Fault tolerance in Parallel Data Processing Systems

vorgelegt von
Dipl.-Inform. Mareike Ruth Höger

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
- Dr. rer. nat.-

Promotionaausschuss:

Vorsitzender:	Prof. Dr. Manfred Hauswirth
Gutachter:	Prof. Dr. Odej Kao
	Prof. Dr. Ivona Brandic
	Prof. Dr. Volker Markl

Tag der wissenschaftlichen Aussprache: 30.10.2018

Berlin, 2019

Abstract

These days data is collected any time and everywhere. The number of devices we are using every day is steadily growing. Most of those devices collect data about their usage and environment. That data is no longer gathered to answer a particular hypothesis. Instead, it is gathered to find patterns that could build a hypothesis. The collected data is often semi-structured, may stem from different sources, and is probably cluttered. The term *BigData* emerged for this kind of information.

Parallel data processing systems are designed to handle BigData. They work on a large number of parallel working nodes. The high number of nodes and the long runtime of jobs lead to a high failure probability. Existing fault tolerance strategies for parallel data processing systems usually handle faults with full restarts or work in a blocking manner. Either the systems do not consider faults at all, and restart the entire job if a fault occurs, or they save all intermediate data before they start the next task.

This thesis proposes better approach to fault tolerance in parallel data processing systems. The basis of the approach reduces restarts and works in a nonblocking manner. The introduced ephemeral materialization points hold intermediate data in memory while monitoring the running job. This monitoring enables the system to choose the sweet spots for materialization. At the same time, the materialization points allow the pipelining of data.

Based on this method the thesis introduces several continuous fault tolerance techniques. On the one hand, it covers data and software faults, which are not included by the typical retry methods. On the other hand, it covers further optimizations

for jobs with stateless tasks. Stateless tasks do not have to reprocess all their input to produce the same output, as they do not have to reach a certain state. The possibility to run a task at any point of the input stream offers the opportunity for further optimizations on the fault tolerance method. In this case it is possible to add additional nodes to the system during recovery or to skip parts of the input stream.

The evaluations of the approaches show that they offer a fast recovery with small runtime and disc space overhead in a failure free case.

Zusammenfassung

Heutzutage werden Daten überall und zu jeder Zeit gesammelt. Die Anzahl an Geräten die wir im alltäglichen Leben verwenden steigt immer weiter. Diese Geräte sammeln Daten über ihre Nutzung und Umgebung. Diese Daten werden nicht gesammelt um eine bestimmte Hypothese zu untermauern, sondern um Muster zu finden die eine Hypothese bilden können. Die gesammelten Daten sind oft semi-strukturiert, können aus verschiedenen Quellen stammen und sind möglicherweise mangelhaft. Der Begriff *BigData* hat sich für solche Informationen herausgebildet.

Parallele Datenverarbeitungs-Systeme wurden entwickelt um mit BigData zu arbeiten. Sie arbeiten mit einer Vielzahl von parallelen Arbeitsknoten. Die groSSe Anzahl an Maschinen und die typischerweise lange Verarbeitungszeit führt zu einer hohen Fehlerwahrscheinlichkeit. Existierende Fehlertoleranz Strategien für diese Systeme nutzen normalerweise komplette Neustarts oder arbeiten blockierendernd. Entweder können sie gar nicht mit Fehlern umgehen und starten den gesamten Job neu, oder sie speichern alle Zwischenergebnisse bevor der nächste Schritt gestartet wird.

Diese Dissertation hat die Absicht einen besseren Ansatz für Fehlertoleranz in parallelen Datenverarbeitungs-Systemen zu finden. Die Grundlage des Ansatzes vermeidet Neustarts und arbeitet in nicht blockierender Weise. Die vorgestellten ephemeral materialization points (Flüchtige Materialisierungspunkte) halten Daten im Speicher, während der Job untersucht wird. Diese Untersuchung ermöglicht es dem System die besten Punkte für die Materialisierung zu finden. Diese Materialisierung blockiert die Verarbeitung nicht.

Aufbauend auf dieser Methode stellt die Dissertation verschiedene Fehlertoleranz Mechanismen für parallele Datenverarbeitungs-Systeme vor. Auf der einen Seite behandelt es verschiedene Fehlertypen die in den üblichen Neustart Methoden nicht behandelt werden können, wie Daten- oder Software-Fehler.

Auf der anderen Seite betrachtet sie Optimierungen für Jobs mit zustandslosen Teilschritten. Die Zustandslosen Teilschritte müssen nicht die gesamten hereinkommenden Daten wieder verarbeiten, da sie keinen Zustand wieder herstellen müssen. Die Möglichkeit einen Teilschritt an jeder Stelle des hereinkommenden Datenstroms neu zu starten eröffnet die Möglichkeit für weitere Optimierungen. Das System kann während der Wiederherstellung zusätzliche Knoten zu dem Job hinzufügen oder Teile der hereinkommenden Daten auslassen.

Die Evaluationen der vorgestellten Methoden zeigen, dass sie eine schnelle Wiederherstellung bieten und gleichzeitig geringe Zusatzkosten in Bezug auf die Laufzeit und den Speicherverbrauch verursachen.

Acknowledgements

It has been a long road to finish this thesis. During that time a lot of people have supported and helped me to achieve this, and I want to take the time to thank them.

First I want to thank my advisor, Prof. Dr. Odej Kao who offered me the opportunity to work in his research group and supported and advised me during the last years. I am deeply grateful for the opportunity, and the many things I have learned from him in those years.

Over the last years, I worked with many colleagues and learned from all of them. I am thankful for all the discussions, meals and chats I had with all of them. However, I want to especially thank Dr. Alexander Stanik, Dr. Andreas Kliem, and Dr. Marc Körner who shared with me the moments of anxiety and excitement of research and non-research life.

Furthermore, I would like to thank my parents Angelika und Ingo, and my siblings. They supported me in all those years and had an open ear for my struggles at any time. I am fortunate to have a family like this. I am deeply grateful for their backing and their unconditional love.

Speaking of unconditional love: I have to thank my girls, Lena and Johanna who tried their best to understand that mom often had to sit behind a screen when she actually should be playing with them.

Finally, my endless gratitude goes to my husband, Christoph. He always listened to my ideas, helped me to get my head straight, took the time to think his way into the depth of parallel execution and even hunted some bugs with me. He raised me up from deep lows more than once, without him I would not have been able to finish this thesis. There are no words that could express my thankfulness for all your support. I am blessed to have you by my side; I love you.

Contents

CHAPTER 1

Motivation

We are living in a world of ever-growing data. On Twitter, users generate about 500 million tweets daily[1], nearly 95 million pictures, and videos are uploaded to Instagram each day. The web 2.0, the rising number of sensors and devices with internet connections and mobile phones used by the bigger part of the world population, lead to an enormous increase in data. They are part of the *internet of things*, a network of objects which collect data 24/7. As sensors become cheaper over time, one can find them almost everywhere. In smart home environments, they are built into windows to detect rain and temperature, and into doors to detect their status. Vendors build sensors into medical devices, cars, entertainments systems, and so on. Those sensors collect data we are producing during our ordinary course of life. Furthermore, our everyday life includes the online world. Social networks produce and link information about people, events, things, and their relationships. Social platforms develop special algorithms designed for the newly emerged use cases of social networks.

However, data is not only generated in an automated manner. For example in the medical sector, large clinical studies collect all possibly interesting data that is related to the health and lifestyle of a patient. Modern studies even collect DNA samples to learn more about common diseases. In January 2015, the United States of America announced the collection of DNA samples from at least one million

[1] http://www.internetlivestats.com/twitter-statistics/

volunteers during the *Precision Medicine Initiative* [1]. At the *Personal Genome Project UK* everybody can donate their own DNA to make it available to the public, to enable researchers to widen the possibility of genetic testing.

The experiments that run on the *Large Hadron Collider* (LHC) in CERN are used to collide proton beams. They produce one petabyte of data per second[2] and share part of that data through the CERN OpenData project. This data is collected to see "if the collisions have thrown up any interesting physics"[2], not to answer a detailed question.

In general, working with data has changed these days. Experiments and data collection are typically used to confirm or deny a given hypotheses. Today, lots of data is not accumulate and filtered for a particular purpose. Instead, the information is gathered just because it is available. In science, this approach is called "discovery science". In discovery science the hypotheses are no more the first step in the process. The data is. The goal is no longer to approve a given theory, but to find patterns or anomalies in a given large data set[3]. Data mining usually does this pattern detection. Data mining includes anomaly detection, regression, classification, association rule learning, clustering, and summarization. However, there are other technologies to work with large datasets depending on the field, e.g., machine learning, time series analysis, or social network analysis.

Moreover, as big data science evolved researchers found that some algorithms although designed for other purposes, like graph retrieval or sampling, can be used in big data analysis fields too. Joel Dudley et al. just published work in which they found three subgroups of type 2 diabetes using a patient network considering high-dimensional electronic medical records[4]. This too included a large data set and a hypothesis-free analysis of the given data.

Additionally to the size of the data, the structure of the data is a new problem. Data produced in the ways described above range from structured over semi-structured up to unstructured data. Unstructured data is not pre-defined by a data model. As data is collected whenever possible, even if it is unsure whether it contains interesting information, it can be complicated or even unfavorable to define a data model. Data can come from different sources with different specifications, like sensors of various vendors or different patient questioning forms in different medical institutes.

Log files, for example, typically collect semi-structured data sets that, without a predefined question to answer. The running system that uses the logging mechanism, saves much information about states and outputs in a semi-structured way. That information is unimportant as long as the system runs smoothly. If any problems emerge, those log files contain valuable information about the issue and how the

[2] https://home.cern/about/computing

system was behaving at the time it was running correctly. Log files of user actions, for instance, collecting every click in a program, are used to analyze user behavior to direct development and payment strategies.

1.1 Processing

With the increase of data a new research field emerged and the buzzword Big Data came up to summarize the issues above. Big Data is still a challenge and has a lot of open questions to be answered. Those questions include the collection, pre-computation/filtering and availability of data, the evaluation of the data, how to combine different data sets and even the question what questions all the data can answer[5].

Once there is an idea how to use the data its size leads to the question of how to assess it. As described, the data is usually not entirely structured and cleaned up in a database, but consists of a lot of semi-structured or unstructured elements which might only contain a small portion of valid information.

The size and the structure of the data ask for new ways of programming. It is no longer possible to evaluate the data on a single commodity personal computer, and high-end servers are expensive. It is necessary to run programs in a parallel manner. However parallel computing is a complex issue and had been usually used with specialized computers and by specialized programmers. Race conditions, dependencies, and manual synchronization are just a few problems that a parallel programmer has to solve. Creating a parallel program is not suitable for unknown and versatile data. Moreover, the evaluation of data sets is no more limited to big companies with specialized personnel. Everyone can get millions of data sets[3] and find out what they want to know.

This situation asks for a new way of programming parallel evaluation programs. As the demand of users working on big data sets is raising, there is a need for a more comfortable solution. Google did the first step in this direction with the Google MapReduce Framework. In 2004 Jeffrey Dean and Sanjay Ghemawat published a very well known paper about Google's MapReduce[6]. They introduce a parallel execution environment that raises the pressure of parallelization from the user's shoulders and enables the user of the environment to run parallel data processing engines on many machines. The main idea leans on the map and reduce concept known from functional programming. The user has to implement two functions, a map function that executes the user code to each record of the input data, and a reduce function, which combines records with a similar key with some user-defined

[3] https://github.com/caesar0301/awesome-public-datasets

functions (UDF). The user does not need to worry about data exchange or dependencies. The execution engine takes care of this.

MapReduce is based on two user-defined functions `map()` and `reduce()`. Those functions are adapted from functional programming. MapReduce uses and outputs key-value pairs. The map function applies the user-defined code to every key-value pair of the input. The reduce function sums up its input by performing is functionality to a list of records with a similar key. Between the map and the reduce function, the system performs a re-partitioning step. The output keys of the map function have to be assigned to a reducer, this is done in a load balancing way, usually by a hash function. Then the data is sorted and distributed between the reducers. Apache invented an open source version of MapReduce called Hadoop[7]. Hadoop is used among others by Facebook, Twitter or Spotify.

Hadoop as an OpenSource project offers anyone the possibility to write jobs that run on a lot of nodes. But there are some drawbacks for MapReduce systems. First of all, not every parallel job can be easily written in one MapReduce job. Use cases may need several connected MapReduce jobs. Hadoop or MapReduce do not provide the possibility to chain jobs automatically. The fact that MapReduce jobs allow only one input makes tasks like joins, that combine every tuple with similar key of two inputs complicated to program. From this drawbacks, other systems and programming tools emerged. Apache introduced Pig[8] that creates Hadoop MapReduce jobs that are described in an SQL-Like language called PigLatin[9]. Apache Hive[10] also offers an SQL-Like language that converts the statements into MapReduce jobs. Apache Spark[11] uses so-called Resilient Distributed Datasets (RDDs), a portion of the data that can be manipulated parallel. Spark leaves the two-staged MapReduce idea for a multi-stage paradigm. Of course, not only Apache created systems that try to make the drawbacks of MapReduce better. Systems like Hyracks[12], Dryad[13], or Nephele[14] go beyond the subscribed the strict Map and Reduce concept.

As a consequence, everyone can write programs that run on many machines, without knowledge of the parallelization and data exchange details. However, although the programs exist, the limitation of computing power may be the next problem. For example, the CERN decided in 2002 to use grid computing to spread the herculean task of data processing to other computing centers. Additionally, CERN provides a volunteer computing platform LHC@home[15] that enables everyone to donor unused computing time of their private personnel computer to the LHC data processing. However, not only CERN but a lot of other research organizations and universities offer @home projects to increase their computing power, including the probably most widely known SETI@home[4][16]. Which are just two example to use external computing power to solves the worlds big questions.

[4] http://setiathome.berkeley.edu/

Nevertheless, for a single company, who may have a big data analysis issue from time to time, it is not possible to build up a grid or an @home project. Furthermore, there might be several occasions when small businesses or end users need additional computing power for a limited time frame (like additional web servers after an advertisement campaign). Investing in hardware for those peaks would lead to under-utilization of servers. On the other hand, just the number of servers covering the usual business will not be able to handle the mentioned peaks.

A solution to this issue is services like Amazon EC2[17] or Open TelekomCloud[18], which offer on-demand computing resources. They are part of the Cloud computing idea, *"ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*[19]. Cloud computing includes several Service models like System as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). IaaS clouds are large data centers with a high number of shared-nothing commodity hardware that run virtualized servers which costumers can book on demand. In the shared nothing architecture, the individual nodes run independently and do not share disk space or memory. The nodes do not have to compete with each other for the resources.

The on-demand booking gives a high amount of freedom to the user. It is possible to get as many CPUs for a big task as needed to solve it quickly, without the needs to maintain a big data center in the own basement. Shortly booked cloud nodes can easily carry slight increases in the usage of a service, e.g., clicks on a shop website. Cloud machines can expand existing data centers, combining the private server architecture with a public cloud environment.

1.2 Availability

Nevertheless, cloud vendors do not promise perfect performance and uptime of a machine. The booked machines in a cloud system are virtual machines, usually with several virtual instances on one physical computer. As those machines share the physical hardware of the host system, hardware that is not designed to be highly available, and as they communicate over the network, failures are likely to occur. Vendors of cloud computing services give their guarantees of availability of the services with Service Level Agreements (SLA). Those agreements cover the expected service behavior, the measurements for the level of service, and the consequences of violations of the contracts.

For example, Amazon guarantees a monthly uptime percentage of 99.95% measured in 1 minute periods. That means to calculate a risk that during one of 20 one-minute

slots all machines will be unavailable within more than one region, or in average once every 33 hours. Here “unavailable" is defined as “when all of your running instances have no external connectivity."[20]. The above calculation does not cover breakdowns of single instances. If the downtime increases above that 99.95%, the user is entitled to financial rewards. Other vendors have similar SLAs, usually their advertise a monthly 99.95% uptime with similar exclusions. This means the systems running on IaaS Clouds will most likely have to deal with breakdowns.

Those uptime promises may be sufficient for running web servers where downtimes only reduce profit, and backup servers can start up quickly. However, they are pretty much useless for parallel processing, as even a failure of one single machine can break the entire job. It is therefore necessary to build these systems in a failure tolerant manner to use them efficiently in the cloud. A system needs to handle breakdown or temporarily unavailability of machines. Irrespective of the breakdown of machines, there are other failure causes in the context of parallel data processing systems. As discussed above, these systems are designed for parallel execution of Big Data input. The very definition of Big Data includes that the data may be flawed. Flawed data can cause failures in the user-defined function if the programmer made the wrong assumptions about the data. Those failures can cause the system to crash or just produce wrong output.

Both cases will lead to a re-execution of the job, which takes up additional resources, adding monetary cost to the user. At the same time, it adds stress to the environment, since the resources that could be idling and potentially shut down[21, 22] have to run on high workload once more. Consuming energy directly and for cooling systems, that are also usually not emission-free.

It is thus with good cause desirable that a system does not crash because of one fault, but recover from the fault and finishes its work with as less additional resource usage as possible. For the budget of the person running the system, as well as for the overall environment.

1.3 Problem Definition

The high number of hardware in a cloud system tends to result in a high possibility of failure. Especially with the use of commodity hardware that is not particularly resistant or secured against malfunction. Failures do occur, and will cause parallel execution jobs to fail, if no fault tolerance mechanisms are applied. This would mean to the re-execution of the entire job, even if just parts of the job failed. This leads to unnecessary usage of resources, which will cost the user money.

This fact asks for the systems that run in cloud environments to be fault tolerant.

Those systems have to be able to deal with all kinds of failures: Flaws in the network or memory, machines that are not reacting anymore or the breakdown of machines including loss of data. Ideally, the system would recognize those failures and handle them in a way the user does not even see. The perfect fault tolerant system would be able to reduce the time of failure recovery in a way that the runtime of the job does not increase memorably. At the same time, it should be not recognizable in case of a non-failure situation, regarding disk space usage and increase of runtime. Of course, it is impossible to have all of this; a fault tolerant system will always be a trade-off between those wishes.

The main question this thesis is trying to answer is:

> *Given the restrains of parallel data flow systems in IaaS Clouds, how can fault tolerance be achieved in a transparent, fast and space-saving manner for several types of faults?*

As IaaS clouds are designed for the customer to pay for any service he is using, one primary interest for the user is to keep the costs of a computation low. This adds to the general efficiency demand that the computation will finish in the shortest processing time possible. An efficient fault tolerant system thus has to add as few additional processing time as possible and take up as less disk space as possible, as the customer will be shared for permanent storage as well.

In summary, there are three requirements:

- **Fault tolerance** The system should be able to react properly to various kinds of failures. That means noticing the failure and recovering from it.
- **Little supplementary costs** The fault tolerance should be achieved with as little additional costs as possible. Especially the runtime increase for saving necessary recovery data and recovery has to be noticeably shorter than the complete restart of the system. Additionally, it should use as little disk space as possible.
- **Transparency** Ideally the user does not even notice the fact that a failure occurred. The entire failure recognition and recovery should be as transparent to the user as possible. And the user should be able to run the same jobs he used to run on the system before it implemented fault tolerance.

1.4 Research Method

Denning et al. state three main paradigms for the computer science discipline: design, abstraction, and theory. Even though the authors state themselves that the three processes are inseparable, the three paradigms have different focuses and steps of work[23]. The *theory* paradigm is a mathematical approach beginning with a definition, followed by a theorem based on this definitions, an attempt to proof the theorem and the interpretation of the results. The *abstraction* is a experimental approach and starts with a hypothesis, followed by a model and a prediction. The next step is to build an experiment with available data, and then check whether the experiments confirm the model. The *design* paradigm is an engineering approach which starts with the collection of requirements and specification, the next step is the implementation followed by testing.

This thesis has its focus in the design paradigm. Each approach for fault tolerance is designed and implemented in an existing data flow system, and the hypotheses tested using example jobs, to show if the requirements and predictions are fulfilled. Even though the jobs are implemented for the particular execution engine, they are based on real world scenarios of parallel execution and adapted from other real world jobs. As can be seen, using hypothesis and predictions also show that each approach is based on an abstraction step.

The main hypothesis of this thesis is that it is possible to achieve fault tolerance in data flow systems, transparently and faster than state of the art approaches, using intermediate data. It is based on the processing model described in section 2.2. Each presented approach is based on an hypothesis how it will be useful to achieve the fault tolerance requirements. However, the design and experiments that aim to prove the hypothesis are made in an existing real world data flow execution engine, which shift this part of the work directly to the design paradigm.

1.5 Outline

The remainder of this thesis is structured as follows:

In **Chapter 2** the basis of used techniques an the environment of implementation is given.

In **Chapter 3** the technique of ephemeral Materialization points and their positioning in the data flow is introduced. The chapter includes the general idea of ephemeral materialization points, the implementation details, and an approach for optimization.

The **Chapter 4** discusses persistent faults types, namely data and software faults, and the fault tolerance possibilities.

Chapter 5 covers optimization for the recovery process for stateless tasks. The approaches covered in this chapter aim to speed up the recovery process.

The conclusion is covered in **Chapter 6**. The chapters provides a recapitulation of the thesis and discusses how it answers the introduced question.

CHAPTER 2

Introduction

Contents

This chapter covers the basic concepts and environments this thesis is placed in. It describes the idea of IaaS Clouds and the implementation details of the data flow

System Nephele. Additionally, the chapter gives an overview of the fundamentals of fault tolerance concepts in general and checkpointing and logging in particular.

Cloud computing has emerged to the state of the art computing infrastructure. It allows the use of shared resources over the internet. The user is no more the owner of the resources like machines, software, or networks. Instead, vendors offer those resources and charge the user in a pay-as-you-go concept. The resources are usually comfortable and fast to provision. The consumer, may it be a private person or an enterprise, does not have to invest in the infrastructure or provide it himself. Especially for small businesses, investments in such computing infrastructures can be a risk factor.

The NIST definition[19] of cloud computing differentiates between three types of service models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service.

The SaaS model delivers centralized hosted software to the customer. The user of the software can access it typically over a web browser and can use it on demand. The service provider fees the customer for the subscription to the service, usually monthly and per user license. Thus the customers avoid a high initial setup cost, for the software and potentially needed hardware. At the same time, the customer has to accept that any data handled with the software reside with the service provider.

The PaaS model offers a platform to run and develop web applications on. The costumer does not have to setup and administrate the infrastructure himself; the service provider takes care of it. In a public PaaS environment, the user can manage the software deployment typically via a web browser. Similarly to the SaaS model, the vendors usually fee the user on a monthly basis.

However, next to these three big service models there are several other possible variations. In cloud computing everything can be offered as a service[24]. Those services can often still somehow fit into the three service models. But there are also other services that are provided under the aaS name, without fitting into the XaaS stack. One example is Humans as a Service (HuaaS), that offers human intelligence as a service to solve issues like image recognition that is easy to solve for humans, broadly known as crowd-sourcing.

As the model of Infrastructure as a Service (IaaS) is the basis of the programming model, it will be discussed in more detail in the following section.

2.1 Concept of IaaS Clouds

There are several vendors of IaaS Clouds these days, for example Amazon EC2 [17], Microsoft Azure[25], Rackspace[26], and GoGrid[27]. The main concept of those IaaS clouds is similar. The vendor offers the user the possibility to obtain a virtual machine (VM) on demand within a few minutes. The user can typically choose between different types of machines, which run in different regions of the world. This regioning can be very important for geographical aware load balancing and replication. Once the customer chooses a machine type, the machine can be started using an API and accessed via a web front-end and per *SSH*. All details of the provided services, the quality, availability, and responsibilities of client and vendor are defined in Service Level Agreements (SLA). In the SLAs, the service provider and consumer agree on the definition of the service and its degree. They include technical definitions for the service including the mean time between failure and the mean time to recover. The SLA should also describe how the customer can monitor the service quality, how a customer has to report issues, and the appointment of penalties if the vendor could not provide the defined service level[28].

The VMs are generated from prepared disk images with the guest operating system. Those disk images can be provided by the vendor, but could also be build customly by the user. This enables the user to build templates that fulfill his needs for the virtual machine, which includes all necessary software and configuration on startup. One could, for example, have a custom web server pre-configured to be able to run a failover instance fast, which can also be called provisioning an instance. One could also have an image with pre-installed parallel execution engine, to bring new nodes into the system quickly.

A running virtual machine in this context is called an instance. Running several instances of with the same disk image is possible. Running several identical machines can be necessary for load balancing or parallel execution frameworks. It also offers the possibility to react to usage peeks or lows. The consumer can provision identical machines or un-provision instances fast, and thus reduce the cost for idling machines, or the loss of unhappy users.

The customer pays for the machine per hour of usage. Usually, there are different machine types with different prices. The instance types differ, among other aspects, in the number of CPUs, and the size of RAM and disk space. This way, the user can book the virtual machine that suits his needs best with the lowest price. The user chooses the necessary image and the instance type he wants and starts up the machines. The vendor then bills every started hour the machines are assigned. Moreover, the region in which the machines are running has an impact on the price as well.

Additionally to the service of the virtual machines some cloud vendors offer cloud space for persistent storage of data. The disks of the virtual machines are not persistent after the machine is unassigned. With the persistent storage, the vendor offers the user a possibility to save data beyond the assignment period of the instance. Any data saved in the local file system will be inaccessible, once the machine is switched off. Therefore data that is needed beyond the lifetime of the virtual machine has to be saved outside of the VM. The pricing for this persistent storage is usually by used gigabytes. It raises the needs to reduce the amount of data that a VM saves in this storage. Furthermore, vendors often price up- and downstream to the persistent storage as well, thus getting the data out of the vendor's storage might be costly too.

Amazon, for example, offers three storage solutions: The Amazon Elastic Block Store (Amazon EBS), Amazon Elastic File System (Amazon EFS), and the Amazon Simple Storage Service (Amazon S3). The EBS and the EFS are designed to work with the EC2 nodes. EBS is a block storage and needs to be formatted by the user. It thus offers the possibility to decide on the file system type. EFS is a network file system formatted in the NTFS format. The S3 Service is an object storage that is independent from EC2 instances. All file storage solutions have their assets and drawbacks, including pricing, availability, and latency. S3, for instance, is publicly accessible, whereas EBS is only accessible via the linked virtual machine, and EFS is only available form AWS services and virtual machines. S3 is the slowest storage service, EBS the fastest and EFS is in between. In contrast to S3 and EFS, the EBS storage is not scalable. The customer has to choose the best fit for the usage purpose. The pricing of that storage is typically per GB and month, depending on the region. Considering the region EU (Ireland) the standerd storage version of S3 costs 0.023$ per GB/month (for the first 50TB), EBS 0.11$ per GB/month and EFS 0.33$ per GB/month, at the time of writing this thesis.

