Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Optimization of execution plans in the FlumeJava model

Master's thesis

2016 Bc. Juraj Macháč

Comenius University, Bratislava
Faculty of Mathematics, Physics and Informatics

Optimization of execution plans in the FlumeJava model

Master's thesis

Study progamme:	Informatics
Study field:	2508 Informatics
Department:	Department of Computer Science
Supervisor:	prof. RNDr. Rastislav Královič, PhD.

Bratislava, 2016

Bc. Juraj Macháč

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Juraj Macháč
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Optimization of execution plans in the FlumeJava model

Aim: FlumeJava (resp. Crunch within the Apache project) is a Java library built over the MapReduce infrastructure (Hadoop in the Apache project). It provides a "parallel collection" class representing a result of a Map-Reduce cycle execution, and supports creation of, and operations on such collections with lazy evaluation: the objects are materialized on demand, and at the time of materialization, an execution consisting of several Map-Reduce sweeps is performed. The goal of the thesis is to study the optimization of the creation of this execution plans. On one hand, different optimizers can be compared, and possible ways of improvement can be identified. On the other hand, the optimization process can be formulated as an abstract optimization problem; it can be evaluated experimentally to what extent this problem corresponds to reality, and the complexity aspects may be studied.

Supervisor:	prof. RNDr. Rastislav Kráľovič, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	doc. RNDr. Daniel Olejár, PhD.
Assigned:	04.12.2014

Approved: 16.12.2014 prof. RNDr. Branislav Rovan, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

93804668

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Juraj Macháč
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Optimization of execution plans in the FlumeJava model
Optimalizácia paralelného spracovania dát v modeli FlumeJava

Cieľ: FlumeJava (resp. Crunch v projekte Apache) je javovská nadstavba nad infraštruktúrou MapReduce (resp. Hadoop), ktorá pracuje s triedou "parallel collection". Umožňuje vytvárať a kombinovať objekty tejto triedy a v istom momente ich materializovať volaním príslušných MapReduce operácií. Cieľom práce je preskúmať optimalizáciu rozvrhovania týchto operácií: jednak porovnať rôzne optimalizátory a prípadne navrhnúť ich vylepšenia a jednak zadefinavať optimalizáciu ako abstraktný problém, porovnať, nakoľko zodpovedá realite a prípadne sa venovať jeho algoritmickej zložitosti.

Vedúci: prof. RNDr. Rastislav Kráľovič, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 04.12.2014

Dátum schválenia: 16.12.2014 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Acknowledgement

Most of all, I would like to thank my supervisor, prof. RNDr. Rastislav Královič, for professional guidance and for keeping me on the right track while writing this thesis. The door of prof. Královič was always open whenever I hit a trouble spot, and I was always provided with very valuable advices and feedback.

I would also like to thank my friends and family for their constant support while I was writing this thesis.

I hereby declare that I have written this thesis independently with use of resources listed in the bibliography.

Bc. Juraj Macháč

Abstract

MapReduce is a paradigm used for processing large sets of data in parallel. FlumeJava, developed by Google, is a library providing an abstraction of this MapReduce in form of Collections and operations on these Collections. The operations defined by programmer form an execution plan, and FlumeJava has to transform this plan into pipelines of MapReduces.

There are many options on how to create these pipelines from given execution plan, and some are better in terms of network usage. This thesis studies the creation of MapReduce pipelines with optimal network usage when given an execution plan.

Abstrakt

MapReduce je paradigma využívaná na spracovávanie veľkého množstva dát paralelne. FlumeJava, vyvinutá v Google, je knižnica poskytujúca abstrakciu MapReduce-u vo forme množín a operácií na týchto množinách. Operácie na týchto objektoch, ktoré sú definované programátorom, tvoria plán výpočtu a FlumeJava má za úlohu tento plán pretransformovať do jednotlivých MapReduce kôl.

Je mnoho možností, ako vytvoriť MapReduce kolá z daného plánu výpočtu, a niektoré sú lepšie v zmysle zaťaženia siete. Táto práca skúma spôsoby vytvárania MapReduce kôl s optimálnym zatažením siete z daného plánu výpočtu.

Contents

Introduction

In this thesis, we study MapReduce algorithms and libraries using these algorithms as a primitive, and aim to optimize the execution plans of these libraries.

In general, MapReduce algorithm is a distributed algorithm used for problems which are very well parallelizable. When defining a parallel program distributed among thousands of machines, the programmer has to deal with problems such as fault tolerance, data synchronization, load balancing, etc. MapReduce lifts the programmer from the necessity of dealing with these problems for exchange of following the MapReduce paradigm. However, using solely the MapReduce may be inconvenient, because real-world computations often require a pipeline of different MapReduce jobs. Therefore, some libraries using this MapReduce algorithm as a primitive were developed (*FlumeJava*, *Apache Crunch*, *Cascading*, *Plume*, *Pig*). Especially, libraries using the *FlumeJava* design (*Apache Crunch*, *Plume*) are meant to deal with data collections and define convenient operations on these collections. Because the collections are often very large (TB, PB) , most operations are parallel - in other words, executed using the MapReduce primitive. The programmer specifies the input sources, operations on these sources, and the output stream. However, the MapReduce requires very specific functions (Map, Reduce), hence some transformation of these operations is required. For this reason, *FlumeJava* uses deferred evaluation. This means that the functions called on some object are not executed directly, but rather registered, and thus form an acyclic directed graph - execution plan. When the program is executed, the library forms this execution plan, transforms groups of operations into MapReduce rounds, and finally runs the pipeline.

Often, there are many solutions for these transformations. The most important aspects of created pipelines are the number of MapReduce rounds, and the amount of data transferred through common storage - network usage. Lower bound for the number of MapReduce rounds can be reached easily, hence this thesis focuses on optimization of the network usage.

We will show that if the network usage during the transformation is not given much attention, the resulting network usage can be higher by a large factor when compared to the optimal.

Chapter 1

Basic knowledge

In the following chapter we will introduce some terms and models we will use throughout the thesis. We will also describe functionality of FlumeJava in section 1.2 and introduce the execution plans we would like to improve.

1.1 MapReduce

In general, MapReduce is a programming model with an associated implementation for processing large data sets in parallel. Initially, the MapReduce framework was developed by Google, but it has recently been under wide adoption and has become a standard for large data analysis. The Google MapReduce itself is a closed source project, but there are also open source implementations with similar behaviour, such as Apache Hadoop. Firstly, we will describe how the MapReduce works.

We shall now explain the basic phases of MapReduce. The basic element in this model is a key-value pair - input and output of any MapReduce algorithm is a set of key-value pairs. Operations are executed in the following three stages: *map*, *shuffle* and *reduce*. Programmers specify functions *map* and *reduce*. The shuffle stage is hidden for programmer, and is handled by the system. The basic work-flow should look like this:

Init - the system divides the input into many parts, determines the number of processors (mappers & reducers) it will use, and distributes a key-value pair based on the input to each mapper (possibly many times)

Map - every mapper takes a single key-value pair as an input and produces an output consisting of a set of any key-value pairs. It is required that the map function is stateless and pure - not depending on any other key-value pairs, and not causing any side effects (e.g. changing a global variable).

Shuffle - the system is responsible for this phase. Output from mappers is grouped by key. Every reducer is supplied with a unique key and all values associated with that key. Note that the shuffle phase can not start before all mappers are done - the system is responsible for this synchronization

Reduce - every reducer takes a key and all values associated with it. As the reducer has access to all values associated with the given key, it can perform sequential computations on these values. Note that parallelism is exploited by the fact that reducers with different keys are independent and can be executed simultaneously. The reducer should eventually output a set of key-value pairs (mostly one).

After all these stages are finished, a possibly smaller set of key-value pairs is ready for either another round or final output. A program in the MapReduce paradigm may consist of many rounds of different MapReduces scheduled one after another.

Additionally, a MapReduce can be generalized by allowing more types of mappers with their associated inputs. This allows for great optimization, which we will introduce in section 1.2.2.

A few simple examples of how a MapReduce program could be used:

- **Frequency statistics:** Every mapper gets some key and one line of document as a value, and for every *word* in a line emits a pair <*word*, *n*>, where *n* denotes number of occurrences of *word*. Following the workflow, every reducer gets a word and number of occurrences in each of the lines, so it only sums them up and outputs the result.
- **Reverse Web-Link Graph:** The mappers emit <*target, source*> pairs for each link to *target* found in page *source*. Reducers concatenate these sources into a list and emit a pair: <*target, list(sources)*>.
- **Distributed Grep:** All the work is done in the mappers - the map function only emits the full line if it matches the given pattern. The reducers are just identity functions, and copy the data to the output.

As we now have a slight image of what the MapReduce algorithm is, we shall introduce a formal definition of this algorithm.

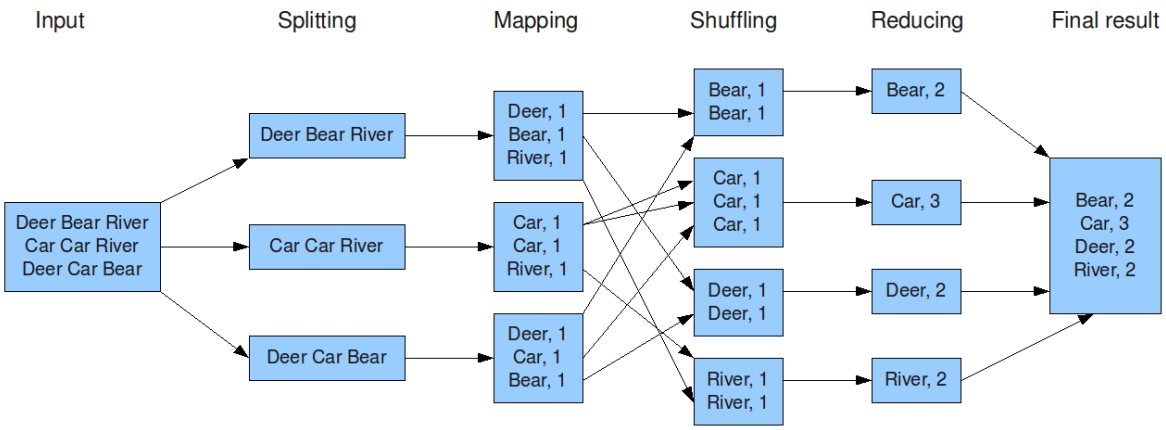


Figure 1.1: Frequency statistics example of MapReduce

1.1.1 MapReduce Class

If we want to formally define a MapReduce Class, we need to properly define mappers and reducers. We will use the definitions stated in [6].

Definition 1.1.1. *A mapper is a function which takes as input* ***one*** *ordered <key,value> pair of binary strings. As output the mapper produces a finite multiset of new <key,value> pairs.*

Definition 1.1.2. *A reducer is a function that takes as an input a binary string* ***k*** *as a key, and a sequence of values* $v_1, v_2, \ldots,$ *which are also binary strings. As output, the reducer produces a multiset of pairs of binary strings* $\{< k, v_{k,1} >, < k, v_{k,2} >, \ldots \}$.

A simple corollary form definitions stated above is, the mappers may change the keys anyhow, whereas the reducers can not. Following these definitions, we shall now formally define an algorithm in *MRC* class.

Definition 1.1.3. *Let input be a finite sequence of pairs* (k_j, v_j)*, then* $n = \sum_j(|k_j| + |v_j|)$ *is size of the input. An algorithm in* MRC^i *consists of sequence* $<\mu_1, \rho_1, \mu_2, \rho_2, \ldots, \mu_r, \rho_r>$ *of operations where exists* $\varepsilon > 0$ *such that:*

- *Each* μ_r *(resp.* ρ_r*) is a randomized mapper (resp. reducer) implemented by a RAM (Random Access Machine) with* $O(\log n)$ *length words, that uses* $O(n^{1-\varepsilon})$ *and time polynomial in n.*
- *Let* U_r *be the multiset of output by* ρ_r*, then the total space* $\sum_{(k,v)\in U_r}(|k| + |v|)$ *used by (key, value) pairs output by* μ_r *is* $O(n^{2-2\varepsilon})$
- *The number of rounds is* $O(\log^i n)$

This definition gives restrictions to memory, it requires the memory of mappers and reducers to be substantially sublinear to the size of input. However, we should also restrict the number of machines - as an algorithm requiring n^2 machines, where n is the size of web, is not very practical. Therefore, we require the number of required machines to be also substantially sublinear in the data size. Note that more than one instance of a reducer may run on the same machine, as the total number of keys may be as large as $O(n^{2-2\varepsilon})$.

1.1.2 Apache Hadoop

Google has recently (August, 2014) been granted a patent for MapReduce algorithm, and their implementation of MapReduce is a closed source. For these reasons, we will not work with the Google MapReduce itself in this thesis, but rather with an open source implementation from Apache - Hadoop.

Hadoop, like many of these open source implementations of MapReduce, is based on the specification of the Google MapReduce model. That said, we will not lose too much by examining this implementation instead of the one from Google. Additionally, Hadoop is currently used by many large companies (Facebook, Twitter, LinkedIn, Yahoo, ... [2]) and is periodically updated, so it is not just some forgotten open-source software. Now we will shortly introduce, what are the components of Hadoop and how it works.

The Apache Hadoop MapReduce is a software framework meant for writing applications processing vast amounts of data in parallel (following the MapReduce paradigm described above) conveniently, and in a fault tolerant manner. Although Hadoop framework is implemented in Java, the map / reduce tasks are not required to be written in Java. They just have to be executables with proper input & output following the specification of MapReduce. As this framework can operate on large data, stored across many clusters, Apache has developed technologies to assure easy synchronization, handle hardware failures, and assure efficiency.

HDFS (Hadoop Distributed FileSystem) is a distributed filesystem designed to run on a commodity hardware. There are many similarities with existing distributed filesystems, but there are also some differences. As the HDFS is usually running on hundreds of thousands of server machines, hardware failures are very common and the HDFS needs to be fault-tolerant - quick and automatic recovery is a core architectural goal of HDFS. Besides that, HDFS is designed to support large files stored across several nodes, and to provide interface for applications to move closer to where their data for computation is stored in order to improve efficiency.

YARN is a structural architecture for Hadoop MapReduce. The idea is to have one global ResourceManager and ApplicationMaster (for job scheduling). The ResourceManager and per node slave NodeManager form the data-computation framework. ResourceManager is responsible for arbitrating resources among all applications, and the ApplicationMaster is library tasked with negotiating the resources from ResourceManager and working with NodeManager-s to execute the tasks.

1.2 FlumeJava

We have introduced MapReduce paradigm, a framework using it, but in real life, using solely the MapReduce for computation is not very easy nor convenient. Most of the real computations need a pipeline of MapReduces. Therefore, additional coordination code to chain together the separate MapReduce stages require additional work to manage the creation and deletion of the intermediate results between the pipeline stages. With all these low level details, the logical computation is obscured, and the computation itself is less transparent and less understandable. Additionally, division into pipelines gets hardcoded and difficult or inconvenient to change later, when the computation logically changes. For these reasons, some libraries helping to resolve these issues have been developed. One of these libraries is FlumeJava. In this chapter, we will present the functionality and the important parts of specification of FlumeJava.

FlumeJava is a model with associated implementation developed at Google. It aims to support the development of data-parallel pipelines. It is a Java library with a few major classes which represent *parallel collections*. These parallel collections support many *parallel operations*, which implement data-parallel computations. These parallel collections abstract programmer from the details of how the data is represented - it does not matter if it is a single or more files, or some database. Similarly, the implementation strategy of the parallel operations is abstracted. These abstractions allow for simple debugging on a smaller set of data using even the standard Java IDE debugger.

