time of the whole process, and it is determined by the last reducer to complete the job, i.e., the straggler.
We achieve it by fooling Hadoop with the introduction of virtual reducers as our "new" workers in the MapReduce operation, where they are located inside the "physical" reduce processes.
Instead of partitioning the data uniformly between the reducers we uniformly partition it between the virtual reducers, and only decides which small chunk, amount of virtual reducers, should be placed in each reducer. Using more virtual reducers than reducers results in dividing the transferred data into smaller pieces, which is easier for tuning the partition of data and reducing the completion time.

Network Aware and Adaptive Multiway Join

- Smartly assign virtual reducers (chunks of data) on the reducers.
- Reducer i hosts v_i virtual reducers.
- B - total communication cost.
- W - sum of the reduce rates.
- C - straggler's finish time.





The basic idea of Nap is simple, exploit the reduce rate of each reducer, and by that minimize the completion time.
This time is defined as the completion time of the whole process, and it is determined by the last reducer to complete the job, i.e., the straggler.
We achieve it by fooling Hadoop with the introduction of virtual reducers as our "new" workers in the MapReduce operation, where they are located inside the "physical" reduce processes.
Instead of partitioning the data uniformly between the reducers we uniformly partition it between the virtual reducers, and only decides which small chunk, amount of virtual reducers, should be placed in each reducer. Using more virtual reducers than reducers results in dividing the transferred data into smaller pieces, which is easier for tuning the partition of data and reducing the completion time.

What is the Optimal Partition?

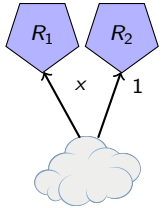
- The completion time with r reducers is lower bounded by $O(\frac{B}{W})$.
- Do we need to use all the r reducers or not?
- Finding the optimal partition of virtual reducers that minimize the completion time includes a connection to Integer Partition and Young Lattice.

What is the Optimal Partition?
- The completion time with r reducers is lower bounded by $O(\frac{B}{W})$.
- Do we need to use all the r reducers or not?
- Finding the optimal partition of virtual reducers that minimize the completion time includes a connection to Integer Partition and Young Lattice.

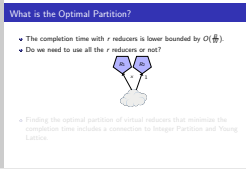
Given this model what is the best thing we should expect for? A uniform finish time and we show in the paper that when we use all the resources, all the r reducers, then the completion time is lower bounded. Sometimes we don't need to use all the reducers because they can not help with the processing, and we prove it in the paper as well. Because the multiway join method we know includes replication of the tables, thus the total communication cost is a function of the virtual reducers and using less virtual reducers will generate less replication, therefore it can be beneficial.

What is the Optimal Partition?

- The completion time with r reducers is lower bounded by $O(\frac{B}{W})$.
- Do we need to use all the r reducers or not?



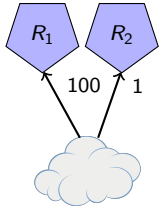
- Finding the optimal partition of virtual reducers that minimize the completion time includes a connection to Integer Partition and Young Lattice.



Given this model what is the best thing we should expect for? A uniform finish time and we show in the paper that when we use all the resources, all the r reducers, then the completion time is lower bounded. Sometimes we don't need to use all the reducers because they can not help with the processing, and we prove it in the paper as well. Because the multiway join method we know includes replication of the tables, thus the total communication cost is a function of the virtual reducers and using less virtual reducers will generate less replication, therefore it can be beneficial.

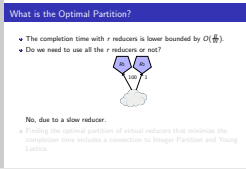
What is the Optimal Partition?

- The completion time with r reducers is lower bounded by $O(\frac{B}{W})$.
- Do we need to use all the r reducers or not?



No, due to a slow reducer.

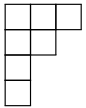
- Finding the optimal partition of virtual reducers that minimize the completion time includes a connection to Integer Partition and Young Lattice.



Given this model what is the best thing we should expect for? A uniform finish time and we show in the paper that when we use all the resources, all the r reducers, then the completion time is lower bounded. Sometimes we don't need to use all the reducers because they can not help with the processing, and we prove it in the paper as well. Because the multiway join method we know includes replication of the tables, thus the total communication cost is a function of the virtual reducers and using less virtual reducers will generate less replication, therefore it can be beneficial.

What is the Optimal Partition?

- The completion time with r reducers is lower bounded by $O(\frac{B}{W})$.
- Do we need to use all the r reducers or not?
  No, due to a slow reducer.
- Finding the optimal partition of virtual reducers that minimize the completion time includes a connection to Integer Partition and Young Lattice.

Integer Partition of $n = 7$ for $(4,2,1) \rightarrow$ 

Finding the optimal partition is not trivial but using Integer Partition and Young Lattice we can find a solution. We offer an alternative that is based on greedy searching for the optimal partition.

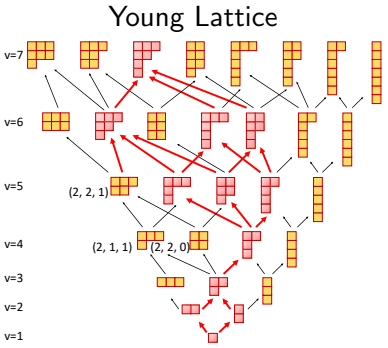
Young Lattice and Optimal Walk

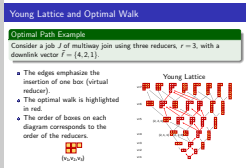
Optimal Path Example

Consider a job J of multiway join using three reducers, $r = 3$, with a downlink vector $\bar{f} = \{4, 2, 1\}$.

- The edges emphasize the insertion of one box (virtual reducer).
- The optimal walk is highlighted in red.
- The order of boxes on each diagram corresponds to the order of the reducers.

(v_1, v_2, v_3)





Each diagram (i.e., virtual reducers assignment), can be translated to a completion time, and on every level n there are all the possible partitions of integer n. So in each level, which denotes the given number of virtual reducers, we color in red the best assignment, i.e., the assignments with the minimal completion time based on the rates vector. Note that several assignments can achieve the minimum. Accordingly, edges that are directed into such optimal assignments are also colored in red.