2.1.1 Pricing

Besides the described model, Amazon offers another alternative to using virtual machines, so-called spot instances. Those instances are often available at a lower price than the usual virtual machine instances. But the user bids a price he is willing to pay for an hour of machine time. The price of the machines varies based on supply and demand.

Once the price of spot instances reaches or falls under the bid price, the instances are made available to the user. The machines are then usable until the spot price raises over the bid price. If the spot price is higher than the bid price, the spot instance will be terminated. Amazon sends a warning two minutes before the termination, to enable the user to save data or handle other necessary configuration changes.

But still, the machine will be terminated without the user being able to change anything about it. In a way, s spot instance is not a “computing-power-on-demand" but “computing-power-at-availability".

These instances offer a new a way of scaling out big data analytics on lower cost. In more detail, the user will be able to set a price he is willing to pay for faster completion of the job if the analytic system can handle new incoming and leaving nodes. Moreover, the system does not only have to be able to un-provision nodes on demand but has to manage nodes, which are suddenly unavailable. To make sensible use of those instances, it is therefore essential to build a fault-tolerant application.

2.2 Data Flow Systems

The fault tolerance techniques are implemented in Nephele[14], a fork of the execution engine of the Apache Flink system[29]. Nevertheless, the main ideas of the approaches are transferable to other data flow systems in this area. There are plenty of data flow systems, each with slightly different focus. Those systems include for example Asterix[30], Dryad[13], and Flink[29].

Those systems are running on clusters or IaaS clouds. The main idea is to have hundreds of -typically virtual- machines and spread the work between them. The user writes Jobs for the data flow system, that consist of several tasks that have to exchange data between them. Each task will be spread to several parallel instances if possible. The system takes care of the deployment and the data exchange between the tasks. Figure 2.1 shows a possible distribution of tasks to virtual machines. Although the implementation differ in detail, they have several general structures in common.

A data flow system usually works in a master/worker pattern and controls several kinds of instances. The typically running black box user code, which causes the processing engine to run code without the information about its internal state. The user code is usually assumed to be deterministic, i.e., it produces the same output if it receives the same input. That is a common assumption; nevertheless, some parallel data processing engines cannot guarantee that the receiving tasks consume the data from the producer tasks in the same order at every execution of the job. This non-determinsm in overall execution comes from the fact that the engine decides based on the data availability which task’s output is read next. Thus, the tasks are expected to be deterministic, but the system may not be deterministic at every point.

Like in the MapReduce framework, the user does not have to take care of parallelism or data distribution. Instead, he programs jobs, usually given as a directed acyclic graph (DAG). The DAG is the representation of the data flow. The vertices are

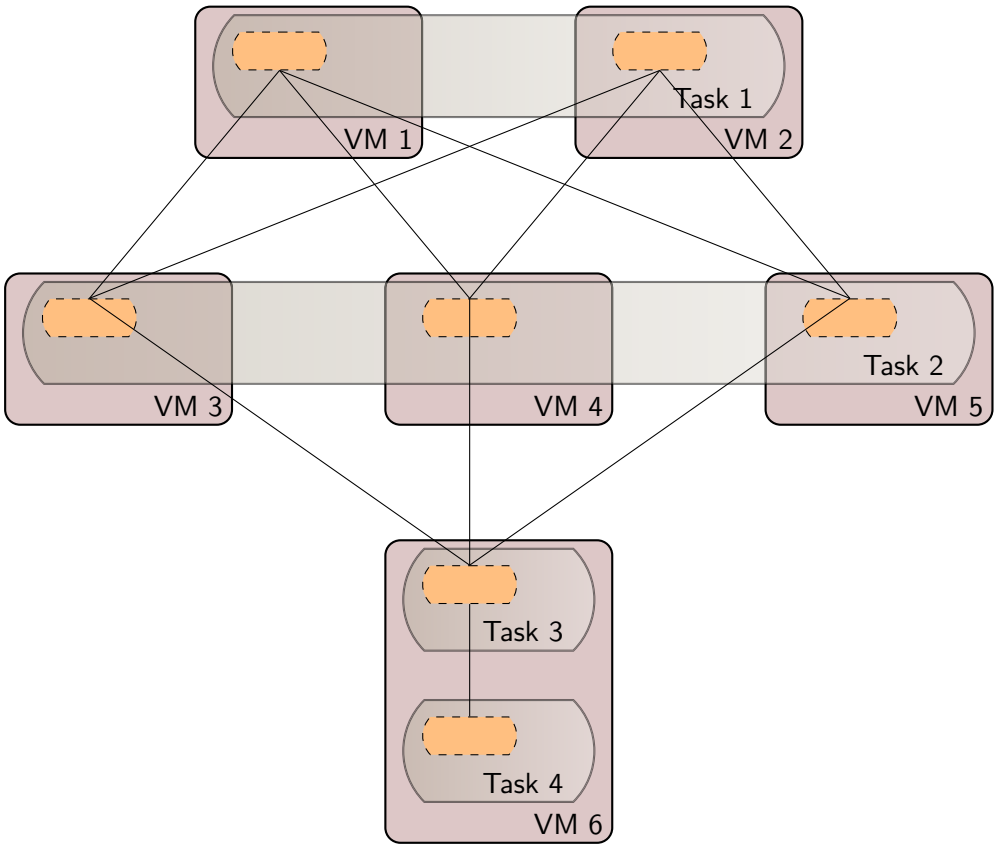

Figure 2.1: Task distribution on VMs

representing the subtasks of the job, and the data flows along the edges. The tasks are written in a sequential user-defined function. The system then uses the graph for deployment of the subtasks to nodes of the cloud or cluster. As mentioned before the general architecture of those data flow systems is based on the master/worker pattern. The master is responsible for deployment and monitoring of the worker nodes. The user-defined function that is given by the user will take a record, do some computation on it and will output a -possibly empty- set of records[31].

The system assumed in this thesis receives jobs described as a directed acyclic graph $G = (V, E)$ where V the vertices are the tasks written by the user and the Edges E are the communication connections between the tasks, implemented as FIFO queues. Each vertex $v \in V$ in the DAG will be split into one or more parallel instances of the task $v_1 - v_n$. A parallel instance will receive a set of records I during its runtime and will output a set of records O. The task receives it's records over a set of input channels P_{v_x} and will output the records to a set of output channels S_{v_x}. The channels are the parallel instances of the edges E in the DAG. The internal state of the task may change after the processing of a record. Thus, a UDF can be described as $udf_t : s_t, r \rightarrow \langle s'_t, D \rangle$. Where D is a set of output records.

Even though this definition is based on the state of the task, it is possible to have stateless UDFs where the state s'_t is equivalent to the previous state s_t. A stateless task does not trace any previous information about other records. In contrast to that, a stateful task remembers information about the previously processed records.

2.2.1 Example

The figure 2.2 shows an exemplary job for a data flow system. It consists of an input task `LineReader`, which takes a directory in a distributed file system that consists text files, and reads each file line by line. Each line is put into a record with the line number and document path. The first task tasks the line and split it into words, producing an output record for each word, including the line number and document path. Those records build the input for the `IndexBuilder`, which produces a list for each word, consisting the documents and their line number the word appears in. It will then hand those lists to the `IndexOutWriter`, which writes the index to the distributed file system.

Considering the example given above the user-defined function are the `LineReader`, `WordSplit`, `IndexBuilder`, and `IndexOutWriter`. The LineReader, for example, takes files as records and outputs records containing line number, file path, and the actual line. The LineReader is a stateless task. The state after the processing of a record is equal to the state before. The IndexBuilder, on the other hand, has to combine the list of documents with equal words. It holds a list for each word, and

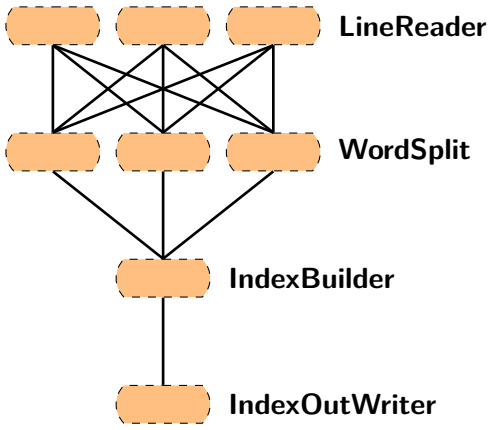

Figure 2.2: Inverted Index Dataflow Job

add the document path of the next record to a list. This means the internal state of the IndexBuilder depends on the previously processed records.

2.3 Nephele

As mentioned above the prototypical implementation of all concepts of this thesis are made in the execution engine Nephele that was designed and implemented at the research group CIT at the TU Berlin under the lead of Odej Kao. The Execution engine was part of a DFN research group called stratosphere that combined it with an entire programming stack, including the PACT programming interface. To give an overview of the concepts of Nephele this subsection describes its characteristics in more detail. Nephele is written in Java and offers a Java API for the programming of *job*s.

Nephele as aforesaid is designed in a master/worker pattern. In Nephele this master is called `JobManager` and the workers are called `TaskManager`. Given a number of IP-addresses or a connection to a cloud API the engine will start up a JobManager on one machine and one TaskManager on every other machine. The user is then able to start a job client, that connects to the JobManager. The user can send a job via this local job client to the JobManager, which will take over the work of parallelization and deployment of the job. The client polls progress updates form the master periodically and presents them to the user.

The programmer writes jobs as a DAG which is called `JobGraph`. Every vertex in

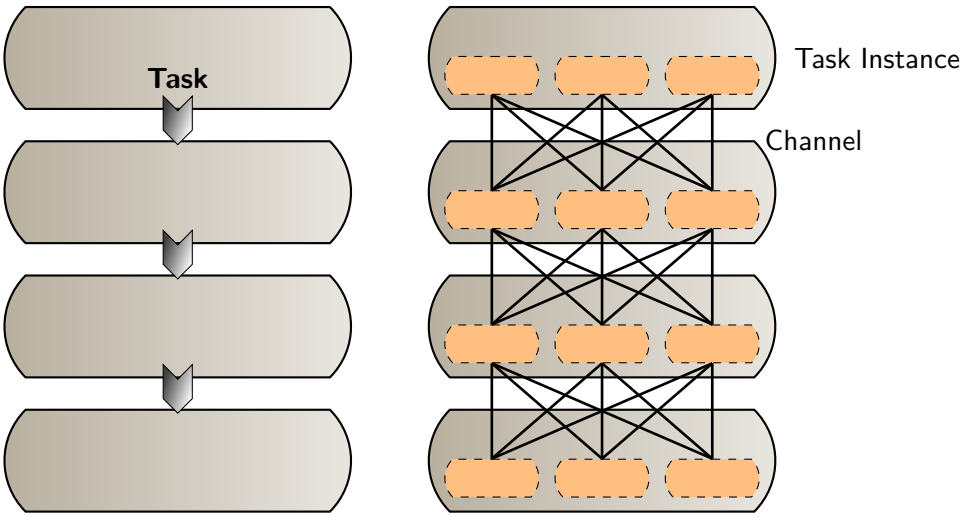

Figure 2.3: JobGraph and ExecutionGraph

the DAG describes a task of the job and is defined by a UDF that is programmed according to the API. The edges of the DAG represent the connections between these tasks and therefore the data stream. The user can give a particular degree of parallelization for each task. It is also possible to define the maximum number of tasks per instance and which tasks should share an instance. There are three types of communication between tasks: Data can be exchanged over a network, a file, or within memory. The latter two types require the two tasks to share an instance. This sharing is ensured by the engine; the user does not have to define it explicitly.

The system translates the JobGraph into a graph that is a close representation of the final execution pattern, which includes the actual degree of parallelization and the particular communication pattern between the parallel instances of each task. The resulting `ExecutionGraph` consists of a vertex for each parallel instance of a task and the tangible connections between them. The vertices in this ExecutionGraph are organized in two layers. One layer is called the `GroupVertex` and represents the tasks of the job. It contains all parallel instances of this task. An instance of a task is represented by an `ExecutionVertex` in the ExcutionGraph.

With this graph, the JobManager assigns all task instances to a virtual machine and deploys the user code to them for execution. It is responsible for building up the communication pattern and starts the job. The TaskManagers are reporting their progress to the JobManager and send heartbeats periodically. A non-reacting TaskManager can be noticed by the missing heartbeats and any exception caught

during execution will be reported to the master.

The tasks (i.e., the UDFs) executed by the workers are black box code to the system. The system does not know the behavior of the code. Effectively a task could, for example, write files, use random numbers, or hold data in its data structure. This task cannot be guaranteed to be deterministic or stateless. Nevertheless, for most of the fault tolerance mechanism described in this thesis, it will be necessary to rely on those assumptions. If so, the assumptions made will be emphasized.

2.3.1 Pipelines and States

As the programmer has any freedom to write his code, it is possible to hold state or write data to disk within the code out of control of the Nephele engine. Thus UDF can be stateful or stateless. This *state property* of a task has a direct impact on the possible fault tolerance mechanism. A stateless task can be restarted at any position of the input stream, without an effect on the output that would be produced. It is also possible to skip parts of the input data for a stateless task and it would still provide the same output for the rest of the input data.

A stateful task, in contrast, depends on all input data to build up its internal state. Skipping parts of the input changes the output of the task, as the internal state of the task will be different after consuming it. Unless the system saves the internal state of the task, the only possibility to restart it correctly is to reprocess the entire input.

Unfortunately, it is impossible to determine the state property of a task. The Nephele system is not able to inspect the state from the black box code given to the JobManager. Therefore any task in the Nephele system is deemed to be stateful by default. However, there is an opportunity to the user or some upper layer to indicate a task as stateless using annotations. A stateless task can offer a broader opportunity for fault tolerance and optimization.

For example, scaling is a typical optimization action to change the system to handle the current workload. A scalable system can handle the increase and decrease of resources. Cloud scalability differentiates between horizontal and vertical scaling. Vertical scaling (scale up/down) is done by adding (or removing) computing power to the existing nodes, e.g., migrate it to a computing instance with more CPU or RAM. Horizontal scaling (scale out/in) means to add or remove more nodes with less powerful computing instances.

Scaling tasks during runtime is only possible with stateless tasks. If data within tasks is saved, for example to group data and send entire groups to the next task, it is impossible to destroy one task to scale-in. On the other hand, if the distribution

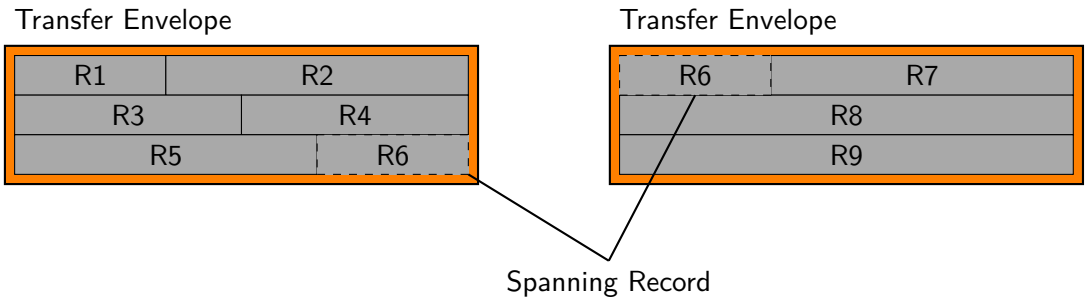

Figure 2.4: Spanning Record over two TransferEnvelopes

of output data for a task depends on the number of consuming tasks, it will not be suitable to scale out the producing task.

Nevertheless, the system does not expect tasks necessarily to be stateless, but it always expects tasks to be deterministic. Given the same sequence of input records, a task is supposed to produce the same sequence of output records.

Data flow systems like Nephele are designed to be pipelined. Each task is processing the incoming data directly without saving it to disk (aside from File-Channels) or waiting for the entire output. Naturally, pipelined execution is only possible if the user code is following this requirement. Unfortunately, not all operations can be executed in a pipelined manner. Tasks like joins or sorts need all input data to work correctly and have to be handled in a non-pipelined fashion. In the remainder of this thesis, those tasks are called *pipeline breaker*.

In the pipelined case, however, the producer sends data directly to the consumer, the moment it is produced. This data exchange happens over network or in-memory, depending on the location of the tasks. To the system, the exchanged data is just a stream of bytes. However, as this data exchange is an important basis for the upcoming work, it is necessary to look into it in more detail.

2.3.2 Data Exchange

For the tasks, a portion of data is called a *record*. The user can define a record using the `Record` interface, which requires the programmer to define serialization and deserialization of the record. Apart from that, the programmer can define his own records depending on the needs of the job. Records thus can be anything from a single integer to a key-value pair, an entire file, or a combination of complex objects.

The execution engine is not aware of the character of the record. Once it receives a record to send it to the designated consumer, it calls the serialization method, sends the bytes to the necessary instance and calls the deserialization method.

However, the records are not sent individually over the network. Records are serialized in ByteBuffers, which size can be configured. Once a ByteBuffer is filled with the serialized data it is wrapped into a so-called `TransferEnvelope` before it is sent to the consumer. The TransferEnvelope (TE) contains the data, information about the source of the data and a sequence number. These ByteBuffers with preconfigured fixed size are analogous to the block size in the HDFS, where data is also split into chunks to handle data distribution efficiently.

Depending on the size of a record, a TE may contain one single record, several small records or just a part of a record. Records are not guaranteed to fit into one TE. If a record does not fit into the free portion of the buffer, it spans over several envelopes. The deserialization of the record then requires all the envelopes that contain parts of the record. The `TransferEnvelope`s are sent to the consumer. The connection between the producer and consumer is called a channel.

Each edge of the `JobGraph` will be translated into two channels between two task instances. Depending on the character of the connection those Channels are either `InMemory-`, `File-` or `Network-Channels`. A link between two task instances is build up by one output channel at the producer side and an input channel on the consumer's side. Note, that `InMemory` and `File` connections are always between only one consumer instance and one producer instance. However, with network connections, the number of output channels from the producer matches the number of consumer task instances. Similarly, the consumer's number of input channels matches the number of producers from which it gets its data. The data in network channels are exchanged over a TCP connection.

For each of these output channels, there are distinct output buffers. Thus the sequence numbers within the TransferEnvelope are counted for one particular connection between task instances. Records can be defined by the programmer and have to implement a *write* and *read* method for serialization and deserialization. A user-defined function receives a stream (iterator) of records that can be processed and can output some records to its output channels. There is no 1:1 relation between the number of records that go into a UDF and the number of records that are coming out. A UDF can generate several, one, or even no output record for a given input record. For example, sentences can be split into words, or filtered for nouns, or combined into paragraphs.

The same relation goes for the size of the records. In the first case above, the size of an output record (a word) is noticeably smaller than an input record (a sentence). The size of the input and output records is not equal, and records of one sequence

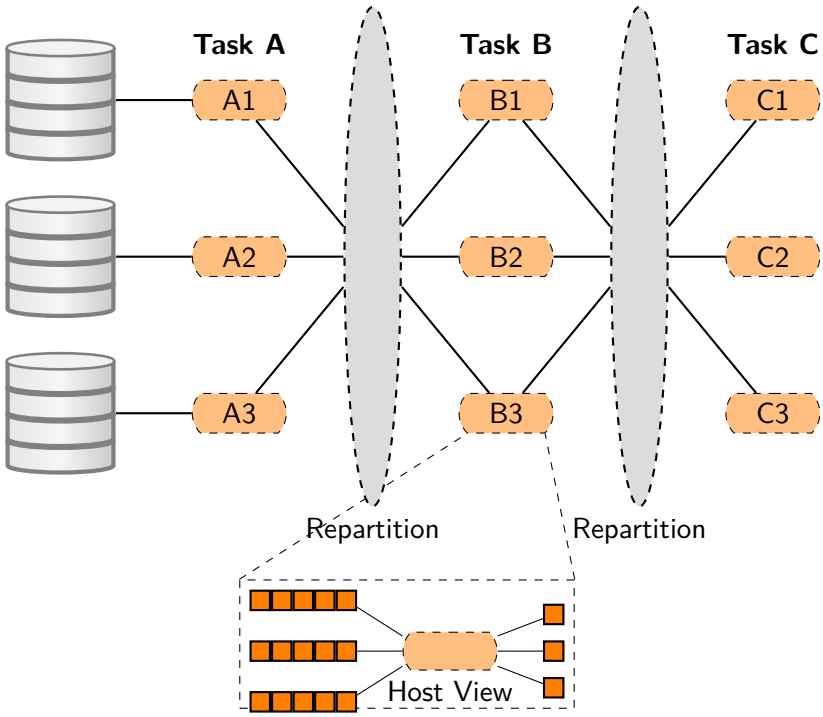

Figure 2.5: Data distribution in job and the tasks view

of input records can vary in size drastically. As the user is able to define records and has to implement the serialization and deserialization, the engine does not know the size of a record before deserializing it. It is therefore impossible to make any predictions about the shape of the data. The data is a black box to the system too.

2.3.3 Example

Listing 2.1 shows an example Job description of a Nephele Job with four tasks. One input and one output task and two worker task. Each task vertex is defined by a Java class, in this example `LineReader`, `WordSplit`, `IndexBuilder` and `IndexOutWriter`. These vertices are connected using a NETWORK channel. This way Nephele users can write a job directly for the Nephele engine and submit it to the JobManager. The Java source code can be found in the Appendix at section B

```
public class InvertedIndex {

        public static void main(String[] args) {
                JobGraph jobGraph = new JobGraph("Inverted Index Job")
                    ;
                JobFileInputVertex fileReader = new JobFileInputVertex
                    ("Line Reader", jobGraph);
                fileReader.setFileInputClass(LineReader.class);
                fileReader.setFilePath(new Path(args[0]));
                fileReader.setNumberOfSubtasks(3);

                JobTaskVertex wordSplit = new JobTaskVertex("WordSplit
                    ", jobGraph);
                wordSplit.setTaskClass(WordSplit.class);
                wordSplit.setNumberOfSubtasks(3);

                JobTaskVertex indexBuilder = new JobTaskVertex("
                    IndexBuilder", jobGraph);
                indexBuilder.setTaskClass(IndexBuilder.class);
                indexBuilder.setNumberOfSubtasks(1);

                JobFileOutputVertex indexWriter = new
                    JobFileOutputVertex("Index Writer", jobGraph);
                indexWriter.setFileOutputClass(IndexOutWriter.class);
                indexWriter.setFilePath(new Path(args[1]));

                try {
                        fileReader.connectTo(wordSplit, ChannelType.
                            NETWORK, null);
                        wordSplit.connectTo(indexBuilder, ChannelType.
                            NETWORK, null);
                        indexBuilder.connectTo(indexWriter,
                            ChannelType.NETWORK, null);
                } catch (JobGraphDefinitionException e) {
                        e.printStackTrace();
                        return;
                }
                Configuration clientConf = new Configuration();
                clientConf.setString("jobmanager.rpc.address",args[2])
                    ;
                clientConf.setString("jobmanager.rpc.port", args[3]);
                JobClient jobClient;
                try {
                        jobClient = new JobClient(jobGraph, clientConf
                            );
                        jobClient.submitJobAndWait();
                } catch (Exception e) {
                        e.printStackTrace();
                        System.exit(-1);
                }
        }
}
```

Listing 2.1: Example Nephele Job

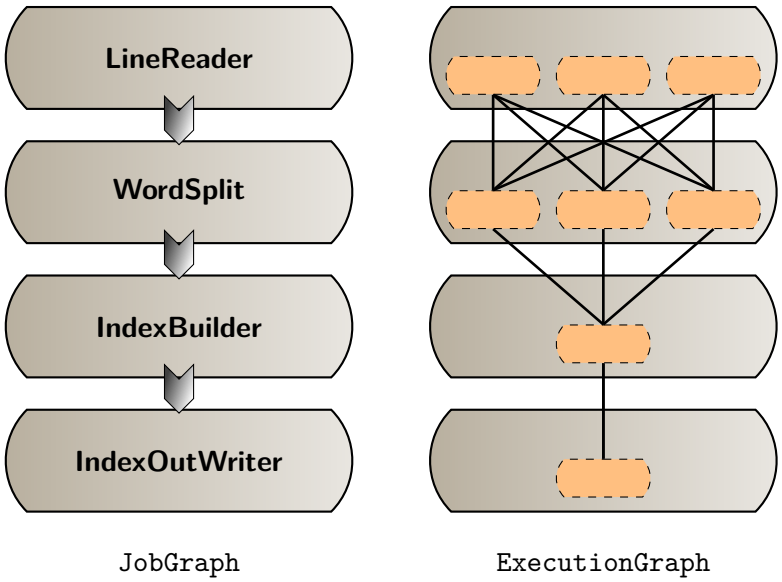

Figure 2.6: Job- and Execution Graph for InvertedIndex Job

2.3.4 The PACT Layer

In the stratosphere stack, the programming of a job is usually done using PACT. PACT is a programming interface that allows writing jobs using Parallelization Contracts (PACTs) and a key/value data model. The PACT programming model is mainly based on an `InputContract` which is a second-order function and takes a UDF and data as input.

The PACT programming model is closer to the MapReduce paradigm, and actually the Map and Reduce functions are examples of PACT contracts. Additionally to Map and Reduce, PACT also provides the Cross, CoGroup, and Match contracts. Cross builds a Cartesian product of its inputs, CoGroup partitions input by keys, and Match matches key/value pairs of its multiple input sources.

Once a PACT job is written, the PACT compiler translates it into a Nephele DAG. The PACT programming model has a declarative character and allows the compiler to potentially translate a PACT program into different execution plans, and thus aim for an optimal setup of the job. During translation, the compiler gives information about the considered channel types and the degree of parallelization using annotations. With these annotations, Nephele builds a fitting `ExecutionGraph`.

Once the system chose an execution plan, it transforms it into a Nephele DAG that consists of vertices running PACT code that wraps the UDF. Wrapping the UDF means the task will not only consist of the original UDF but have added PACT runtime code. That code will be used to pre-process the incoming data to fulfill the given guarantees about the input data that are provided by the input contract. From the viewpoint of the Nephele system, this runtime code runs like regular user code. Therefore the Nephele engine does not know the behavior of the PACT runtime code or the way the original UDF is invoked by PACT, which typically differs from the invocation in Nephele. In Nephele a UDF is invoked once and fed with its portion of the input data. In PACT a UDF is typically invoked repeatedly with portions of the data, for example, data with one particular key. The consequence is that most of the Nephele tasks that are compiled from a PACT program are stateful tasks, even though PACT asks for the UDF to be stateless between invocations, the input contract usually has some state to group the input data.

However, the information about the state property of a task could come right from the PACT layer, and it could annotate it to the task. This annotation gives the opportunity to use proper optimization and fault tolerance strategies for these tasks. This feature is possible, but not implemented in the initial system.