More importantly, FlumeJava uses *deferred evaluation* in order to achieve better performance and to actually transform the plan into MapReduce pipeline. When a programmer writes a logical code to be executed, it may not always be the most effective plan among the variations of plans with same logical output. Additionally, as the user is abstracted from the MapReduce itself, a proper mapping to the MapReduce rounds has to be found. Therefore, the deferred evaluation essentially means that the invocations of operations are not evaluated immediately, but only after the whole execution plan is constructed. All the operations

are just chained together with their logical outputs to form an execution plan for the whole computation. This allows the library to firstly optimize the plan before the computation is actually executed. FlumeJava even chooses the strategy to implement for each operation in the runtime, based on the size of intermediate data (e.g. local sequential loop vs. remote parallel MapReduce) and can perform independent operations in parallel. It also handles all the creation / clean-up of intermediate files created withing the computation. It is much more effective, and easier to understand or change than hand-optimized chain of MapReduces. As the Google states,

"The optimized execution plan is typically several times faster than a MapReduce pipeline with the same logical structure, and approaches the performance achievable by an experienced MapReduce programmer writing a hand-optimized chain of MapReduces, but with significantly less effort."[3]

FlumeJava uses the MapReduce algorithm just as described in previous section, but with a few modifications:

- Map phase is generalized - multiple input sources with associated mappers are allowed.
- If the Reducers first combine all the values using an associative and commutative operation, *Combiner* function may be specified by the user so that some partial combining is performed in the **Map** phase. The map workers will keep a cache of key-value pairs emitted by the mappers and try to combine as many as possible before sending them to the shuffle phase.
- In the **Shuffle** phase, each group of key-values pair is sent to a deterministically chosen reduce worker. As there may be many more distinct keys than workers, many distinct keys may be sent to the same worker machine (for different reduce instances of course), typically by using a bucket hash function. Alternatively, user can define a *Sharder* function to aid load balancing by specifying, which reducer should receive the group for given key.

1.2.1 Core classes

The major class is *PCollection<T>*, which is a huge immutable collection of elements of type T. It can be either ordered or unordered. This collection can be created either from a *Java Collection<T>* or by reading a file (multiple files spread across the cluster), which may be even binary if the process is given specification on how to read such file.

Another core class is *PTable<K,V>*, which is essentially a *PCollection<Pair<K,V> >*, and it is a large immutable multimap of pairs of specified types. The reason for creation of such class even if there is an equivalent representation by another class is, that it can implement different methods working only with the given type of data in the collection.

Last class worth to mention is *PObject<T>*. This class is a container for a Java object of type T. As the above-mentioned object can be either deferred or evaluated (we will get to this in the next section), the result is stored in this *PObject<T>* and the value can be extracted using *getValue()* after materialization.

For manipulation of these classes, the main way is to invoke a parallel operation on it. FlumeJava defines only a few primitive operations and the others are defined based on terms of the primitives. Together, there are only a few primitives:

1. **parallelDo()** - implemented by *PCollection<T>*. It is an elementwise computation on the whole input which produces a new output of type *PCollection<S>*. This operation takes a function-like object *DoFn<T,S>* as a main argument specifying, how to map each object from the input to values in output. This mapping function is specified in the *process()* function of *DoFn<T,S>*. The function *process()* will be passed an element from the *PCollection<T>* and an instance of emit function *emitFn*, which has to be invoked for every element which should be in the result. Second argument for *parallelDo()* is a type of object returned by the *DoFn<T,S>*. As there may be many instances of *DoFn<T,S>* running concurrently, it must not access any global variables, and ideally, should be a pure function.

2. **groupByKey()** - implemented by *PTable<K,V>*. It converts a multimap into a map of type *PTable<K, Collection<V> >* (more specifically, *PGroupedTable*), where each key is mapped to a collection of all values occurring in the former *PTable<K,V>* associated with the key.

3. **combineValues()** - implemented by *PGroupedTable<K, Collection<V> >*. It takes as an argument an associative and commutative combining function on the values of type *V*, and returns a *PTable<K,V>*, where each *Collection<V>* was reduced to a single value *V* using the given function. This function is, generally, just a special case of *parallelDo*. However, the associativity of given function allows part of this combining be done in mappers and finished in reducers, which is faster than doing it all in the reducer.

4. **flatten()** - function that merges the given list of *PCollection<T>* into one large *PCollection<T>*. More precisely, it does not copy all the inputs as this would be very

ineffective, but rather just creates a logical view of them as one *PCollection<T>*.

5. **operate()** - used to embed an arbitrary computation withing a FlumeJava pipeline, so that it is executed in a deferred fashion. This function can be used to manipulate contents of *PObject<T>* within the execution of a pipeline. Basically, it takes a list of *PObject* as an argument and returns another list of *PObject* as a result.

Indeed, there are many more operations defined for every of the mentioned class, but they are defined using these primitives, no different than just helper methods a programmer could write (e.g. *count()* is implemented using *parallelDo()*, *groupByKey()* and *combineValues*).

1.2.2 Optimizer

As mentioned before, FlumeJava uses deferred evaluation to enable optimization. Every *PCollection* (or any other of the classes in previous section) can be either in deferred or materialized state. A deferred object holds reference to the operation, which computes it. A deferred operation holds reference to its arguments and the objects, where the results are to be stored. As we can see, result of executing some series of FlumeJava operations is a directed acyclic graph. This graph is called an *execution plan*. An example execution plan may look like the following, as stated in *FlumeJava: easy, efficient data-parallel pipelines*[3]:

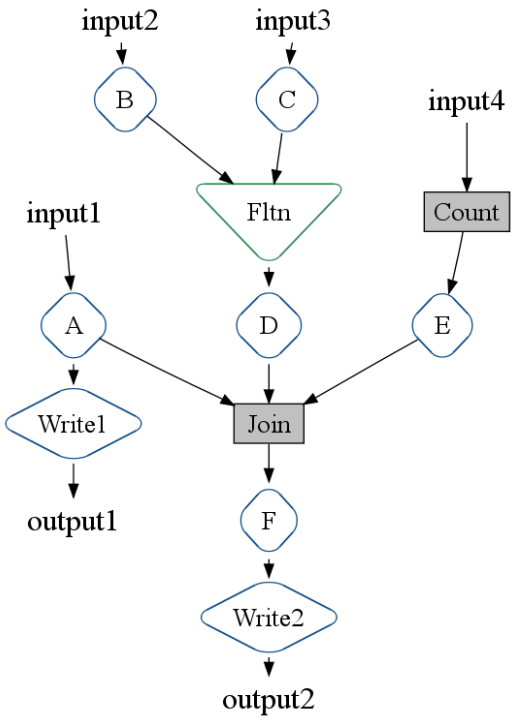


Figure 1.2: Example execution plan[3]

Nodes *A,B,C,D,E,F* represent different *parallelDo* functions.

1. Inputs 2 and 3 are fed into *parallelDo B* and *C*. Input 4 is counted using special operation *count* and the result is processed by *E*. One copy of input 1 is written to an output file and the other copy is processed by *A*.

2. Results of *B,C* are flattened together and pushed as an input for *D*.

3. Results of *A,D,E* are joined together using derived operation *join()* and the result is processed by *F*.

4. Result of *F* is written to an output file.

Programmer specifies this pipiline using object *MRPipeline* and can eventually invoke materialization of all objects in this pipeline by calling *MRPipeline.done()*. This will at first start the built-in optimizer, which will try to optimize the execution plan. The main aim of the optimizer is to map groups of operations to MapReduce rounds in the most efficient manner, in terms of time and resources. Eventually, after the optimization, the plan is run and all the given objects are materialized. We'll now take a closer look into the optimizer strategies.

ParallelDo Fusion

Fusion is very simple and intuitive, intended to reduce the number of *parallelDo* functions. As *combineValues* operation can be thought of as a special case of *parallelDo*, these fusions apply on that operation too. We distinguish between two types of fusions:

1. **producer-consumer** - Let f and g be two *parallelDo* functions. If the result of f is used as an input for g, then these two *parallelDo* functions can be replaces by a single multioutput *parallelDo*, which computes $g \circ f$.

2. **sibling** - Two or more *parallelDo* operations use the same input - they can be fused together into one multioutput *parallelDo* operation, which computes all the results in a single iteration over the input.

Sink flattens

In the execution plan, there might be a flatten directly followed by set of subsequent *parallelDo* operations O. For the optimizer it is better to create a *parallelDo* operation for every operation from O before the flatten for every input source and flatten them afterwards. Optimizer then has better opportunities of constructing the *ParallelDo Fusion*. In terms of functions f, g, h, which operate on multisets: $h(f(a) + g(b))$ is transformed to $h(f(a)) + h(g(b))$, so that the optimizer has the opportunity to fuse it to $(h \circ f)(a) + (h \circ g)(b)$.

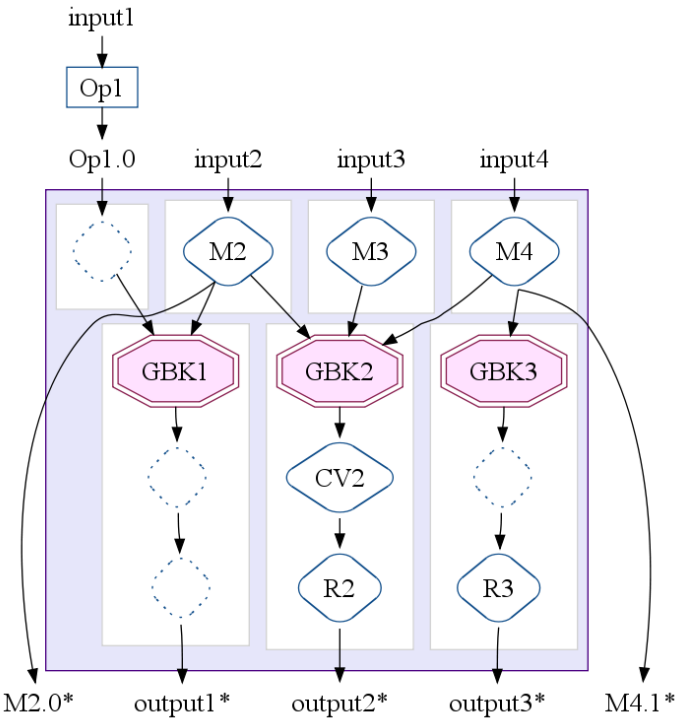
MSCR (Map Shuffle Combine Reduce) Fusion

This is the core optimization pattern in FlumeJava. It transforms combinations of *parallelDo*, *groupByKey*, *combineValues* and *flatten* into one MapReduce. As the algorithm may not be obvious from the first sight, let us introduce the *MSCR* operation. The *MSCR* consists of M input channels and R output channels. All of these input channels perform one (or any amount, because of the producer-consumer fusion) *parallelDo* (or as to say, *map*) operation with R outputs. These outputs may go to one or more of the R output channels. All inputs per channel are flattened, and in the next stage we allow for two types of output channels:

1. ***groupByKey*** ("shuffle") will be perfomed, and optional *combineValues* and *parallelDo* (as a "reduce") may be performed afterwards.

2. **Outputting** the value is required. No grouping nor *parallelDo* is performed and the values are just pushed through the channel.

After both cases, the data from every channel is pushed to one of the *R* output channels.



MSCR is a generalization of one MapReduce round. We can think of the initial *parallelDo* phase in every input channel as the *map* phase, of the *groupByKey* phase as the *shuffle* phase and eventually, of the *combineValues* and *parallelDo* as the optimized *reduce* phase in MapReduce. It is a generalization, because it allows more types of mappers and reducers. This generalization has less constraints and is thereby a better target for optimizer.

Figure 1.3: Optimized *MSCR* from figure 1.4[3]

We will show one way, how *MSCR* can be performed in a single MapReduce round.

- **Map phase** - as we assume a generalized MapReduce model, which allows multiple input sources with associated mappers, we can use different mappers for each input channel. For the next stage to work properly, without mixing the elements between channels, we can tag key from every emitted pair with a channel (e.g. from < K, V > make < < Channel, K >, V >). If *combineValues* is specified for a channel, the mappers will try to combine as much as possible before emitting a pair to the channel, as specified in 1.2.

- **Shuffle stage** - this is where the *groupByKey* will be executed in every channel. Because of the previously done marking, all of these keys will stay in their appropriate channels and the distribution in channel will be exactly the same as if the *groupByKey* was running independently (not in optimized *MSCR*).

- **Reduce stage** - Reducers will receive pairs with a channel specified in the key. This way, they can determine the channel they work with and the reducer they should mirror. If the *combineValues* was specified in the channel, reducer firstly finishes combining, removes the channel from key, and runs the mirrored reducer on the new pair. The output from the mirrored reducer is again tagged by a key, so that it could be sent to proper output channel.

Now only remains to show, how to create the described *MSCR* fusion. Firstly, we need to find a proper pattern in the execution plan. We will start with a few definitions to ease the description of searched pattern.

Definition 1.2.1. *We say that a PCollection is an input source for groupByKey operation, if it is one of the arguments of the preceding flatten operation, or is a direct argument of the groupByKey operation.*

Definition 1.2.2. *We say that two groupByKey operations are source related, if any of their input sources is common (produced by the same parallelDo operation). Furthermore, if we assume a transitive closure (Φ) of this relation, we say that a set M of groupByKey operations is source related, if $\forall x, y \in M : (x, y) \in \Phi$.*

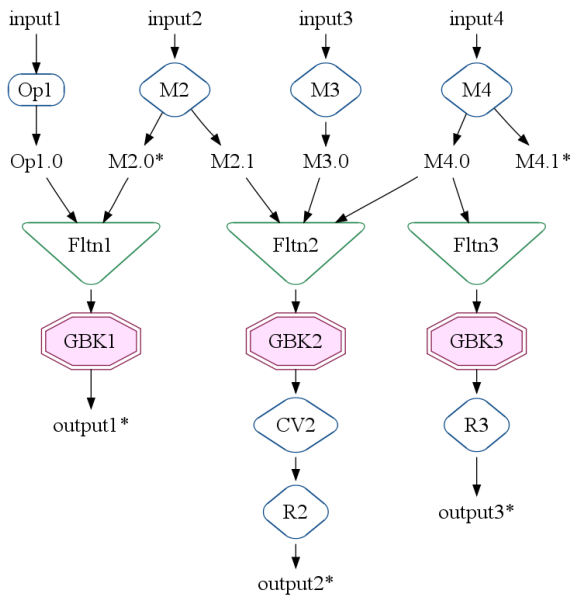


Figure 1.4: Example pattern for *MSCR*[3]

We find a *groupByKey* operation, and then the largest *source related* set containing this operations. Every of these *groupByKey* operations will form a distinct *groupByKey* type channel. Then, we find all the *parallelDo* operations, where one of their outputs is used as an input source for any of the considered *groupByKey* operations, and construct a new channel for every output of these operations, which is not consumed by any of these *groupByKey* operations. These new channels will be the *outputting* type channels. The input channels will be all the channels, which are input sources for the considered *parallelDo* operations.

The output channels are now logically equivalent to the corresponding output channels in the former execution plan, but the plan can be executed in a single MapReduce round.

Eventually, when all the *groupByKey* operations are replaced by appropriate *MSCR* operations, the optimizer creates a trivial *MSCR* operation for every remaining *parallelDo* operation. Afterwards, there remain only the *MSCR*, *flatten* and *operate* operations.

Optimizer strategy

Now that we have defined all the optimizations the optimizer uses, we shall explain the algorithm how the optimizer actually optimizes the execution plan.

1. **Sink Flattens** - optimizer prepares transforms the execution plan to one with more opportunities for *ParallelDo Fusion*

2. **Lift combineValues** - if a *combineValues* is about to happen immediately after *groupByKey* operations, the *groupByKey* remembers it, which allows for combining optimization while grouping. The original *combineValues* is then treated as a regular *parallelDo*, able to fuse.

3. **Create fusion blocks** - Preparations for *producer-consumer* and *MSCR* fusions. The *MSCR* fusions can be dependent - two *groupByKey* operations can be connected by a chain of *parallelDo* operations. The optimizer has to decide, which *parallelDo* operations should it fuse "up" and which "down". The optimizer firstly estimates the size of intermediate *PCollection*-s along the chain and tries to find one with minimal size. This will be the boundary between the two *MSCR* fusions.

4. **ParallelDo fusion** - fuse *parallelDo* operations with respect to the fuse blocks, using either *producer-consumer* fusion or *sibling* fusion.

5. **MSCR fusion** - create *MSCR* operations from the fuse blocks.

6. **Dispose of *parallelDo*** - convert any remaining *parallelDo* operation to trivial *MSCR* operation.

1.2.3 Optimizer imperfections

The optimizer described above is not perfectly optimal. As any optimizer, even this has some space for improvements. As stated in *FlumeJava: easy, efficient data-parallel pipelines*[3], there are a few imperfections:

1. **redundant operations** - sometimes, the users write execution plans containing duplicate operations or more subsequent *groupByKey* operations (maybe disguised as *join*), resulting in less efficiency. The optimizer should be able to detect such cases, and remove the unnecessary operations.