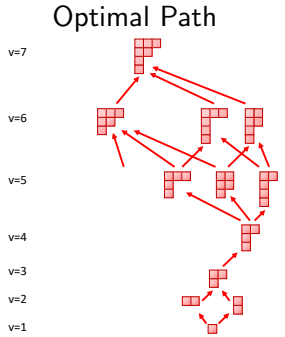
*For instance, the leftmost diagram on level four (integer partitions of $n = 4$) has two virtual reducers on R_1, one virtual reducer on R_2, and one virtual reducer on R_3, $(2,1,1)$, and to the right on this diagram there is a partition of four virtual reducers, two on R_1 and two on R_2, $(2,2,0)$.

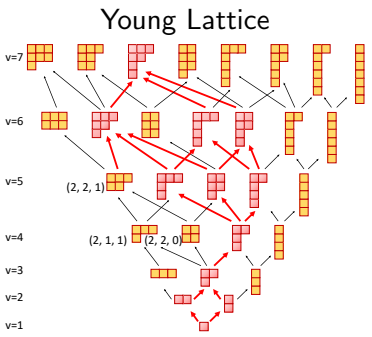
These walks are the basis for a greedy search, for an optimal assignment, when we greedily insert virtual reducer based on $\bar{f}$.

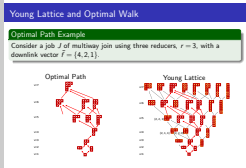
Young Lattice and Optimal Walk

Optimal Path Example

Consider a job J of multiway join using three reducers, $r = 3$, with a downlink vector $\bar{f} = \{4, 2, 1\}$.







One significant diagram from the figure is the leftmost diagram on level five, $(2,2,1)$, which doesn't have an ancestor with minimal completion time on level four. It is not a minimal partition, because for partition of integer five it has two stragglers and there other optimal partitions (on the same level) with only one straggler.

Endless Loop?

- Iteratively "walking" on the optimal path would result in optimal partition, but when should we stop?
- We can stop after $W - r$ comparisons, and eventually the running time would be $O(W \cdot \log r)$.

Endless Loop?
- Iteratively "walking" on the optimal path would result in optimal partition, but when should we stop?
- We can stop after $W - r$ comparisons, and eventually the running time would be $O(W \cdot \log r)$.

Finding the optimal partition for partition of one is simple by brute force.
We can add optimally on each step, but when should stop?

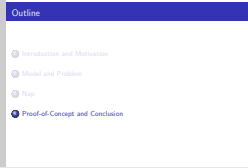
Endless Loop?

- Iteratively "walking" on the optimal path would result in optimal partition, but when should we stop?
- We can stop after $W - r$ comparisons, and eventually the running time would be $O(W \cdot \log r)$.

Endless Loop?
- Iteratively "walking" on the optimal path would result in optimal partition, but when should we stop?
- We can stop after $W - r$ comparisons, and eventually the running time would be $O(W \cdot \log r)$.

Greedily adding one virtual reducer by another until we reach W when the adding compares r options for the insertion of one virtual reducer.

Outline



First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Implementation

Problem

How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?

Implementation
Problem
How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?

The idea behind it is to decrease the completion time, straggler's finish time, by sending less data to the straggler and more data to some other reducers but how can we do it when we don't know where they are? (without updating the RM code for different scheduling of the containers in the cluster).

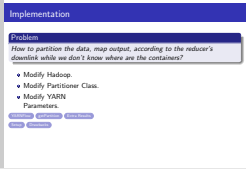
Implementation

Problem

How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?

- Modify Hadoop.
- Modify Partitioner Class.
- Modify YARN Parameters.

YARNFlow | getPartition | Extra Results | Setup | Drawbacks



- Write to HDFS the containers location in the heartbeat function.
- Override the getPartition function.
- Enables using the new Partitioner Class and uniformly distribute the mappers.

Upon Job execution, the RM decides where to allocate each container inside the workers.
This scheduling process is oblivious to the end user and for updating this process I had to understand the core of scheduling (which is not easy and not well documented in the web) and then I had to do the following three goals.
We have changed the starting point of the reduce containers and the shuffle phase to the same time of allocating mapper containers, therefore all the containers must be allocated in parallel and it enables our code (Partitioner Class) to distribute the data according to the computer's downlink.

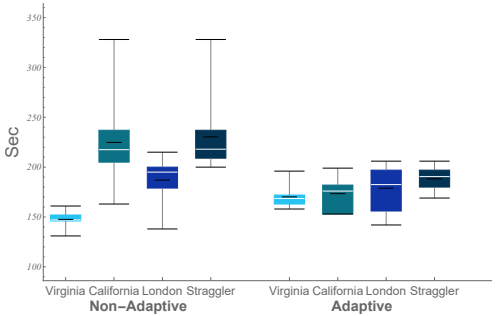
Implementation

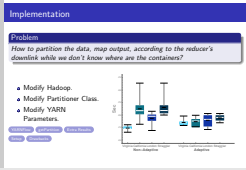
Problem

How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?

- Modify Hadoop.
- Modify Partitioner Class.
- Modify YARN Parameters.

YARNFlow | getPartition | Extra Results | Setup | Drawbacks





The boxplot displays the mean value as a black strip, the median value as a white strip, and two other strips for the boundaries of each box. In fact, the slowest job in the adaptive scenario (10, 211) is roughly as fast as the fastest non-adaptive job (1, 205).

Partit	no-adapt	adapt
Mean	236	192
Variance	1462	143
Median	224	191
Max	331	221
Min	205	174

*There is a minor difference of time between the results, due to the time from submitting the job to allocating the reduce containers. Although the reducers are allocated right after the mappers, there is no slow start for the shuffle nor allocating the reducers, the elapsed time of a straggler from all the reducers will not include a few seconds before the reducers

Implementation

Problem

How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?

For more see my Nap repository on Github, https://github.com/razo7/Nap.

Implementation
Problem
How to partition the data, map output, according to the reducer's downlink while we don't know where are the containers?
For more see my Nap repository on Github, https://github.com/razo7/Nap.

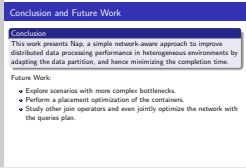
Conclusion and Future Work

Conclusion

This work presents Nap, a simple network-aware approach to improve distributed data processing performance in heterogeneous environments by adapting the data partition, and hence minimizing the completion time.