2.4 Fault Tolerance

The ideal goal of fault tolerance is to have the system recover from a failure completely transparently to the user and to be not noticeable in case of flawless execution. This is, of course, impossible to achieve. Every achievement in fault tolerance leads to a drawback somewhere else; the system will be slower, take more disk space, utilize more machines or increase other costs. Therefore fault tolerance is always a trade-off between the cost and the degree of fault tolerance.

The way to achieve fault tolerance in this context is typically a form of rollback recovery. In case of a failure, the system is reset/rolled back to a previous consistent state. The most straightforward fault tolerance technique for a data flow system is to restart the entire system automatically once a failure occurs, as the initial state of the system is a consistent state. This method is easy to program, and it is not noticeable if no failure occurs. In case of a failure, it leads to a significantly longer job run and thus longer leases time for the computing instances. If the system shall not roll back entirely, it is necessary to save consistent states of the system.

The other extreme is to save all intermediate data between tasks to non-volatile memory before giving it to the consuming task. Saving all intermediate data is equivalent to saving the system state after the execution of each task, and then to restart the failed task. This routine is fast during recovery, as only the failed task

has to be reset, but slows down the system significantly in case of a failure-free run. In principle, MapReduce follows this schema as it stores all data after the map task before the reduce tasks start.

The sweet spot for fault tolerance is somewhere in between those two extremes: Saving states occasionally, not after every step. This intuitive solution leads to several demands to the user-defined functions and the nature of the saved state.

This section first covers the terminology in the context of fault tolerance. In subsection 2.4.2 it discusses the failure model, that the fault tolerance approaches are based on. Subsection 2.4.3 describes the fault tolerance techniques of checkpointing and logging in message passing systems, which are the base of the presented fault tolerance approaches for data flow systems.

2.4.1 Terminology

In the context of fault tolerance, there is a clear distinction between faults, failures, and errors. This terminology is described in several sources [32, 33, 34] and analog to those descriptions this section covers the used vocabularies in fault tolerance.

If a system differs from the expected behavior, this is a *failure* of the system. To see that deviation the normal functioning of the system has to be specified. Unless there is a description of what the system is expected to behave like, there can be no failure. If the system conducts in any other way, than the specification asks for, there is a failure. A failure might involve the system being unreachable or producing incorrect output.

Failures can be consistent or inconsistent, where inconsistent or byzantine failures in contrast to consistent failures, do not appear equally to every user of the system. If a failure manifests differently to different users or viewpoints, it is inconsistent[32].

An *error* is an incorrectness of the system that may lead to a failure. Errors do not necessarily cause failures. Errors can be detected in the system before they even produce a failure. At this point, fault tolerance mechanisms can prevent the system to run into a failure under an error.

Errors, however, occur because of a fault in the system. A *fault* is an incorrectness in the system. For user code, that might be code which deviates from the correct syntax, missing null checks or even just working code that does not fulfill the requirements. Every system is suspected to have faults, but not every fault will manifest into an error. This could, for example, be a fault in user code that the system never reaches during the runtime. Those faults are called *latent*. Once a fault causes an error the fault is *active*. On the other hand, every error that occurs is caused by a

fault in the system. Several different faults can lead to the same error.

Faults can be either transient or permanent. The transient fault will only occur for a period, and therefore just cause errors temporarily. Whereas permanent faults, do not disappear over time. Due to the time component of transients faults, they are complicated to detect[33].

Coming from these definitions, fault tolerance is the ability to prevent failures of the system even though faults exist in the system and parts of the systems fails. This means the system should be working according to its specification even if components of the overall systems fail.

Fault tolerance is usually running through several phases: error detection, damage confinement, error recovery, fault treatment. First of all, an error has to be detected, to do the right things to avoid failure caused by this error. Once the system detects the error, it must prevent that the error spreads through other components. After that, the error must be removed. Otherwise, the system would run into failure. In case of permanent faults, it is not enough to handle the error, as the error will always occur again if the fault is not removed.

As mentioned above the fault tolerance techniques in this thesis aim to provide fault tolerances for Hardware-, Software- and Data-faults. Hardware-faults are caused by errors or the breakdown of hardware. This can be, for example, disk failures, engine failures, or breakdowns of switches. Hardware faults can be permanent or transient.

Software-faults are emerging from programming errors in the user code or third-party libraries. A software error might not cause a fault for every record, but if it occurs for a record once, it will happen every time that record is processed.

Data-faults are flaws in the Input data that cause the tasks to fail. Note that Software-faults and Data-faults go hand in hand if deficiencies in the data cause a UDF to crash, it is because the code made assumptions about the data without handling possible deviation.

2.4.2 Failure Model

This thesis separates failures according to two categories: transient and detectable. A failure is either transient or not, and it might be detectable or not. Transient failures are those, which only occur for an amount of time and will not happen again afterward. In contrast, permanent faults will always occur in the same manner. A detectable failure is one that the system recognizes either through exceptions, or missing heartbeats. A non-detectable failure is a failure, that is no identifiable by the system. Such a failure can only be recognized by the user.

1. **Transient faults, which are detectable**
   In this thesis those faults are covered under Hardware-faults, as the hardware is the typical cause of this fault type. As the fault is transient, it will not occur again after a restart. They are covered in chapter 3.

2. **Non-transient faults, which are detectable**
   Those failures will occur at every run of the job equally. The system will detect some kind of error, e.g., an exception, and can, therefore, handle the failure. They are covered in chapter 4.1.

3. **Non-transient faults, that are non-detectable (by the system)**
   Obviously the no system is able to recover automatically from non-detectable failures. However, these failures may be detectable by the user. As they are not transient, the user may be able to handle the error. A possible solution, on how the system can support the user in cases of these failures is discussed in 4.2

4. **Transient faults, that are not detectable (by the system)**
   This type of failure, may be detectable by the user (e.g., a flawed output of the job), but as it is transient, the only solution is to restart the job. The system is not able to handle those faults.

In data flow systems, there are generally three types of faults. Either the hardware has flaws, this could range from an energy break down, which causes machines to fail, over connectivity problems to flaws in network communication. Sahoo et al. presented an analysis of a 400 node cluster with different workloads and found 664 permanent and 143841 temporary hardware related errors in a year[35]. This shows that those errors are fairly common in distributed environments and have to be addressed.

Software-faults are usually called bugs and are mistakes in programming or configuration, for example a missing null check on Objects. Data-faults are flaws in the input data, missing values in data sets are typical problem, that would lead to a fault if it is not addressed by the user code. Sahoo et al. found 1000 software related faults in their year of observation which let to 123 faults[35]. Another example would be type conversion problems. Consider a field that is expected to contain a Integer represented as a String containing some string that cannot be parsed to an Integer. Data flaws should be handled by the user code, however if it does not, one flawed record will cause the entire job to fail permanently.

Simulating Faults

In the supposition failure model, Hardware-faults are transient faults, which occur randomly. They are not bound to a particular job or input. Hardware faults can happen at any time, and will not necessarily occur in the same manner again. In order to simulate hardware faults for evaluation purposes, the hardware must of course not be actually damaged. Instead, hardware faults will be represented by either killing the running processes of the worker node or killing the user-defined function. The first simulation is from the recovery point of view equal to the crash of a virtual machine, the physical machine, or the operating system, as the connection to the worker processes is no longer been given in any case. Those faults are typically detected by missing heartbeats or failing connections. The second simulation (killing the UDF) is equivalent to hardware faults that do not cause the machine to crash. Those are transient faults, which are detectable.

Data-faults and Software-faults go hand in hand. It is not always possible to distinguish between both kinds of failures. A record which has, for example, unexpected empty fields may cause a crash of the UDF. However, it can only do so if the possibility of an empty field was not considered during programming. This behavior could be a bug and therefore a Software-fault. Data-faults are permanent faults. A record, that causes a breakdown of the UDF will do this every time the UDF is invoked with this record. In the failure model, a Data-fault is expected to cause an unhandled exception in the UDF. For evaluation purposes, Data-faults are simulated, by changing a UDF so that it will throw an exception for a particular record. Those are non-transient faults, which are detectable.

Software-faults which are not Data-faults are not detectable by the engine. A Software-fault or bug, may not crash the system at all, it may only produce the wrong output data. It may, for example, only output the results of the first incoming record for every other record, or it might never terminate due to an infinite loop. An UDF that runs smoothly will not appear faulty to the system. Therefore Software-faults can only be detected by the user. He can identify that the final output is not correct, or that the UDF runs longer than expected.

As a Software-fault is not detectable by the system, the requirement of transparency is not achievable in this case. In fact, the entire fault tolerance approach is a different one in this case. As the system cannot detect these faults, it can naturally not recover from them. The question that raises with these kinds of faults is: Can the system be built so that it can support the user in the recovery process?

Master Failures

As mentioned above the fault tolerances mechanisms described in this thesis are supposed to handle worker failures. However, the master node is, of course, a possible point of failure as well. If the master (`JobManager` in Nephele) fails or is not reachable during execution, the job completion will fail. In Nephele status updates and heartbeats are sent to the `JobManager` using an RPC call. Sending RPC requests will fail if the `JobManager` is not reachable, and therefore cause an IOException at the `TaskManagers`. Thus the master is a single point of failure.

Even though a master failure will fail the complete system, there is plenty of research and solutions to fault tolerance in the case of master failures. The most common solution for this issue is the preparation of secondary master nodes that will take over the work of the original master. Typically, the secondary master will run in parallel to avoid delays because of the startup time. To be able to take over for the failed primary master, the secondary master has to keep up with the state of the primary. This can be done in different ways, by doing periodic checkpoints that can be replayed on the secondary, or status updates from the primary to the secondary node. This way, the secondary node will provide enough state information to take over the work quickly. The granularity of information updates and checkpoints usually depends on the consistency guarantee that has to be achieved.

Googles MapReduce checkpoints the master state to GFS and will start a backup master from this checkpoint.[36]

However, since master failures and backup nodes are a well-known solution, master failures will not be the focus of this thesis. Known solutions can be implemented in Nephele similarly to other systems and will offer fault tolerance to the JobManager. The presented fault tolerance mechanisms are based on intermediate data and focus on failures of the other components in the system.

2.4.3 Checkpointing and Logging

In recovery, the saved state of a process is called a checkpoint. The checkpoint includes all necessary information to roll back the process to a consistent state and restart processing from this saved state. Any operation which was done after the writing of the checkpoint is not represented in the state of the checkpoint and is either lost or has to be re-executed.

However, in distributed systems, it is not as simple as rolling back one process that has failed. The states of processes depend on each other, rolling back just one process would mean that the states of other processes are no more consistent

with the rolled-back process. The failed process might have sent messages after the checkpoint was written. If it rolls back individually, the system would consist of messages that a process has received without a process that has sent them. This issue can be adapted to data flow systems. Records flowing through the DAG can be considered to be messages, and the tasks are the processes in a distributed system. Even though message exchange patterns are very restricted (only sending records from the producer to the consumer) at any failure, the states of the tasks have to be reproduced so that a consistent state is reached.

Thus the main challenge in a data flow system is to define the sum of information, which are essential to rollback the complete system to a consistent state. As tasks exchange information between each other, every task may have dependencies upon others. Rolling back the failed task may hence cause other tasks to rollback. This effect is called *rollback propagation* and can cause the system to roll back to its initial state[37].

One solution to this problem is to let tasks create their checkpoints coordinated by a master. This coordination must ensure that the checkpoints represent a consistent state of the overall system. A consistent state is qualified by the fact, that for each received message in that state the sending of the message is contained in the checkpoint of the sending process.

The data flow system is a special case of a message passing system. As the jobs are described by directed acyclic graphs, one task instance can only depend on some of the other processes, namely its predecessors and its successors. Significant message exchange for the rollback of this task instance happens merely between itself and its pre-/successors. Thus a consistent state does not have to consider checkpoints for every task of the job.

The other specialty in the data flow system is that a state of tasks can only be described by the data it has already processed. This has two implications: Saving the output-data of a task qualifies as saving its state. Second, it is necessary to save all data from the first produced tasks to the last. It may be possible to discard records if all outputs based on this record is saved to another checkpoint, but it would mean to monitor each record over the entire job.

In detail, for the given situation, a set of checkpoints that covers a consistent system state is a set of checkpoints that includes a checkpoint for every path from the failed task to the input nodes. This consistent checkpoint coverage is covered in detail in chapter 3 which describes the approach of ephemeral materialization points.

Fault tolerance techniques typically add logging to the checkpointing technique, where all message passing between two checkpoints is saved to a local log file. With the log file, the process can reprocess its state, from the checkpointed state to the

pre-failure state. A checkpoint can be seen as a compressed version of the log reprocessing up to that point. Every time the state of the processes is checkpointed it can start a new log, as all entries of the current log are represented in the checkpointed state.

2.5 Detailed Design Goals and Scope

There are three goals that the design of the fault tolerant system and the prototype should reach. First of all, to be transparent to the user, the fault tolerant system has to run with the same jobs, that would work in the system without the fault tolerance approach. The user should not change anything in the job.

Second, the fault tolerance mechanism should work without additional information from the user or the upper layer. It can be beneficial for the fault tolerance performance to get information. However, the general mechanism should work without additional input.

The third goal is that the fault tolerance implementation should use the existing mechanisms of the system whenever possible and change the internals of the system as little as possible. The goal is to achieve fault tolerance in the Nephele system, not to optimize the system itself.

As described before, the work focuses on worker or tasks faults, not on faults for the master node. The fault tolerance strategies are designed for pipelined batch processing systems; they do not consider stream processing or iterative processing.

2.6 Contribution

This thesis introduces a technique of materializing and saving intermediate data in dataflow systems, that adapts from checkpointing and logging techniques. Using this method, the system can recover from various failures and achieve Hardware-, Software- and Data-fault tolerance.

By using intermediate data, the engine can detect and recover from the first two types of faults, at least in a restricted environment. The first part of the implementation covers the idea of `Ephemeral Materialization points` which introduces intermediate data as a first-class citizen. It includes an approach to save intermediate data efficiently, for jobs of black box UDFs. Additionally, it offers an efficient recovery process. The ephemeral materialization points are a general approach to cover any transient and detectable fault.

The second part of the implementation focuses on the non-transient faults, and how the materialization points can be used to achieve fault tolerance in those cases or the case of non-detectable non-transient faults. It discusses the opportunity to support the user during the manual error detection and recovery.

Third, the work takes a look at additional optimization techniques that can be adapted given some restrains to the tasks, and offer an opportunity to speed up the recovery. If the system can rely upon particular task characteristics, it is possible to step away from the general recovery process and use optimization techniques. The details of the approaches including the requirements to the task will be described in chapter 4

The techniques described, are evaluated considering the requirements described above. The first question to ask is if the system is able to detect and recover from a simulated failure. In order to answer this question, faults will be generated in running jobs, and the recovery processes monitored. The transparency of the approaches is discussed, as transparency is not a measurable value. Does the user of the system have to give additional information or input to the system, or does he have to engage in the recovery process?

The third requirement is a measurable value. The supplementary cost includes additional runtime and disk space usage. The question to answer here is whether an approach adds additional cost and if this cost is acceptable as the benefits of the fault tolerance outweigh the cost. This is done by runtime measurements for test jobs that run on a private cloud environment.

All approaches have been implemented in *Nephele*, a massively parallel execution engine, which started as a research prototype and developed into the execution engine of Apache Flink and will be described in detail in section 2.3. All experimental evaluations are presented in the corresponding chapters.

2.6.1 Earlier Publication

Parts of this thesis have been published in the following publications:

Mareike Höger, Odej Kao, Daniel Warneke, Philipp Richter
Ephemeral materialization points in stratosphere data management on the cloud
Adv. Parallel Comput, 23, 163-181.

Mareike Höger, Odej Kao
Memoization of Materialization Points
Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on. IEEE, 2013

Mareike Höger, Odej Kao
Progress Estimation in Parallel Data Processing Systems
Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences. IEEE, 2016.

Mareike Höger, Odej Kao
Record Skipping in Parallel Data Processing Systems
In Cloud and Autonomic Computing (ICCAC), 2016 International Conference on (pp. 107-110). IEEE.

Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, Daniel Warneke
The Stratosphere platform for big data analytics
The VLDB Journal 23(6), 939-964

Alexander Stanik, Mareike Höger, Odej Kao
Failover Pattern with a Self-Healing mechanism for high availability Cloud Solutions
In Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on (pp. 23-29). IEEE.

CHAPTER 3

Ephemeral Materialization Points

Contents

This chapter introduces the idea of *ephemeral materialization points*. That is, a unique logging strategy in the scope of parallel data processing engines. With ephemeral materialization points it is possible to monitor the behavior of a job during runtime to make a well-grounded decision about saving the task's output data or not. This approach is a fault tolerance technique with a small overhead and without the needs of comprehensive statistical information about the job. This approach is designed and implemented in the Nephele system, which is described in section 2.3. And offers a new approach besides the usual fault tolerance approaches in data-flow systems

There are two possible approaches, either to save intermediate data or to make snapshots of the entire task or VM. For the ephemeral materialization points, the key is to save intermediate data. Data flow systems usually do not keep intermediate data; they hand the data over to the consumer right after production. Thus, unless directly forced by the user (like using `FileChannel`s in Nephele), the data will be fugacious and only exist as long as the consumer does not process it. Once the consumer used a portion of data only the result coming from this data exists in the system. The only persistent data are the input source of the job and the result, which the job usually reads from and writes to a persistent file system.

As there are no persistent data in the system during execution, it can only reproduce lost data from the job's input. As a task is a black box to the system and might have an internal state, it is necessary to restart it and reprocess its entire input to enter the same state it was in before the failure. This means the system has to reproduce the task's full input. The system is not aware which output is coming from which input data, the only solution of reproducing the input of a failed task is to restart the predecessors. This predecessor restarting leads to a rollback propagation and causes all tasks upwards the data stream up to the input to restart.

Additionally to that the data distribution in those data flow systems is not deterministic: One record may run over another connection at any other job execution. Unfortunately, this also applies to partial restarts of a job. Therefore the order and the location of the data may change after the restart of the tasks. Considering an exactly-once consistency model, that asks for each record to be processed exactly once, this causes all followers of the failed task to restart as well. That is because the successors cannot decide whether the data they receive is new unseen data or data they have already processed. That means, one failed task in the pipeline will lead to a restart of the entire job.

3.1 Idea

Saving intermediate data is a form of log-based rollback recovery. Usually checkpointing means to save the state of the processes to disk. In addition to that, it is possible to log all events between two checkpointing events. Thus it is possible to rollback to the state of the checkpoint and replay all logged events.

In opposition, with materialization points there are no states saved during execution. The system only knows about two states of the task: The current and the initial state. The recovery logic uses the initial state as the last saved state, and the materialization points as logged events. Events in this context are the sent messages containing the data. Recording the events means to write the messages to disk. They already include all information that the recovery process needs.

If intermediate data is stored on disk or in a distributed file system, it is possible to use this data to reduce the number of tasks that have to restart. The naive solution is to save the output data of each task to disk. That would enable the failed task to read directly from the saved data of its predecessors, without the needs of restarting any other task. Unfortunately saving data to disk causes a high overhead and increases the runtime of a job. Some tasks in the job may blow up the data volume drastically and cost a lot to be saved. It is important not to save all data to provide fault tolerance and an acceptable runtime in case of a failure-free job run.

This automatically raises the question, where to position materialization points. Setting a materialization point at every third task may hit precisely the tasks that produce a high amount of data and are expensive to materialize. The materialization points should position themselves at those tasks, which are cheap to store to disk. Meaning those tasks that produce a small amount of data, and the tasks that are expensive to rerun.

The issue with this requirements is that the described data flow system does not have that information. The execution engine runs black-box user code. Before runtime, it does not have any data on the behavior of the job and its tasks. It is thus impossible to make a proper decision before the job is running.

The crux is, when the job is running, it already produces data. And as the system is supposed to behave in a pipelined manner, the producer sends its data directly to the consumer. Intermediate data in data flow systems are highly fugacious. This fact asks for a solution, which enables the possibility to monitor the job during runtime and still preserve the pipelined character of the system. This solution is called ephemeral materialization points.

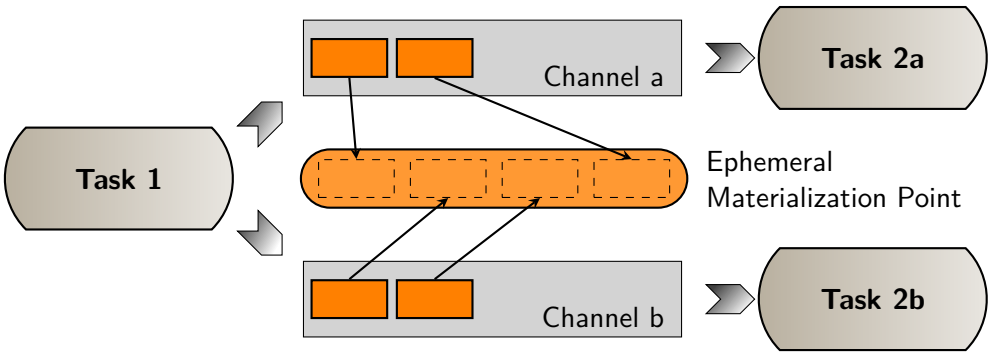

Figure 3.1: Saving envelopes in the materialization point

With ephemeral materialization, a copy of the pipelined data is held in main memory during the beginning of a task run. Data that is produced by the task runs over the network to the consuming task; simultaneously, it stays in memory. During this time the task is monitored to retrieve information of its characteristics and behavior. If the limitation of the assigned memory is exceeded, a decision is triggered. Based on the collected task information, the ephemeral materialization point decides whether the data held in memory is saved to disk or discarded. Every task is considered as a candidate for keeping intermediate data. During runtime, it will be determined whether the ephemeral materialization point will discard the data or the materialization point is made permanent. This is discussed in detail in 3.3.

In the used system data is not exchanged as single records but, wrapped together into one transport container that includes besides the data, information about sender-receiver, and a sequence number. The container is called `TransferEnvelope`. Therefore the materialization point does not write data to disk with every produced record. Instead, it writes the data to disk every time a TransferEnevlope is filled and ready for network transfer. The materialization point writes the envelope to disk and hands it to the network connection afterwards. The consumer receives the data almost at the same speed as without materialization, just delayed by the time it takes the producer to write the envelope to disk.

Materialization points can have different states. They can either be complete or partial. The Materialization point is complete if it has written all data of the producing task to disk. This is naturally the point after a task has finished its execution, and it sends the last TransferEnvelope. As long as the task still produces data and writes it into the materialization point, it is partial. The recovery is not only possible from a complete materialization point, but also when it is still partial, and writes data to disk. As the recovery mechanism does not have to wait for a

materialization point to be complete, the system will not block during recovery.

3.2 Recovery

The recovery from ephemeral materialization points relies on a log-based rollback recovery. If a failure occurs, the failed task rolls back to the initial state. The logged events are replayed. Replaying the events means that the recovery logic has to resend all `TransferEnvelope`s that the materialization point has written. Thus the rolled back task receives all its input data again.

As mentioned in section 2.3.1, all tasks have to be considered to be stateful, as it is not possible to find out about the state property of a task unless the user provides that information. At this point of the fault tolerance approach it must be suitable for any job that is given to the system, may it be stateful or stateless. Thus, in the initial implementation of the rollback recovery, all tasks are considered to be stateful.

The materialization points do not save tasks states but the intermediate data. Thus the only opportunity to bring a task back into the same state it was before the failure occurred is to roll it back to the last known state, which is the initial state, and from here use the intermediate data to reproduce the state the task was in before the failure. Accordingly, the materialization point is similar to a log, that keeps all necessary information to achieve a state from the last saved state.

For a task t in a state s_t^x the corresponding materialization point is a set of record $R = (r_1, r_2, ...r_n)$ that create the state switches, which bring the task from the initial state s_t^{init} to the state s_t^x.

$s_t^{init}, r_1 \rightarrow \langle s_t^i, D_1 \rangle$

$s_t^i, r_2 \rightarrow \langle s_t^{i+1}, D_2 \rangle$

...

$s_t^{x-1}, r_n \rightarrow \langle s_t^x, D_n \rangle$

This can lead to *rollback propagation*, as the system does not ensure that all tasks log their events. If the failed task needs its input again, but the producing tasks do not have written a materialization point, they have to rollback themselves to reproduce all output, and so on. Additionally to that, all followers of a rolled back task have to be restarted.

If a task rolls back, it reproduces all output again from the first record on. As the

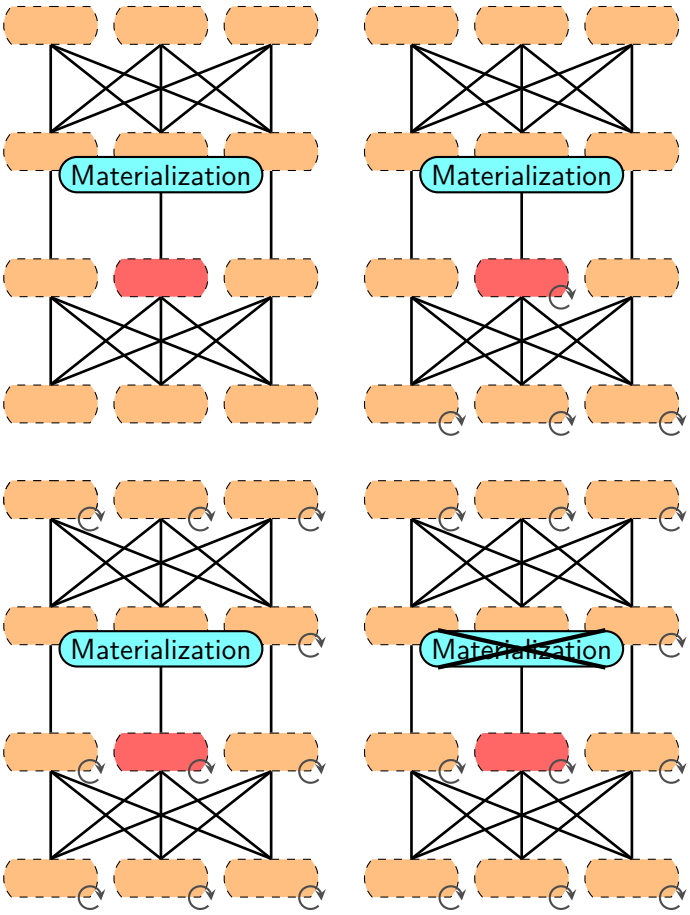

Figure 3.2: Rollback propagation

distribution pattern of the data is not necessarily fixed, the successor tasks probably do not receive the same data again. They therefore have to restart execution and discard all produced data. This rollback propagates further to the successors. The rollback propagation will lead to a lot of restarts and may even lead to the *domino effect*[37] and cause a complete restart of the job for one failed vertex.