2. **user-code analysis** - this optimizer does no analysis on user-written code. While determining the boundary of a fusion block, it only decides based on the structure of the execution plan and some optional hints provided by the user on the size ratio of input and output. The optimizer can be improved by running a static analysis of the user code and get better expectations for the size ratios.

3. **modifying the user-code** - the optimizer does not modify user-code. Better performance may be achieved by generating a new code, which would represent the appropriate composition of the user's functions.

1.2.4 Execution

FlumeJava executor now supports batch execution. This means, that the independent operations in forward topological order are executed concurrently. This way, it enabled a kind of task parallelism.

We will now take a look, how the executor runs the newly constructed *MSCR*. Firstly, FlumeJava decides, whether to run the operation locally, or as a remote - parallel MapReduce. This decision is made based on the size of input sets. FlumeJava estimates the sizes based on the hints about the input / output ratio provided by user and the structure of execution plan. As there is some overhead in launching a remote job, the local, sequential evaluation is preferred for smaller inputs, while the remote, parallel MapReduce, is used mainly for large input sets. If the FlumeJava decides to run the operations as a remote, it automatically chooses a reasonable number of workers (mappers & reducers) based on the observations of input data and the optional user hints.

FlumeJava also makes it easier when finding a bug in a large pipeline: it supports a *cached mode*, where the FlumeJava will rather use the result from the previous run, than recompute the whole operation from scratch. This will happen, when the FlumeJava determines the reuse of previous results is safe.

1.3 Apache Crunch

In the previous section, we have introduced the FlumeJava model, along with its features. However, FlumeJava is a closed-source implementation bound by a software patent, so we will not be able to work directly with the associated implementation of FlumeJava by Google. On the other hand, some open-source implementations based on FlumeJava model have been developed after the release of FlumeJava (Apache Crunch, Plume).

Apache Crunch claims to be based on the FlumeJava model - as is the optimizer, which will be the main point of interest in this thesis. Indeed, according to the documentation in [1], the API and workflow is very, very similar.

In the next chapter, we will take a closer look at the abstraction of the optimization problem itself, and discuss its complexity.

Chapter 2

Optimization of network usage

Basically, the main aim of the optimizer is to prepare a plan, which processes the data in the fastest way possible while using the least resources. These jobs are mostly run on a large network with huge number of workers. As the MapReduce is highly scalable, it only makes sense for the companies to have one (or very few) large network used for MapReduce jobs. Also, there are companies offering to run these jobs as a service. In both cases, there is a high possibility that more jobs will be running concurrently on one large network. Hence, one of critical points of the optimizer should be to minimize the network usage.

During the execution of a plan, there are basically two options for what is happening to the data at any point. The data may be:

1. processed or moving within a worker
2. sent to another worker via network

There is no network usage in the first case, because all the data is only moving within the worker, without touching the network. This happens when the data is being processed in a worker by a particular (possibly fused) map or reduce operation.

For the latter case, there are again two possibilities where it can take place: the data is travelling from worker at the end of one MapReduce round to a worker at the beginning of another, or the data is being sorted by distributed sort in the shuffle phase. We cannot influence the amount of data sorted in the shuffle phase by modifying the creation of MapReduce pipelines from given execution plan, because if we want to create an equivalent plan in terms of input and output, the total amount of data flowing through an operation should stay the same. The amount of the data travelling between two MapReduce rounds directly depends on where the fusion block is inserted. The optimizer is responsible for dividing the plan to fusion blocks. In this chapter, we will study how to improve it.

2.1 Execution plan as a graph

As we've previously stated, the execution plan consists of derived, and a few primitive operations. After all these derived are replaced by a combination of primitive operations, the plan eventually consists only of these four primitive operations: **parallelDo***(PD)*, **combineValues***(CV)*, **groupByKey***(GBK)*, **flatten**.

Firstly, we will consider a very straightforward graph representation of an execution plan:

1. we create a vertex $v \in V$ for each of the primitive operations
2. $\forall u, v \in V$: if there is an edge between u, v in execution plan, then $(u, v) \in E$
3. contract the edge between every flatten operation and the operation it is flattening into (right after it in the plan)

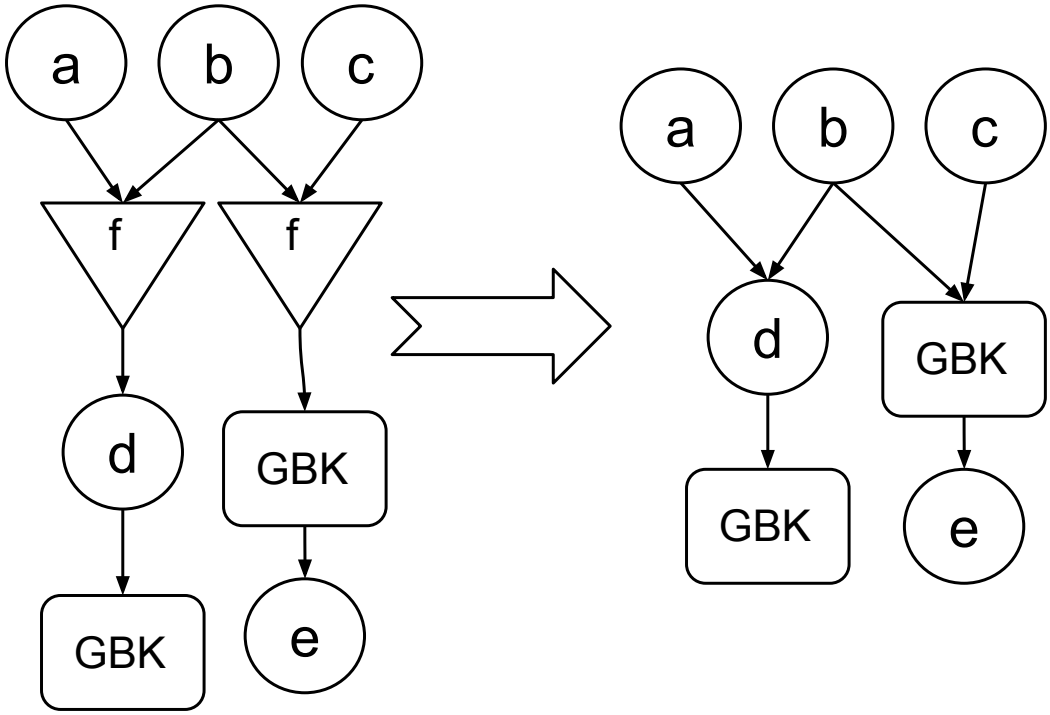


Figure 2.1: Example transformation of plan to a graph, where *a, b, c, d* are **parallelDo**, *f* is **flatten**, *GBK* is **groupByKey**, and *e* is **combineValues**

Additionally, for every edge $(u, v) \in E$ we define the capacity function $c(u, v)$ as the amount of data flowing between the operations corresponding to u and v in the original execution plan. Moreover, as the **combineValues** operation is just a special case of **parallelDo** operation, we will treat them both equally - as a **parallelDo**. This yields quite simple **DAG** representation of an execution plan, which we will use from now on.

As we have this representation now, we can define a following simple property:

Definition 2.1.1 (dependency)**.** *Given a graph G as a representation of an execution plan, we say that operation* ***b*** *is* ***dependent*** *on operation* ***a*** *if there exists a path in G from the vertex corresponding to* ***a*** *to the vertex corresponding to* ***b****.*

2.2 Partitioning for basic plans

If there is at most one *GBK* operation, the final execution plan produced by the optimizer is very simplistically optimal, because everything there can be done in a single *MapReduce* round. The network usage in this case is optimal, as there are no intermediate writes to common storage.

We shall now consider a simple class of execution plans with exactly two *GBK* operations. Additionally, to ensure that the plan can not be executed in a single *MapReduce* round (or two parallel), we consider only plans where one of the *GBK* operations (b) is *dependent* on the other (a).

As stated in previous section, the optimizer can affect the network usage by dividing operations into fusion blocks. As we now have only two of these *GBK* operations, there will only be two fusion blocks, and the aim of the optimizer is to minimize the data transferred between these blocks. The operations which are *dependent* on *GBK B* have to be in the same block as the *GBK b* and, similarly, the operations on which the *GBK a* is *dependent* have to be in the same block as *GBK a*. Therefore, these operations will never affect the choice of fusion blocks, and it is unnecessary to consider them in this graph. Thus, the graph is reduced only to these two *GBK* operations and operations on the path from a to b.

Before we continue any further, we should remind ourselves of the definition of a **Minimum s-t cut**

Definition 2.2.1 (Minimum s-t cut)**.** *Given a directed graph $G = (V, E)$ with edge capacity function $c(u, v)$, a cut refers to a partition of the node set into two nonempty parts S and $N - S$. An* ***s-t cut*** *$C = (S, T)$ is a partition of V such that $s \in S$ and $t \in T$. The* ***capacity*** *of and s-t cut is $c(S, T) = \sum_{(u,v)\in(S\times T)\cap E} c(u, v)$. The* ***minimum s-t cut*** *problem is to determine sets S and T such that $c(S, T)$ is minimal.*

2.2.1 Direct application of min-cut

While still considering the reduced graph as in previous section, it may seem obvious that the minimum network usage is the minimum cut in this graph. Hence, we will now analyze this minimum cut between *GBK a* and *GBK b*.

As an example, consider the graph on the right, and that the minimum cut divides the operations to sets **S** and **T** as shown on the image. The edges going from **S** to **T** will be the only edges transferring data between these two MapReduce rounds. However, there can also be edges going from **T** to **S** - and as the block **S** has to be processed strictly before the block **T**, there is no way to transfer data from an operation in **T** to an operation in **S**, and we have to handle this case differently. At this moment, we make use of the *stateless* property of *PD* operations. This means that the *PD* operations should only depend on their input, and nothing else. Therefore, if we interpret the operation as a function f getting inputs a and b, then $f(a+b) = f(a) + f(b)$. In other words, the operations behave as linear functions, and we can transform the data by parts.

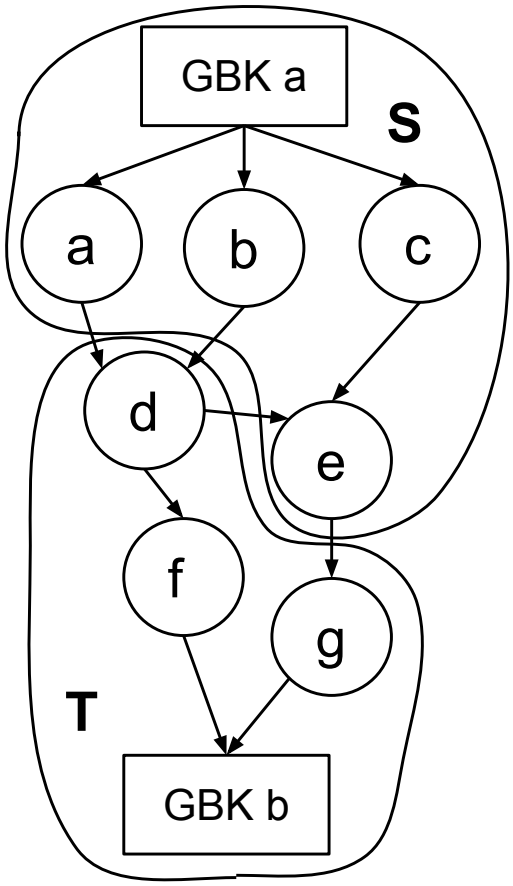


Figure 2.2: Partitioning the execution plan

Before we make any modifications to a plan, we should define what it means when two plans are *equivalent*.

Definition 2.2.2 (Equivalency)**.** *Two plans are equivalent, if for every input dataset, both plans output the same dataset.*

Taking into account this case, where only two **dependent** *GBK* operations are present, we assume that the input dataset is the output from *GBK a* and the output dataset is the input for *GBK b*.

Operation division Consider an operation f taking inputs a and b. We make a copy, f' of f, and make a the input for f and b the input for f'. As for the outputs, the capacities of both $f(a)$ and $f(b)$ must have changed, because they receive less input sources. However, there is

no way to decide the sizes of these outputs from the capacity function as it is now defined. Hence, we will now define the capacity function in a different manner.

Definition 2.2.3 (Dataflow coefficient)**.** *For every pair of edges $e_1 = (u, v), e_2 = (v, w), e_1, e_2 \in E$ we define the dataflow coefficient $df(u, v, w) \in R^+$ as the ratio between the size of the output to w and input of v, provided that the input comes from u.*

Using this definition, the capacity function can be computed using the following recursion: $c(v, w) = \sum_{(u,v)\in E} c(u, v)df(u, v, w)$, where $\forall e = (GBK\ a, u) \in E$ the $c(e)$ are sizes of outputs from *GBK a* given in the graph.

We will now use this division of an operation to dispose of edges from **T** to **S**. Whenever there is such edge between $u \in T$ and $w \in S$, we divide w, and place the copy w' connected to u into T. If w was connected to v, then the corresponding outputs of the w' and w are divided, and they merge back as inputs for v. By making use of the *stateless* property of operations, the plan after this step is *equivalent* to the original plan. If we repeat this until there are no edges from **T** to **S**, we finally have a graph, which is valid in terms of creating MapReduce rounds.

However, the sum of capacities of the edges from **S** to **T** in the final graph will always be lower (if there were any edges from **T** to **S** in the former graph) than in the former. Thus, it might happen that there is a cut which is not minimal in the former graph, but will eventually (after dividing the necessary operations) be smaller in the final graph.

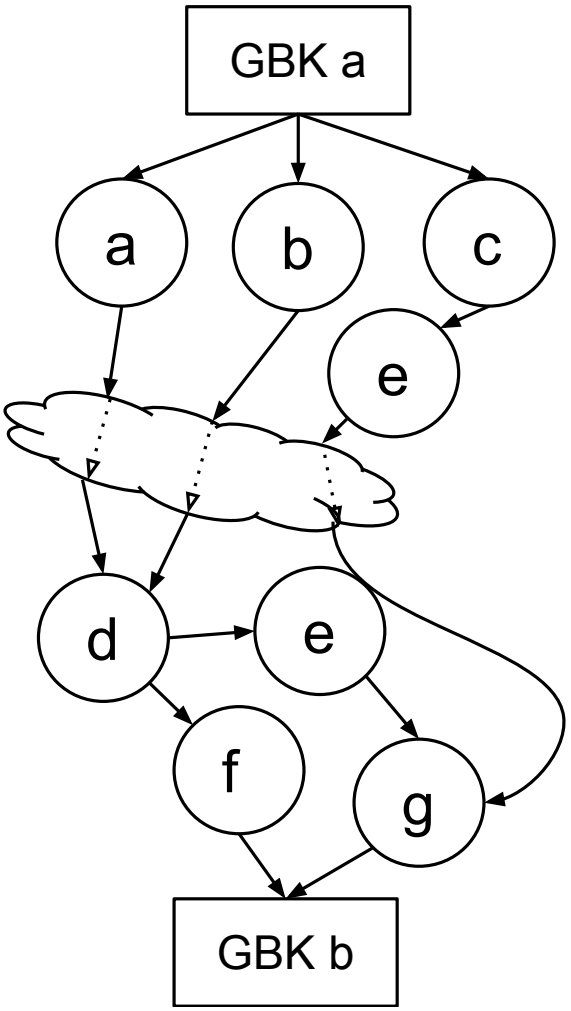


Figure 2.3: Operation division

Note that after an operation is divided, they both represent the same operation. Although there are now two vertices representing the operation, they both maintain the *dataflow coefficient* of the original operation.

We will elaborate on this hypothesis using an example. Consider a graph in figure 2.4. Weights on edges illustrate expected amount of data (the units are now irrelevant) flowing between the operations. For every two vertices x, y in the graph, we will now denote the data going from x to y as $p(x, y)$. Consider that the *PD* d has the following behavior: $|d(p(c, d))| = 3$ and $|d(p(a, d))| = |d(p(e, d))| = 1$. The minimum cut of weight $3 + 5 + 2 = 10$ is obtained by partitioning the vertices into sets $\{a, c, d\}$ and $\{b, e\}$. Then, after the graph is prepared for insertion of fusion blocks, the total data passing through common storage is $3 + 4 + 2 = 9$.