Future Work:

- Explore scenarios with more complex bottlenecks.
- Perform a placement optimization of the containers.
- Study other join operators and even jointly optimize the network with the queries plan.



This kind of work can be related mainly to Hadoop's shortcomings due to its lack of network consideration.

- Consider a different implementation of multiway join and try to optimize th shuffle phase also for Apache Spark.
- Try to find a suggestion for the number of reducers, r.
  We presented a formal performance analysis of our approach and reported on a proof-of-concept implementation.
-

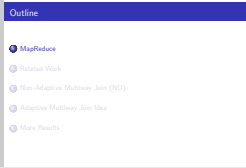
1. Consider a different implementation of multiway join and try to optimize the shuffle phase also for Apache Spark.
2. Given our work with AD^*, try to find a suggestion for the number of reducers, r.

We believe that our work opens several interesting avenues for future research in addition to the remarks I had mention before. We presented a formal performance analysis of our approach and reported on a

Thank you

Thank you

Outline



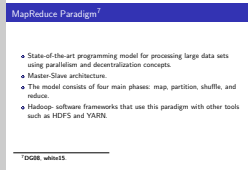
First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

MapReduce Paradigm[7]

- State-of-the-art programming model for processing large data sets using parallelism and decentralization concepts.
- Master-Slave architecture.
- The model consists of four main phases: map, partition, shuffle, and reduce.
- Hadoop- software frameworks that use this paradigm with other tools such as HDFS and YARN.

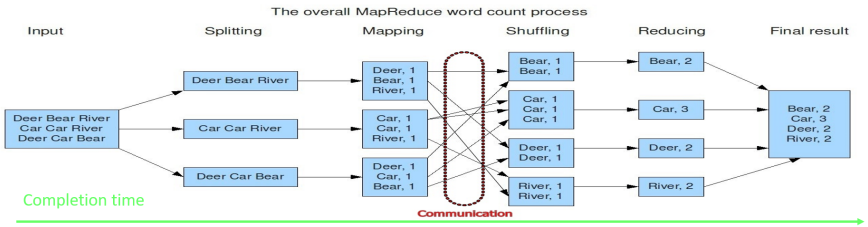
[7] **DG08**, **white15**.

[a] **DG08**. **white15**.



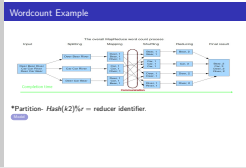
- This paradigm is highly useful and efficient tool for large-scale fault toleran data analysis with crash recovery mechanism. This model has a Master-Slav architecture with one master that manages all the slaves/workers.
- All the phases are performed locally except the shuffle, when there is a communication between mappers and reducers and this is the most consuming phase and can be a bottleneck for the whole execution.
- Apache Hadoop (batch processing with massive amounts of data) and Apache Spark (overcome latency and the inability to stream data by exploiting the memory) are software frameworks that use this paradigm.

Wordcount Example



*Partition- $Hash(k2)\%r$ = reducer identifier.

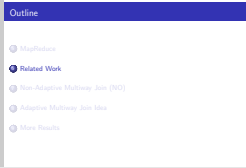
Model



A well-known example of word count in the corpus.

- A map phase- Each machine takes a split of the corpus, and for each word, key-value tuple is created (k2,v2). The key is the actual word from the corpus, and the value is one as the number of appearances for that word so far(Combiner).
- A shuffle phase- Tuples are sent from the mappers to the reducers, according to the Partitioner, which uses the tuples' key to determine the destination of each mapper's output. The shuffle is performed in a way tha does not account for the communication; thus it cannot balance the load o each reducer, which results in longer C.
- A reduce phase- The reducers collect all the tuples and perform the reduce function, which aggregates all the tuples by their key. Then the reducer saves the result locally and in the distributed file system.

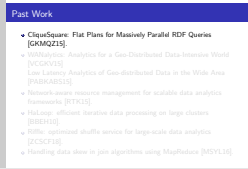
Outline



First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Past Work

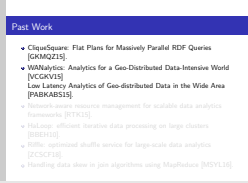
- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].



Our idea and contribution covers many topics that has been studies intensively over the years for the way of minimizing the C.
Worth mentioning is the work of CliqueSquare on flattening the operator tree to improve response time for RDF (A query language for databases) queries. When RDF tends to involve in many joins.
Resource Description Framework (RDF)- flexible data model introduced for the Semantic Web and his database can be seen as a directed labelled graph. RDF queries tend to involve more joins than a relational query computing the same result.

Past Work

- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].



The two Microsoft works (WANalytics and Low Latency Analytics of Geo-distributed Data in the Wide Area) that captures the importance of the network in Geo-distributed cluster with a placement optimization in Spark.
The model of the designers includes a proxy layer (with Apache Hive) and cache for the optimization in each DC. The analyst sends an analytical queries (SQL) to a WANalytics command layer which creates a distributed execution to the partitions (the partition includes some DCs) then the proxy layer (in each DC) manages analytics stack, cache and support data transfer optimally
Iridium is a system for low latency geo-distributed analytic and it uses a greedy heuristic for a data and task placement of queries. In addition, it is implemented on Spark framework with HDFS, for task placement they override the default Spark's scheduler and the experiment have been used 8 EC2 in different regions around the globe for geo-distribution.

Past Work

- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].

Past Work
- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].

The (Network-aware resource management for scalable data analytics frameworks) article that focus on the importance of sharing cluster resources between multiple workloads using a network-aware container placement approach.
The current framework are based solely on compute resource profiles for their work without taking information on the network topology and input data locations into account (also try to load balancing by sharing the containers with many nodes). The solution uses a weighted cost function which consider data locality, container closeness and balance over avaliable resources.

Past Work

- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].

Past Work
- CliqueSquare: Flat Plans for Massively Parallel RDF Queries [GKMQZ15].
- WANalytics: Analytics for a Geo-Distributed Data-Intensive World [VCGKV15]
  Low Latency Analytics of Geo-distributed Data in the Wide Area [PABKABS15].
- Network-aware resource management for scalable data analytics frameworks [RTK15].
- HaLoop: efficient iterative data processing on large clusters [BBEH10].
- Riffle: optimized shuffle service for large-scale data analytics [ZCSCF18].
- Handling data skew in join algorithms using MapReduce [MSYL16].