Figure 3.2 shows a rollback propagation for one failed vertex, that causes the entire job to restart, even though the predecessor of the failed task has materialized intermediate data. The failed task has to restart, therefore the followers have to restart. As the restarted tasks need the input again, the predecessors have to restart, and their predecessors have to restart, and thus any of their followers have to restart. In this case the materializing task has to restart as well, and the materialization point

is discarded.

The recovery logic can reduce those restarts if the output stream of a task is forced to be equal after a task restarts. If a task replays its input, it will receive its input from the first record and therefore produce its output data from the first record. Thus the consuming task will receive its input data from the first record and so on. Unfortunately, the order of the records after a recovery process cannot always be assured to be the same order as in the original job run. This is because a job will be deterministic, but not determined. Meaning that a job will always produce the same output for an equal input, but it may do so with different internal states during the various runs. The output of the job will still be the same, but it is not necessarily guaranteed, that a task instance will receive the exact data. Note that every task is considered to have deterministic behavior.

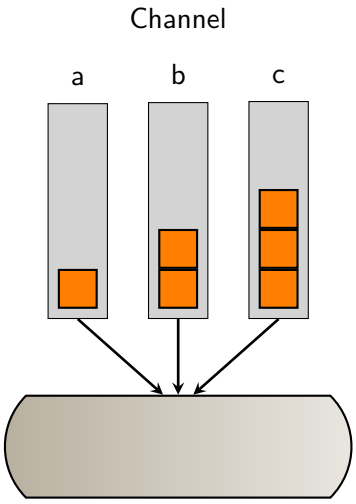

Figure 3.3: Reading from several Channels

One record and its resulting records do not necessarily flow over the exact same path through the graph; this is because of the reading behavior of tasks. To optimize the performance an being able to handle different speeds of tasks, a task is not reading envelopes in a round robin fashion. Instead, it reads all available envelopes of a channel, and then switches to the next channel that has data available. If envelopes are available, all are read, if not, the task checks the next channel.

Figure 3.3 shows a possible input for a task. The task has three input channel a, b, and c. At the channel a there is one envelope available, at channel b it could read 2 envelopes, and the channel c has 3 envelopes available. The task will read the

envelope from channel a, then it switches to channel b. At channel b both available envelopes are read, before the task switches to the third channel, where it reads all three available envelopes.

This way the reading of data is pretty efficient. The system does not block if one producer is slower then the other. If one producer lags, the consumer takes the data of the other producers first. And reading all available data at once reduces switches. However, this technique gives the possibility that the task reads data in different order between two job runs if for any reason the speed of the data production varies. Especially in the recovery case, this is the point. During recovery, all input data is available immediately, as the replaying threads read it from the checkpoint. Thus all available data from the first producer would be read at once, before the others. This ordering most likely differs from the original one.

3.2.1 Enforcing Deterministic Data Flow

In order to enforce a deterministic behavior, it is necessary to force the same reading order for the already-processed data. That makes it essential to save the reading order during the job runtime. The execution engine achieves this saving with a technique called consumption logging. With consumption logging, the order in which the tasks consumes the data is written to disk and is therefore available after a restart of a task, allowing to follow that same order after a restart. The logging mechanism is pretty straightforward. A list is saved, with the `channelID`s of the envelope that the consumer read at this position. During restart of the system, envelopes are saved in a list at the position equal to the list with the `channelID`s. If the head of the list is filled the envelope will be used as input until an empty space is reached and the task has to wait for the next envelope in order.

Consider a task with four predecessors that produces its input data, as shown in figure 3.4. The task has four input channels, A, B, C, and D. The first producer processes data and outputs a large record that spans over three envelopes. The second producer fills one envelope. The task then first reads all three envelopes from the first producer before it checks for the next channel. It then reads the one envelope from the second producer. The third producer might be slower and might not have yet filled any envelopes. Then the fourth producer sends an envelope which the consumer reads, In this case followed by the second producer again. The Consumption log would look like this: (A, A, A, B, D, B)

After a restart, the producers three and four may be faster and send their first envelope. Both envelopes are saved, but not yet forwarded to the User Defined Function (UDF). The first three envelopes of the channel A are received. These three envelopes can be sent directly to the UDF, as they are the first on the list.

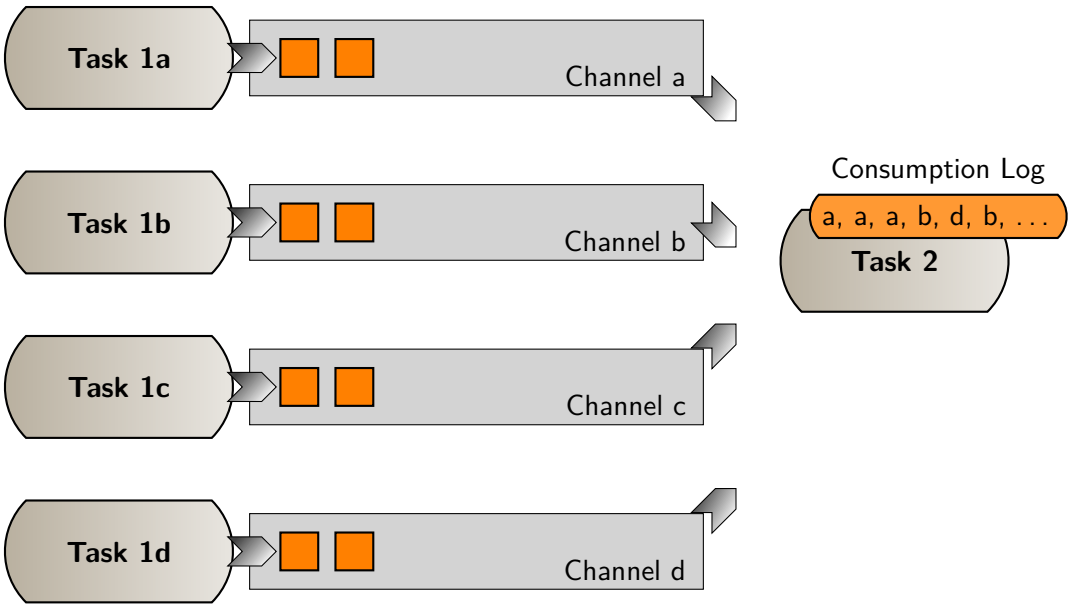

Figure 3.4: Consumption log

However, the saved Envelopes from C and D are still not forwarded as the envelope from channel B is still outstanding. Once an envelope from Channel B arrives the envelope from B and D can be handed to the UDF. The envelope from channel C however, has still to be saved until the second envelope from channel B was received. The consumption log saves any incoming envelopes during that period. After the second envelope from B has arrived, all saved envelopes are free to be processed in the order of their arrival. Note that the order between the different channels is not important at that point anymore, as the data in those envelopes is new data.

This mechanism could raise a deadlock. Each task has only a limited number of memory buffers and thus a limited number of buffers it can save before it blocks. Thus if all available memory buffers have filled up before the envelopes at the head of the list arrive, the task would block and stop accepting new envelopes, which it would need to free the buffers. Therefore, during the consumption logging replay, the buffers are saved to disk to keep the memory free for upcoming envelopes.

Preventing Restarts

With this approach, all restarted tasks are replayed in the exact same way as in the previous run until they are are in the pre-failure state. This part of the execution, the replay from the initial state to the pre-failure state, is now not only determined but deterministic. Note that the tasks are not forced to any order once they produce data beyond the failing state. From this moment on, the arriving data is new to the tasks and thus new to the successors and thus can be read and processed in any order. Of course, the consumption logging is still going on, and the order of the reading is still logged, to be prepared for any other failure in the system.

Depending on the position the failing task has in the graph and the positioning of the materialization point, the consumption logging technique can prevent a high number of restarts while being a lightweight solution with a negligibly small overhead during a nonfailure run. As this method is preventing restarts of tasks that have their position behind the failed task, it will naturally have the most benefit the earlier in the job the failure occurs, as it has the more following tasks.

Additionally, the consumption logging gives the opportunity to cover another kind of failure and skip flawed records. This possibility and technique is described in the following chapter 4.1

3.2.2 Global Consistent Materialization Point

As the decisions where materialization points materialize is made more or less uncoordinated (section 3.3 covers the details of the decision-making process), it is necessary to find the suitable materialization points for recovery in case of a failure. This set of materialization points is called a *global consistent materialization point*. This global consistent materialization point is the set of materialization points that includes all inputs for the failed task and the tasks that have to rollback due to rollback propagation.

Definition

A global consistent materialization point is defined as follows:

A global consistent materialization point in a DAG $G(E, V)$ is a set of local materialization points, which cover all paths from the failed Task ($f \in V$) to the data sources ($S = \{s \in V | \forall v \in V : \nexists e \in E : e = (v, s)\}$).

A recovery-path p is defined as
$p = (v_1, ..., v_f) : v_1, ..., v_f \in V, v_1 \in S \wedge \forall i \in \{1, ..., f-1\} : (v_i, v_{i+1}) \in E$

Param	Description
G(E,V)	Job graph as directed acyclic graph with edges E and vertices V
f	Failed Task
S	Datasources. The vertices reading the input for the overall job
p	Recovery-path in the DAG. Any connection between a Datasource and the failed Task
M	Set of Materialization points. Tasks writing outputs to disk
K	Global consistent materialization point
$\bar{P}$	Set of all Paths containing f

Table 3.1: Parameters

As we can define a partial order $(V, \leq)$ (according to the direction) we have the following requirements for the global consistent materialization point $K \subseteq M$ with $M \subseteq V$ as the set of all materialization points, and P the set of all recovery-paths.

$$\begin{array}{lrll} \mathit{Correctness} & \{\nexists k \in K & : f < k \wedge f = k\} & \wedge \\ \mathit{Completeness} & \{\forall p \in P, \exists k \in K & : k \in p\} & \wedge \end{array}$$

Correctness:

To find that global consistent materialization point, it is necessary to prove that it does not include a failed task. A failed task is not able to replay and is thus, not suitable for recovery. A global consistent materialization point must not contain the failed task, or any follower of the failed task.

Completeness:

Secondly, all input of the failed task has to replay, and the materialization point has to include a local materialization point for every path that contains the failed task. If a path exists, that includes the failed task, but does not provide a materialzation point, the replaying will miss part of the input.

Search

Finding a global consistent materialization point is done by breadth-first search upwards the graph, starting at the data source. In this search, the recovery logic marks all tasks without materialization points to be restarted, and keeps every materialization point for recovery.

An ephemeral materialization point can have one of four states: *UNDECIDED* if

it has not yet made the decision *NONE* if the decision was not to write data, *PARTIAL* if the decision is to write the data, but the task is still producing data, and *COMPLETE* if the materialization point materializes the data entirely

Ephemeral materialization points that have not been decided yet, that are found by the search are triggered to make their decision. The materialization point makes this decision based on the profiling information collected so far. This method avoids restarts of tasks that would decide to materialize their data.

3.2.3 Task and Machine Failures

The recovery mechanism described above focuses on task failures. The system detects task failures by caught exceptions. The engine catches the exception in the thread that started the UDF, marks the task as failed, and starts the recovery process.

If the task fails before it has received any data, it will restart without any other rollback propagation. As the task did not produce any data, it is not necessary to restart any other task.

In all other cases, the recovery logic has to find a global consistent materialization point. Afterwards, the rollback has to propagate to all tasks between the failed task and the found global consistent materialization point. All those tasks have to set themselves to their initial states. The restart has to be propagated to all followers of the failed task as well, because the tasks may not read the data in a deterministic way, and thus may not produce the data in the same order as in the first run.

Machine failures are special cases of a task failure, where all tasks from one machine fail simultaneously. A machine failure is detected by a missing heartbeat. Every worker periodically sends a heartbeat to the master. If a heartbeat is missing, the master marks the worker and all tasks that where running on that worker as failed. The master removes the worker from the list of available worker nodes, until it receives a heartbeat from the machine again. The recovery process, for this, is similar to single task failure. The recovery logic has to calculate the restarts and replays for every failed task. The restarted tasks are deployed to a free machine if one is available. If no machine is available it marks the job failed.

Additionally, the system has to deal with the fact that every locally written materialization point from this machines becomes unavailable. However, this is just an issue if a materialization point was complete. As all tasks from this machine are failed, they are not suitable for replay, and the system would discard the materialization points anyway.

The recovery can, however, use complete materialization points for recovery in case of a machine failure. It is therefore sensible to transfer locally written materialization points to a distributed file system once the task has finished and completed the materialization point.

3.3 Materialization Decision

As described above, the decision whether to make an ephemeral materialization point permanent is made locally based on the task's and job's characteristics. The primary challenge is to make a proper decision, whether to write a materialization point or not. As the engine does not know the tasks before they are running, it is necessary to observe the task during execution. The tasks characteristics are monitored during the execution of the task until no more data can be kept in main memory.

3.3.1 Monitoring

There are several possible characteristics of a task that the system can monitor. The Usage of Memory and CPU, and the I/O operations to disk to name just two. However, not all monitoring results may be suitable information for a decision about materialization points. The central question at this point is which characteristics of a job are essential to optimize the positioning of the materialization points.

Materialization points are supposed to be at positions that are particularly helpful during recovery. A materialization point covers previous executions and prevents those to be re-executed. It will prevent all tasks before the materialization point from restarting if it participates in the recovery. A materialization point decision is always made for one particular task. Therefore a materialization point is especially valuable if it covers tasks that are hard or expensive to re-execute. Thus a materialization point decision should take into account the cost of re-execution of a task in some way.

The other aspect of materialization points is that they ideally should themselves be inexpensive, both during writing and during recovery. The cost of a materialization point is dominated by its size, as the cost of writing and reading the materialization point defines its cost. Thus, the size of data that the materialization point writes to disk is an important factor in the decision. However, as the task is monitored during runtime and there is no information of the amount of data that will run through a task at this point it is not possible to make a decision based on the absolute size of the data that would be written in the MP. Therefore the size of the materialization

point has to be estimated in some way. The estimation at this point is made by the ratio of input and output of a task. The idea behind this estimation is that a task decides to materialize its output if it reduces the data noticeably.

Thus the monitoring concentrates on the data input and output of a task and the CPU usage. Even though the information about network and memory usage is collected to present it to the user of the system, these factors do not influence the decision. The characteristics that affect the decision are the user-CPU time, the number of received bytes, the number of bytes sent and the distance to the last materialization point.

3.3.2 Decision

Data should be materialized if

- materialization of the data is inexpensive ⇒ rate of In-/Output is high
- reprocessing is costly ⇒ high CPU usage

  If both are undecidable, the default decision is not to materialize, unless:
- the task is part of more than one path in the graph

In general, there are two types of tasks to consider. One is the pipelined task that streams at least one input through the processing. The task processes every record of the input on its arrival, produces the output immediately, and pipelines it to the consumer of the output. The other task type is the pipeline-breaking task. This task has to receive its entire input before it can start its execution.

The pipelined task allows an assumption about the multiplicity of a task. The ratio of input and output size indicates the increase or decrease of the overall amount of data at the task. In contrast, the pipeline breaking task makes it impossible to make such an assumption. As it consumes all input data before it produces the first amount of output data, it is not possible to translate that ratio to a multiplicity. In case of a pipeline breaker, the average size of the input and output records is used. If the record size decreases drastically after processing it is assumed that the size of the entire output will be comparably small.

For the decision, the system uses thresholds that the user can configure. The user can define a lower and an upper bound for the CPU usage and the input/output ratio. An ephemeral materialization point will change its state to permanent if the rate of input and output is higher than the upper bound and discarded if it is smaller than the lower bound. If the ratio lays between the given bound the CPU usage is

used for decision making, and the materialization point will be made permanent if the CPU usage is higher than the configured CPU boundaries.

If the CPU usage is under this value, the decision logic checks the position in the graph. If the task covers more then one output path (i.e., it has more output gates than input gates), the materialization point writes its data to disk. Because this task can be a global consistent materialization point for faults in both paths. Thus, it is beneficent to write a materialization point if the cost boundaries do not disallow it.

Additional Hints

The user or upper processing layer can additionally give some hints to the fault tolerance mechanism.

- The materialization decision for a task can be switch to true or false
- Materialization can be switched off completely
- Materialization can be forced for all tasks.
- The size of input and output records can be propagated to the system

If the user forces a decision, the execution engine applies it without checking other conditions in the profiling. The hint described in the last point offers the possibility to make measurements more accurate. The user can pass the size of each consumed and produced record to the materialization system. This information provides a better base for the decision, especially for stream-breaking tasks.

To avoid the domino effect, there is a coordinated component in the decision-making process. Each task does its own profiling and may come to a decision. As a typical distributed pattern in job graphs is the $n : n$ distribution it would be beneficial if all tasks in one group came to the same conclusion. One task that comes to a negative decision may make the other materialization points useless for recovery. Therefore the first task deciding it materialization will force this decision to all other tasks in the group, even though it may be the only task coming to this conclusion.

Enforcing the decision onto other group members has the benefit that it avoids the case where a few tasks would not materialize and thus make all additional materialization overhead useless. However, one could argue, that it may also lead to the case where one task would decide to materialize even though all other tasks, would determine that this is too expensive and thus increase the overhead. Although this is theoretically possible, it does not occur as the task would come to this conclusion

because of its high selectivity. If only one task in the group is highly selective, it will not produce as much output as the other tasks. As a task comes to its decision once it is not able to hold any more data in memory, the less selective ones would come to a decision first.

3.4 Implementation

The implementation of the general approach of ephemeral materialization points is prototypically implemented in the Nephele execution engine described in section 2.3. Even though this implementation is done in one particular system, the general idea can be adapted to other systems and be implemented similarly.

As described above, the materialization of intermediate data has to start from the first record. Therefore all produced data is kept in memory until the space is exhausted. Nephele sees data as records. Records can be anything from a single integer to complex types combining several objects. A record can be implemented by the user if he implements the given record interface. The interface includes a read and a write method for serialization purposes. This write method serializes the produced records into `ByteBuffers`. Those buffers are sent to the next task. To send a byte buffer it is wrapped in a `TransferEnvelope` which includes additional information about the sender and receiver, and a sequence number.

Each computing node has a fixed number of buffers used for serialization. During the execution, each task requests an empty buffer every time one is needed. At the beginning of the execution of the task, the filled byte buffers are not sent to the consuming task immediately but held in memory until all buffers fill up. This is not only used for fault tolerance purposes but also for lazy deployment and optimization. During this time the tasks behavior is profiled. Once the task has filled all buffers, the profiling information suits as input for the decision making process.

If the decision is to discard the ephemeral materialization point all filled buffers are freed, and the task sends upcoming buffers to the consuming tasks without further actions.

If the decision is to change the materialization point from ephemeral to permanent, the system writes the buffers that are still in the memory to disk. It starts a write thread, which receives the `TransferEnvelope`s and writes them to a file in the local or in a distributed file system. The user can configure the directory, in which the system should save the materialization points. For every materialization point, the writing thread creates a metadata file and a file that contains the data. Each file is named with the prefix “cp_" indicating it as a materialization point, followed by the `vertexID` and a suffix. The metadata file is an empty file, which has a

suffix that indicates the status of the materialization point and is either “_partial" for an unfinished materialization point or “_final" for a completed materialization point. The system saves data for a materialization point in one or more files with an incrementing suffix “_0" - “_n". These files contain all `TransferEnvelopes` from all output channels.

3.4.1 Consumption Logging

The logger that writes the consumption log takes action every time the task receives an envelope. Any `InputChannel` that receives a `TransferEnvelope` will report this to the *EnvelopeConsumptionLog*. The `ConsumptionLog` stores the `gateIndex`, the `channelIndex` within the gate, in an integer. If the task receives several inputs, the inputs are received over different gates. The gates collect the input from several parallel instances from the input channels. The integer holds the gate index in the first 7 bit and the channel index in the following 24 bit. Additionally, it stores a boolean, that indicates whether the data of that envelope is available. The logger does not use this boolean while it writes the consumption log. The recovery logic will use it when it replays the consumption log. The logger writes the integers into a fixed size `IntBuffer`, and writes the buffer to disk every time it is filled. It writes the file into the given temporary directory, and each file is named with the “cl_" prefix, indicating that it is a consumption log, followed by the `vertexID` of the writing vertex. The naming of the log file enables a restarted vertex to find a consumption log of a previous run. After the logger has stored the needed data in the buffer, it announces the availability of data to the input gate.

If a task restarts, it searches the temporary directory for a consumption log file with its `vertexID`. If it finds the file, the order of the read envelopes are replayed according to the log. Therefore the logging mechanism loads the first buffer from the file. After the restart, equal to the initial run, the channels announce any received envelope to the consumption log. However, in contrast to the initial run of the task, the consumption log replay saves envelopes in the `InputChannel` without reporting to the input gate that data is available. Usually the `InputChannel` informs the `InputGate` that it has data available once it receives an envelope. During the consumption log replay it is necessary to replay the order in which the channels announce the availability according to the initial order. Therefore the `InputChannel` saves the received envelopes, and only announces data availability when it is it’s turn.

The consumption log checks whether the announced envelope is the first in the list of the log. If so, it announces the availability of data from that channel to the input gate. Otherwise, the logger only marks the entry in the log as available. It will set the mentioned bit in the corresponding integer. Once the first envelope in the list arrives, the logger can report the availability of it immediately. Afterwards it

checks the next entry in the list. If it is marked available it can be announced and the next entry is checked until a entry is not available, and no further envelopes can be announced until it arrives. If the last envelope in the log is announced, the other saved envelopes are announced corresponding to the order of their arrival. Note that the order between the channels is not important at that point as, those envelopes contain new data, that was never seen by the succeeding tasks. However, the consumption log will go on logging the next envelopes as described above, in case another failure occurs. The log, only containing integers, is a lightweight solution for consumption logging.

Algorithm 1 Consumption logging algorithm

```
1:  procedure REPORTENVELOPEAVAILABILITY(inputChannel)
2:                                  ▷ The inputChannel that received an Envelope
3:      outstEnv ← loadOutstandingEnvelopes
4:      if outstEnv ≠ empty then                ▷ Still Envelopes to handle from log
5:          for IntEntry candidate : outstEnv do
6:              if candidate.getChannel() == inputChannel then
            candidate.setDataAvailable();
7:              end if
8:          end for
9:          for IntEntry envelope : outstEnv do
10:             if envelope.hasDataAvaiable() then
            envelope.getChannel().announceAvailability();
11:             else
            break
12:             end if
13:         end for
14:     else
            inputChannel.announceAvailability();
            announcedEnvelopes.put(IntEntry(inputChannel);
15:     end if
16: end procedure
```

3.4.2 Materialization Decision

As described in 3.3.1 the decision whether to materialize data or not depends on different profiled data and thresholds. The system implements the statistics about the size of input and output in three different ways. First, the reporting of the data ratio from the upper PACT-Layer, which collects the size of each record given to and the size of each record emitted by the user code and computes the rate.

If this information is not available the Nephele framework collects the statistics using

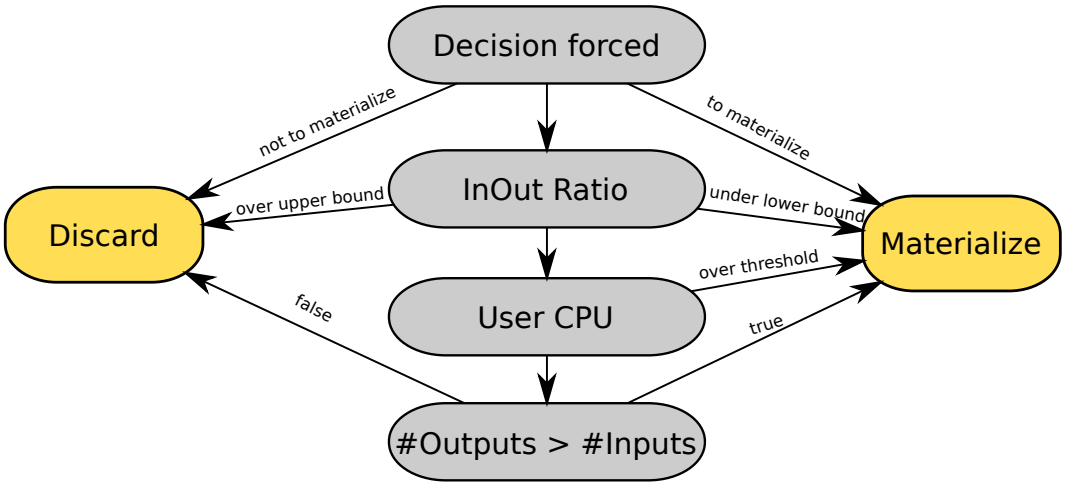

Figure 3.5: Decision process

monitoring data. Therefore the system collects the number of bytes received and sent over the network by each task. It sums the number of bytes contained in each *ByteBuffer* the task sent and each *ByteBuffer* the task receives during processing and build the arithmetic ratio.

That approach works well for tasks that stream data. If a task is a pipeline breaker and has to collect all its input data before it starts, this mechanism is not suitable. In this case, the ratio would be between the total input size and the first portion of processed output and thus misleadingly indicate a decrease of the data size.

To handle those cases correctly, the system checks whether the task already received all input data, i.e., if the task has closed all incoming channels. To be able to make an assumption about data size, in this case we compare the average size of the incoming and outgoing records, by collecting the number of records which have been serialized or deserialized from or to the *ByteBuffers*.

For the decision the collected ratio will be compared to the thresholds, a materialization point writes its data to disk if the ratio is smaller than the lower bound. If the ratio is higher than the the upper bound it discards the data. In between these bounds we use the CPU usage for the decision. The *ThreadMXBean*[38] interface provides information about the amount of CPU used by the user-code. We consider a task to be a CPU bottleneck if the CPU-user-time is over 90% of the overall CPU usage and write a materialization point in this case.

3.4.3 Rollback

Two possible scenarios trigger the recovery logic. One failure is a missing heartbeat of an instance. The instance manager checks periodically if it has received heartbeats from all instances. If it misses a heartbeat, it reports the corresponding instance as dead to the scheduler. If the vertex was writing a materialization point, the scheduler sets the status of the vertex to NONE, as the checkpoint is not usable for recovery. Afterwards, it marks the vertex as failed.

The other possibility is that the user defined function throws an exception. In that case, the `RuntimeEnvironment` which is responsible for the execution of the UDF catches this exception and reports the failure to the master. In both cases, the corresponding state listener starts the recovery logic. As described before, the recovery searches for the last global consistent materialization point. It begins at the failed task and explores a breadth-first search against the stream direction. If the preceding task has written a materialization point, it will replay its data, and if it is undecided, the recovery logic demands the decision immediately. If the predecessors have decided not to write a materialization point, they have to restart.