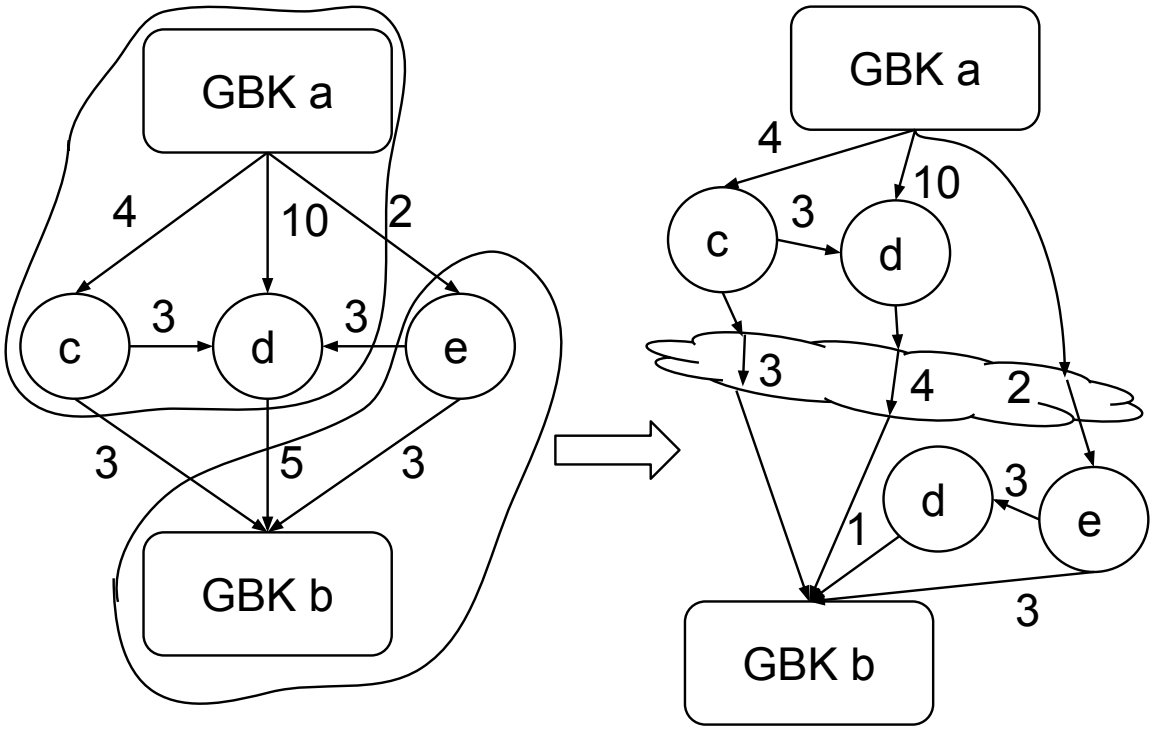


Figure 2.4: Direct application of Min-cut

However, this naive direct application of min-cut does not take in account the behavior of operations on particular inputs, but only on all inputs merged together. If we consider also the outgoing data flow for every particular input for operations with more than one input (currently, only d), we might get better final results. The operation d has, basically, 3 outgoing edges of weights 3, 1, and 1. Previously, we let the edges of weights 3 and 1 pass through the common storage. It might be more beneficial to run the heavy edge (3) locally, and let the light edges pass through the common storage.

Consider partitioning this graph into sets $\{a, d\}$ and $\{c, e, b\}$, as seen in figure 2.5. Then, the cut has weight $4 + 5 + 2 = 11$. However, after the graph is prepared for insertion of fusion blocks, the data passing through common storage is expected to be only of weight $4 + 1 + 2 = 7$, which is less than the partition found by direct application of min-cut.

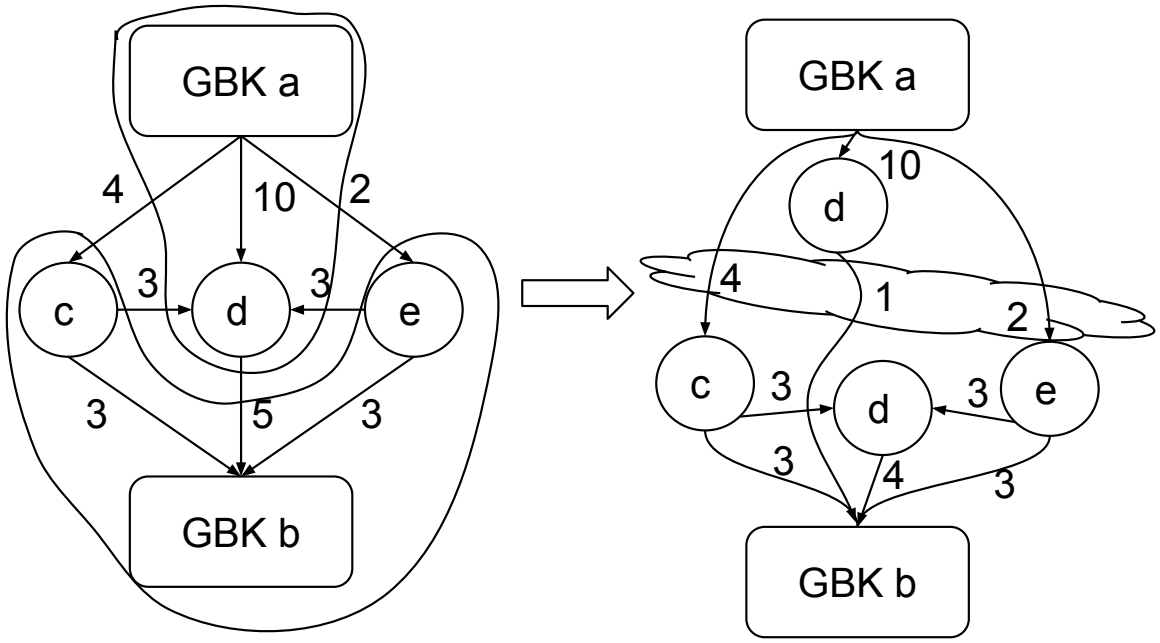


Figure 2.5: Counterexample for direct application of Min-cut

Therefore, we need to find a way how to take these outgoing edges into consideration when calculating this Min-cut.

2.2.2 Optimal partitioning

The problem in previous section was that the graph, after it was partitioned, needed to be adjusted for insertion of fusion blocks, and therefore its final dataflow through a common storage might have changed. We will try to pre-modify this graph in such a way, that this problem will be eliminated.

Definition 2.2.4 (Normalized plan)**.** *We call and execution plan **normalized**, if every parallelDo and combineValues operation has at most one input stream.*

Lemma 2.2.1. *Consider a normalized plan. The weight of Min-cut on a graph corresponding to this plan is also the minimum data flow in the normalized plan.*

Proof. We will prove this by contradiction. Denote the partitions S and T as in previous section. Find a Min-cut on the graph and consider the opposite. Then, there must have been an edge from $t \in T$ to $s \in S$, otherwise the plan would be prepared for fusion blocks without any modifications, and the data flow would not change. Consider a graph H induced by vertices in S reachable from s. By definition of H, all outgoing edges from H end in T and thus contribute to the cut. Denote these edges E_{HT}. We now move all vertices in H to T. As all of the vertices have exactly one incoming edge, there are no new edges between S

and T. However, the weight of the cut is reduced by the sum of weights of edges in E_{HT}. Contradiction with the previously found Min-cut. □

Finally, what remains is to normalize any given execution plan. Then, using the lemma 2.2.1, we will find find the Min-cut of the normalized plan, and construct the final and optimal execution plan, prepared for insertion of fusion blocks.

Corollary 2.2.1.1. *Consider execution plans G and H. If for every path from GBK a to GBK b in G exists a path in H consisting of the same operations in the same order, and vice versa, the plans G and H are equivalent.*

Lemma 2.2.2. *For every graph G of an execution plan consisting of distinct vertices and two dependent GBK operations, there exists a normalized plan H, such that G and H are equivalent.*

Proof. We will present an algorithm for finding a normalized plan. Consider a reverse topological order of vertices in G. Such order must exist, as the graph is directed and acyclic. Process vertices in the defined order as follows:

1. Let v be the currently processed vertex, and P_v be the set of predecessors of v.
2. Create $|P_v|$ copies of subgraph induced by vertices reachable from v.
3. For every vertex in P_v, remove the edge to former v and connect one copy as a child.

After all vertices are processed, we might end up with more copies of *GBK b* - we merge these into a single one. Note that it follows from the order defined, that each time a subgraph is copied, every vertex in the subgraph was already processed. Therefore, this algorithm produces a normalized plan H in a finite time. What remains to show is that G and H are equivalent. Consider a path p from *GBK a* to *GBK b* in G. Let E_p be the set of edges on this path. Every edge $(x, y) \in E_p$ is also in H, because it was created when y was being processed. Vice versa, consider a path q from *GBK a* to *GBK b* in H. Every edge $(w, z) \in E_q$ is again in G, because the edge (w, z) was created when z was processed, and therefore w must be parent of z in $G \implies (w, z) \in G$.
Lastly, we will show that even though there are duplicate vertices in H, there are no duplicate paths. Notice that every vertex has at most one child with given label, because every vertex is processed exactly once and his copies are attached to distinct parents. It follows that there must be no two paths distinct in terms of vertices, but matching in terms of labels (labels of duplicated vertices stay the same), as there would be a vertex having 2 children with the same label. It follows now from corollary 2.2.1.1, that G and H are equivalent. □

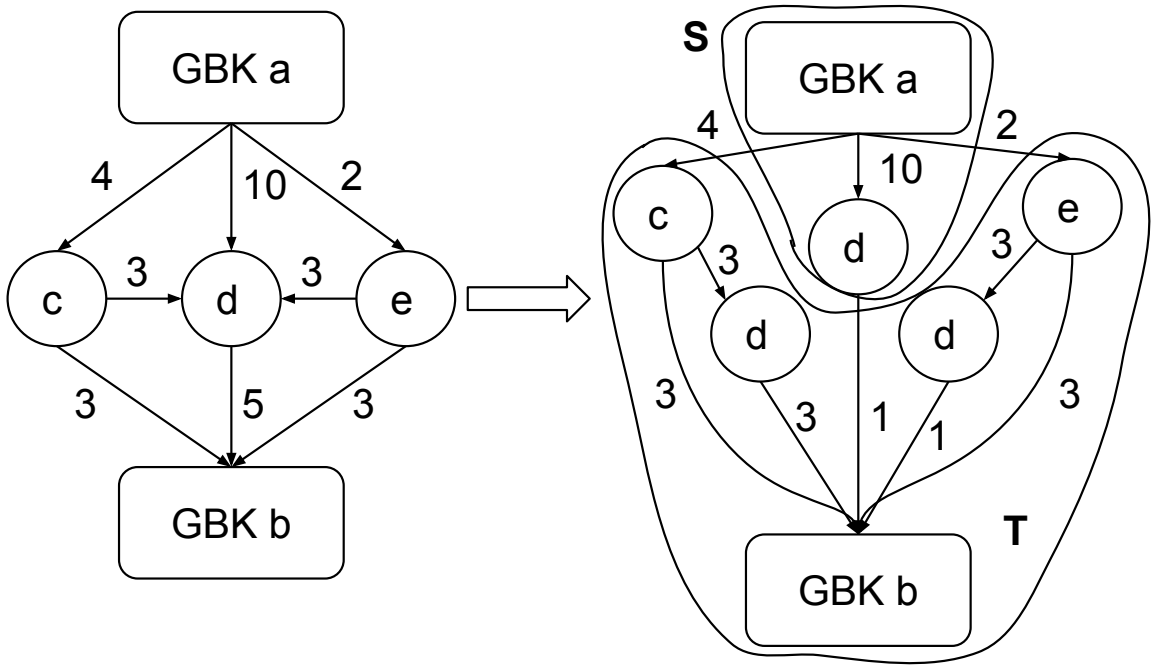


Figure 2.6: Algorithm for plan normalization

Lemma 2.2.3. *Consider a normalized graph, and a vertex u in this graph. The edges on all pairs of paths starting at distinct outgoing edges of u, and ending in GBK b, are completely disjoint.*

Proof. If we omit the *GBK b*, the graph is, essentially, a rooted tree with U as root. Therefore, for every node, the subtrees rooted in its children are completely disjoint. □

Lemma 2.2.4. *The Min-cut between a node u and GBK b in a normalized plan can be computed using the following recursion:*

$$mc(u) = \sum_{(u,v)\in E} min(c(u,v), mc(v)),\ where\ \forall(u, GBK\ b) \in E : mc(u) = c(u, GBK\ b)$$

Proof. Every cut starting at an outgoing edge from u has to be cut somewhere. Following the Lemma 2.2.3, these paths are independent per child of u. Hence, cutting an edge in subtree of child x of u does not cut any edge in subtree of any other child y of u. The cut can be then calculated recursively for each child x of vertex u by choosing the minimum between cutting the edge (u, x) or separating x from the *GBK b*. □

Corollary 2.2.4.1. *The value mc(GBK a) stores the Min-cut of the given normalized plan.*

Complexity

Running time of this algorithm is linear $O(|V|)$, as we touch every edge at most twice (one can imagine this as a DFS traverse). Slight modification can be made if we want the edges

of the cut - each node would also remember the choice it made for every edge (it has either added that edge to the cut, or some edge from the subtree on the other end). The cut can then be reconstructed again in linear ($O(|V|)$) time.

Computations on the *normalized* graph are linear, however, the transformation of a graph to the *normalized* form can exponentially increase the number of vertices. See example 2.7.

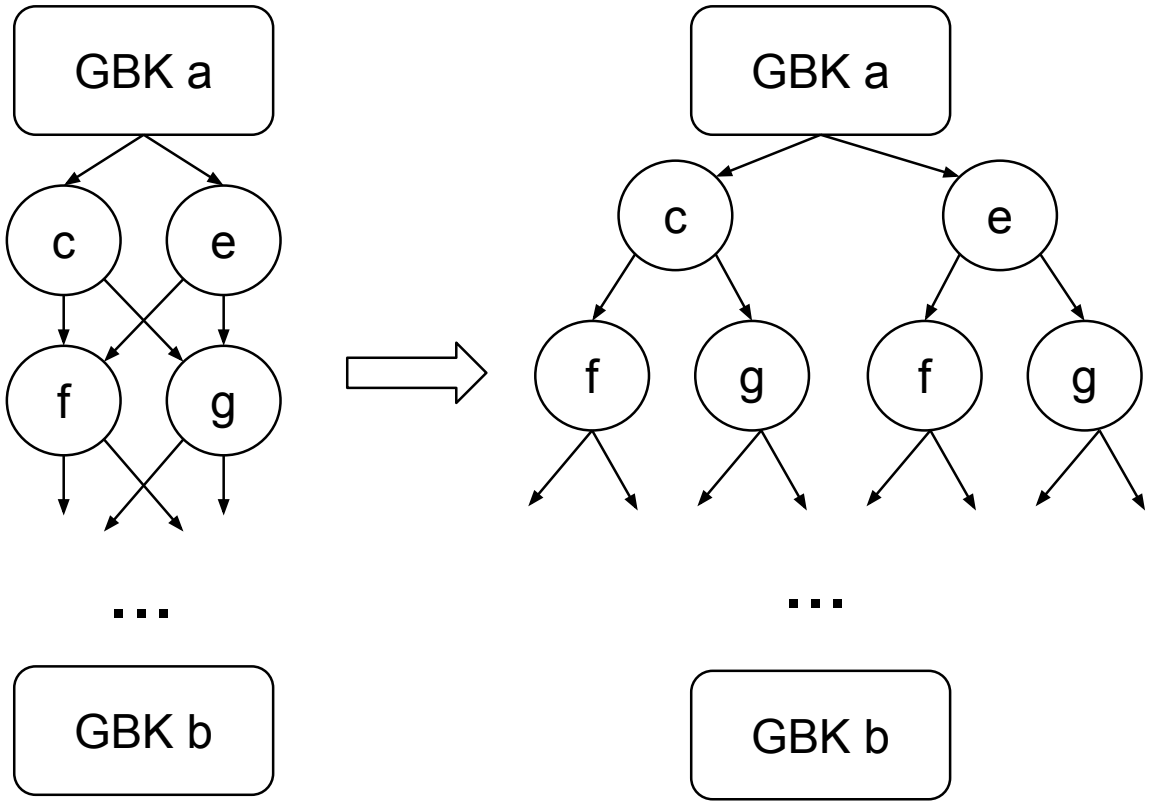


Figure 2.7: Exponential increase in graph size

The worst case, in terms of graph expansion, is when the input graph is a complete directed acyclic graph - adding additional edge would create a cycle. The *normalized* form of this graph, also contains an exponential number of vertices.