Next there are works on Hadoop's network problem more specifically the shuffle phase, with I/O overhead or data skew and even skew in join algorithms, when the data is not load balanced.
For instance, Haloop that initiates the work for optimizing current frameworks to iterative jobs, it decrease the running time and shuffled data regarding a workflow of iterative jobs. Haloop jointly reduces the waste on time (from I/O, CPU, and network bandwidth) and detects a termination condition of loops.
The article suggests Multi-dimensional range partitioning (MDRP) which combines the ideas of the last methods and decrease the skew. It samples the data (make it a workflow) and don't repartition the data cells without output (can not make a join).

More Related Work

- SmartJoin: A network-aware multiway join for MapReduce [SHCY14].
- Optimizing multiway joins in map-reduce environment [AU12].

But there are two articles that are much more closer to our work (AU and SmartJoin).
SmartJoin, also, present a network-aware multiway join algorithm for map-reduce but for a different multiway join. Their join joins two large tables using one joint attribute in a reduce side join fashion. And the network aware reference relates to a late join between many small tables (on the reduce function) using hash join between the reducers. SmartJoin dynamically redistributes tuples directly between reducers, while we optimize the way of partitioning the data in the shuffle phase (where there is a replication problem) and we join tables with two joint attributes.
Moreover, we relate to the network by the downlinks of each worker, where SmartJoin relates to the structure of the cluster, e.g., the switches' bandwidth between nodes and racks.

More Related Work

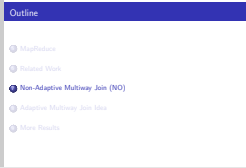
- SmartJoin: A network-aware multiway join for MapReduce [SHCY14].
- Optimizing multiway joins in map-reduce environment [AU12].

More Related Work
- SmartJoin: A network-aware multiway join for MapReduce [SHCY14].
- Optimizing multiway joins in map-reduce environment [AU12].

Afrati and Ullman's scheme, which we show next, is oblivious to the downlinks vector (i.e., assuming all downlinks are the same); thus we consider this as a non-adaptive scheme. Also, we denote the scheme by NO.

Afrati and Ullman present a model for computing multway joins in map-reduce, accounting for communication costs by changing the data partition. However, their approach is non-adaptive as it assumes that all link capacities are equal, we suggest to remove this restriction and generalize the model. Their work inspired us for the multiway join analysis and implementation as we will see on the next section.

Outline

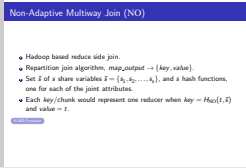


First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Non-Adaptive Multiway Join (NO)

- Hadoop based reduce side join.
- Repartition join algorithm, $map\_output \rightarrow \{key, value\}$.
- Set $\bar{s}$ of s share variables $\bar{s} = \{s_1, s_2, \ldots, s_s\}$, and s hash functions, one for each of the joint attributes.
- Each key/chunk would represent one reducer when $key = H_{\text{NO}}(t, \bar{s})$ and $value = t$.

H-NO Function



The scheme (NO) performs a cascade join in the reducers, but for making a proper join each reducer must have all the rows with the matching joint attributes. Thus, the map output pair of $\{key, value\}$, for each row in the tables, will have a key with s values, where s is the number of joint attributes in the join, and a value as the row itself. For making the key they use $H_{\text{NO}}(t, \bar{s})$ which is a set of s hash functions. There is share variables vector with one variable for each joint attribute, when each variable defines a degree of replication for the corresponding joint attribute (number of buckets that the attribute is hashed to). Therefore, the rows are duplicated according to size of share variable and it's related missing joint attribute, as we can see on the next slide.

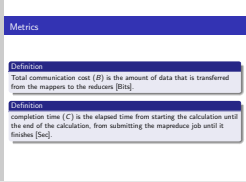
Metrics

Definition

Total communication cost (B) is the amount of data that is transferred from the mappers to the reducers [Bits].

Definition

completion time (C) is the elapsed time from starting the calculation until the end of the calculation, from submitting the mapreduce job until it finishes [Sec].



One can assume that the computation time in the computers (local computation) is negligible in comparison to the communication time (transfer of the data), due to the rapid improvement in computing capabilities. There is a trade off between these two metrics for making the multiway join:

1. Minimum B at cost of long C, perform a local cascade join, don't distribut the work and maybe some concurrency (a queue might build up when it depends only on the downlink of a single reducer).
2. Maximum B with short C, replicate the tables to all the reducers with a cos of sending vast amount of data which is unutilized. High concurrency between computers (and not only processes) but at some point it doesn't decrease the C any more and only produces a large output from each reducer .

Minimization of the Completion Time-1

Minimize B → *Minimize C*

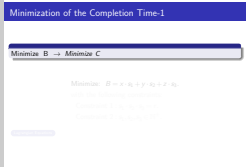
Minimize: $B = x \cdot s_1 + y \cdot s_2 + z \cdot s_3$.

with the following constraints:

Constraint 1 : $s_1 \cdot s_2 \cdot s_3 = r$,

Constraint 2 : $s_1, s_2, s_3 \in \mathbb{N}^+$.

Lagrangian Equations



This leads to Afrati and Ullman approach for minimization of completion time by first minimizing the total communication cost.

This is analysis example of the multiway join problem for three tables when each table is missing only one joint attribute.

Minimization of the Completion Time-1

Minimize B $\rightarrow$ *Minimize C*

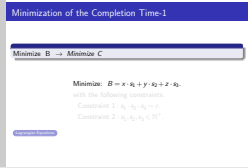
Minimize: $B = x \cdot s_1 + y \cdot s_2 + z \cdot s_3$.

with the following constraints:

Constraint 1 : $s_1 \cdot s_2 \cdot s_3 = r$.

Constraint 2 : $s_1, s_2, s_3 \in \mathbb{N}^+$.

Lagrangian Equations



This leads to Afrati and Ullman approach for minimization of completion time by first minimizing the total communication cost.

This is analysis example of the multiway join problem for three tables when each table is missing only one joint attribute.