After the breadth-first search, the recovery logic holds a set of vertices, which form a global consistent materialization point and a set of all vertices between those vertices and the failed one. The latter ones restart. Therefore each of those tasks need to be canceled first. All canceled tasks, the failed task, and the tasks that will replay their materialization points are then deployed again. During the deployment, the `TaskManager` checks whether a task with the same ID is already running. If so the task is wrapped by a `ReplayTask`, which takes care of the replaying while the original task keeps running. If it finds no original task running, it checks whether a completed checkpoint is available. If it finds a complete materialization point, the system starts the task as ReplayTask, if not, the system deploys the task as usual.

At this point not only that task has to restart, but any task down the stream from the materialization point. This step is the rollback of the processing to the last consistent state, and replay all events, that the materialization point saved.

Reading from Materialization Points

During recovery, the system reads the materialization points. It can recover with both complete and partial materialization points. To be able to read from partial materialization points, i.e., materialization points that currently receive and write data to disk. materialization points are split into several files. Each finished file can be read during replay. If all available files are read, the reader waits for the next file to be finished.

As a materialization point contains the `TransferEnvelope`s that are usually routed to the corresponding channel and send over the network, reading a materialization point means to deserialize the TEs, find the correct channel, and sent it over the network. Recovering a materialization point that has not yet finished writing means that the producer task that is writing the checkpoint still produces new TEs. As the replaying is active, the recovery logic has to keep newly produced envelopes from being sent over the network. Otherwise, the task would send them twice, and the receiving task would discard them anyway as they have too high sequence numbers.

The writing and sending of TEs in Nephele are done with a so-called *ForwardingChain*; this chain is a modular sequence of processing steps that the envelope goes through. In the normal run case, an envelope enters the chain; the first step is to write the envelope into the materialization point on the disk. Afterwards, it is forwarded to the next part of the chain, which sends the envelope to the network. The final step of the chain recycles the buffers. In case of recovery, the recovery logic removes the network sending step from the forwarding chain, and envelopes only go into the materialization point and are recycled afterward. The replay functionality is the only process doing network transfer in case of a replay.

3.5 Evaluation of Ephemeral Materialization Points

In order to evaluate the functionally of materialization points, there are two main questions to be answered.

1. Will the dynamic materialization decision find the “right" tasks to materialize?
2. Will it perform better or equal to other materialization approaches?

The first question is not too easy to answer because “right" is not directly measurable. However, assuming the programmer knows the job and its characteristic complexity and selectivity of tasks, and that he would make a suitable decision, the example tasks are analyzed and the best-assumed positions from the programmers view are compared with the actual choice of the system.

The evaluation answers the second question by comparing different versions of materialization; one is to materialize at every task in the job, the second not to materialize only if the data flows over the network, and the third to make the decision dynamically as described here. Note that in contrast to MapReduce materialization, the Nephele system still pipelines all the data during materialization. This pipelining is already expected to have a significantly smaller overhead.

These materialization approaches compare four different failure scenarios, each with errors at different times in various tasks. The errors will be triggered at the first quarter, the half, and after three-quarters of the failure-free runtime. Additionally to that, the runtime without failure compares to the three variations and the runtime without writing any intermediate data at all.

As examples serve two different jobs, the TPCH-Query3 and a triangle enumeration job, written in PACT described in the following.

3.5.1 Triangle Enumeration

Enumeration of triangles is a typical job for social networks. It is used to indicate indirect connections between users or cumulative interactions in the network. The algorithm finds those connections by finding triangles in a connection graph. In order to do so, two steps are necessary: Finding those edge pairs that have one node in common, and from that checking if the edge between the other two nodes exists in the task[39]. The job is written in PACT and runs on sample data from the 2009 Billion-Triple-Challenge [1] which was converted to integers.

The PACT job for the enumeration of triangles has nine individual tasks. After reading data from a file, each row in the file will be a record that represents an edge in the graph. Then all edges ([a,b]) are projected to two edges for each direction of the edge ([a,b][b,a]) and handed to a reduce task which counts the edges for each node and appends the node to the record.

The next reduce step joins those counts and then on one side a task removes the counts from the records. This is necessary to get just the edges in the record for the final match operation. On the other side, the task sorts the edges, so the node with the smaller degree is first in the record and therefore the grouping-key for the following reduce, which builds triangle-candidates from the given edges.

In particular, for each tuple of edges [x,y][y,z] the job processes a tuple [x,z] as a possible existing edge to close a triangle. In the last match step, these edge-candidates are matched with the existing edges of the graph to find the existing triangles.

The interesting part of this job is that the task, which builds the triad candidates highly increases the data size and a user would never choose it for materialization. Therefore it should not be materialized by our approach as well. Instead, the materialization should be made at the task after the *BuildTriads* task, *Close Triads*, where the output is relatively small, and the work that was done by the expensive

[1] http://vmlion25.deri.ie/

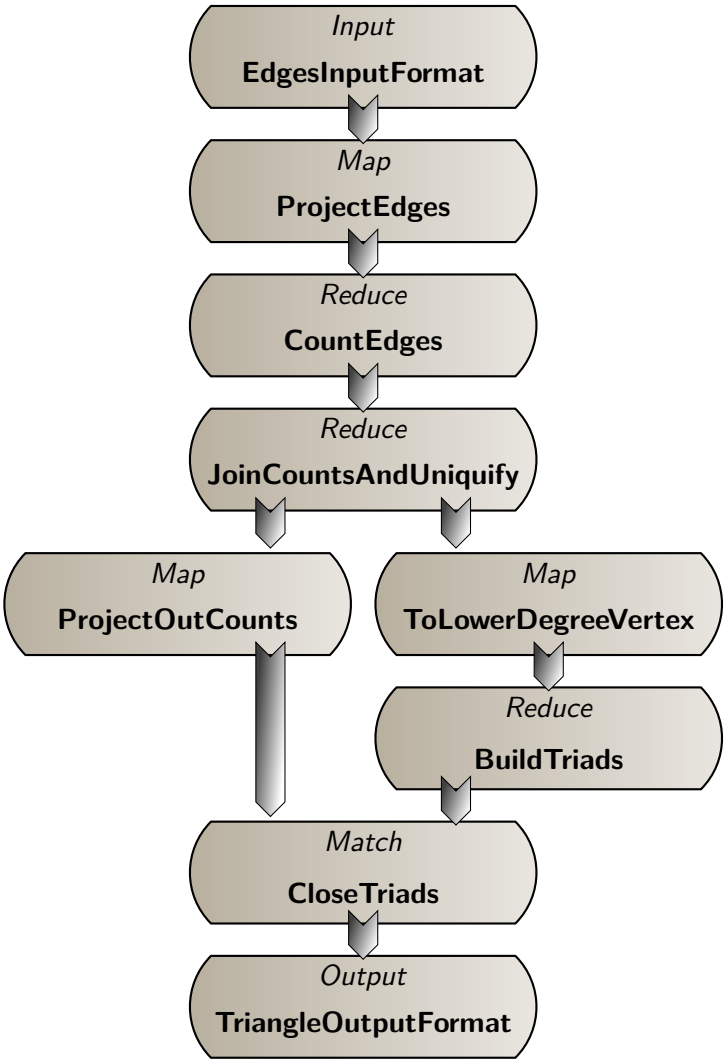

Figure 3.6: Triangle Enumeration JobGraph

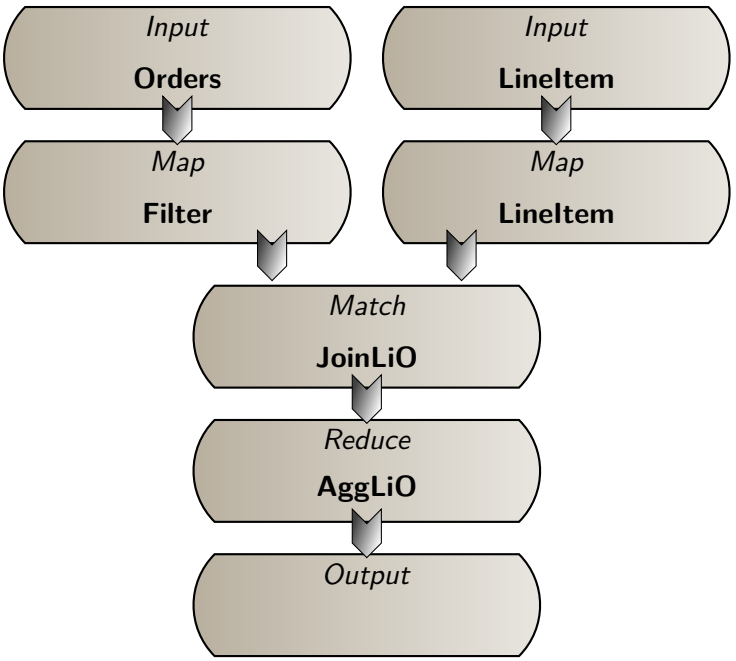

Figure 3.7: TPC-H JobGraph

BuildTriad task is saved.

The test runs show, that the dynamic decision approach sets materialization points at the tasks *JoinCountsAndUniquify* and *CloseTriads*. The materialization of the *JoinCountsAndUniquify* output is the optimal materialization point at a predecessor of the *BuildTriads* tasks because it covers both parallel execution paths (*ProjectOutCounts* and *ToLowerDegreeVertex*/*BuildTriads*). This is the decision reason for the algorithm, as well. The data did not grow over the threshold, and the task covers two paths. The decision for the *CloseTriads* is based on the data ratio. The task reduces the data, as it filters the actual triads from all triad candidates.

3.5.2 TPCH-Query3

Another test job to evaluate the overall performance of the ephemeral materialization is the Query3 of the TPCH-Benchmark[40]. The project source code contains the PACT implementation of the TPCH-Query3 as a PACT example.

The job consists of two map tasks selecting and projecting the input, followed by a

matching task executing the join and a reduce task for the aggregation. In the TPCH Query3 for PACT the join is implemented as a one-sided stream, e.g., the orders input is fully read from disk, hashed, and afterwards matched with the streaming input from line items. Therefore the *Orders* input is distributed to all nodes over the network. The output of the join task is distributed over the network as well. For data generation, the generator provided by the TPCH benchmark suite was run with a scale factor of 100.

The test shows that the dynamic decision approach positions the materialization points after the Filter of the *LineItems* input and at the aggregation step *AggLiO*. Both tasks reduce the data relative to their input data which brings the ephemeral materialization point decision to materialize the output data.

3.5.3 Measurements

The evaluation runs where started on our private cluster. Each machine is running a Quadcore Intel Xeon CPU E3-1230 V2 3.30GHz with 16 GB RAM running CentOS and devices are connected via 1 GBit Ethernet. An HDFS is spanned over the nodes and contains the input data; tasks write all temporary data (including the consumption logs) on local storage. The job was run exclusively on these machines, but not exclusive in the cluster.

To compare the efficiency of the ephemeral materialization points technique, the evaluation has to compare the behavior in case of failure.The main question is: Do the ephemeral materialization points reduce the runtime in case of a failure in comparison to other materialization techniques? The measurements shown in this evaluation compare three types of materialization: *ephemeral*, which is making a dynamic decision described above, *always* which fill force materialization at every task in the job and *network*, which will only force materialization on network channels. As the benefits of the techniques are highly dependent on the position and the time of the fault occurrence, runtime measurements are made for different tasks failing and at different times in a non-failure runtime.

Triangle Enumeration

All measurements were made with failures once at *BuildTriads* task, at the *CloseTriades* task, or the *TriangleOutput*. This makes it possible to see the difference in efficiency depending on the position of the failure in the job graph. The failures were triggered at on quarter, half and three-quarter of the failure-free runtime.

The runtime measurements in figures for the *Triangle Enumeration* in figure 3.8

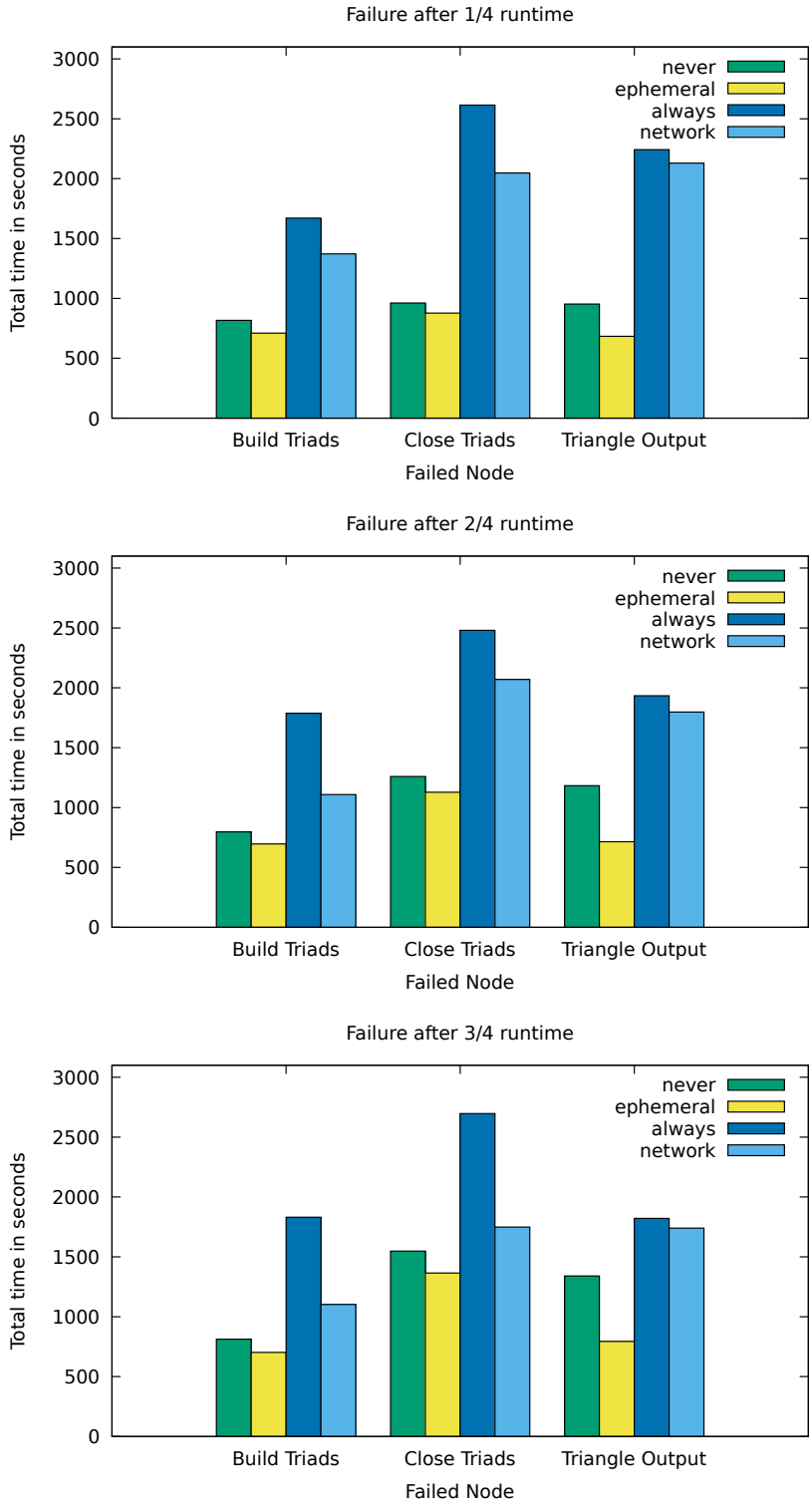

Figure 3.8: Runtime measurements for Triangle Enumeration

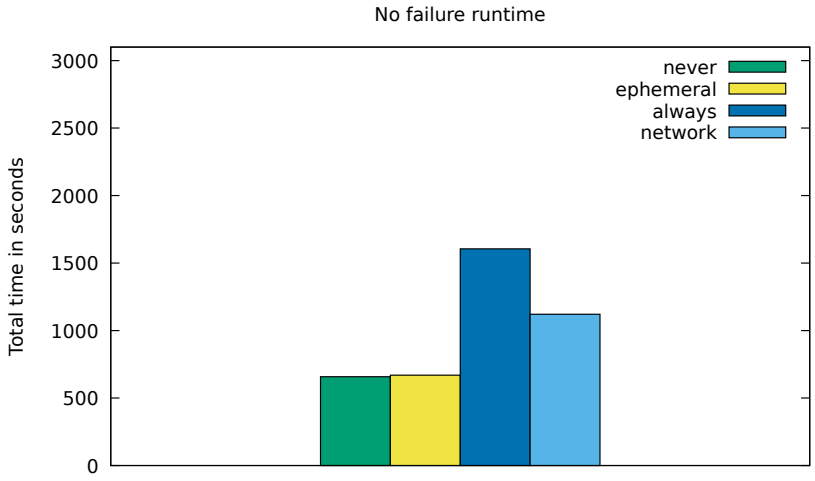

Figure 3.9: Fault free runtime measurements for Triangle Enumeration

show that the total runtime in case of a failure is significantly smaller than the other two materialization strategies. The dynamic approach ranges between 59% and 91% of the corresponding runtime without materializing, whereas the runtime for always materializing ranges between 135% and 271%, and the network approach between 113% and 223% of the runtime without materialization in case of a failure. The results show that the difference is way smaller if a task fails that is writing a materialization point itself. This is because the recovery logic discards the written materialization point and the restarted tasks will come to the same decision whether to write the materialization point, in this case the *CloseTriades* task. The overhead of the first part of writing adds to the overall cost. As assumed the materialization of data is especially beneficial if a fault occurs late in the processing.

Figure 3.9 shows the average runtime for job runs without failure. The measurments show that the dynamic materialization approach adds minimal runtime. The runtime is 102% of the non-materialization job runs. The other two materializing strategies have significantly higher overhead, *always* with 243% and *network* with 170% of the runtime without materialization.

TPCH

Similar to the evaluation for the first job, the evaluation for the TPCH job consists of job runs, for different materialization strategies, with failures at the *LineItems* task and the *JoinLIO* task. Again the failure occurred at one quarter, half, and three quarters of the fault-free runtime. The measurements also include the comparison,

of job runs without a failure.

The runtime measurements for the TPCH job show similar results. The tasks that where killed are the *LineItems* task, and the *Join* task. Again the tasks where killed after a quarter, half and three-quarter of the runtime.

The runtime measurements show that the total runtime in case of a failure is still smaller than the other two materialization strategies, although the difference to the “network" approach is not as significant as in the other Testjob. With these and other Test jobs we found that the advantage of our approach is higher the more tasks are in the tested job, the longer the job is running and the bigger the intermediate data is.

This is because the main advantage of the given approach is not to write every task's data and, therefore, reduce the writing overhead. More tasks in a job lead to a higher probability of tasks that are not chosen to write their output. On the other hand, a job containing only tasks which produce much bigger output data relative to the input may cause no intermediate data writing at all.

The measurements without failure show the same small overhead for the ephemeral technique. The runtime of the *ephemeral* approach is 101% of the runtime without materialization. In contrast to that, the overhead for the *always* strategy is 160%.

The evaluation shows that the ephemeral materialization point can reduce the runtime in case of a failure compared to the runtime without materialization. In comparison to other materialization techniques both the failure and the fault-free runtime are reduced. At the same time, the runtime gain in the fault-free case are shown to be negligibly small.

3.5.4 Evaluation of Consumption Logging

In this evaluation, the runtime of a job run with consumption logging is compared with the runtime without consumption logging in a failure-free case. This should give an indication for the over the runtime overhead of the consumption logging technique.

To measure the overhead of the technique, the evaluation compares the runtime of a fault-free job run once with consumption logging and once without. The job is an easy optical character recognition job, that takes pictures, executes a OCR, and builds a PDF from the recognized text. Figure 3.12 shows the job graph of the OCR job. The job reads picture files hand those to the optical recognition task that extracted the text from the pictures and hands them to the PDFCreator. The text is also split into words and sent to the Inverted Index task. As the names indicate

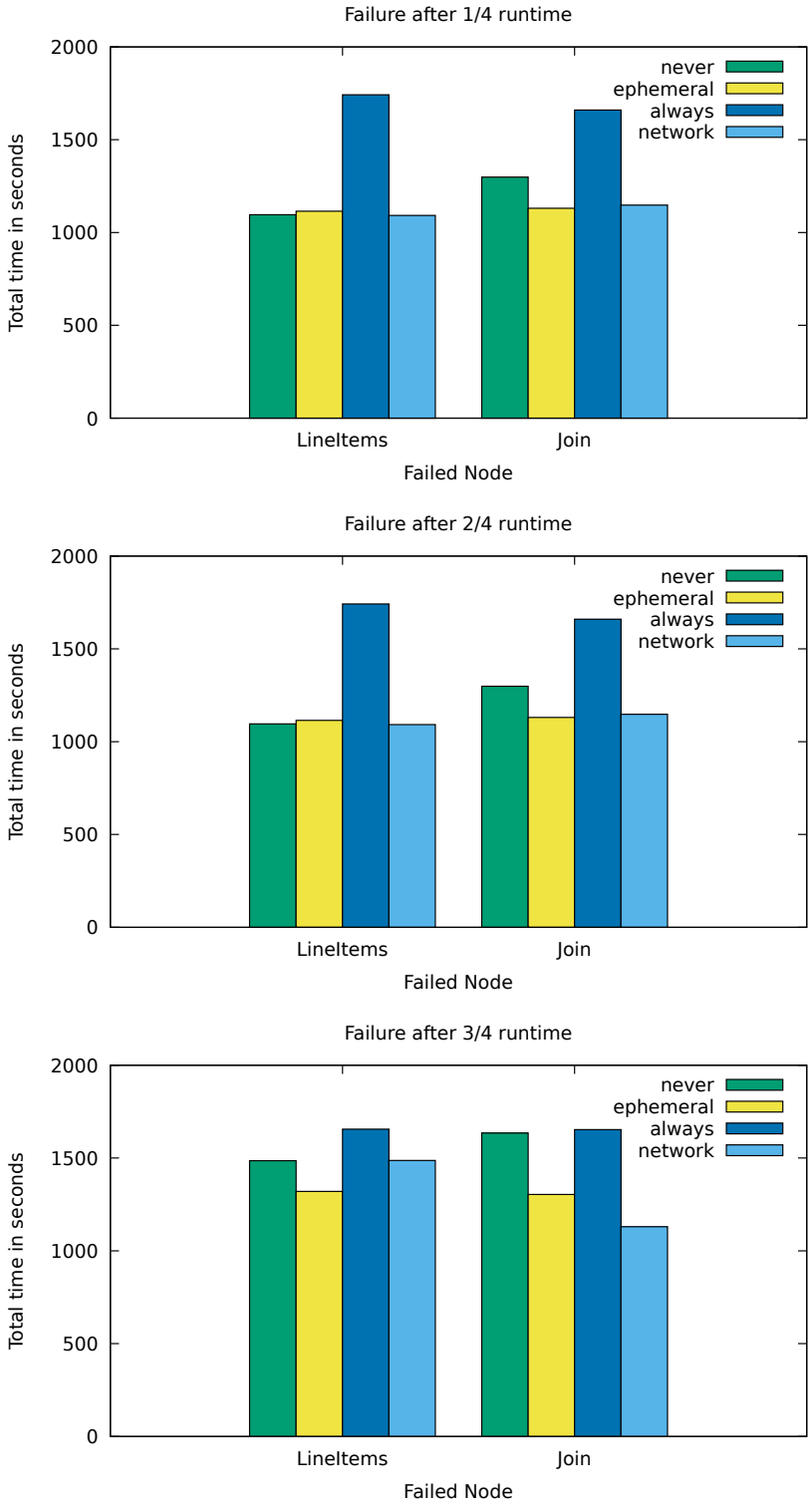

Figure 3.10: Runtime measurements for TPC-H

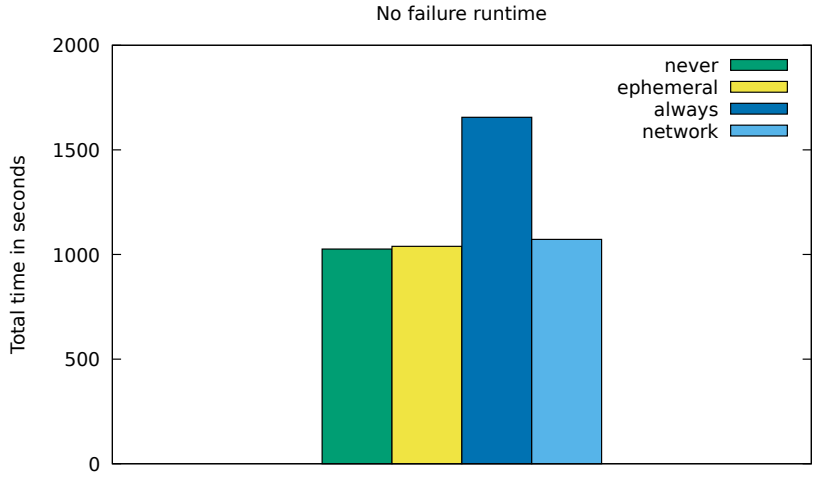

Figure 3.11: Fault free runtime measurements for TPC-H

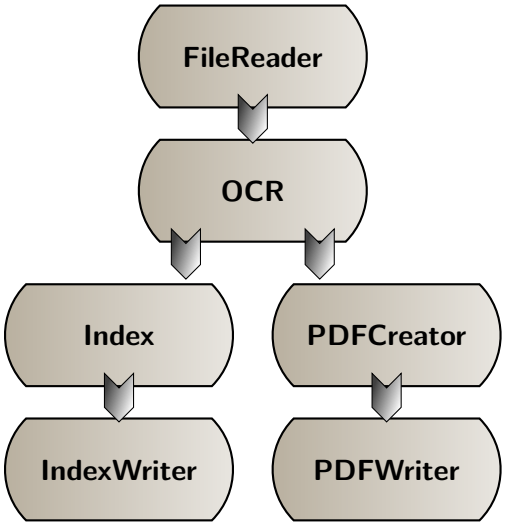

Figure 3.12: Jobgraph of OCR Job

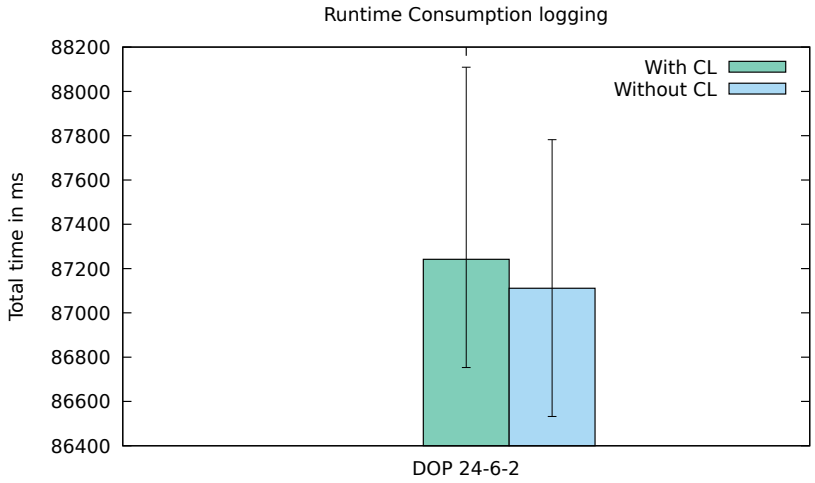

Figure 3.13: Runtime OCR Job with and without Consumption logging

the PDFCreator task creates a PDF file from the given text and the inverted index builds an inverted index over the entire input.