Lemma 2.2.5. *There are as many as 2^{n-1} vertices in a normalized form of a complete directed acyclic graph with n vertices.*

Proof. We will prove this by induction. Base - graph with one vertex is already normalized. By adding a new vertex v, connected to each of the previous vertices, the normalized graph can be constructed in the following fashion - construct a normalized graph rooted in each of the former vertices, and connect the roots of each to v. Therefore, the size of the normalized form of a graph with n vertices can be expressed recursively: $a(n) = 1 + \sum_{0 \le i < n} a(i)$, where $a(1) = 1$. Hence, $a(n+1) = a(n) + 1 + \sum_{0 \le < i < n} a(i) = 2a(n)$. □

Although this increase in graph size can be exponential, the algorithm is still much better than a brute-force, which we will analyze in the next section.

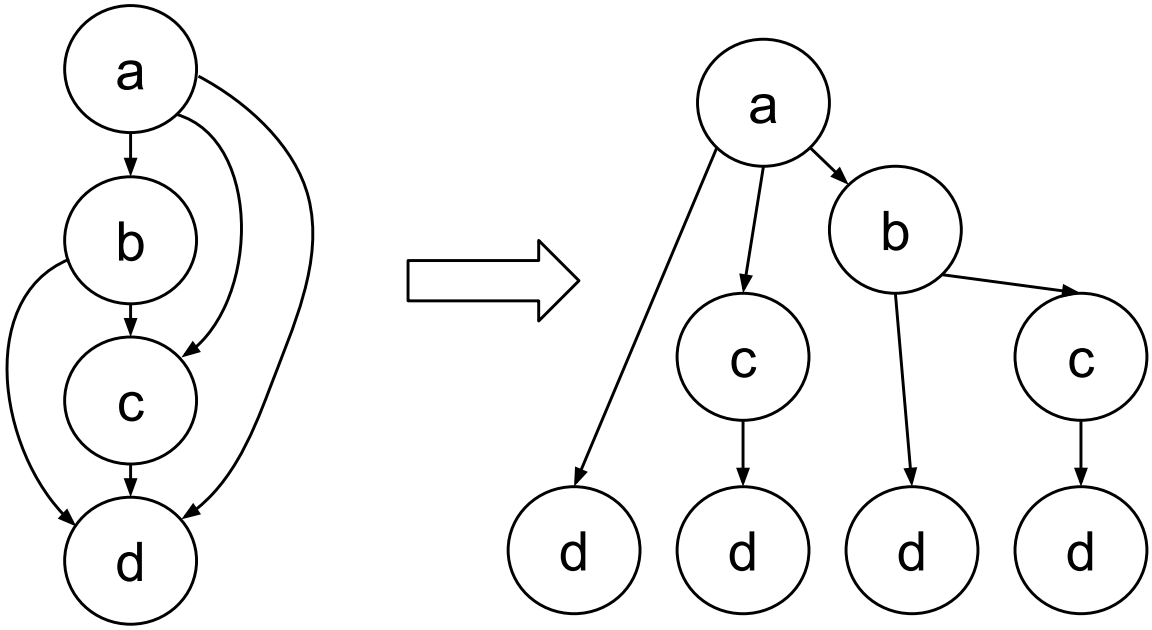


Figure 2.8: Normalized complete directed acyclic graph

2.2.3 Comparison with brute-force algorithm

Definition 2.2.5 (Viable)**.** *A partitioning of a graph $G = (V, E)$ to partitions S and T, such that $\nexists (u, v) \in E : u \in T \wedge v \in S$, is called **viable**.*

Corollary 2.2.5.1. *Every **viable** partitioning directly defines fusion blocks without any necessary changes to the plan.*

A brute-force algorithm would work in the following fashion - find every viable partitioning, compute the weight of the cut, and return the smallest one. However, the number of viable partitionings may be very large. Basically, as we are able to split any node by its incoming and outgoing edges, we can also consider a very strict form, which consists of only vertices with a single incoming and outgoing edge. This representation contains a path consisting of unique edges for every path between *GBK a* and *GBK b*. Every viable cut has then cut every of these paths. If we interpret the *GBK v* as a leaf for every path, the number of viable cuts can grow as large as $\Pi_{(GBK\ a,u}(height(u) + 1)$. Hence, the number of viable partitionings in 2.7 is $\left(\frac{n-2}{2}\right)^{2^{\frac{n-2}{2}}}$, where $n = |V|$.

This expression really denotes the total number of viable cuts, because there are no backward edges if the cut is constructed as described, and the MapReduce rounds can be con-

structed. However, there are edges in different paths which contain identical data, so if one is cut, it only makes sense to cut the other (in terms of finding the minimal cut) - in other words, some data could flow through the same operations (in terms of types), but end up in different nodes. The given calculation does not take this into account. A more precise enumeration of *meaningful* viable cuts can be calculated on a graph representation, where all the data which has flown through the same sequence of operations ends up in the same node (with the exception of *GBK b*). This property is maintained in the *normalized* graph representation.

Lemma 2.2.6. *In the normalized plan, all data which was computed by composition of the same sequence of operations end up in the same vertex. (except for GBK operations)*

Proof. There are no duplicate children of any vertex in the normalized plan, because there would have to be duplicate operations in the original plan, as the operation type of children of any vertex was not changed. Additionally, as there is only one incoming edge to every vertex u, there exists a unique path between the root node (*GBK a*) and vertex u. □

Lemma 2.2.7. *Number of viable cuts between any node u and node GBK b in the normalized graph can be computed by the following recursion:* $cuts(u) = \Pi_{(u,v)\in E}(cuts(v)+1)$*, with initial value* $cuts(GBK\ b) = 0$.

Proof. As every node except the *GBK b* has a unique parent, the sets of edges in subtrees rooted in children of u are disjoint. Therefore, every child v of u has to be either cut somewhere in the rooted tree of v, or the edge (u, v) has to be cut. Additionally, as the sets of edges for children are disjoint, the cuts for each child are independent, hence the multiplication. □

Following the Lemma 2.2.7, the number of meaningful viable cuts of the *normalized* plan in 2.7 can be computed by the following recursion $c(2n) = (c(n) + 1)^2, c(2) = 1$, where n denotes the number of vertices. We will use the recursion $a(n) = (a(n - 1) + 1)^2$ which was solved by Aho and Sloane in *The Online Encyclopedia of Integer Sequences* [7] labeled *A004019* - $a(n) \approx b^{2^{n-1}} - 1$, where $b \approx 2.258518$. The definitions imply the following: $c(n) = a(\lg(n)) \approx b^{\frac{n}{2}}$. If the original plan in 2.7 has n vertices, the *normalized* plan has $2^{\frac{n}{2}}$ vertices $\Rightarrow$ the number of meaningful viable cuts in given example is $O(c(2^{\frac{n}{2}})) \approx O(b^{2^{\frac{n}{2}-1}})$.

Theorem 2.2.8. *A brute-force algorithm for finding the optimal partitioning by enumerating all viable cuts would in worst-case have to enumerate approximately* $\lfloor c^{2^n} \rfloor$ *cuts, where* $c \approx 1.59791$

Proof. Similarly, using the Lemma 2.2.7 for the complete directed acyclic graph in example 2.8, and approach used in the proof of Lemma 2.2.5, the number of *meaningful viable* cuts can be calculated using the following recursion:

$$a(n) = \Pi_{0\leq i<n}(a(i) + 1)$$

Hence,

$$a(n+1) = (a(n)+1)\Pi_{0\leq i<n}(a(i)+1) = (a(n)+1)a(n) = a(n)^2 + a(n)$$

This recursion was solved and published by *Benoit Cloitre* in *The Online Encyclopedia of Integer Sequences* [7], yielding $a(n) = \lfloor c^{2^n} \rfloor$, where $c \approx 1.59791$ □

The proposed algorithm is better by the factor of the number of viable cuts in the normalized version of a graph, while the number of these cuts may be exponential.

2.2.4 General model

In the previous section, we've supposed that all the data flowing from through an edge between two vertices in the graph is unique - in other words, that there are no two edges carrying the same data. However, if in the program there are more operations performed on a *PCollection*, which is the result from some operation u, then in the plan these outgoing edges from u carry identical data. Therefore, if there's a cut through more of these outgoing edges, it is only necessary to transfer the data once and copy it upon reading, as they are all identical. We will call this model the **General model**.

Additionally, as all the output edges from a vertex carry identical data, then for the *dataflow coefficient*, the following holds: $\forall u, v \in V, \exists c \in R^+, \forall (v,w) \in E : df(u,v,w) = c$. In other words, the last argument in df is irrelevant, as it necessarily yields the same result for all possible values. Hence, we will use the short form of $df(u,v)$ in this section.

Similarly, we will also modify the capacity function, as it does not matter, where the output leads to.

Definition 2.2.6 (Output function)**.** *For a given plan, we define the out function* $out(w) \forall w \in V$ *as the amount of data flowing from the vertex w, when given all input data for w.*

Corollary 2.2.8.1. *The output function* $out(w)$ *for each* $w \in V$ *can be computed recursively from given dataflow coefficient (provided that the input data from GBK a is normalized to 1)*

$$out(w) = \sum_{(u,w)\in E} (df(u,w)\ out(u)),\ out(GBK\ a) = 1$$

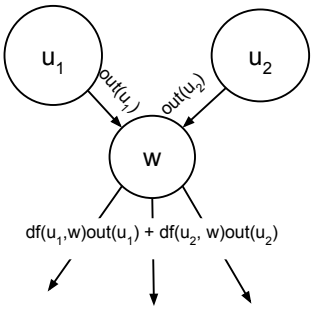


Figure 2.9: Out function of operation

Similar problem with edges like these was solved in [10]. The authors H. H. Yang and D. F. Wong propose a polynomial-time algorithm for finding a form of a Min-cut (minimizing the number of edges from S to T) in such graph. The algorithm firstly transforms all the outgoing edges from each vertex into a structure with similar properties as a hyperedge, and then computes a Min-cut on the transformed graph. However, even if we generalized the solution to finding a regular Min-cut, we would still face the same problems with backward edges as in section 2.2.1.

Algorithm for optimal partitioning

As in the previous model, we can solve this problem by transforming the plan into a *normalized* plan. Although there are many copies of the same vertex in the transformed plan, their outputs are independent. They do not transfer the identical data, because the path to each of the copies of a given operation is different. Hence, given a normalized plan we can compute the Min-cut in a similar manner.

Lemma 2.2.9. *Consider a normalized plan, where all output channels from an operation carry the identical data. The Min-cut between any node u and GBK b can be computed using the following recursion:*

$$mc(u) = min(out(u), \sum_{u,w)\in E} mc(w)) \text{ where } \forall(w, GBK\ b) \in E : mc(w) = out(w)$$

Proof. Similarly, as in Lemma 2.2.4, for every edge (u, w) of node u, either the edge has to be cut, or the subtree rooted in w has to be cut. In this case, however, we can cut all outgoing edges from u for unit price. Hence, we take the smaller from cutting the outgoing edge, and cutting all subtrees rooted in all children of u. □

Complexity of this algorithm stays exactly the same as in section 2.2.2, for the same reasons. And again, the normalization step of a plan can exponentially increase the number of vertices, so the worst case running time is again $O(c^{|V|})$ for some constant c.

2.3 Partitioning for general plans

In previous section we have only discussed the most simplistic case, where only 2 *GBK* operations were present in a plan, and these operations were **dependent**. However, most of the plans contain many more *GBK* operations, and some of them may not even be **dependent**. Therefore, we need to generalize the former algorithm in such a way, that it would be applicable for any execution plan.

2.3.1 Abstraction of general execution plan

Firstly, we will try to use the methods which worked in the previous section for the most simplistic case - *normalization*.

In the previous section, we have supposed that the data is created in the *GBK a* and consumed in *GBK b*. In other words, that there is no input for *GBK a* and no output for *GBK b*. However, in the plans in the real world the data has to get to each of the *GBK* operations somehow, and, similarly, every output of every operation is either consumed by another operation or outputted somewhere. Additionally, there might exist some operations which would like to read input directly, rather than the output of some other operation. Therefore, we will introduce a new *Input* vertex and *Output* vertex.

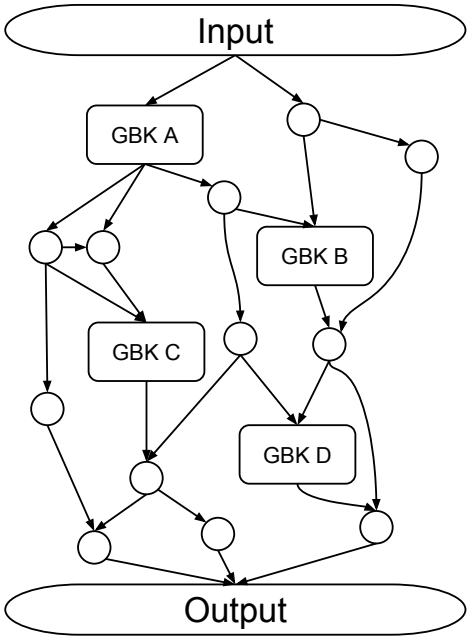


Figure 2.10: Example of a plan with more GBK operations

Definition 2.3.1 (Input)**.** *The Input vertex provides inputs for the FlumeJava program. Input vertex in plan has no incoming edges, and may have many outgoing edges, which do not necessarily provide the same data. There is always just a single Input vertex in any plan.*

Definition 2.3.2 (Output)**.** *The Output vertex is used as a stream for outputting (from the FlumeJava program) any data from its inputs. Output vertex in a plan has no outgoing edges, and may have many incoming edges.*

Since we have these new vertices, it is worth to mention that a plan with an operation without any **incoming** edge can be reduced to a smaller, *equivalent* plan by removing that operation - as it takes no inputs, its outputs are also empty, and it is meaningless. Similarly, a plan with an operation without any **outgoing** edge can be also reduced - if the output of an operation is never used nor outputted from the program, the operation is meaningless. Hence, we will now consider only graphs, where each operation (apart from Input and Output) has at least one incoming and outgoing edge.

2.3.2 Approaches to partitioning the plan

The plan might contain many *GBK* operations. Some of these operations might be *dependent* on other *GBK* operations, some do not have to be *dependent* on any. We will denote the set of *GBK* operations as M. As stated in previous section, no two *dependent* operations can be processed in single MapReduce round. Therefore, we need to separate each of the pairs $g_1, g_2 \in M$, where g_1 is *dependent* on g_2. However, as finding such cut in the original plan did not yield any reasonable results even for the simple case yet, we try to *normalize* the plan beforehand.

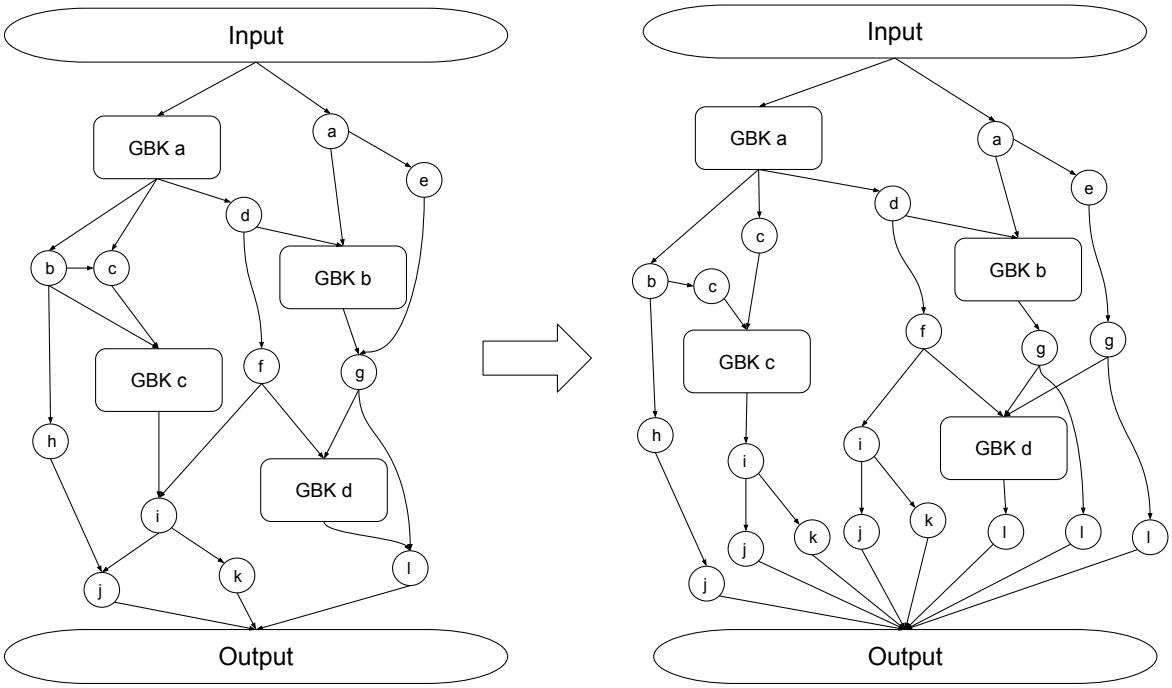


Figure 2.11: Normalized general plan

Although we have introduced new vertices (*Input* and *Output*), the definition of a *normalized* plan stays the same. However, as the plans may now potentially contain more *GBK* operations, we need to change the *normalization* algorithm. The *GBK* operations should not be divided, and it does not make sense to divide them, as we are looking for MapReduce block for each *GBK*. The data traveling through more *GBK* operations should only have memory of the last *GBK* operation in went through. In other words, if there are more data streams incoming to some *GBK a*, and we can not split it, then there is no sense in copying the subtrees rooted in *a*, as there would only by identical duplicities. Hence, when the normalization algorithm encounters a *GBK* operation, it will not split it nor copy its subtree. As for the *Output* operation, there should be only single one in the whole plan, and so it can be treated like a *GBK* operation. *Input* type vertex has no incoming edges, which makes it be skipped by the *normalization* algorithm. Although we do not copy the subtrees under *GBK*-s

for *GBK*-s they are *dependent* on, the graph may also exponentially increase in its size.