Minimization of the Completion Time-1

Minimize B $\rightarrow$ *Minimize C*

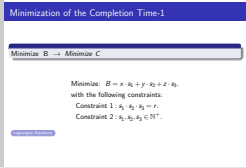
Minimize: $B = x \cdot s_1 + y \cdot s_2 + z \cdot s_3$.

with the following constraints:

Constraint 1 : $s_1 \cdot s_2 \cdot s_3 = r$.

Constraint 2 : $s_1, s_2, s_3 \in \mathbb{N}^+$.

Lagrangian Equations



Also, it uses the share variables for defining the size of replication for each table, based on the missing joint attribute.

Note that these two constraints connect the number of tables replications to the number of reducers, where using more reducers results in a larger replication of tables and larger total communication cost.

Afrati and Ullman use Lagrange Multiplier(a method in mathematical optimization), λ, for finding local minima of the function, subject to equality constraints.

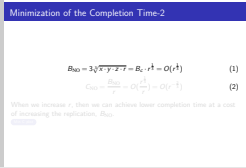
Minimization of the Completion Time-2

$$B_{\mathrm{NO}} = 3\sqrt[3]{x \cdot y \cdot z \cdot r} = B_c \cdot r^{\frac{1}{3}} = O(r^{\frac{1}{3}}) \quad (1)$$

$$C_{\mathrm{NO}} = \frac{B_{\mathrm{NO}}}{r} = O(\frac{r^{\frac{1}{3}}}{r}) = O(r^{-\frac{2}{3}}) \quad (2)$$

When we increase r, then we can achieve lower completion time at a cost of increasing the replication, B_{NO}.

Min C plot



Eventually they find that there is linear proportion between the total communication cost and the number of reducers while there is also a communication constant that captures the size of tables, $B_c = 3\sqrt[3]{xyz}$.
B_{NO} increases with the number of reducers.
I would like to mention that this nice results might be a product of rounding and approximations have to be done for credibility of constraints.

Minimization of the Completion Time-2

$$B_{\mathrm{NO}} = 3\sqrt[3]{x \cdot y \cdot z \cdot r} = B_c \cdot r^{\frac{1}{3}} = O(r^{\frac{1}{3}}) \quad (1)$$

$$C_{\mathrm{NO}} = \frac{B_{\mathrm{NO}}}{r} = O(\frac{r^{\frac{1}{3}}}{r}) = O(r^{-\frac{2}{3}}) \quad (2)$$

When we increase r, then we can achieve lower completion time at a cost of increasing the replication, B_{NO}.

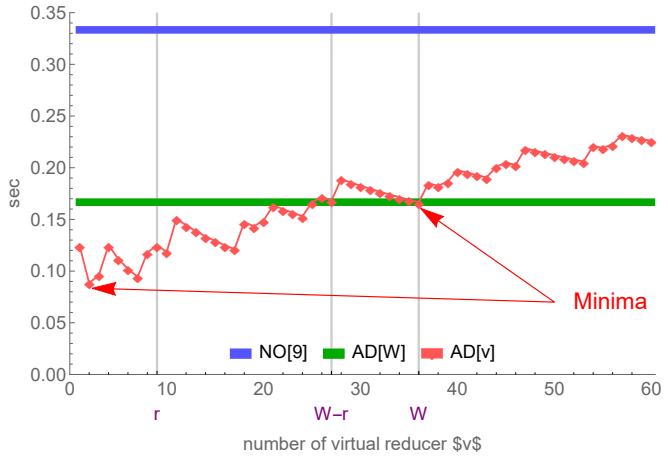
Min C plot

Minimization of the Completion Time-2
$B_{NO} = 3\sqrt[3]{x \cdot y \cdot z \cdot r} = B_c \cdot r^{\frac{1}{3}} = O(r^{\frac{1}{3}})$ (1)
$C_{NO} = \frac{B_{NO}}{r} = O(\frac{r^{\frac{1}{3}}}{r}) = O(r^{-\frac{2}{3}})$ (2)
When we increase r, then we can achieve lower completion time at a cost of increasing the replication, B_{NO}.
Min C plot

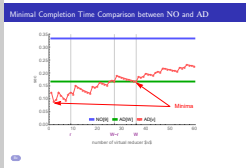
Then, they assume having a hash functions, that partition the records uniformly, when using the default, uniform, Partitioner, then the minimal C by NO is ...

Each reducer will receive the same amount of data, an equal sharing, and since they are all alike, they will finish together at the same time. There is a trade off between the two metrics.

Minimal Completion Time Comparison between NO and AD



Bc



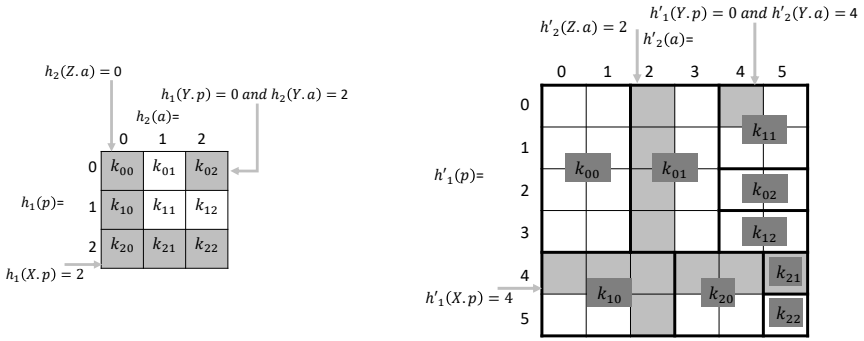
For further discussion and highlighting the theoretical results of the next sections, we consider again the ACM multiway join example in the Introduction with $\bar{f}$ vector as in the last Figure and for the sake of simplicity $B_c = 1$. The sum of downlinks, number of reducers, and their difference is highlighted with purple on the X axis and three vertical lines. There are two arrows towards the local and global minima of AD[v]'s C, at $v = W = 36$, and $r = 2$. Each point in the figure shows the minimal completion time out of all the λ partitions of v.