It is analog to the archiving job Derek Gottfrid wrote for the New York Times archiving[2]. The job generates searchable PDFs for articles from 18511922 given in a TIFF format. He wrote a Hadoop job which reads the 11 million articles data from storage and generates the PDFs. Gottfrid deployed the job to an Amazon EC2 Cluster. This job is interesting, because of the real world issue it solves. The archiving of data that is not available digitally is a common task these days.

The input was 477MB of data, in 515 .bmp files. The job was executed with different degree of parallelism and input sizes as indicated in the measurements.

Figure 3.13 shows the average runtime with and without consumption logging of failure-free job runs, for parallelism of 24 OCR tasks, 6 *PDFCreator* tasks, and 2 *InvertedIndex* task. The average overhead of the consumption logging method was 235 ms which is 0.26% of the average runtime without consumption logging. As can be seen in the graph, the variation of runtime between the usage of consumption logging and the job run without consumption logging is way smaller than the variation between individual job runs. The difference between identical job runs could come from the variation in network traffic due to other usages of the cluster.

In Figure 3.14 the DoP is changed to 3 OCR Tasks, 2 PDF Creator Tasks and one *InvertedIndex* Task. The difference in runtime was 468 ms in this case, which is 0.24% of the runtime without consumption logging.

[2] open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

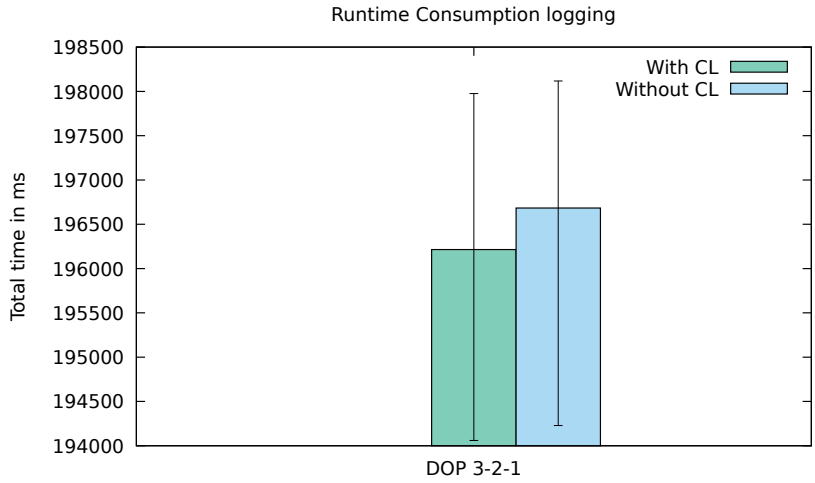

Figure 3.14: Runtime OCR Job with and without Consumption logging

The OCR job also runs with a larger data-set of Wikipedia articles. The data-set contains 16000+ random Wikipedia articles that were converted from HTML to BMP files. The size of the files ranges from 136KB to 30.6 MB, with a total of 4GB.

All measurements show that the consumption logging has a negligible small overhead and is thus an efficient approach to increase fault tolerance.

Additionally to the overhead from writing the consumption log there may be overhead for the recovery phase. As the engine is no more free in the distribution of data, it is forced to use the same pattern as before. During a failure-free run, the order in which the task reads data depends on data availability. Using consumption logging, the tasks are forced to read a fixed number of `TransferEnvelope`s from one channel at a time. This could cause the task to block even though other data in other channels are available. To avoid this, the incoming envelopes are written to disk during consumption log replay. This behavior could slow down the execution if the envelope that has to be read is not available.

However,this is unlikely to happen in practice. In contrast to a normal job run, the data of the restarted intermediate tasks are immediately available. The input is read from materialization points and is thus immediately available, and the task can process it directly. Nevertheless, network flaws or other outside events may cause such a slow down anyways. Unfortunately, this possible overhead is hard to measure. It highly depends on the job and the environment and is thus not comparable between different jobs. Therefore we are not able to quantify this possible overhead.

The evaluation shows that the consumption log technique has a small overhead,

while avoiding a possibly high number of restarts. Naturally the advantage of the consumption log approach is the higher the more tasks are behind the failed tasks. Without consumption logging all tasks that follow the failed task have to be restarted. The consumption log avoids those restarts. The measurements show that the overhead is negligibly small, this makes the consumption log a beneficial optimization for ephemeral materialization points.

3.6 Related Work

Rollback recovery with checkpoints is a standard technique to increase fault tolerance in distributed and database systems and is an often discussed topic[37, 41, 42, 43]. In massive parallel processing like MapReduce[6], Dryad[13], or Nephele[14] these techniques are inapplicable as they do not take into account issues encountered in this type of system, like vast amounts of intermediate data and the network as the scarce resource.

MapReduce[6] With the widely cited paper *MapReduce: simplified data processing on large clusters* Dean and Ghemawat introduced Googles MapReduce: A parallelization framework that hides the complexity of parallelization and fault tolerance. The MapReduce framework operates on a master-worker pattern, with the worker nodes executing either a map task or a reduce task. The map and reduce functions are written by the user, the framework expects them to be deterministic. The map tasks write their output to the local file system. The reducer workers can remotely read their input from the local file system of the mappers.

The fault tolerance mechanisms of MapReduce handle two cases: worker failures and bad records. Worker failures are detected as the master node pings each worker frequently. These worker failures have to be separated between map-workers and reduce-workers. A failed map worker has to be re-executed entirely, as the intermediate data is stored in the local filesystem and not available anymore. The reduce task will be informed about the change so that any reduce worker that did not read its input from the failed mapper can now read it from the alternative mapper that was re-executed.

As reducers will store their data in the global file system, it is not necessary to re-executed a finished reducer. However, a reducer that has not completed its work has to be re-executed. Reducers write their output in temporary private files and will rename the file to the final output file once they finish. If a reducer is re-executed even though it was struggling but not actually failed would not change the result of the job, as the renaming would be done on the similar file. This fault-tolerance method is based on the fact, that all intermediate data is saved to disk, even with several MapReduce jobs in a row. This prevents the system from restarting more

than one task, but it also has a high overhead. In contrast to that, the ephemeral materialization points, work with pipelined jobs.

Additionally, to worker failures, MapReduce can skip bad records. If a map or reduce task fails, it sends the sequence number of the current record to the master. If the master receives a failure for the same record a number of times it tells the next worker to skip this record. This will be discussed in detail in chapter 4.1

RAFT[44] introduces an advanced checkpointing technique for MapReduce. RAFT uses the intermediate data produced by MapReduce anyway and uses it to react to task and worker failures more efficiently. On the one hand, the RAFT concept pushes intermediate results to worker nodes, and on the other hand, keeps track of the offsets of input key-value to intermediate data. Thus the system can partially recompute data if necessary (in case of multiple node failures) and thus reduce the recovery overhead. Nevertheless, the RAFT paradigm relies on the intermediate data produced by MapReduce which already is a high-overhead method.

ISS[45] Ko et al. introduce an Intermediate Storage System (ISS) that aims to minimize the runtime overhead of MapReduce jobs, by optimizing the usage of intermediate data. ISS is implemented as an extension to HDFS which uses asynchronous replication, rack awareness, and selective replication.

However the described approaches aim to optimize the MapReduce materialization techniques, that materializes data after every task. The ephemeral materialization points try to avoid to save all intermediate data. The listed optimization could possibly adapted to ephemeral materialization points too.

SGurad[46] is a fault tolerant Stream Processing Engine(SPE), which uses rollback recovery as fault-tolerance mechanism. For that end, the SGuard system uses passive standby and checkpoints the state of the nodes in a distributed file system. In contrast to other passive standby techniques, SGurad avoids a full suspend of the operator during the checkpointing of the state. To copy the state of the operator to disk while it is still running, it takes control of the memory belonging to the stream operator. It partitions the memory into pages and copies those pages to disk using an application level copy of write. Pages are marked as read-only, and if an operator touches a not yet copied page, this pages is copied first. Thus the suspension time of the operator is reduced.

PPA[47] Li Su and Yongluan Zhou introduce a Passive and Partially Active (PPA) approach for Massive parallel SPE. They use passive checkpoints and backup nodes for all nodes of the parallel stream processing and active checkpointing for a subset.

In **“Storm @ Twitter"** Toshniwal et al. introduce Storm a real-time resilient distributed stream processing engine[48]. Storm can use an “at most once" or an “at

least once" semantic using acknowledgment of tuples. Despite the fact that storm is also a non-deterministic data flow, they introduce something similar to a sequence number. Each tuple is getting an initial number. After flowing through one task, each record produced from one record will get a sequence number, which computes from the parent records sequence number.

The authors compare their approach with handwritten java code. For comparison, they used storm once with and another time without message reliability mechanism. Although the authors argue that their approach without message reliability does not have significant overhead to native java code, the numbers show that the message reliability in their system adds three times more CPU utilization and need two more machines for the same message passing speed.

The last three approaches focus on streaming environments, which are out of scope for the work in this thesis.

3.7 Summary

This chapter introduced ephemeral materialization points as a low overhead materialization strategy, according to both runtime as well as to disk space usage. The system decides whether to materialize a task or not, based on the profiling of a task and predefined boundaries. The results show that the presented approach for materialization points finds the same tasks to materialize which a programmer would prefer for materialization. The approach is working without any knowledge of the user code or the input data before the job starts running.

The materialization and the recovery in case of failure are working in a nonblocking manner and allow therefore pipelining, seeing results incrementally, and continuous queries. A materialization point writes the data to disk or in a distributed file system. Afterwards, the engine sends it over the network to the consuming task. This way, the materialization point technique, ensures that the following tasks do not wait until the materialization point is ready. The experimental results show that this materialization method reduces the runtime during failure significantly compared to other materialization techniques that save data after every task.

This chapter also introduces the idea of consumption logging, which reduces the number of restarts during recovery. Consumption logging forces a deterministic input of data for a task even after a restart. This deterministic data stream enables a task to skip previously processed chunks of data and leads to the fact that tasks succeeding the failed and therefore restarted task do not have to restart themselves.

Evaluations show that this consumption logging does not increase the processing

time of a job noticeably, in case of a failure-free run. Still, it offers a significant reduction of restarts and thus reduces the recovery time in case of a failure.

CHAPTER 4

Data- and Software-Faults

Contents

As described in section 2.4.2 the ephemeral materialization points described in the previous chapter offer fault tolerance for transient and system-detectable failures. It is possible to recover from transient faults, by restarting a job or parts of the job. This is not possible for persistent faults. A fault that occurs on every run of a task causes the system to fail eventually, to show a failure in the output, or to run the recovery process in an infinite loop. To avoid the latter, recovery processes usually have a defined number of retries, after which the job finally fails, and the recovery re-tries end.

As described before, the transient faults can also be subdivided into system-detectable

and non-system-detectable faults. Even if a failure is persistent and appears at every job run it is not necessarily detectable. A UDF that produces a wrong output for a particular record, and will, therefore, spoil the job's output, may still run properly without any exception rising. The system is unable to detect the wrongdoing of the UDF in this case. And as the system is not able to detect it, it is impossible to recover from it automatically. The system must rely on the user or upper layer system to erase the failure. However, it may still be able to support the user in the recovery process.

The upcoming sections present solutions for data- and software faults. Where data faults are permanent faults that are detectable, and software faults are permanent but not detectable. Note that both fault tolerance techniques request a veer away from some of the desired design goals described in 2.5.

4.1 Data Fault Tolerance for Flawed Records

Massive parallel data processing engines are designed to handle a massive amount of data. One key aspect of BigData is that it can be flawed, and it is not easily possible to oversee the structure of the entire data altogether. Execution Engines and programmers can not rely on the assumption that data is typically well- structured and quality-controlled as they might have been in databases. Working with BigData means to accept data that is inconsistent and dirty. Even though the programmer of a job should keep that in mind, it is not always possible to foresee the flaws. Especially working with third-party libraries may cause unexpected errors.

Jobs that must handle data with some flawed records would usually fail entirely, using the typical re-execution attempt. Even after restart attempts, the job would still fail at the exact same record every time. Furthermore, the simple restart of a task would not even lead to the information that a particular record caused the failure. In a worst-case scenario, a job will fail after several restart attempts, probably giving the user or programmer nothing more than an exception coming from a third party library, with no knowledge about the structure of the flawed record.

4.1.1 Skipping flawed Records

Considering an exactly once consistency model, there would not be a chance to change this behavior. The job will ultimately fail at any given run until the negative record is changed or removed from the input. Still, optimization of this behavior would be to give the user information about the record that caused the fault. This would ask for the engine to identify and log the record. This would still result in

a failed job, but would probably give the programmer the necessary information to improve the UDF. However, if the job allows twisting the consistency model to a more open "at most once" consistency, it would be possible to recover from this failure and skip this record at the point of failure. This way the job would finish, at least with a part of the input data.

Therefore, the engine must be able to identify a record. What might sound trivial, can usually just be achieved with a high amount of overhead. As data can flow through the graph over any path, it is not possible to identify a record by its position in the stream. A task would not be able to decide that it is continuously failing at its input record number x as the record it receives at position x might be a different record on each run. The records would have to be identifiable by themselves.

Identifying Records

The most intuitive approach would be to give each record a unique ID. This ID, however, must be recomputable, as is needs to be the same between two computations of the record. A random unique ID would not be suitable for this application, as it would change between runs. A record would not be identifiable after the partial restart during recovery. As the identifiers have to be unique but equal on every occasion, it must depend on the records data itself. The data the record holds is the only part that identifies it uniquely. In practice, of course, it would be possible to have records holding the same data. However, this would be beneficial in this case, as records containing the same data will cause the same error. Nevertheless, creating an ID that depends on the records data gives just one possibility: computing checksums.

That means to compute a checksum for each produced record which will lead to high overhead. Depending on the record size, calculating a checksum can be very cost-intensive. That approach would add runtime for every single record in the system. Still, this method is a possible solution to the given issue and may be a suitable solution for some cases, especially if the job deals with a lot of data-identical records.

Note that the overhead for jobs that do have those failures is negligible. That is because the job would never finish without this method. Trying to compare an overhead increase of a checksum implementation to a failing job is impossible. The comparison of runtime can thus only be made by comparing the nonfailure cases. The question is: How much additional runtime do we have to accept if no failure occurs?

But fortunately, there is another possible solution, using the benefits of consumption

logging. As described in chapter 3.2.1 it is possible to force the data stream to be identical after a restart of a task. The consumption logging guarantees that each task receives the same data stream, with the same records in the same order on every restart attempts. That guarantee enabled the engine to skip the head of a reproduced stream if the task has already processed the data.

Additionally, to that the consumption logging technique offers the opportunity to skip records depending on their position in the stream. A task that fails because of one particular record restarts during the recovery. After the restart, the replay reproduces its input data either be reading it directly from a materialization point or by its predecessors which read their input from the materialization point. Using consumption logging, the input is reproduced identically to the original input. This means the task would fail, after the same number of records, and would again trigger the restarts and recovery.

Unfortunately, this is only possible for pipelined tasks. A pipline breaker as described in section 2.3.1 consumes all its input before it starts the processing of the data. Even though it might fail at the same record every time, the engine is unable to detect the record, as it has no access to the internal state of the UDF. Using record skipping for pipeline breakers would only be possible if the UDF hands the information of the currently processed record to the engine.

Recovery

As the inputs are identical, it is thus possible to identify a record by its position in the stream. The task can now count the number of records it has consumed and report the number of the record that caused a failure (if the system can recognize and handle the failure, of course). If a task reports the failure with the same record number several times, it can be notified to skip this record at the next try. The engine saves the record for a detailed failure report, which will give the programmer all he needs to figure out the problem.

This mechanism is not possible for all kinds of tasks. As described in section 2.3.1 tasks can be either pipelined or pipeline breakers. The characteristic of pipelined tasks are that they consume one record and output the results of the records immediately. Pipeline breaker, in contrast, consume all the input data before actually starting the processing. In case of pipeline breakers, the engine cannot link the failure to a specific record. From the viewpoint of the execution engine, the task fails after it has read its entire input. As pipeline breakers can be easily spotted during runtime, the records skipping is only be done if a task is pipelined. A data flaw that will cause a pipeline breaker to fail will, unfortunately, cause the job to fail.

After all, this mechanism would cause an extended recovery time. To be sure, that failure is caused by the record the system has to restart and recover at least three times. The first time the suspicious record can be reported, the failure, however, could have occurred for any other reason that has nothing to do with the record. For example, a flaw in memory, that might even occur regularly but not necessarily on the same record or might not even occur again. Skipping a record after the first occurrence of failure could mean to change the consistency model unnecessarily. If a task fails two times at the same record, is a good hint, that the record is the culprit. Still, the number of tries until a record is marked as flawed, can be left by the user.

Depending on the position and the number of failures, this might lead to a remarkably higher runtime than the original run. Notwithstanding, that this is a cost-intensive fault tolerance mechanism, it enables the system to finish a job that would otherwise fail entirely and not produce any output data.

Datastream After Skipping

Note that even though the record skipping changes the data stream after restarts, it does not go against the consumption logging. The consumption logging guarantees an identical stream after recovery restarts. However, the engine can only give this guarantee for data that has been seen previous to a failure. As a flawed record will create a failure at the exact same position in the stream, and the task does not produce any output for the flawed record, the skipping of a flawed record will indeed change the overall stream. However, it will not change the stream that has already been sent to the successors of the failing task. As any data that the task produces after the skipped record was not received by the successors of the task previously.

Thus, even though the overall stream of data is changed, compared to a fault-free implementation of the job, it just changes parts of the stream that are not affecting the consumption logging. Nevertheless, if an error occurs after the skipping of the record, and the stream must be reprocessed, the record obviously hos to be skipped again. On the one hand, because it would cause an error again, on the other hand, because the task has to reproduce its changed output stream.

Fault Treatment

The fault that causes an error that is masked with record skipping is a permanent fault. It will occur on every run of the job, at the same state. Using record skipping prevents system failure, i.e., the crash of the job. However, it does not prevent the system from failure at any other run of the job. To this end the user must remove the fault. This means to filter the flawed data from the stream or to change the

UDF to handle the flawed data in a suitable manner.

As the engine is handling black box user code, it is impossible to do this automatically. The user must change the UDF, add filtering tasks, or remove the flawed records from the original input. To enable the user to take up the correct actions it is important to give all information possible about the error that occurs. Thus, it is not enough to skip records. The system must save the skipped record to present it to the user. Additionally, of course, the detectable error has to be presented to the user. In this case, this will be most likely some Java exception, that includes a stack trace and probably a hint on the code line that cause the error and might be faulty.

However, the fault treatment should usually get even further by indicating the flawed part of the jobs input, and from there indicate the source of the flawed data. This could be one particular data producer, as input data for Big Data applications could come from different producers. Or it might be a fault in the data collection already.

Especially if a task has to skip not one but several records, the saved records can give important information about the structure of a record that is flawed. However, it is only possible to save the input record of the failing task. As there is no 1:1 ratio between records, an input record of a task cannot directly be linked to an input record of the overall job. Giving the user the opportunity to observe the records that activated the fault, can give him the ability to indicate the corresponding record in the original input and probably remove faults that initially lead to the input data.

If every skipped record contains the same value for one field, it can be an indicator that this value is not properly handled in the user code. This could for example be a converter that returns a null value for an unknown format. The fault may be a missing null-value handling. But for the quality of the jobs output it would be necessary not only to fix the null value handling, but to add a proper conversion for the format.

4.1.2 Implementation

The implementation of the record skipping technique relies on the previously introduced consumption logging. The introduction of the consumption logging ensures the that the input `TransferEnvelope`s (TE) received from all channels are always in the same order even after several restarts of a task. The system deserializes records included in each `TransfereEnvelope` in the same order. Hence the consumption logging naturally guarantees an identical stream of records. With this identical record stream, it is possible to identify the records by their position in the stream.

In the Nephele system, the *InputGate* deserializes the records. It receives TEs from

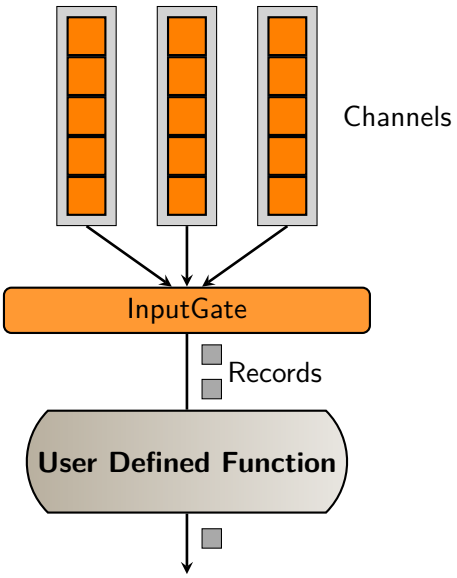

Figure 4.1: InputGate deserialization

the different InputChannels. In the first run, the TEs are consumed depending on the availability. After a restart the order of the consumption is replayed, as described before in chapter 2.4.3. The gate deserializes all fully contained records of a TE and passes them on to the UDF. At this stage, the records are counted. If the UDF crashes after it has received a record, this record's number is saved and reported to the `JobManager` in a failure report. This way, the `JobManager` can keep track of the number of crashes for a particular record. After the second time a UDF fails at the same record, the `JobManager` will advise the `InputGate` to skip the record. To do so, the `JobManager` passes a list of record numbers to skip to the input gate. During counting, the `InputGate` checks whether the record number is on the skip list and if so it discards the record and deserializes the next record.

4.2 Software Fault Tolerance

Jobs written for Big Data analytics may run very often on different kinds of data. However, they may not be sturdy forever. The data input may change, adding sources of data with a slightly different schema. Changed APIs of third-party libraries, bugs in the code, all this can lead to the additional development of jobs. However, these changes may only have an effect on single tasks in the overall job,

Algorithm 2 Record skipping algorithm

```
1: procedure ReadRecord(target)
2:     record ← deserializeNextRecord()
3:     num ← num + 1
4:     skipList
5:     if skipList contains num then
6:                                          ▷ do not return record, get the next
7:         record ← deserializeNextRecord()
8:         num ← num + 1
9:     end if
        return record
10: end procedure
```

and may not change any behavior of other tasks. For this, it can be beneficial to use materialization points between two job runs. Especially if it is unclear what part of the data exactly caused an error; it is helpful to start the job with the same data. If the beginning of the job is still unchanged, this could easily be skipped by using materialization points of a previous run and thus start the job partially.

We call this technique *memoization of materialization points*. This is analogous to the memoization in programming which caches previous execution results in order to speed up execution. In memoization, results of function calls are cached and returned if a call with the same input occurs. The name memoization is coming from the Latin term “memorandum" and should not be confused with the similar word memorization as *memoization* covers a specialized technique in programming. In some programming languages, is called memoization tabling.

4.2.1 Memoization of Intermediate Data

Similiar to the memoization in programming, the written materialization points from one finished job run could be cached and reused for another run of a job with similar parts and the same input. A job which is identical from the input up to a materialization point can start from the materialization point instead of the input. This can save the recomputation of parts of the job. This technique can be a beneficial tool in the development of a data analytics program. As programming faults, updates in third-party libraries, or faulty assumption about the data may ask for changes in parts of a job. In those cases, just single tasks have to be changed while the rest of the job and the input data stay the same. Then it is possible to use previous materialization points as the input of parts of the updated job.

In order to achieve the possibility to memoize materialization points two things are

necessary:

- The system must save materialization points beyond the lifetime of the job.
- The system must identify the job and its changes

The first point, saving materialization points after a job finishes is obvious. However, materialization points are typically written to the local file system, and removed after the job has finished. This applies to successful and failed job runs likewise. This is done to keep writing the materialization points fast and the usage of disk space low. Furthermore, writing materialization points to a local file system means it is not available once the virtual machine is terminated. To use the materialized data after a job run, the mechanism that saves intermediate data has to change. The intermediate data must be saved to persistent storage and not removed after the job ended. The engine can either achieve that by writing the materialization points to a distributed file system directly or by copying it to persistent storage once a job ended. However, this should not be the solution of a regular job run. Instead of saving intermediate data after a job has finished, it should be kept in development mode only.

Finding Job Similarities

The second point is far from trivial. Identifying a job as similar would automatically mean to ensure the input data is identical, the UDFs are identical and which UDFs have changed.

To ensure that the input is identical, we would have to compare the input of the new job to the input of the job with the memoized materialization points. Even though the basic idea sounds simple, it is unfortunately not. Obviously, the same path of the input does not guarantee that it is equal to a previous version of the file or the directory. A solution would thus be to compare both inputs directly. That would mean to keep not only the intermediate data but also a copy of the input data of the job to compare against. Fortunately, there are other solutions than just a byte-wise comparison of data. It is possible to calculate checksums for the data to assure it is unchanged. However, as we are dealing with BigData, checksum calculation might be a costly solution.

The same issue applies to the identification of the job. The user sends a job to the system as a jar file. It contains one or more class files that implement the job and all necessary classes. In the best-case scenario, the programmer writes each task in an individual file. In this case, it may be possible to compare each class and mark a task as updated, if a class differs from the previous job jar file.

It is possible to execute a simple diff of the two Java class files, but this method leads to some problems:

- First this does not cover dependencies of classes. As the task is associated with a UDF (i.e., an invokable class), changes of a class that the UDF depends on would not mark the task to be changed. The dependency analysis of java classes is a, ongoing field of research. Tools as *classcycle*[1], *Class Dependency Analyzer*[2] or *Dependency Finder*[3] work on this field.

- Second, it will not consider external libraries. There may still be third party libraries that may have changed, and thus change the output of a task.

- Third, the possibility to write inner classes in Java. Writing inner classes in Java will cause changed class files for all declared inner classes of a class even if only one inner class changed. This would lead to false-positives when searching for changed classes.

- Fourth, even with distinct class files, it can lead to false positives: Changes that do not change the behavior of a task may mark it as updated. Note that the last two issues do not change the correctness of the approach but may reduce the benefit noticeably.