One direct approach would be to apply *Min-multi-cut*, however, this problem is known to be *NP-hard* [4], so we would like some better solution on a possibly large graph.

We will use similar approach as in the previous section - for every node we computed the cost of partitioning the vertex from the *GBK* used as *sink*. However, we now may have many more *GBK* operations in the graph. What we would like to achieve is that every *GBK* is separated from its *dependent GBK* operations. In terms of these vertices, we would like to separate the given vertex from all *GBK* operations, which are reachable from the vertex.

Definition 2.3.3 (Direct reachability)**.** *A GBK a is directly reachable from vertex u if there exists a path from u to a not going through any other GBK operations.*

Corollary 2.3.0.1. *If a vertex is separated from all of its directly reachable GBK operations, it is separated from all reachable GBK operations.*

Consider now a *GBK* operation a in the *normalized* plan. According to corollary 2.3.0.1, it is only sufficient to separate the vertex from all directly reachable *GBK* operations. This can be achieved by dynamic programming similar to the version used for simple graphs with 2 *GBK*-s.

Algorithm

In every vertex, we will keep an information about how much does it cost to separate it from all of its *directly reachable GBK* operations. One can easily see, that the set of directly reachable *GBK*-s from an operation u is union of the directly reachable *GBK*-s of the children of u. Additionally, by Lemma 2.2.3, if some *GBK* a is directly reachable from more children, edges on each of these paths are completely disjoint. Hence, if we would like to separate the vertex from the set of *GBK*-s, we need to either separate each of its children from its directly reachable *GBK*-s, or separate this vertex from its children. Because we are looking for a Min-cut, we choose whichever is smaller, and thus we propose almost the same recursion as in Lemma 2.2.9, but with initial values adjusted for more *GBK*-s:

$$mc(u) = min(out(u), \sum_{(u,v)\in E} mc(v)), \text{ where } \forall a \in M : \forall (u,a) \in E : mc(u) = out(u)$$

If we compute this for every *GBK* in the plan, then, by Corollary 2.3.0.1, all *GBK*-s are separated from their reachable *GBK*-s. In other words, every pair (a, b) of *dependent GBK* operations is separated (if they were not, there would be a path from a to b, and a would not be separated from (directly) reachable a - a contradiction).

The Map and Reduce phases for each *GBK a* (fusion blocks) can then be reconstructed in the following fashion:

- **Map** - traverse the edges in *opposite* direction starting from a and avoid passing through any vertex which is in the global cut. All encountered vertices are part of the Map phase.
- **Reduce** - traverse the edges starting from a until any of the vertices in cut is reached. All encountered vertices are part of the Reduce phase.
- **Fusions** - vertices of the same type in the same phase of one fusion block can be merged into operation with more incoming and outgoing streams. These operations are then subjects to further merging by fusions into a single (or more - per input stream) operations.

Complexity

Consider a vertex u in the normalized graph. This vertex is touched when the algorithm travels into it by its incoming or one of its outgoing edges. As all vertices have only one incoming edge, there exists (at most) one *GBK* operation, from where u is reachable. Hence, the algorithm touches u only on at most one of the runs (computations per *GBK*). As similar as this algorithm is to the previous one used for the simplistic case, this also touches each edge at most twice. Hence, if we run the algorithm for every *GBK* in the graph, every edge is touched at most twice, and hence the algorithm has running time of $O(|E|)$.

The proposed algorithm has potentially exponential time and space complexity, but it finds the optimal partitioning for each *GBK*. In the next section, we will propose another realistic model with relaxed constraints, and strive to find algorithm with better running time for this special case.

2.3.3 Relaxed model

Until this moment, we have supposed that each operation can behave differently on each of its inputs (it may increase the data in size by different coefficients). Although this is possible in FlumeJava, it is not very common. Mostly, the streams are treated as one, and the *dataflow coefficient* is the same for all these inputs. Hence, we will now study this model.

Once again, as we have relaxed the *dataflow coefficient*, we have to redefine it slightly. In this model, as we consider it, the following holds: $\forall u \in V, \exists c \in R^+ : \forall (v,u) \in E : df(v,u) =$

c. Hence, we will use the short form of $df(u)$ in this section, which simply denotes how much does the data increase in size anytime it passes through vertex u.

Additionally, we set the $df(Output) = 0$, as it really has no outgoing edges, and such definition is helpful for algorithm proposed in next section. The amount of data outputted can be calculated by summing the weights of edges incoming to the output node.

We could use the *normalization* trick even in this model, as it is only a special case of the previous one, but it will not be any more efficient. Rather, we shall study what exactly happens during the *normalization*.

When the algorithm reaches a vertex with n incoming edges, it wants to make n copies of the subtrees rooted in this vertex and assign one to each of its parents. However, when the algorithm later tries to find the Min-cut, it computes the cut for all of these copies separately, even though they are identical.

2.4 Polynomial time algorithm for optimal partitioning

In this section we will exploit the fact, that many computations are redundant when applying the *normalization* method. We will introduce a polynomial time algorithm computing the optimal partitioning for the relaxed model, and later generalize it for the previous models.

2.4.1 Relaxed model

Firstly, we will analyze and propose an algorithm working for the relaxed model introduced in the previous section. The main observation allowing this optimization is summarized in the next Lemma.

Lemma 2.4.1. *Consider the optimal cut and a vertex u in normalized graph of the relaxed model. If the subtree rooted in u is cut, then every copy of subtree rooted in u is cut in the same (corresponding) vertices.*

Proof. As they are all identical copies, the only difference between them is the input data for U. Suppose that the input data is of size d. The size of outgoing data from any vertex in the subtree can be computed by multiplying the df coefficients on the path and finally by the size of incoming data, d. The price of any cut of vertex U is the sum of outgoing data from all vertices in the cut, hence the expression can be modified into the form $d \cdot S(U)$, where $S(U)$ is an expression not dependent on d. Notice that $out(U) = d \cdot df(U)$. Then, by these definitions and the algorithm:

$$mc(U) = d \cdot S(U) = d \cdot min(d \cdot df(U), \sum_{(U,V)\in E} d \cdot df(U) \cdot S(V))$$

$$\Downarrow$$

$$S(U) = min(df(U), \sum_{(U,V)\in E} df(U) \cdot S(V))$$

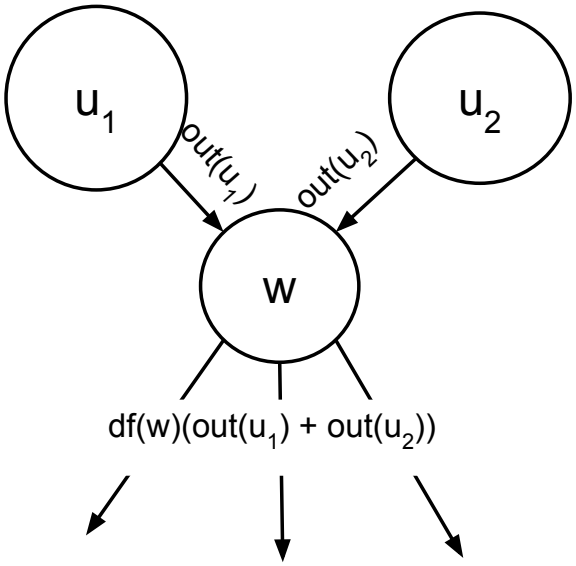


Figure 2.12: Vertex in relaxed model

Hence, the decision on cutting the vertex or its children does not depend on the incoming data, d. □

According to Lemma 2.4.1, the algorithm does not need to compute the cut for each of the copies, as the cut does not depend on the incoming data. Rather, for each vertex u, it can remember how much the cut would cost if the incoming data to u were of size 1 (this is exactly the meaning of the function S in the proof of Lemma 2.4.1).

Definition 2.4.1 (Subtree dataflow coefficient)**.** *Consider a vertex u and a subtree rooted in u in the relaxed model. We define function $S(u)$ as the ratio between the weight of optimal partitioning between u and any reachable GBK, and the size of data incoming to u.*

Corollary 2.4.1.1. *The function S can be computed recursively by:*

$$S(u) = min(df(u), \sum_{(u,w)\in E} df(u) \cdot S(w))$$

where

$$\forall a \in M : \forall (u,a) \in E : S(u) = df(u)$$

$$S(Output) = 0$$

Note that all the data passing through edges incoming to the *Output* vertex will have to be written to the shared storage. Those edges should have no effect on finding the Min-cut,

hence we set the $S(Output) = 0$.

We will now propose an algorithm for finding the cost of the optimal partitioning, along with the vertices in cut. The algorithm will only be able to process plans, where the Output is not directly reachable from Input. This case will have to be handled differently, which we will discuss later. Additionally, we will define the term *supertree*, as we will be using it in the algorithm.

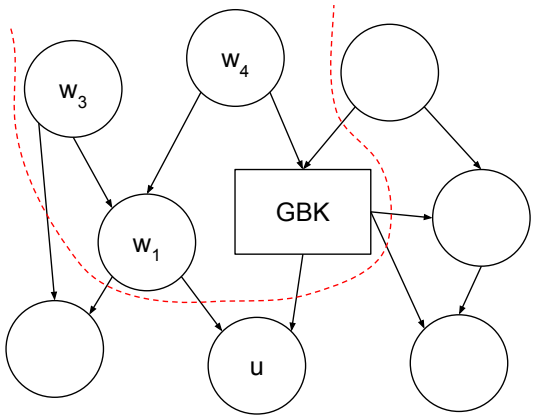


Figure 2.13: Supertree of u

Definition 2.4.2 (Supertree)**.** *Let u be a vertex in a plan. Let $H = \{w \mid w \neq u \wedge u$ is directly reachable from $w\}$. We will call the graph induced by H a supertree.*

Algorithm 2.4.1. *Consider a plan, where the Output is not directly reachable from Input. In each of the GBK a, we are interested in the value $S(a)$. We compute this recursively starting in every GBK. Additionally, each vertex will remember, which one of cutting itself or all of its children was optimal. Note that not all of vertices will be touched during computation of S. Before creating the MapReduce rounds, we precompute the following functions:*

- *$FIR(u)$ - denotes if u is directly reachable from Input (fromInputReachable)*
  - $FIR(u) = \bigvee_{(w,u)\in E} FIR(w)$
  - $\forall b \in M : FIR(b) = false, FIR(Input) = true$
- *$OR(w)$ - denotes if Output is reachable from w (outputReachable)*
  - $OR(w) = \bigvee_{(w,u)\in E} OR(u)$
  - $\forall b \in GBK : OR(b) = false, OR(Output) = true$
- *$C(w) \Leftrightarrow u$ was touched during computation of S (computed)*
- *$T(u)$ - denotes the choice the vertex made for minimal cut (traverse)*
  - *$T(u) \Leftrightarrow$ vertex u chose to cut its children*
- *$AT(u)$ - denotes if all vertices in supertree of u chose to cut children (allTraverse)*
  - $AT(u) = \bigwedge_{C(w)\wedge(w,u)\in E} (AT(w) \wedge T(w))$
  - $\forall b \in M : AT(b) = true$

- *margin*(u) *- denotes margin nodes (part of cut)*
  - $margin(u) = \neg T(u) \wedge ((\bigvee_{C(w) \wedge (w,u) \in E} (AT(w) \wedge T(w))) \vee (u \in M))$

Then, the vertices for Map and Reduce phases for each GBK operation a can be found in the following fashion:

- ***Reduce*** *- build graph by traversing vertices reachable from the GBK. Only traverse into children of a vertex u if* $T(u) = true$*. Copy every traversed vertex into a new graph. Output of the vertices which were cut should be written to shared storage. Define function* $outputs(u)$*:*

  1. *add u if u was not yet added, else return.*
  2. $\forall (u, w) \in E$: *if* $OR(w)$ *then* $outputs(w)$

  Run $outputs(a)$*, while adding these vertices into a separate graph. Eventually, merge this graph with the former.*

- ***Map*** *- Define recursive functions* $add(u)$*:*

  1. *add u if u was not yet added, else return.*
  2. $\forall (w, u) \in E$: *if* $\neg AT(w)$ *then* $add(w)$

  $marginInputs(u)$*:*

  1. $\forall (w, u) \in E$: *if* $margin(w)$*, expect input from copy of w for all children of w*

  And $inputs(u)$*:*

  1. *add u if u was not yet added, else return.*
  2. $\forall (w, u) \in E$: *if* $FIR(w)$ *then* $inputs(w)$

  Lastly, run $add(a)$*, and copy all added vertices to the Map phase of a. Then, run* $marginInputs(a)$ *and gather inputs for the vertices. Eventually, run* $inputs(a)$ *while adding the vertices into a separate graph - merge the graphs afterwards.*

In order to prove the time complexity and correctness of the algorithm, we will note and prove a few properties the functions defined in the algorithm have. Also, any node for which $margin(u)$ defined in the algorithm is *true* will be called **margin node**. The next few Lemmas will all be bound to Algorithm 2.4.1.

It is worth to mention that if node u is in cut, then $T(u)$ defined in Algorithm 2.4.1 must be *false* - otherwise the cut would be somewhere in each of its children. Moreover, note that in optimal partitioning, every path between two directly reachable *GBK* operations a and v has a continuous path for Reduce phase of a ending after reaching the first vertex with $T(u) = true$, inclusive. In a valid partitioning, the Map phase for b should contain all vertices on this path starting from u, exclusive.

Lemma 2.4.2. *Every **margin node** is in at least one cut in Reduce phase in the resulting plan.*

Proof.

Let U be margin node. By definition, $T(u) = false$. Additionally, either u is a *GBK* operation, or there exists a parent w of u, such that $T(w) \wedge AT(w)$. If u is a *GBK* operation, then it is part of trivial Reduce phase (with identity Reducers). If it is the latter case, then on every path between some *GBK* and directly reachable w, the value of function T on any of those vertices is *true*. Hence, u is first with $T(u) = false$ on any continuing path, and so it is part of the cut in Reduce phase for these paths.

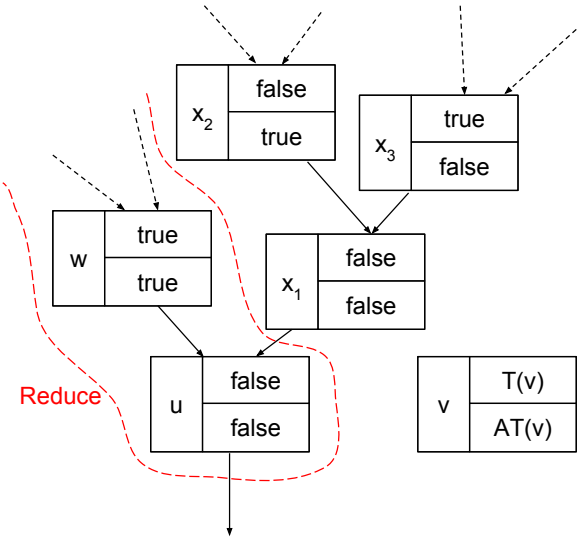


Figure 2.14: Margin node u in reduce phase

□

Lemma 2.4.3. *If $AT(u) = false$, then supertree of u contains a margin node.*

Proof. We will prove this by induction on the length k of the longest path from some w, where $AT(w) = true$, to u.