Blue line - when we partition equally the data between nine reducers, whereas the **green** line is the completion time for AD[W], which utilizes all the downlinks of the nine reducers. AD[v] outperforms NO[9], because it can partition the data and utilizes the network better. $v = 26$.AD* is achieved by selecting only two virtual reducers on R_1 and R_2, which leads to the lowest C, and **identical finish time** between them. **Overall, $C_{\mathbf{AD[2]}}$ outperforms $C_{\mathbf{AD[}W\mathbf{]}}$ by 50% and $C_{\mathbf{NO[9]}}$ by 75%.**

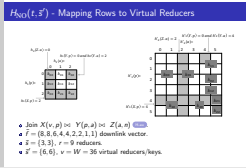
Outline

First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

$H_{NO}(t,\bar{s}')$ - Mapping Rows to Virtual Reducers



- Join $X(v,p) \bowtie Y(p,a) \bowtie Z(a,n)$ H-au.
- $\bar{f} = (8,8,6,4,4,2,2,1,1)$ downlink vector.
- $\bar{s} = \{3,3\}$, $r = 9$ reducers.
- $\bar{s}' = \{6,6\}$, $v = W = 36$ virtual reducers/keys.



Now we return to the example we had for NO scheme, when we implicitly assumed that the downlink rates are all equal and one, and when each reducer is identified with a single key.
On the right is AD scheme, that assumes that the downlinks are known to be $\bar{f} = (8,8,6,4,4,2,2,1)$ for $r = 9$ reducers, thus it uses $v = W = 36$ virtual reducers.
Now, each cell on the matrix represents one virtual reducer and there are two different hash functions $h'_1(p), h'_2(a)$, and two different share variables $s'_1 = 6, s'_2 = 6$. The map output keys would represent the virtual reducers and afterwards the partitioner would use new function for partitioning the keys/virtual reducers according to the reducers' downlinks. This way R_1, which has a key (0,0) as before, now has 8 virtual reducers, while R_9, which has key (2,2) as before, will receive only 1 virtual reducer since its downlink is much slower.
Afterwards the basic join method in reducers stays the same as in Afrati and Ullman and every two rows that need to join will end at a unique

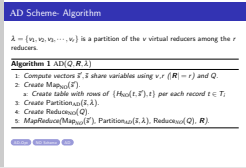
AD Scheme- Algorithm

$\lambda = \{v_1, v_2, v_3, \cdots, v_r\}$ is a partition of the v virtual reducers among the r reducers.

Algorithm 1 $\mathrm{AD}(Q, \boldsymbol{R}, \lambda)$

1: *Compute vectors* $\bar{s}', \bar{s}$ *share variables using* v,r ($|\boldsymbol{R}| = r$) *and* Q.
2: *Create* $\mathrm{Map}_{\mathrm{NO}}(\bar{s}')$.
   a: *Create table with rows of* $\{H_{\mathrm{NO}}(t, \bar{s}'), t\}$ *per each record* $t \in T_i$
3: *Create* $\mathrm{Partition}_{\mathrm{AD}}(\bar{s}, \lambda)$.
4: *Create* $\mathrm{Reduce}_{\mathrm{NO}}(Q)$.
5: *MapReduce(*$\mathrm{Map}_{NO}(\bar{s}')$, $\mathrm{Partition}_{AD}(\bar{s}, \lambda)$, $\mathrm{Reduce}_{NO}(Q)$, $\boldsymbol{R}$*)*.

AD-Opt | NO Scheme | AD



The $\mathrm{AD}(Q, \boldsymbol{R}, \lambda)$ scheme extends NO scheme with a change to the partition function, $\mathrm{Partition}_{\mathrm{AD}}(\bar{s}, \lambda)$, and the addition of virtual reducers' input partition.

The map function creates a $\{key, value\}$ pair with the same value as we have seen before, but now with a different key.

The new key of the table row t is mapped to new keys, and it is replicated to v workers using a $\bar{s}'$ vector.

Next, the new partition function, $\mathrm{Partition}_{\mathrm{AD}}(\bar{s}, \lambda)$, maps from new keys to NO keys and then to the reducers according to λ, the partition of the virtual reducers, and $\bar{s}$ vector.

This results in partitioning of the new keys, according to the network, followed from λ, but $\mathrm{Partition}_{\mathrm{AD}}(\bar{s}, \lambda)$ insures that we distribute the values (rows) to r reducers instead of v virtual reducers.

The scheme ends with running a MapReduce job...

Outline

First, I introduce the motivation in general, then with a join example, and I will give some empirical motivation.
Next, I cover the model for the problem and the problem itself.
Then, I go over what is Nap scheme with it's relation to Young Lattice.
In the end I go over the implementation, it's difficulties and introduce some points for future work.

Setup

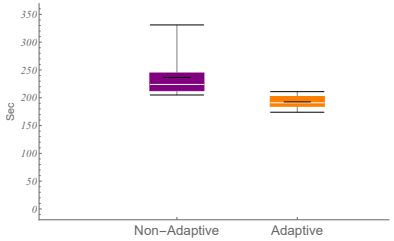
- Modified version of Hadoop on AWS multi region cluster.
- EC2 t2.xlarge and M4.xlarge instances.
- Wonder Shaper for fixing a downlink rate.
- HDFS and YARN daemons.

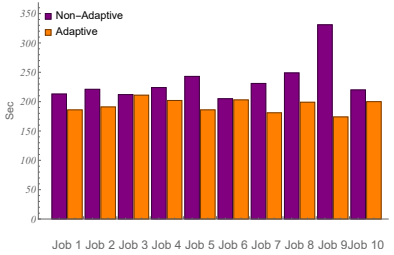
EmpSetup Implementation

We implemented a prototype of Nap and conducted some basic experiments on EC2 that serve us as a proof-of-concept.
Wonder Shaper is a command-line utility that limits the adapter's bandwidth.
The master instance is in charge of the whole computation by running the NameNode (NN), and the Resource Manager (RM) daemons, and the workers are responsible for storing the data and running the workload within containers.
HDFS has a a replication factor of three; thus, the input resides in every worker and we even managed to split the input (almost) evenly around the 25 mappers (which was not straightforward).
Why the master is only "mastering"? $\rightarrow$ The NN and RM daemons have been selected to run on a different machine, the master machine, because we have wanted to separate the monitoring work from the workers by keeping them less busy and don't prioritize any locality communication on one of the regions (e.g., AM with RM or DN with NN).
Default block size (b) =128 MB

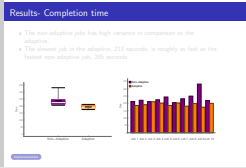
Results- Completion time

- The non-adaptive jobs has high variance in comparison to the adaptive.
- The slowest job in the adaptive, 211 seconds, is roughly as fast as the fastest non-adaptive job, 205 seconds.