Given all those stumbling blocks, it is evident that the identification of similarities and changes within a job are far from trivial. Therefore, the responsibility to identify changes and give the info about the reference job is moved to the programmer. Any changes made must be declared by marking the changed UDF or changed classes the task's UDF depends on. If a job is claimed to be a modified version of a previous job, any task that does not have the "updated" mark and executes the same class (identified by its name) will be considered to be equal. With the usage of the memoization technique and the indication of a reference job, the system also expects the user to guarantee that the input data is unchanged.

Note that the input of the user makes this fault-tolerance technique violates the transparency requirement. However, the technique may be useful for upper layers in the programming stack. If an upper layer like PACT for the Nephele framework runs the job, it may be able to provide the required information without knowledge of the user. The development mode might then be useful in the context of trying different execution plans. The upper layer can change the execution plan without losing the work done before the changes.

[1] http://classycle.sourceforge.net/index.html
[2] http://www.dependency-analyzer.org/
[3] http://depfind.sourceforge.net/

Using saved Materialization Points

Once an updated job is sent to the system and the updated tasks are identified, it can reuse all intermediate data from predecessor tasks. The system cannot use any intermediate data that a changed task has produced, as the intermediate data that the updated task will produce is not equal. Reusing the materialization points from the reference job is similar to the recovery of the new job considering the first updated task as failed. During recovery, the recovery logic searches for the last global consistent materialization point upwards the stream and replays it. The failed task instance receives its entire input from its predecessors or directly from the found materialization point.

After the engine received an updated job, it will also search for a global consistent materialization point. However, in this case, the search only allows completed materialization points. If the search finds a suitable materialization point, the corresponding task will replay. All predecessors of the replaying task will be initially marked as finished and thus will not start processing at all.

Note that this approach does not work with partial materialization points as the system cannot predict which parts of the input are already present in the materialization point. As the states of the task are not saved, the new task has no knowledge which envelopes have been completely processed by the previous task and must process all envelopes again, discarding the already written data. Even with a previous consumption log, the task could replay the input envelopes it has processed, but it cannot ensure that the materialization point contains all resulting data of the last received envelope. Moreover, this would most likely not gain much benefit, as the tasks preceding the partial materialization point would have to be restarted anyway. Given these issues, partial materialization points are not used for the memoization.

4.2.2 Implementation in Nephele

To reuse the recovery logic that is used in case of a failure, there are some tweaks necessary to the new job. Materialization points are found by the IDs used in the `ExecutionGraph` of the job. The deployment of the new job must use the same IDs for those parts of that job, that are marked to be identical. Additionally, the materialization points contain `TransferEnvelopes` which will be routed using the source `ChannelID` stored in the TE.

The system compares the updated Job then to the saved data for the given `JobID`, i.e., it identifies the updated vertices, and marks all followers as updated too. During the construction of the `ExecutionGraph` all non-updated vertices are compared, checking whether they execute the same class and whether they have the same

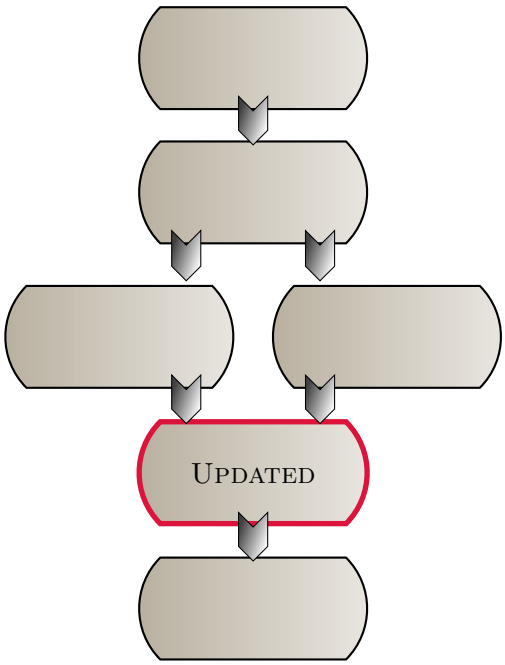

Figure 4.2: Given Job

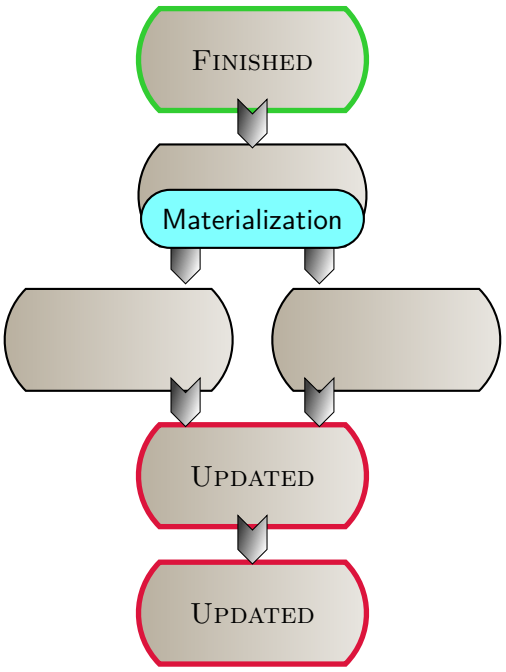

Figure 4.3: Reconstructed

number of subtasks and connections. Otherwise the memoized materialization points cannot be used.

In order to use the recovery logic and materialization points, it is necessary to use the identical IDs for vertices and connections. This is because materialization points are sets of `TransferEnvelope`s, which store the source of the data using the `ChannelID`. Materialization points themselves are files for each vertex named after the `VertexID`. To reuse the IDs the initial job must save them after it failed. When the user sends an updated job, the system reuses the IDs for all vertices, which it did not mark as updated.

As mentioned above, the system marks the updated tasks and their successors as updated. From the updated tasks a breadth-first search for a completed materialization point is done against the stream. If COMPLETE materialization points exist, all predecessors of the corresponding vertices are switched to the state FINISHED. Figure 4.2 and figure 4.3 show a given updated job example and the reconstructed Job after comparison.

The task that has written the materialization point changes to the state FINISHED after it has replayed the materialization point. For any submitted task of a newly

submitted job, the framework searches for materialization points in case the task is re-submitted during recovery. Therefore, the materialization points for an updated job are found automatically.

4.2.3 Evaluation

In order to evaluate the memoization of materialization points, it is interesting to compare the runtime of a job using memoization vs. the runtime for a complete restart of the job. As the submission of a job using memoization includes reading and applying the job information of a previous run, the setup time is an important value to monitor. It is expectable that the runtime decreases with the use of memoization. The decrease is expected to be the smaller the earlier the changes are in the `JobGraph`. However, the approach itself has an overhead, caused by the rewriting of the `ExecutionGraph` and the reading of the materialization points. The runtime measurement can show whether and when the memoization is profitable.

Optical Character Recognition

The OCR job described in section 3.5.4 serves as test job. It shows the advantage of memoization very well because it begins with the most time and resources consuming task, the OCR. All following tasks are comparably lightweight, especially the index-creating task. Therefore, one would expect that changes of the PDFCreator or the index-creating task would be considerably faster to recompute using the memoization approach. For the job's input we used pictures of the pages of a PDF Version of the King James Bible[4]. The OCR task parallelizes to 24 instances, and the PDF generator parallelizes to 6 instances.

For the test runs one task was marked updated, but the code was left unchanged, to make the run times comparable. The tests ran with the OCR; the PDFCreator and the Index task were marked as updated respectively. The test covers two measurements: the runtime and the setup time. The runtime is measured after the job has scheduled until all tasks finished. The setup time is the time between the submission of the `JobGraph` and the scheduling of the job, including the reconstruction of the graph.

The setup times are an average from all runs, with updated tasks and the same number of runs without updates. As shown in figure 4.4 the setup time using the updated mode is slightly longer than in regular mode. The mean extension for the setup amounts to 47 milliseconds, about 8% of the regular runtime. The

[4] http://www.davince.com/content/blogcategory/14/33/

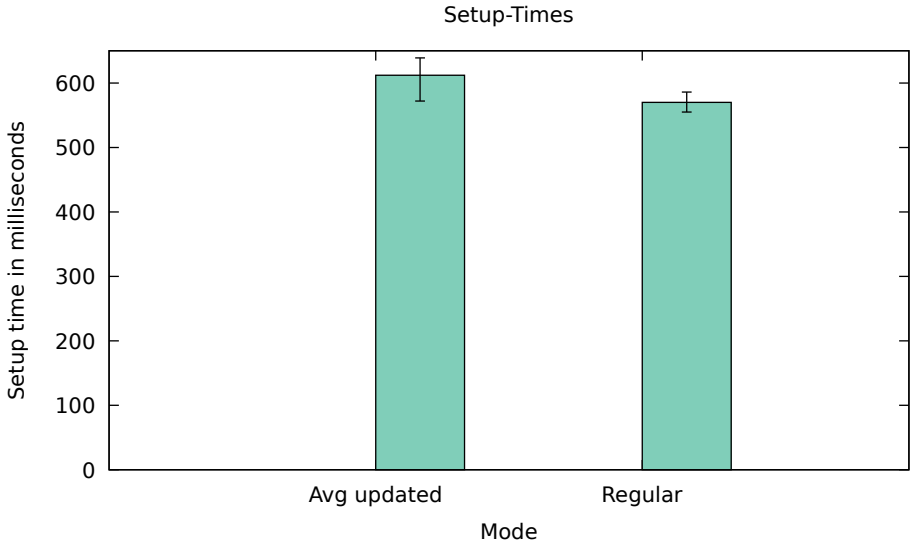

Figure 4.4: Setuptimes

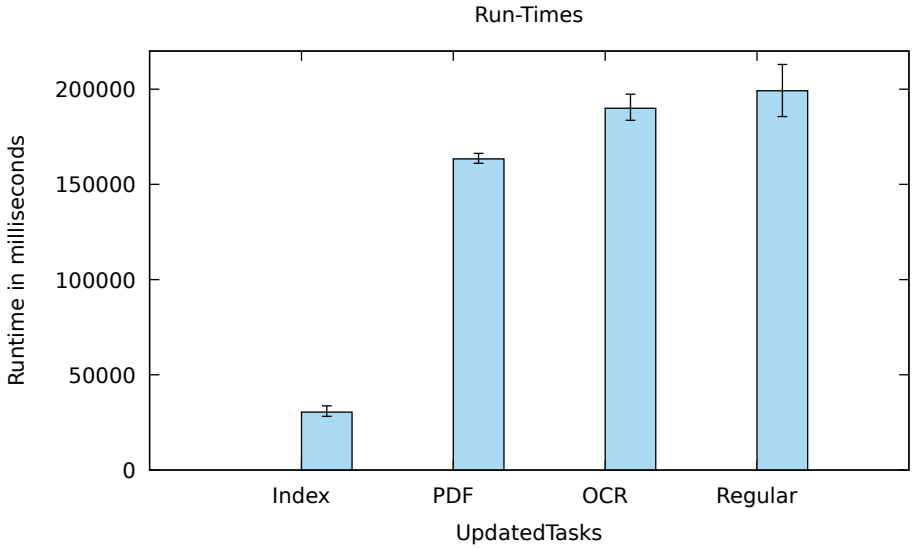

Figure 4.5: Runtimes for OCR with updated tasks and regular restart

difference between the longest updated runtime and the shortest regular runtime is 84 milliseconds, which is an increase of 15%. This increase is of no consequence compared to the gain of runtime.

Figure 4.5 shows the average run times for the test runs with the updated tasks and the regular restarts runtime. The measurements are from 10 runs for each updated task and 30 regular restarts. As one can see, depending on the updated task, the decrease of runtime can be huge. Memoization for the updated *Index* task leads to a runtime of only 15% of the restarted task. Updates in the *PDFCreator* task reduce the runtime to 81% and updates in the OCR task still lead to 95% of the restart runtime.

This is because the main time-consuming work is in the OCR and the *PDFCreator* task. The *Index* task is fast, as long as all data is available. Updates in the OCR task lead almost to a complete restart and thus do not benefit much of the memoization approach.

Triangle Enumeration

We also tested a triangle enumeration job written in PACT. This job is used to observe the runtime behavior in a worst-case scenario. The triangle enumeration job is a worst-case job because the main time-consuming work is in the last tasks in the pipeline. Updates on the job will therefore only skip the less time-consuming tasks in the pipeline. In this case, it may be possible, that the overhead is higher than the decrease of runtime. The job is described in detail in section 3.5.1.

As expected, the test runs for the triangle enumeration job did not show the significant runtime decreases as for the OCR job. However, the updated runtimes where in average still between 85% and 97% of the restart runtime as shown in figure 4.6.

Nevertheless, the gains of memoization can be huge, and as the runtime decrease outweighs the increase in setup time it is a useful technique, especially in a job's development phase.

4.3 Related Work

MapReduce itself has a record skipping mode. In this mode, the failed task sends the sequence numbers of the records to the master. If the master has seen more than one failure for a particular record, it will demand the task to skip the record. In contrast to the MapReduce system, our record skipping techniques cover systems that do not require strict input schema and contain more than two steps of execution. As

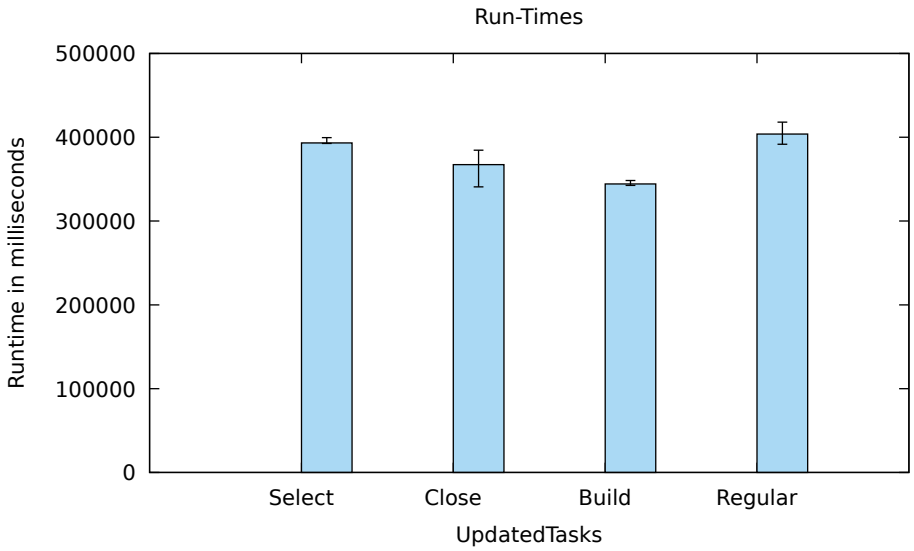

Figure 4.6: Runtimes for triangle enumeration with updated tasks and regular restart

described in section 2.2 the data flow is by default not deterministic. It is therefore not trivially possible to introduce sequence numbers to records as described above.

In “Storm @ Twitter" Toshniwal et al. introduce Storm a real-time resilient distributed stream processing engine[48]. Storm can use an ”at most once" or an “at least once" semantic using acknowledgement of tuples. Despite the fact that storm is also a nondeterministic data flow, they introduce something similar to a sequence number. Each tuple is getting an initial number. After flowing through one task, each record produced from one record, gets a sequence number computed from the sequence number of the source record.

The authors compare their approach with handwritten java code. For comparison, they use storm once with and another time without the message reliability mechanism. Although the authors argue that their approach without message reliability does not have significant overhead to native java code, the numbers show, that the message reliability in their system adds three times more CPU utilization and need two more machines for the same message-passing speed.

Memoization

As briefly mentioned, memoization or functional caching is a well-known technique to avoid re-execution of functions or incremental computation by caching results[49, 50]. In contrast to the classic memoization our technique does not cache single results of

a task's function evaluation, but the complete output of a task.

Elghandour and Aboulnaga introduc ReStore, a system to reuse intermediate data of single MapReduce Jobs within a workflow of MapReduce jobs[51]. The implementation is an extension to Pig[8]. It rewrites the MapReduce jobs compiled by Pig and stores intermediate data for reuse in similar jobs. Unlike the Pig extension the approach described in this thesis is independent from the higher-layer language.

Popa et al. describe DryadInc an incremental computation for append-only data sets. It has two functionalities: either the user has to give a merge function for previously and newly computed data, or the system fully automated reuses previous results for a new computation. The automatic so-called *IdenticalComputation* is a form of memoization and similar to the approach described in this thesis. In contrast, it is only based on additional data input to avoid re-execution of the previously executed data[52].

Nectar is an approach to manage automatic caching for computations to improve data center management and resource utilization, which also includes the reuse of sub-computations, by rewriting programs to be able to use cached data for sub-expressions[53]. As distinguished from this idea, our approach does not consider resource management but runtime optimization.

4.4 Summary

This chapter introduces data fault-tolerance using the record-skipping technique. The main challenge in record skipping is to identify the bad record between several recovery restarts. This can be a resource consuming issue, considering the non-deterministic data exchange between the producing task instances and the consuming task.

The technique described here relies on the consumption logging technique that ensures a deterministic data stream for each task, on every recovery restart. With this assurance the identification of a record can simply be done by counting the input records of an individual task and use this count to detect the flawed record detected by previous faults.

The chapter also covers the idea of memoization of materialization points. This technique enables the user to reuse materialization points of a previous job run, if only parts of the job have changed. With the information about the changed tasks, the system can find a suitable materialization point to use. The job is then restarted from this materialization point. The predecessors of the materialization point can be skipped.

CHAPTER 5

Recovery Optimization

Contents

As mentioned before, all tasks must be considered stateful, unless the user or upper layer of the system indicates it differently. The engine has no knowledge of the internals of the UDF and can therefore not predict the state property of a task. However, if the system gets the information that a task is indeed stateless, it can use additional optimizations in the recovery process. The engine can scale a stateless task freely without changing the results, and it does not have to roll it back entirely, as the results of one input record do not depend on the other inputs.

The previously described fault tolerance mechanism, has one major drawback: Some tasks have to redo work that is discarded by their followers. Considering the case when the failed task is a successor of a materialization point, it rolls back to it's initial state and redoes all work from the first record on. The successors of the failed task informs the failed task about the data they expect next, and the failed task throws away any produced output until it reaches data that is unseen by it's followers. During this time, the task is working to produce data, that is not needed and is not able to work on the data that it should have worked on in a failure free case.

In case of a stateless task, there are opportunities for optimization. The upcoming sections cover two optimization approaches. One is an adaptive recovery process. The idea is to survey the possibility to add additional resources during the recovery process to speed it up. It focuses on additional resources for a failed vertex that work on the data that should have been processed by the failed vertex. The second approach is to stop the failed task from reprocessing the head of the stream in the first place. It is not necessary to reprocess the entire input of a stateless task, as it does not have to reach an internal state again. Due to the n:m relation between input and output records, and the data distribution patterns this approach must tackle the challenge to put the output data into relation to the input data.

5.1 Adaptive Recovery

While the optimization discussed in the previous chapters, allows the system to recover from several failure types, there is still room for an optimization of the recovery time. The main issues with the restart recovery approach is the fact that other nodes may have to wait for new data while the restarted nodes process the previous data until they are in the pre failing state again.

As the consumption logging approach reduces the number of restarts drastically the recovery time is reduced already. However, the tasks that do not have to restart may now wait for new data and do no work at all. As described in chapter 3.2.1 restarts can be prevented because the stream is forced to be identical after a recovery restart. Thus, the tasks succeeding the failed task can keep on running, and just discard the data portions they have already consumed. Moreover, when a still running task receives a data package that it has already seen it informs the predecessor about the next expected envelope. Doing so, the producer of the envelopes can discard them right after production without sending them over the network if necessary. This way the data producer still computes the entire stream from the first record on, but only starts to send data over the network once it has produced data that has not been seen by the consumer.

Even though this approach reduces unnecessary network usage, it still includes recomputing output that the consuming tasks have already processed. Furthermore, the consuming tasks, which gave the information about the next expected envelope may have to idle if all their predecessors had to restart. Additionally, the failed task is now redoing work it has already done and which is thrown away, and the work it should be doing is left undone. Both cases slow down the completion of the job.

As described in chapter 3 ephemeral materialization points can be PARTIAL or COMPLETE. (In a strict sense a materialization point could still be undecided. However, during recovery and the search for a global consistent materialization point, any undecided materialization point is forced and thus immediately changes the state to PARTIAL.) The adaptive recovery approach described in this chapter only works with partial materialization points, as it presumes that additional resources can take up work that the failed task was not yet able to do. With a materialization point in the state COMPLETE, the failed vertex has at least almost all data processed and needs to process it again to produce the initial data stream once more.

5.1.1 Adding vertices during recovery

Therefore it seems to be beneficial to allocate additional resources to speed up the job, that is now slowed down due to the recovery process. However, the benefits of scaling-out are limited in the context of systems that handle many tasks and communicate over the network. Scaling-out a task to speed up the computation at that point can cause a bottleneck somewhere else in the job or cause the new machines to idle most of the time.

Adding resources to a task is only beneficial if the producing task can offer more data in the same time, otherwise the additional resources only steal hand able work form other workers. Additionally, if the scale-out is reasonable, because the producer can offer more data at the same speed, the successors of the scaled-out task might not be able to handle the additional amount of data, and thus become a bottleneck. In fact, there is an implementation of a bottleneck detection for the Nephele system which offers the user the possibility to detect bottlenecks for a job configuration. This enables the user to adapt the degree of parallelization (DoP) for another run of the job. Even if this approach can offer information about the bottlenecks in the system, it cannot assume the best DoP for a task itself[54]. Furthermore, adding resources can change the stream, as the output of a task is distributed in a round-robin fashion between all output channels. If the system adds a consumer for a task, the distribution of data changes and a consuming task does not get the same records as it would have before the restart of the failed vertex.

This issue was discussed in detail for the Nephele system in *Detecting bottlenecks*

in parallel DAG-based data flow programs[54]. The authors show an algorithm that indicates bottlenecks and offers this information to the user. They distinguish between CPU- and I/O-bottlenecks. CPU-bottlenecks are those, where the limited CPU causes a task to slowly output records and I/O-bottlenecks are those where the limit of the communication channels are not capable of processing all records in a time unit that a task tries to send. However, the user still has to adapt the DoP manually, and the detection algorithm does not give hints about the best DoP. The user has to adapt the job with a trial and error method. This is obviously not a solution for the recovery process.

Lohrmann et. al[55] use life scale out in streaming environments to guarantee latency constrains. This technique could be adapted for the elimination of bottlenecks at this point too. However, as mentioned in section 2.5, the optimization of the Nephele system itself is out of scope of this work. Thus the adaptive recovery process only focuses on counterbalancing the drawbacks of the recovery process.

When the user has already run the sample runs, detected the bottlenecks of the system, and found the sweet spot in degree of parallelization, adding resources is only sensible as far as they take up the work that would otherwise be neglected during the recovery of the failed task. The failed task replays its input and thus processes all input records, even though it discards some of the corresponding output, because the output data was already processed by the consuming tasks. During that time interval, from the processing of the first record, up to the record that produces the next portion of new data, the work, that should have been done by the failed task at this point is postponed. Adding a resource, that does that work, while the failed vertex recovers, is beneficial regardless of the bottleneck issue. That is because the added task only consumes and outputs the same amount of data as the now recovering task would have done.

As mentioned in section 2.3.1 it is not trivial up to impossible to scale out stateful tasks as they may depend on a particular data distribution pattern. Therefore in order to adapt the job during recovery, the tasks are considered to be stateless for this recovery optimization approach. Another assumption is that the system recovers from partial materialization points. In particular, not all tasks of a job have to be stateless for the adaptive recovery to work properly. As the distribution changes at the predecessor and the successor of the scaled-out task but not with other tasks, it is only necessary to have those three stateless. If any of the other tasks in the job is stateful it does not affect the correctness of the approach.

Scaling out a task can be split in two cases: one where the failing task is a direct successor of a materialization point and the other where predecessors of the failed task must be restarted.

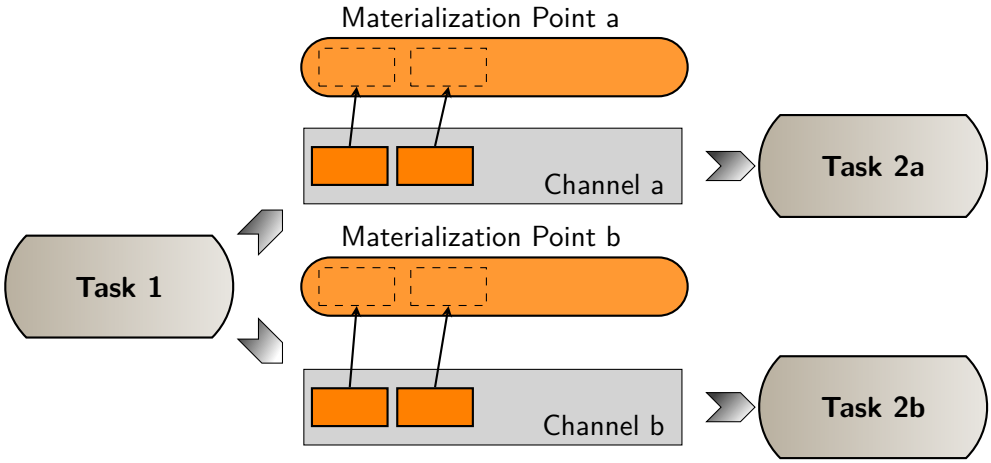

Figure 5.1: Materialization points for each channel

Failing Task Behind Materialization Point

The first scenario is the simpler scenario. In this case it is possible to start a duplicate vertex for the failed vertex itself if it and its successors are stateless. This duplicate vertex receives data from the part of the stream that has not been processed by any other vertex yet and thus takes up work that should be done by the failed vertex. For this to work the materialization points are no longer written to one file per vertex, but must be split in separate files for each output channel. Figure 5.1 shows the writing of individual materialization points for each channel, in contrast to the single file materialization points shown in figure 3.1.

As described in chapter 3, materialization points contain all data produced by a task independently over which channel it was sent. This also means a materialization point can only be read for all channels together. Recovering from a materialization point is done by reading all contained envelopes one after the other. This is a major disadvantage for adaptive recovery. For adaptive recovery the added vertex has to process data that was not yet processed by any other vertex. It cannot process other data, as this would run contrary to the consumption logging technique. Processing data that has already been processed by another task, would change the stream and raise the need to somehow rebuild the stream equal to the previous stream. Additionally, trying to adapt the degree of parallelization with one single materialization point would necessitate to rewrite the serialized envelopes, as they contain the routing information.