- **Base** - $k = 1$: because all of the paths start in some *GBK*, $\forall a \in M : AT(a) = true$, all of the parents w of u must have $AT(w) = true$. Additionally, as $AT(u) = false$, there must exist a parent w, such that $T(w) = false$ - this is the margin node.
- **Step** - $k+1$: at least one the parents w of w must dissatisfy the condition $AT(w) \wedge T(w)$. If $AT(w) = true$, then $T(w) = false$ and w is margin node. Otherwise the margin node is in supertree of W by induction.

□

Lemma 2.4.4. *For every u reached during construction of Map phase for given GBK a: $\neg AT(u) \Leftrightarrow u$ is in the Map phase of a.*

Proof. We will divide this proof into two cases, by evaluation of $AT(U)$:

- If $AT(u)$ is *true*, then all of the nodes in the supertree of u decided to cut their children rather then themselves. Hence, none of the vertices in the supertree are in cut. Additionally, u is not part of any Map phase - every path between two directly reachable *GBK* operations, which is passing through node u, has to be cut either in u or in successors of u, as the $AT(u)$ is *true*.
- If $AT(u)$ is *false*, then by Lemma 2.4.3, supertree of u contains a margin node w. Hence, by Lemma 2.4.2, there exists a path between w and u, such that w is in cut of some Reduce phase, and, therefore, all other vertices on this path from w (including u) are part of Map phase.

□

By Lemma 2.4.4, the function *add* defined in 2.4.1 will add all and only those vertices to the Map phase, which belong there. As for the *inputs* function - by Lemma 2.4.2, every **margin node** is part of some Reduce phase, hence the nodes in Map phase have to expect inputs from some copy of this **margin node**.

We will now analyze the time complexity of this algorithm.

Theorem 2.4.5. *Consider a plan, where Output is not directly reachable from Input. Let M be the number of GBK operations, V the set of vertices and E the set of edges in the plan. Then, the Algorithm 2.4.1 computes MapReduce rounds for every GBK, such that amount of data written to common storage is minimal, in $O((M+1)\cdot|E|)$ expected time.*

Proof. We will divide the algorithm into two parts - precomputation and round construction:

- **precomputation -** while pre-computing the $S(u)$ for every vertex u, we can avoid computing the same value twice by using *memoization*. After computing the $S(u)$, we remember this value in a Hashmap along with values $T(u)$ and $C(u)$ while using the vertex as a key. As the $S(u)$ is oblivious of the *GBK* which started the computation, the Hashmap is shared for all *GBK*-s. Every edge is traversed at most twice, and the lookup & insert into Hashmap is $O(1)$ expected, hence the pre-computation of S takes $O(|E|)$ expected. Similarly, all other functions $FIR, OR, AT, margin$ are also oblivious on the *GBK* which started the computation, and can be shared. The computations of these functions can be also done in $O(|E|)$ expected time.
- **round construction -** for each *GBK* we run a separate, independent process. All edges in construction of the reduce phase are traversed at most twice, and each vertex

is copied to the new graph exactly once. While constructing the Map phase, each edge is also traversed at most twice in computation of function *add*. The check if the node was already added can be done in $O(1)$ expected by keeping the added nodes in a Hashmap. The added vertex is copied to the new graph only once, and each of these vertices is traversed only once in function $inputs$. As the lookup for function AT and *margin* is now $O(1)$ expected, the construction of Map and Reduce round for a *GBK* is in $O(|E|)$ time, hence $O((M+1)\cdot|E|)$ for all *GBK*-s together.

□

Although Algorithm 2.4.1 does not count with plans, where the *Output* is directly reachable from *Input*, these cases can be easily solved.

If the *Output* is directly reachable from *Input*, then it means there exists a path between *Input* and *Output* not passing through any *GBK* operation. Hence, we need to create a separate MapReduce round with all these operations in Map phase, and trivial Shuffle and Reduce phases. To achieve this, we will make use of the FIR, and $inputs$ function defined in the algorithm. The graph for Map phase can be created very simply, just by running $inputs(Output)$. Then, we have created a separate MapReduce round which is independent of any other, and does not change the total Network usage in any direction (because the data would have to be written to Output anyways).

2.4.2 General model

The difference from the **relaxed model** is only that the *dataflow coefficient* also depends on the input stream the data came from. Hence, we can't use exactly the same approach as in the previous section, because the amount of data flowing out of some node can depend on the input stream.

In order save some computations again, we need to lift the dependency of function S on its inputs. Therefore, we will use a slightly different approach, and redefine the meaning of S.

Definition 2.4.3 (Subtree dataflow coefficient for general model)**.** *Consider a vertex u and a subtree rooted in u in the relaxed model. We define function $S_G(u)$ as the ratio between the weight of the optimal partitioning between u and any reachable GBK, and the size of data* ***outgoing*** *from u.*

Corollary 2.4.5.1. *The function S_G can be computed recursively by:*

$$S_G(u) = min(1, \sum_{(u,w)\in E} df(u,w) \cdot S_G(w))$$
$$where$$
$$\forall a \in M : \forall (u,a) \in E : S_G(u) = 1$$
$$S(Output) = 0$$

The df function used in Corollary 2.4.5.1 is the *dataflow coefficient* function for **general model**. Also, note that $S_G(u)$ does not depend on input stream to u, hence it is identical for all input streams and we can use the same approach as in Algorithm 2.4.1.

2.4.3 Optimizing the reads

Until now, we have only discussed about making the plan optimal in terms of the data written. The Algorithm 2.4.1 creates a plan, where for each *GBK* operation exists one MapReduce, and each of the MapReduces has its own copies of every operation it uses.

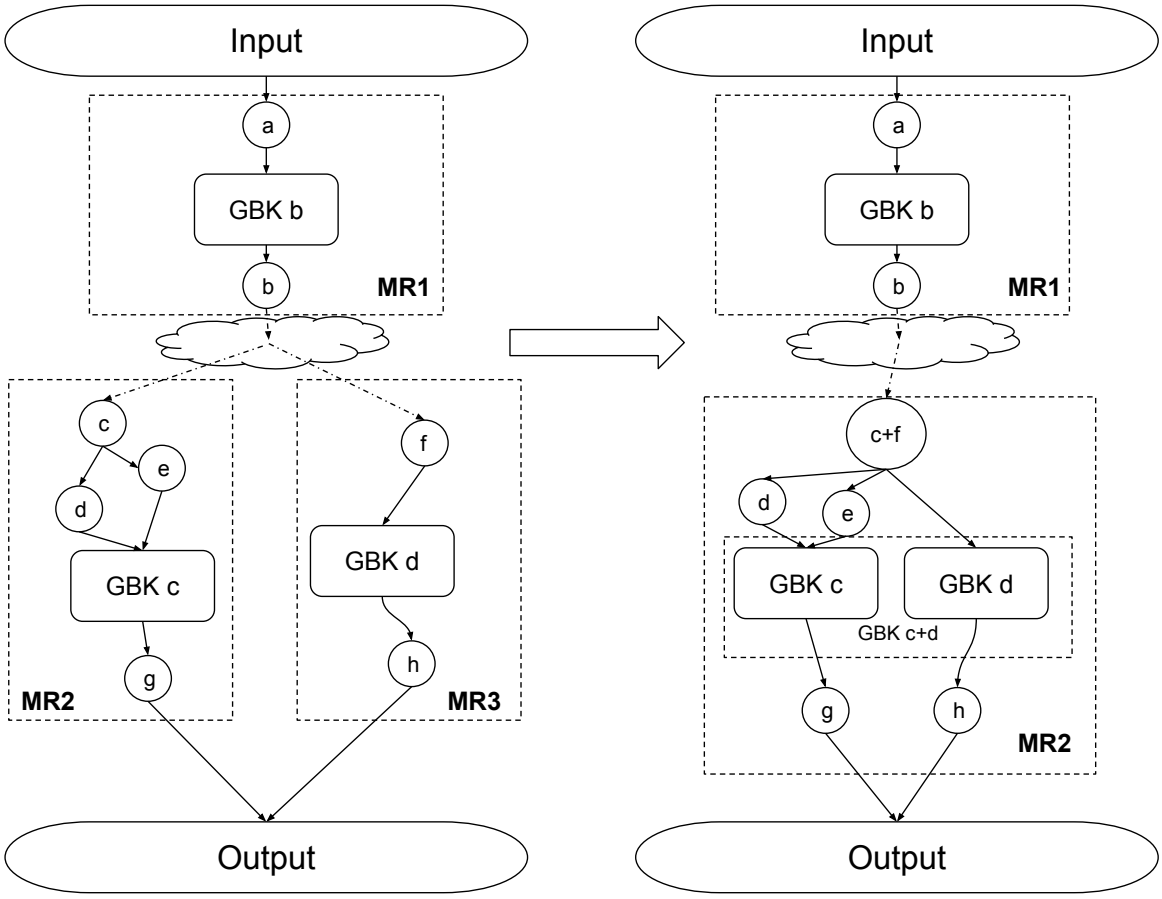


Figure 2.15: Merging independent MapReduce rounds

However, if two *GBK* operations are *independent* and they share some of their inputs are identical, the *GBK* operations can be executed in the same Shuffle phase by using channels, similarly as in *MSCR* optimization shown in Figure 1.3. Then, the operations taking the

same input can be merged into one using the *sibling fusion* introduced in section 1.2.2, thus decreasing the amount of data read from common storage.

Still, there might be many independent *GBK* operations, hence many different possibilities for merging the MapReduces. We can enumerate all the possibilities, calculate the amount of data read for each, and return the minimal.

If we think only about the *GBK* operations and their dependencies, we have to decide, which groups of operations should be performed in a single MapReduce round. For this purpose, we will distribute these *GBK*-s into several *buckets*, and execute all the *GBK*-s in one bucket in the same MapReduce round. Eventually, we will order these buckets and execute the pipeline by running each bucket one after another. Of course, no two *dependent GBK* operations can be in the same bucket, more specifically, if *GBK a* is *dependent* on *GBK b*, *a* has to be in a bucket which is run after the bucket *b* is in.

Consider the *GBK* dependency graph, where the *GBK* operations are vertices and there is an edge from *GBK a* to *GBK b* if *b* is *dependent* on *a*. Since we would like to run the *buckets* one after another (none in parallel), we would like the number of buckets be as low as possible. Because an operation has to be run only after every operation it depends on was run, we need at least as many buckets, as is the number of vertices on the longest path in this *GBK* dependency graph. Firstly, we will propose an algorithm for finding every valid distribution of these *GBK*-s into *buckets*.

Algorithm 2.4.2. *Consider the GBK dependency graph* $G = (V, E)$ *of an execution plan. For every node u in the plan, compute its height (longest path starting in u). Let* $m = max\{height(u) \mid u \in V\}$. *Create m buckets labeled* $0, \ldots, m-1$. *Define function:*

- *place(H, Buckets)*

    1. *If H is empty, return Buckets.*
    2. *Let u be a vertex in H which has no parents in H.*
    3. *If u has no parents in G, let low* $= 0$
       If u has parents in G, let low $= max\{i \mid (v, u) \in E \wedge v \in Buckets[i]\}$
    4. *Let high* $= m - height(u)$
    5. $\forall i, low < i \leq high$: *output place(H with u removed, B with u added to B[i]) as a solution*

Finally, let Buckets be an empty array of length m, and return place(G, Buckets)

Algorithm 2.4.2 returns all possible valid distributions of the *GBK*-s into buckets, in time asymptotic to the number of these solutions.

Now we will propose an algorithm for finding a plan, such that the amount of data read and written to common storage is optimal.

Algorithm 2.4.3. *Consider an execution plan. Find every valid distribution of GBK-s into buckets using the Algorithm 2.4.2. For each of these distributions, compute the amount of data read in the following manner:*

- *Firstly, find all the Map and Reduce phases for optimal writes using Algorithm 2.4.1*
- *Define readsInBucket(Buckets, i)*
  1. $\forall g \in Buckets[i]$*: find its Map phase and for each vertex in its Map phase add all outside vertices it expects input from into S*
  2. *Remove duplicates in S*
  3. *return* $\sum_{u \in S} out(u)$*, where* $out(u)$ *is the expected outgoing flow from operation u.*
- *Eventually, return* $\sum_{0 \leq i < m} readsInBucket(Buckets, i)$

Complexity

The complexity of this algorithm is not easily determined, as it highly depends on the structure of the graph. However, given a distribution of *GBK*-s into the *buckets*, the algorithm crawls all the operations in the Map phase of each *GBK* once. Since each of these *GBK* operations can have at most $|V|$ vertices in its Map phase, the algorithm has running time of $O(M \cdot |V|)$. Thus, if there are s solutions for distribution of *GBK*-s into *buckets*, the complexity of Algorithm 2.4.3 is $O(s \cdot M \cdot |V|)$.

For some graphs, however, the amount of valid distributions can be as large as $\frac{M}{2}^{\frac{M}{2}}$ (dependency graph containing one path $\frac{M}{2}$ long, and another $\frac{M}{2}$ independent *GBK*-s).

2.5 Constraints

For these optimizations to work, we require the knowledge of all the *dataflow coefficients*. Nevertheless, the companies often run the same plans with structurally similar data many times, and some machine learning could be used to determine these coefficients quite accurately.

Besides that, we suppose the data is *homogeneous* - every computation of a function on an input takes approximately the same time. Hence, we suppose that all the workers in the Map and Reduce phases are finished approximately at the same time, because of the load balancing the MapReduce does. However, if the data is not *homogeneous*, it might happen that some worker in Reduce phase of one MapReduce would get a slow block of data, and the other workers would have to wait for him to synchronize after they are done. If we knew that the operation the worker computed was getting a large fraction of small datasets, we could move the operation into the next Map phase and, possibly, there would be less time spent by workers synchronizing, hence the data would be processed faster. On the other hand, the network load is optimal the way the proposed algorithm has computed it, and the homogeneousness of data has no effect on the solution.

Chapter 3

Experimental results

In this chapter, we will analyze the solutions proposed for optimization of network usage on various execution plans. However, the actual implementation of FlumeJava's optimizer is not known to public, and the optimizer in Apache Crunch's implementation is not discussed, nor can it be easily analyzed.

Nevertheless, Apache Crunch allows outputting the graph in *.dot* format [9], and user may create an image representation using another tool for transforming *.dot* into image, such as *Graphviz* - `http://www.graphviz.org/`. This is useful for the programmer to see, what the created execution plan looks like, and how do the MapReduce rounds look. However, it is not very suitable for comparing the plans with the proposed algorithm, because Crunch gives *ids* to the vertices in some order of execution, and this order may be differ per solution if there are independent operations. Additionally, it gives no information whatsoever about the expected network load. Hence, as we can only hypothesize the actual implementation, we will compare the proposed solution to a naive heuristic solution, and show that in some plans, the network load can change drastically if the partitioning is not given enough caution.