Implementation



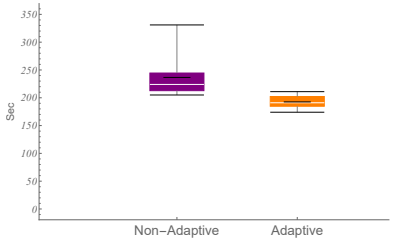
We have used Job History server REST-API for gathering all the statistics and Wolfram Mathematica for the plots.
Non-Adaptive (uniformly), purple, and Adaptive (nonuniformly), orange, $\lambda = (7,\ 6,\ 6))$, with 1.6 GB shuffled data.
The boxplot displays the mean value as a black strip, the median value as a white strip, and two other strips for the boundaries of each box. In fact, the slowest job in the adaptive scenario (10, 211) is roughly as fast as the fastest non-adaptive job (1, 205).

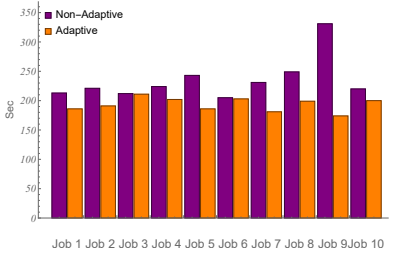
Partit	no-adapt	adapt
Mean	236	192
Variance	1462	143
Median	224	191
Max	331	221
Min	205	174

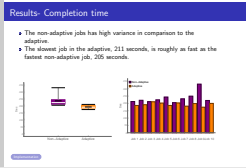
*There is a minor difference of time between the results, due to the time from submitting the job to allocating the reduce containers. Although

Results- Completion time

- The non-adaptive jobs has high variance in comparison to the adaptive.
- The slowest job in the adaptive, 211 seconds, is roughly as fast as the fastest non-adaptive job, 205 seconds.







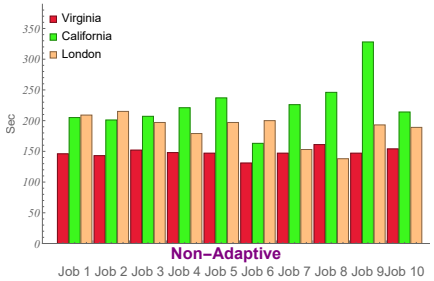
We have used Job History server REST-API for gathering all the statistics and Wolfram Mathematica for the plots.
Non-Adaptive (uniformly), purple, and Adaptive (nonuniformly), orange, $\lambda = (7,\ 6,\ 6))$, with 1.6 GB shuffled data.
The boxplot displays the mean value as a black strip, the median value as a white strip, and two other strips for the boundaries of each box. In fact, the slowest job in the adaptive scenario (10, 211) is roughly as fast as the fastest non-adaptive job (1, 205).

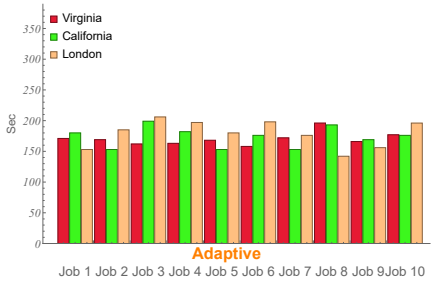
Partit	no-adapt	adapt
Mean	236	192
Variance	1462	143
Median	224	191
Max	331	221
Min	205	174

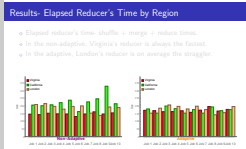
*There is a minor difference of time between the results, due to the time from submitting the job to allocating the reduce containers. Although

Results- Elapsed Reducer's Time by Region

- Elapsed reducer's time- shuffle + merge + reduce times.
- In the non-adaptive, Virginia's reducer is always the fastest.
- In the adaptive, London's reducer is on average the straggler.





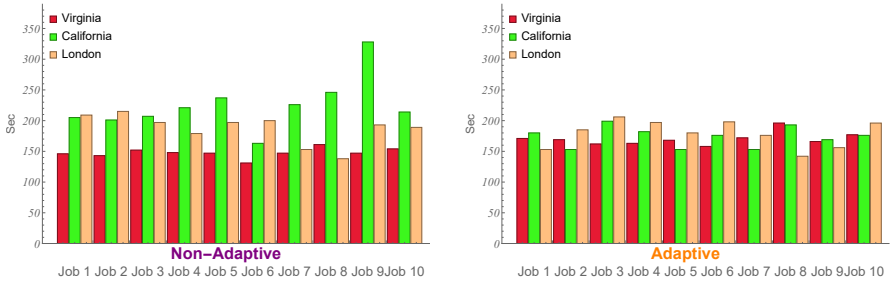


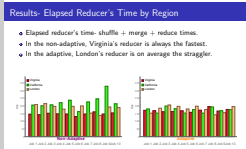
We have seen an average of this results in the empirical motivation but now we can see that for the non-adaptive California's average finish time is more than 30% higher compared to Virginia's, and more than 15% compared to London (147, 225, and 187 seconds, respectively).
Virginia is constantly the fastest and there are many time fluctuations in jobs 5, 8, and 9 for California's reducer (elapsed time is 237, 246, and 328 seconds respectively).
The adaptive partition reduce these large fluctuations when sending less 100 MB towards California that act as a bottleneck in the non-adaptive partition.

The reducers are finishing in almost identical finish time when London's reducer is on average the straggler.

Results- Elapsed Reducer's Time by Region

- Elapsed reducer's time- shuffle + merge + reduce times.
- In the non-adaptive, Virginia's reducer is always the fastest.
- In the adaptive, London's reducer is on average the straggler.





We have seen an average of this results in the empirical motivation but now we can see that for the non-adaptive California's average finish time is more than 30% higher compared to Virginia's, and more than 15% compared to London (147, 225, and 187 seconds, respectively).
Virginia is constantly the fastest and there are many time fluctuations in jobs 5, 8, and 9 for California's reducer (elapsed time is 237, 246, and 328 seconds respectively).
The adaptive partition reduce these large fluctuations when sending less 100 MB towards California that act as a bottleneck in the non-adaptive partition.