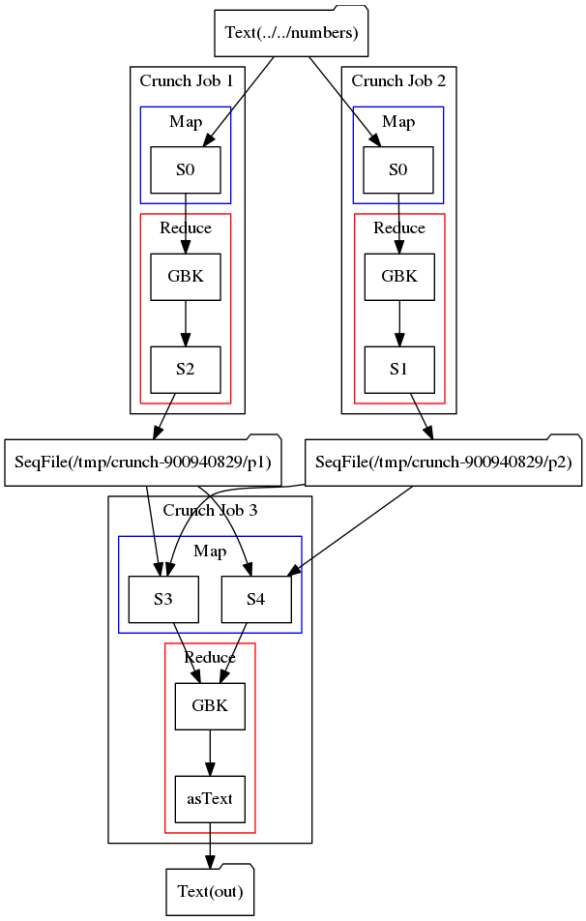


Figure 3.1: Entwined graph resulted in cloned Map and GBK phase, computed as 2 MapReduces

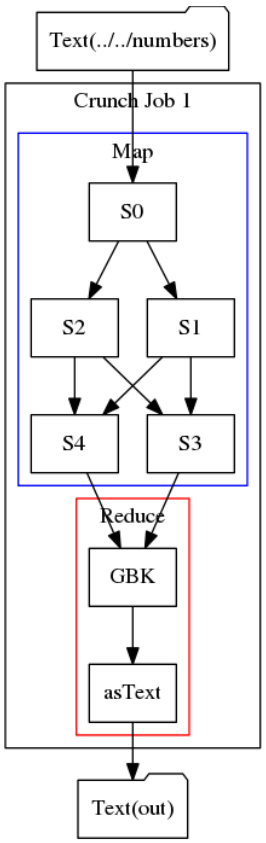


Figure 3.2: Single MapReduce round

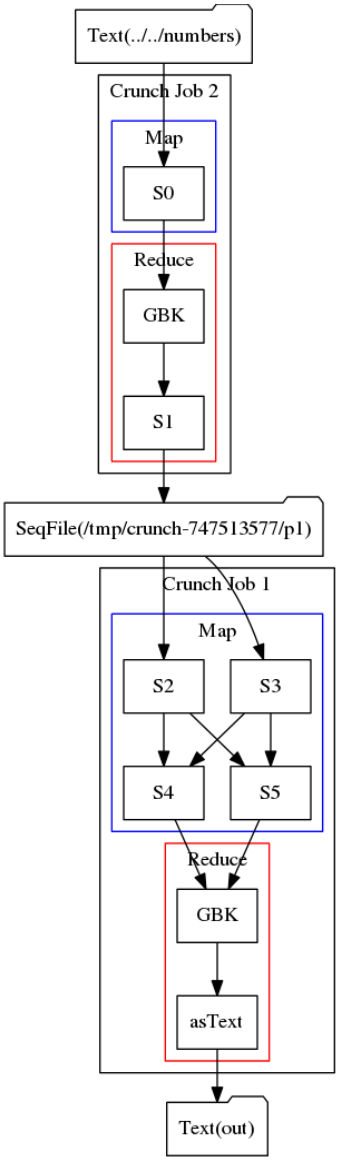


Figure 3.3: Partitioning of an almost entwined graph

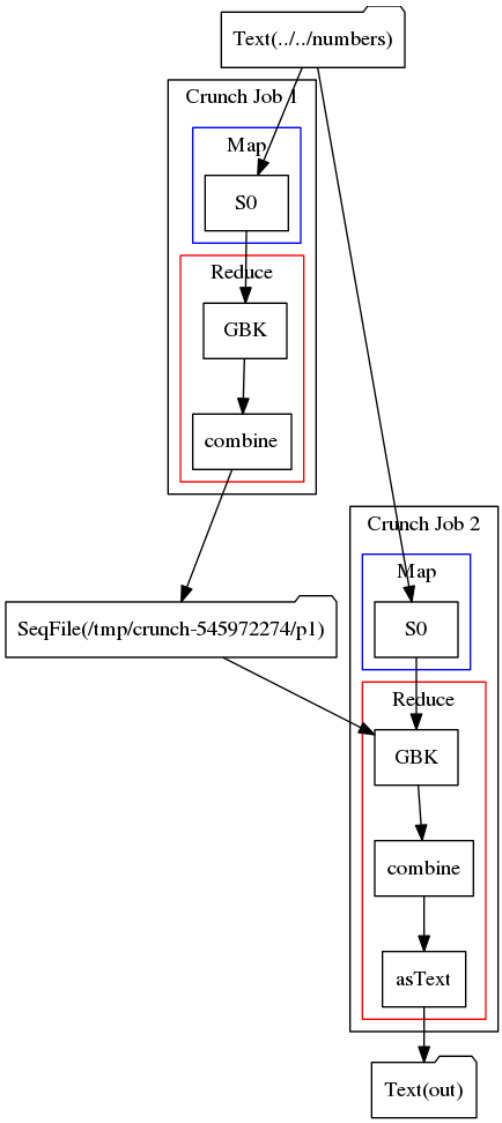


Figure 3.4: Dependent MapReduce rounds

Many generated plans had the *GBK*-s in their cuts. Therefore, in order to compare the optimal cut to just some cut, we will introduce a naive algorithm cutting the graph either before or after a *GBK*.

Algorithm 3.0.1 (Naive heuristic)**.** *Suppose an execution plan. For every GBK a in the plan, find all of directly reachable GBK-s. Then, cut either the edges outgoing from GBK a, or all the edges incoming to these GBK-s, which are on path from a. The Map and Reduce phases can be built similarly as in Algorithm 2.4.1.*

3.1 Library for generating plans

Basically, we would like to generate directed acyclic graphs (DAG) with 4 types of vertices - *GBK*, *PD*, *Input* and *Output*, with *Input* having no incoming and *Output* no outgoing edges. We will create a library for generating pseudo-random DAG-s with these properties, and also a special type of graphs, *entwined*, shown on Figure 2.7.

3.1.1 Pseudo-random directed acyclic graphs

Firsly, we omit the *dataflow coefficients*, and focus on the vertices and edges in the graph. We could generate the graphs based on these parameters:

- **Number of vertices** - total number of *GBK* and *PD* operations (as there are always one of each *Input* and *Output* vertices, we will omit those)
- **Number of GBK-s** - number of *GBK* operations. The more *GBK* operations are in graph, the less options for cuts are available.
- **Edge probability** - probability, that edge between given two vertices exists.

Additionally, we would not like the graphs to be completely random. We would like the graphs to be similar to execution plans. For example, it is senseless to send output of one *GBK* into another, as the data is already grouped. Even if such plan was constructed, where output of *GBK a* is sent to *GBK b*, the optimizer could make *b* take all the input *a* has, and remove the edge from *a* to *b*. Hence, we will generate only graphs, where no edges between *GBK*-s exist.

Datafow coefficients - after generating the graph, we will give each edge its coefficient. There may exist plans where the coefficients are of mostly low or mostly high factor (either increasing or decreasing), some may have mostly increasing or decreasing coefficients, and some may be completely random. We would like to study all of these types, and thus we generate the *dataflow coefficient* for an edge based on these parameters:

- **Highest factor (high)** - bottom threshold of factor from interval $(0, 1]$, by which the coefficient can be either decreasing or increasing
- **Lowest factor (low)** - upper threshold of factor from interval $[low, 1]$, by which the coefficient can be either decreasing or increasing
- **Decreasing probability (dprob)** - probability that the factor is decreasing

Using these parameters, we can generate the *dataflow coefficient* by choosing a random number $f \in [high, low]$ and random $d \in [0, 1]$, where the output is either f if $d \leq dprob$ or $1/f$ otherwise.

Test cases

We have tested graphs with various parameters, up to 30 vertices and selected a few, which show radical improvements in network load. All statistics per setting are taken from 1000 generated graphs with given parameters. The statistics we have been studying for each setting are:

- **Average improvement ratio (Avg.)** - percentual improvement compared to the naive algorithm, computed as an average through all runs
- **Improved cases (Cases)** - percentual ratio of graphs, where the optimal solution was different from the naive (graphs which were improved by more than 0%)
- **Maximum improvement (Max)** - maximum percentual improvement by the optimal algorithm encountered in the 1000 generated graphs with given setting

Vertices	GBK-s	Edge prob.	High	Low	dprob	Avg.	Cases	Max
28	2	0.6	0.01	0.01	0.9	422.81%	30.3%	4801.48%
26	5	0.2	0.01	0.01	0.32	310.35%	10.1%	9900%
25	3	0.3	0.01	0.04	0.7	307.32%	20%	58526.3%
28	5	0.2	0.01	0.04	0.64	206.64%	23.1%	28858.2%
30	7	0.5	0.01	0.01	0.32	194.39%	5.3%	9899.76%
28	3	0.4	0.01	0.04	0.64	194.27%	18.1%	6586.39%
28	4	0.2	0.04	0.04	0.7	172.63%	20.1%	19921.8%
22	2	0.3	0.04	0.04	0.7	158.56%	14%	2400%
19	2	0.5	0.04	0.16	0.9	156.02%	15.9%	4241%
20	2	0.3	0.04	0.08	0.64	107.77%	11.8%	7738.82%
24	2	0.3	0.01	0.04	0.16	86.02%	2.8%	8384.64%
19	4	0.4	0.01	0.08	0.8	77.55%	16.1%	6563.43%
30	2	0.2	0.08	0.16	0.64	64.66%	13.6%	5447.26%
25	4	0.3	0.04	0.08	0.8	58.05%	24.4%	1756%
26	4	0.2	0.02	0.16	0.64	44.99%	17.1%	2564.54%
30	2	0.2	0.08	0.16	0.64	43.12%	11.6%	1142.98%
28	3	0.3	0.04	0.32	0.7	24.71%	22.9%	670.13%

Table 3.1: Tests for DAG-s with various settings, sorted by average improvement rate (Avg.)

The tests show that if the *dataflow coefficients* are of a high factor, the optimal algorithm tends to rapidly improve the plan, compared to a naive solution. Also, the *maximum improvements* show that in some cases, the optimal solution can be better by a huge factor, than some random (or *slightly sophisticated*) cut. Although the *naive* algorithm finds the optimal solution in most cases, the number of *cases*, where the optimal algorithm is better is for some types of graphs quite high.

3.1.2 Pseudo-random special graphs

In this section, we will study special *entwined* graphs similar to the graph shown on figure 2.7, as these graphs show even more radical results.

Entwined graph consists of two *GBK* operations, one directly connected to *Input* and the other to *Output*. Between these *GBK*-s are numerous levels, each level consisting of the same amount of *PD* operations, where all outputs from every *PD* on level k are used as input for each of the *PD*-s on level $k + 1$.

Similarly, we will generate random graphs based on a set of parameters. The parameters for *dataflow coefficients* stay the same as in previous section, and the graph itself will be generated based on the following parameters:

- **Height** - number of levels
- **Width** - number of *PD* operations in each of the levels

Note that the number of vertices (except *Input* and *Output*) in the generated graph is $height * width + 2$.

We have also tested these graphs with various parameters, up to 60 vertices, and we selected a few, which show tremendous improvements in network load. Again, the statistics are taken from 1000 generated graphs with given parameters.

Height	Width	High	Low	dprob	Avg.	Cases	Max
20	2	0.01	0.01	0.9	$1.09 \times 10^9\%$	62.5%	$9.91 \times 10^{11}\%$
17	2	0.02	0.02	0.64	$1.63 \times 10^6\%$	48.3%	$6.47 \times 10^8\%$
15	2	0.01	0.04	0.64	$1.59 \times 10^6\%$	52.4%	$1.48 \times 10^9\%$
19	2	0.02	0.16	0.8	$4.11 \times 10^5\%$	82.6%	$2.81 \times 10^8\%$
9	2	0.02	0.02	0.64	79300%	46.2%	$3.75 \times 10^7\%$
18	3	0.08	0.08	0.9	20394%	83.9%	$2.63 \times 10^6\%$
6	2	0.02	0.02	0.64	20316%	42.3%	$3.06 \times 10^6\%$
10	4	0.01	0.04	0.9	19520%	76%	$3.593 \times 10^6\%$
9	3	0.04	0.16	0.9	1643%	82.8%	$1.45 \times 10^5\%$
15	2	0.02	0.6	0.8	347%	69.9%	12790%

Table 3.2: Tests for entwined graphs with various settings

The naive algorithm would cut this graph only after the first *GBK* or before the second, whichever yields smaller cut. However, the *dataflow coefficients* could be mostly decreasing until some point in the graph and from that point mostly increasing (see Figure 3.5 . Then, the optimal dataflow might drastically lower in that point, compared to outgoing flow from *GBK a* or incoming to *GBK b*.

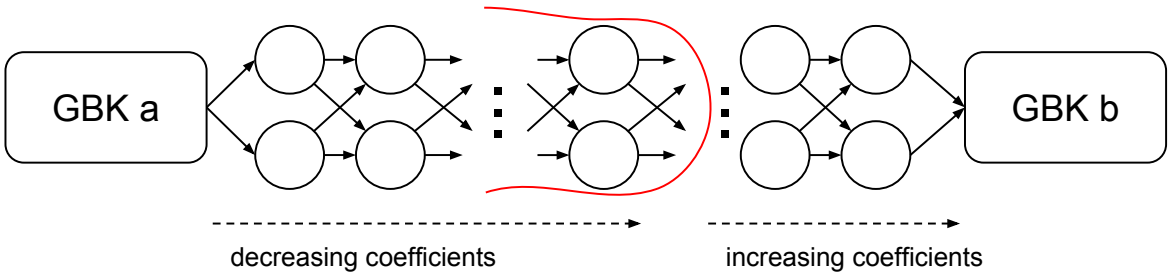


Figure 3.5: Entwined graph with radically better optimal than naive cut

The results also show that the number of cases where the optimal algorithm is better is quite high. Hence, the phenomena of decreasing and increasing coefficients is very likely, the only difference is the extent. The higher the factor is, the higher probability of some greatly improved graph. Indeed, if we found a graph which has the optimal cut somewhere in the middle, l levels from the start, and then generated the same graph but with k times greater factors (decreasing coefficients would be k times smaller and increasing k higher), this graph would have the optimal cut approximately k^l times smaller than before. Thus, by increasing the factors the improvements can get any high even for small graphs.

Conclusion

Execution plans in FlumeJava tend to have many different mappings to MapReduce pipelines. We have analyzed the problem of finding the optimal in terms of network usage, and found a proper and viable abstraction into a more transparent graph problem, which is not very far from the reality. We have proposed several approaches to solving this graph and, eventually, found a polynomial solution. Hence, the partitioning with optimal network usage can be found in almost trivial time, provided the fact that the graphs are usually not too large (< 100 vertices) in real life computations.

Additionally, we have shown that if the network load during partitioning is not given much attention, the results given by some trivial algorithm can in some cases (even up to $\approx 30\%$ for random graphs) deviate by a tremendously large factor from the optimal solution (as shown on Table 3.1). Hence, finding a partitioning with optimal network usage can lead to great optimizations, when compared to the case, where this aspect is neglected.

Future work - as mentioned in section discussing the constraints (2.5) we assume throughout this thesis, we assume that the data is *homogeneous*. This, however, may not be true in the real life computations, and it may make some operations be waiting for synchronization, thus making the pipeline running time longer. To consider this aspect, one could propose and study a whole different model, which, aside from the network load, also takes into account the *non-homogeneousness* of data.

Bibliography

[1] Apache. Apache crunch user guide. `http://crunch.apache.org/user-guide.html`.

[2] Apache. Users of hadoop. `http://wiki.apache.org/hadoop/PoweredBy`.

[3] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375. ACM, 2010.

[4] Shuchi Chawla, Robert Krauthgamer, Ravi Kumar, Yuval Rabani, and D. Sivakumar. On the hardness of approximating multicut and sparsest-cut. *Computational Complexity*, 15(2):94–114, 2006.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[6] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 938–948. SIAM, 2010.

[7] N. J. A. Sloane. The online encyclopedia of integer sequences. "`https://oeis.org/`".

[8] Elmar Teufl and Stephan G. Wagner. Enumeration problems for classes of self-similar graphs. *J. Comb. Theory, Ser. A*, 114(7):1254–1277, 2007.

[9] Ben Watson. Apache crunch toolkit #2: Viewing pipeline execution plan visualisations. "`http://www.hadoopathome.co.uk/Apache-Crunch-Toolkit-2/`".

[10] Honghua Yang and D. F. Wong. Efficient network flow based min-cut balanced partitioning. In Jochen A. G. Jess and Richard L. Rudell, editors, *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1994, San Jose, California, USA, November 6-10, 1994*, pages 50–55. IEEE Computer Society / ACM, 1994.