The reducers are finishing in almost identical finish time when London's reducer is on average the straggler.

Hadoop Versions

- Hadoop version 1 shortcomings - scalability, Cluster utilization, Locality awareness, and input diversity.
- In version two, there are Application Master (AM) and Resource Manager (RM) daemons which "break" the old JobTracker to two components. The scheduling process has been changed also when it begins with AM container (JVM), one per job, who runs on one of the workers and communicates with RM for allocating the next containers (map and reduce containers).

Implementation

Hadoop Versions

- Hadoop version 1 shortcomings - scalability, Cluster utilization, Locality awareness, and input diversity.
- In version two, there are Application Master (AM) and Resource Manager (RM) daemons which "break" the old JobTracker to two components. The scheduling process has been changed also when it begins with AM container (JVM), one per job, who runs on one of the workers and communicates with RM for allocating the next containers (map and reduce containers).

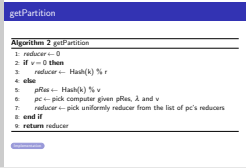
Implementation

getPartition

Algorithm 2 getPartition

```
1: reducer ← 0
2: if v = 0 then
3:     reducer ← Hash(k) % r
4: else
5:     pRes ← Hash(k) % v
6:     pc ← pick computer given pRes, λ and v
7:     reducer ← pick uniformly reducer from the list of pc's reducers
8: end if
9: return reducer
```

Implementation



The modification of Hadoop and our partition class in Section ?? can be used to modify the map output keys for any Hadoop job, not necessarily multiway join. This can be used as a standalone system for adaptive and network aware Hadoop jobs where the programmer only needs to select, beforehand, the λ of the keys on the reducers.

Our current implementation requires clusters with enough RAM for allocating all the containers in parallel. Because setConf is waiting for reading the HDFS file, which is written once, right after all the containers are running. Thus, if setConf does not find this HDFS file, it will not continue to the rest of the code, and the whole job is stuck. Therefore, it can make sense to study more memory-efficient solutions. Running Hadoop with our multiway join, *Nap*, begins with splitting each line of input (from each table) by the split input and record reader. For example, the map function of Table X computes a hash value for the joint attribute, p, by using hashCode function (from class String) and modules it with s. Afterward, the result

YARN -Workflow of MapReduce Job

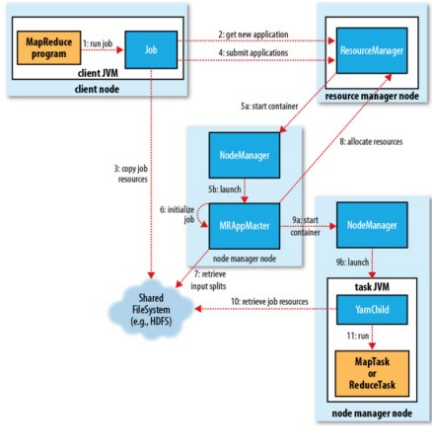
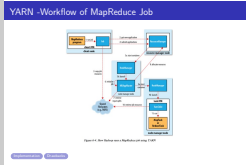


Figure 6-4. How Hadoop runs a MapReduce job using YARN

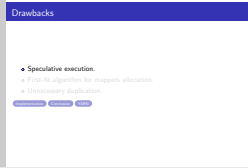
Implementation Drawbacks



Drawbacks

- Speculative execution.
- First-fit algorithm for mappers allocation.
- Unnecessary duplication.

Implementation Conclusion YARN



We emphasize that our prototype implementation and experimental results should be understood as proofs-of-concept. Our main contribution lies in the conceptual and theoretical side. In particular, the prototype still has many limitations, and our experimental results are not representative.
Modifying Hadoop:
RM is monitoring the performance and progress of the containers by the heartbeat messages between RM and NM of each computer in the cluster. These messages are sent every second (by default), and if one of the containers is not responding with a heartbeat for a threshold amount of time or its progress percent is below some threshold, then RM allocates a new container in a different computer, a speculative container. Those containers would race with the original containers, and when one of them is finished the second one is killed. Consequently, a cluster with slow containers would have many speculative containers that could help reduce the straggler's time at the expense of overloading the server.

Drawbacks

- Speculative execution.
- First-fit algorithm for mappers allocation.
- Unnecessary duplication.

Implementation Conclusion YARN

Every Hadoop job begins with RM waiting for a heartbeat message from each NM, then RM allocates all the requested containers in each computer if it is capable, in the order of receiving heartbeat messages, and it can also be seen as a First-fit algorithm from bin packing . Because by default, Hadoop version two uses Capacity scheduler with a yarn.scheduler.capacity.per-node-heartbeat.maximum-container-assignments filed that is set to infinity. When AM is the first container to allocate, the mappers are next, and the reducers are initialized after some slow start time (a percent of the whole map progress, mapreduce.job.reduce.slowstart.completedmaps). Thus, setting the maximum amount of containers per heartbeat to $\frac{m}{\#nodes}$ will result in equal sharing of the mappers in the cluster, a better distribution of mappers in comparison to the default one. But it is possible, when each computer can allocate $\frac{m}{\#nodes}$ containers.

Drawbacks

- Speculative execution.
- First-fit algorithm for mappers allocation.
- Unnecessary duplication.

Implementation Conclusion YARN

Drawbacks
- Speculative execution.
- First-fit algorithm for mappers allocation.
- Unnecessary duplication.
Implementation Conclusion YARN

When using more keys than the number of reducers, there is a possibility of sending data to different virtual reducers that reside on the same computer;
Thus, adding more logic to the mappers for checking this kind of problem before writing the output to HDFS, in the map function, could reduce this overhead.
Furthermore, it introduces a trade-off between the cost of increasing the map function time and the benefit of shuffling fewer data, and reducing the completion time.

Improvement

The idea of MapReduce that some of the keys must go to the same reducer for efficiency and correctness of the job, limits the programmer options for partitioning and controlling the size of tuples under each key, in the reduce function. Our adaptive way for spreading the data replicates the input to more buckets (keys) than in the non-adaptive, when we use $v > r$ buckets. However, this can be improved when finding