Providing the duplicate task with new data is not possible in parallel to the reexecution of the failed task. Adding another consumer to a producing task spreads

the output data that is not yet written to the materialization point between all consumers. A vertex that is replayed, stops to send data over the channel directly. Instead all produced data is only written to the materialization point. The sending of data over the network is suppressed, once the has replay started. The replay mechanism is reading it from disk and sends it to the consuming task. If a task only uses one materialization point for all its successors, data for the newly added vertex is written behind the data that was produced until the failure occurred. Following this, as reading from the materialization point is done one envelope at a time, the newly added vertex receives its first portion of data just when the recovery of the failed vertex has finished. This would not give any benefit to the system and instead add another idling vertex.

Thus, the writing of materialization points must be changed from one single materialization point for a task to several individual materialization points for each output channel of the task. Then it is possible to read data independently for each output channel that has to be replayed.

As the writing of materialization points is changed to a materialization point for each output channel, the reading of materialization points is now also changed to be individual for each channel. The additional task can receive its data immediately.

Restarting Predecessors of the Failed Task

The other case with predecessors that have not written a materialization point and therefore must be restarted, is a different issue. In this case it is not possible to duplicate the failed task and keep on with consumption logging. As the predecessor receives its entire input again to produce the necessary output for the replay, it produces its entire output again. Using consumption logging the head of the stream can be ignored as it is equal, to the previous data stream. Adding an additional vertex to the consumer of the restarted task changes the data distribution. This has the consequence that streams are no more identical. Without identical streams every task after the materialization point must restart.

Thus, adding vertices while also using consumption logging is only possible for the tasks directly after the materialization point. It would thus be possible to add another vertex that works on data that the now restarted vertices cannot process, and therefore give input data to the tasks that skip the head of the stream while it is reprocessed. However, the benefit of the scenario is expected to be smaller than a failure of a direct materialization point successor.

If more than one task lays between he failed task and the materialization point, adding additional tasks does not give any benefit to the system. As the additional

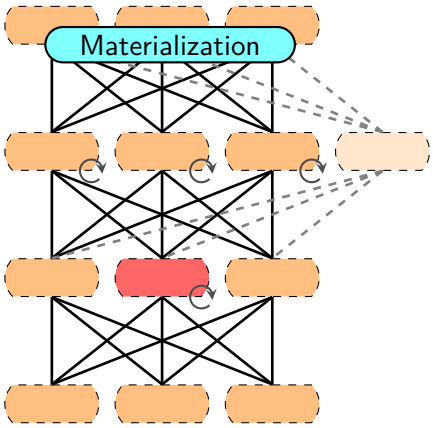

Figure 5.2: Additional vertex at predecessor

task can only be added as a successor of the materialization point, the predecessors of the failed task may produce data faster with an additional vertex. But at least the failed task has to reprocess the data of the other tasks first to reproduce its output stream and cannot process the data of the additional vertex until then.

Note that adding extra vertices to the direct successor of a partial materialization point does not influence the consumption logging. Even though an additional producer is added to a task it does not change the previously produced stream, as the new vertex only receives new data that has not yet been processed by any other producer. The consumers are not restarted and skip any envelopes of the other channels that they have already seen. Any data read from there on is new unseen data and therefore does not change any part of the stream that has already been processed. Any scaling other than the one described here changes the stream and data distribution and does therefore not allow the usage of consumption logging.

Materialization Point Splitting

If the materialization points are written into different files, they can be read independently. For every output channel a file is produced once data for this channel arrives, and can be read then. This means in case of a duplicate vertex a successor is added for the replaying vertex. As the replaying vertex is still running it produces data and eventually writes those data to the materialization point of the output channel which is connected to the duplicate vertex. This happens while the failed vertex is receiving its original data from its materialization points.

Usually a task that is writing a materialization point produces data and puts this

data into a buffer, which is wrapped into `TransfereEnvelope`s. These TEs are designated to a particular source channel, which is paired to a particular input channel of the consumer. The data is written to the channels in a round-robin fashion. A TE which is ready to be send runs through a processing chain. It starts with the writing of the materialization point, and is then handed over to the next step, which sends it to the network and afterwards frees the buffer so it can be filled with data again. During replay of a materialization point, the second step of the chain is removed. The TE is written to the materialization point and afterwards the buffer is freed directly.

Splitting the materialization points does not affect this behavior at all, it is just a tweak in the writing process. Instead of writing everything to the same destination file, each envelope is written to a destination based on its corresponding source channel. Changing the number of output channels adds another option in the round-robin choice and thus spreads any new data between the new number of consumers. However even this newly enabled connection is replayed from disk instead of sending over the network directly.

5.1.2 Cost Analysis

Making it possible to add vertices to the system during the recovery process can give an opportunity to speed up recovery time. However, it also increases costs. Using additional resources in a cloud environment does directly cause additional cost. Setting up an additional node, including starting up a task manager, takes time. Furthermore, adding another node causes additional usage of network bandwidth and could cause a bottleneck.

Due to these issues it is important to consider the cost when deciding whether to scale out and how many additional nodes should be used. In general, additional vertices are beneficial when the covered vertices must reprocess a considerable amount of input but there is still enough input left, so that the startup time does not exceed the time the duplicate vertex is running, and the duplicate vertex is not idling the major amount of lease time. Leasing an instance for an entire hour and only using it to speed some seconds may not be efficient.

However, this would make it necessary to get an idea about the progress of a task, which is not possible in the initial implementation of Nephele. The Nephele engine has no functionality to give information of the progress of a job. This is mainly because it is not a trivial task: A running job consists of black-box user code. It is not possible to predict the behavior or the running time of such a black box before it is running. And even during runtime, the behavior can only be considered for the part of input that has been processed so far.

Nevertheless, information about the progress of one task are necessary to make a suitable decision regarding the provisioning of extra machines. Therefore in addition to the adaptive recovery mechanism the fault tolerance mechanism includes an approach of progress estimation in Nephele.

Progress Estimation

Progress estimation is not only suitable to be used from the fault tolerance mechanism. It is also a possibility for the user to see unwanted behavior in the system and thus, spot mistakes in the user code, even if they do not cause detectable failures. A programmer of a job might have made false assumptions about the data or the behavior of third party libraries, that cause an expected long running task to skip over lots of records or -the other way around- work on single records unexpectedly long. Progress estimation itself gives the user of the system the ability to spot failures early.

Progress estimation is not trivial in a system with black-box user code and black-box data, and unknown data selectivity of tasks. There are too many unknowns for a clear and accurate progress estimation. Before a job is running the system has no clues about an average processing time per record for a task. Furthermore there is no 1:1 relation between input and output records of a task. Even if there was an indicator for the average processing time per record, it would still be impossible to make assumptions as the number of records that have to be processed is not yet known.

Because of theses drawbacks, progress estimation in those systems is usually done using previous runs of the same job, with either previous productive runs, if the job is a recurring job, or with sample runs, done with only a part of the actual input. Obviously the first approach does not work for the first (and probably only) run of the job, and is therefore not an option in every scenario. The second approach raises another problem: Finding suitable samples of the input. Choosing sample data is not as easy as taking the first x MB from the input, as this first part of the data might not be representative. Finding good samples is an ongoing research problem[56, 57, 58, 59].

To avoid this issue sample-run based progress estimation approaches often expect the user to come up with a representative portion of data. This is a convenient approach for the system designer, but expects the user to be able to oversee the characteristics of the input data. Considering the context of BigData this is quite unrealistic, instead the user does most likely give a sample of data that he already used as example for the programming of a job. In terms of fault tolerance this is a huge problem, as a fault in the user code does usually occurs because the programmer

makes false assumptions about the data. The benefit of seeing unexpected behavior early because of a hint in the programming progress is completely lost in this case.

Another issue with samples is to define their size. The minimum size is usually given in terms of representatives of the productive input. The main question is: Is the smallest amount of data enough to do a meaningful sample run. The point here is: If the data portion or the intermediate data portions are to small, they might be kept entirely in main memory, and thus give a false assumption about processing speed. On the other hand, running sample runs with large amounts of data may take almost as long as a productive job run and not be beneficial. Furthermore, one single sample run might not be enough to make an estimation, as complex operators like cross join do not increase linearly with input data size. Unfortunately, it is not possible to see a non-linear increase with just one job run, it takes several runs to recognize this behavior.

In order to be able to make an estimation of the progress with the first run of an unknown job, one must find another solution. A first step to a progress estimation during the first job run is input-split based progress estimation, which uses the known size of the input and propagation of progress information in order to make assumptions of the progress of each individual task.

Before the runtime of a job, the only fix and measurable value is the input of the job. As the system has to handle black box user code, there is no other information left. The input of a Nephele job can either be a single file, or multiple input files. However the size of the entire input is always known before the job runs. The Nephele system reads the input for a job in so called splits. A split might be a part of or an entire file of the input. Each individual task instance of the input task is initially assigned a split of the overall input. Once it has finished processing this split of the input it requests another split, which is assigned to the task from the remaining splits. Split assignment is done in a topology aware manner using the topology benefits of the underlying HDFS system.

This behavior offers great flexibility if some splits are more time consuming to process, and other splits are faster to handle. If that situation occurs, the faster task probably works on more input splits during the job's runtime. And it can be used for a new progress estimation technique during runtime, which we call progress forwarding.

The main idea of progress estimation during runtime is based on the percentage of input splits processed. The main assumption is: Once a new split is requested, the already read splits can be considered to be processed. This is of course a simplification, as the other tasks might still be working some time on their input splits. The calculation of the progress reading the input is pushed through the graph, and gives a hint of the overall progress for an individual task. Once the progress information

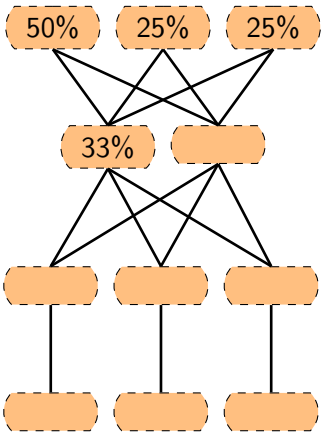

Figure 5.3: Estimated progress

has reached the output tasks, this information can be considered to be the progress of the overall job.

This progress information is added to the `TransferEnvelope`s at the input tasks. Every time an input task requests a new split, it considers all assigned splits to be processed and calculates the progress as the percentage of the processed data to the overall input size. Then it receives its next split. While processing the split, it adds the calculated progress information to the outgoing data at each output channel. A consuming task that receives a TE with progress information takes this information to calculate its own assumed progress. As every instance of the input tasks adds progress information to their output, every consumer receives progress information over each input channel. All progress updates from a tasks input channel are saved, and an average of all received input channel progresses is calculated. The resulting average is considered to be the progress of this task.

Imagine a task with three predecessors. Initially all progresses are seen as 0%. If it receives a progress information of 25% from one predecessor. The task now has 25% of one of its three inputs processed, and its overall progress is estimated to be about 8,3%. If it receives a 25% progress information of all channels, its progress is estimated as 25%. If it then receives 50% from one of the channels its progress calculates to 33%. The estimated progress is then added to outgoing data and sent to the consuming task, which also calculates its progress and propagates the information further. Figure 5.3 shows this scenario in an execution graph.

Note that because of the reading behavior (A task first reads all available data from one channel before it chooses the next channel that has data available.) it is possible that a task has a progress of 100% at one channel but 0% at the other channels.

Also, taking the progress of all predecessors and calculating the average considers slower paths in the graph into the progress estimation. Furthermore, it may be possible, that due to calculation of average, the progress is calculated as 0% even though data has already been received. These 0% progress events are ignored and not propagated to the consuming tasks.

The main assumption of this approach is that if the output task receives the information that they have received a percentage of their input this percentage is equal to the progress of the job. However, it is not safe to translate this information directly to processing time, as the processing time between two records can vary drastically. If the first amount of data does not come through a filtering function, processing time of the first amount of data is notably shorter than for the rest of the data. Nevertheless, the progress propagation approach is self-regulating. If the next progress update is received later or indicates a smaller progress for a similar time interval, the estimated time is adapted.

But independable of the overall runtime of a job, this approach has the benefit, that it works from the first run, and that it leaves progress information at every single task in the job. During the adaptive recovery, it is unimportant how long the overall job runs, but important to have a hint of the considered progress or to be more accurate the amount of work that needs still to be done.

Evaluation of Progress Estimation

The evaluation was done on a sample-run-based approach and on the progress forwarding described above. The sample-run-based approach expects the user to come up with a suitable sample of the input data. For the evaluation we used the first 500 Files as a sample. For the OCR job that was about 462,31 MB from the overall 2,60 GB of input. During the sample, information about the average runtime of each task per record were collected.

To estimate the job runtime the system detects the critical path in the job. A job path is a path through the DAG from the input vertex to the output vertex. One job can have multiple paths, if it splits the data flow somewhere. For the runtime estimation it is necessary to consider only the longest running path. The algorithm for the detection of the critical path using a breadth-first search. The search starts at the jobs output vertices and checks for the vertex with the latest finishing time. Form the specified vertex, all predecessors are checked for the one with the latest finishing time and so on.

During the runtime of the job only the critical path is considered for estimation. The runtime is then estimated with the average runtime of the sample run in ratio to the

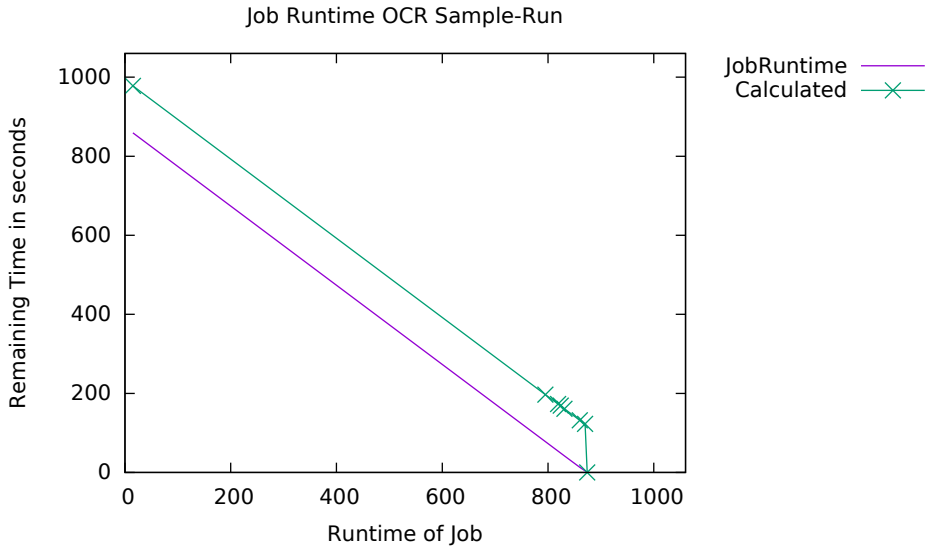

Figure 5.4: Calculated Job Run time Sample Runs OCR

input data. The input and runtime ratio are expected to be straight proportional. The estimated runtime is update with the actual runtime of a task if it is finished.

Figure 5.4 shows the calculated remaining runtime compared to the actual remaining runtime at particular points of the job runtime for the OCR Task.

For comparison the runtime estimation for a sample run with the ideal data set i.e. the complete input is given in figure 5.5.

Two things are noticeable: First the difference in the estimation, depending on the input sample. This shows the importance of the quality of the sample. The second conspicuous thing in the results is the distribution of the estimation points. One estimation is done during job start, the next just shortly before the end of the job. This is because the sample run contemplates one pipeline of the critical path and can therefore only update its estimation if one pipeline is finished. As the OCR job consists of one pipeline the estimation is just updated at the end of the runtime.

The deviation of the estimated runtime was about 13% for the sample of about half of the actual input.

The main benefit of the sample run based approach is that it is suitable for jobs that contain pipeline breakers. That is why we added tests for sample runs with pipeline breaking jobs.

As the files have roughly the same size, the number of processed records is a good

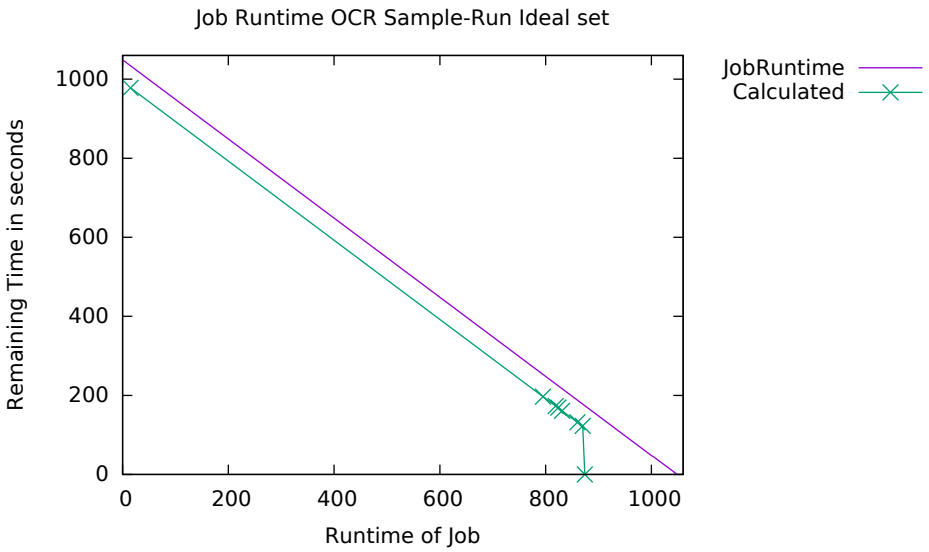

Figure 5.5: Calculated Job Run time Sample Runs OCR Ideal Set

indicator of the progress of a task. In order to compare the calculated progress to the actual progress we look at the processed records for each calculated percentage. The graph 5.6 shows the calculated progress compared to the actual progress. The calculated progress at each task differs only up to 4% from the actual progress. Based on the percentage of the output task we calculated the remaining runtime of the job.

Figure 5.7 shows the calculated remaining runtime compared to the actual remaining runtime at particular points of the job runtime. The points of the calculated runtime graph describe measurements the moment the writer task announces the progress, starting with 5% in 5% increments. The calculated remaining runtime differs only up to 23 seconds. The first information about the jobs progress is received 92 seconds after the job started. Within the first quarter of the job, the system is able to make an estimation about the remaining runtime, which is accurate up to about 5%.

Cost Model

Assuming there is a more or less precise progress estimation at a failing node the adaptive recovery can use this information to decide whether to use a backup node for the failing node. As mentioned above, adding another node to the job needs startup time and may add additional network usage. In the specific recovery scenario the network bandwith is only an issue as long as the original and the backup node are both sending data. With consumption logging and the possibility of a forwarding

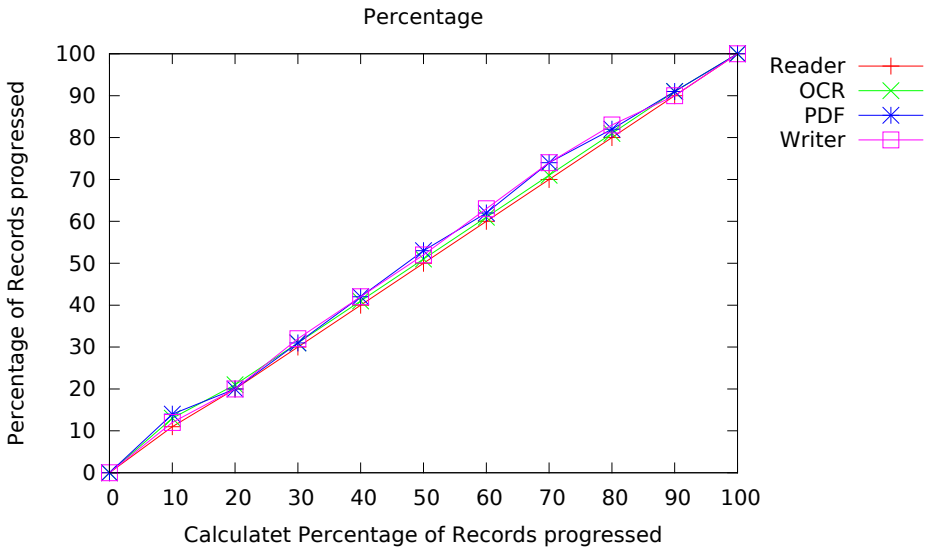

Figure 5.6: Progress at tasks

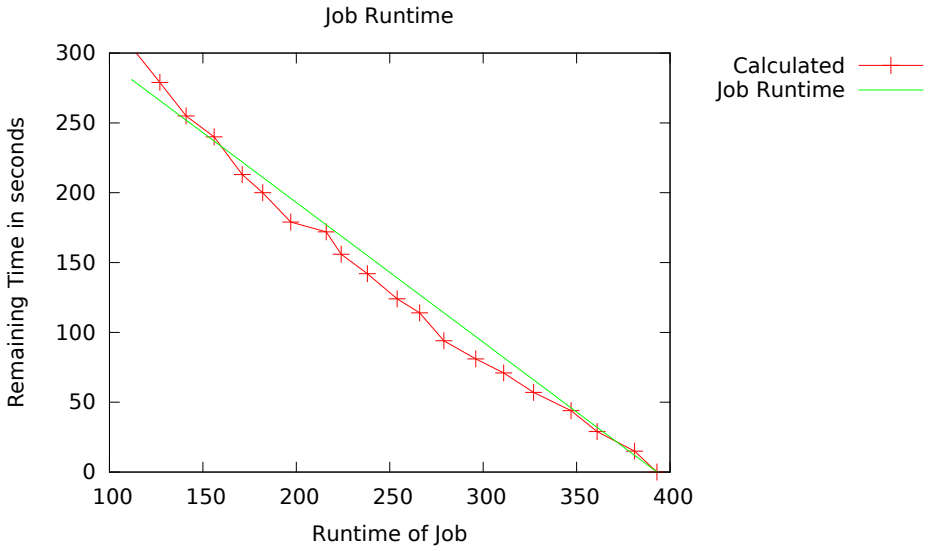

Figure 5.7: Calculated Job Runtime

barrier, the recovering node does not send anything over the network until the recovery is finished. Using a backup node only during that time shutting it down once the failed node is back at work would not change the bottlenecks of the system.[1] Therefore, the start up time is the lower bound. If the failed task has only to redo work that is equal to or smaller than to the startup time of a node, it is not beneficial to add another node.

The other question is, where the additional nodes are coming from. In a fixed cluster with idling nodes this is not an issue. However, in a cloud environment, where the system might have to provision another machine and add cost, to the job processing cost, it can be.

Even in an IaaS Cloud environment it is possible to have idling machines available. As the payment is usually done hourly, a task finishing before the payed hour is reached, gives the opportunity to use this extra spare time to speed up the recovery process, and to unprovision it when the time slot ends.

For any other scenario the user has to provide the information about the money he is willing to put into the faster completion of the job. If the cost boundary is not reached, it is possible to provision additional nodes for this task, probably using a cost efficient solution like Amazon Spot instances. The system would then only use additional machines if they are under a specific price at the point of failure.

5.1.3 Implementation

The implementation of the adaptive recovery concept is linked into the recovery logic. The recovery for a failed vertex can duplicate the failed vertex during its restart, if it is a direct follower of a materialization point. The evaluation of the number of additional vertices was based on a configuration value for the number of duplicates. This expects the availability of the number of instances, that was configured. If the same number of instances is not available, the system just duplicates to the number of available instances. In the cluster mode of Nephele the available instances are a fixed set of nodes that have to be set in the configuration. The Cloud mode of Nephele, that offers support for Amazon EC2, can however lease additional resources using the EC2 API[2]. In this case it is necessary to differ between two scenarios: The system still has nodes available, that idle but did not extend the lease time yet, and the system does not have any free instances.

In the first case, the EC2Manager of Nephele has a list of so called *FloatingInstance*s.

[1] Note that this is of course always with the assumption, that the job is configured in the most efficient DoP regarding to network and CPU bottlenecks.

[2] http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.htm

These instances are allocated instances that idle and are not assigned to any task. It is possible to use these instances for tasks until terminated. Every instance has a lease period that can be configured by the user. The default is one hour as this is the charging period for EC2. The instances are used as long as they are assigned to a task. Once the work is finished and an instance is no more assigned to a task it is converted to a floating instance if the lease period is not yet exceeded. If the instance is leased for the lease period it is terminated.

If the Cloud manager has floating instances, they can be used for duplication. It is the perfect opportunity to use the idling computation power for the speed-up of the recovery process. If no idling instance is available at the manager, the Cloud manager has the possibility to allocate new resources from the EC2 Cloud. Allocating the new instances causes additional fees to the user. In order to avoid that the instance idles most of the time, the duplication is only done if the estimated remaining work time of the failed vertex is longer than the half lease period.

The recovery process checks therefore if the system is running in Cloud mode. If so, it checks if any suitable floating instances are available for the duplication. If not, it checks for the estimated remaining runtime of the failed task. If the remaining runtime is at least half as long as the configured lease period it starts a duplicate vertex. Starting the new vertex automatically triggers the allocation process in the Cloud manager. The Cloud manager requests a new instance from the Cloud.

Each duplicate vertex is created in the same way, as a vertex would usually be started up in the initial start. It is created and added to the group vertex. The difference is that the connections of the vertex have to be set up during starting. In the initial start of a job, the execution graph is constructed all at once. It contains a group vertex for each task, and an execution vertex for each parallel instance of the task. The connections between those tasks are set up during creation. The duplicate vertex must add those connections during duplication. This consists of two steps: One is to add these connections in the execution graph, to construct the abstraction of those connections and enable the system to iterate through the graph using the edges. The second part is to register the connections to the routing service. The routing service needs the information about the newly triggered connections to be able to route the data over the network as needed.

During recovery the `ExecutionGraph` is updated and all necessary edges are added to the graph. This includes adding the channels to the input and output gates of the successors and predecessors. Adding a channel to the `OutputGate` of the data produced, i.e. the task that replays from the materialization point causes any newly arriving data to be distributed between the new number of channels. The writing to the channels is done in round robin manner, and now includes the new channel.

As the data is written to the materialization point before it is sent over the network,

no data is lost, even though the consuming task is not yet running. As the task now distributes the data between all channels, it creates a new materialization point for the new channel and writes the data into the new files. Once the replay task for the task is started, it replays all materialization points and the newly duplicated task receives its input. The registration at the routing service is updated during the start of the replay task. The successors of the failed and duplicate task, wait for any channel to provide data. Once the new vertex is started, it sends data to the follower where it can be read.

Each vertex is only duplicated once. If it fails another time the duplicating algorithm is not triggered again. The same goes for duplicated vertices. A failed duplicated vertex does not duplicate. This could otherwise lead to a huge growth in resources as an already duplicated vertex has a vertex that takes up work. It does not make sense to duplicate it again. Think of a persistent fault that occurs on the same record over and over again. Duplicating the vertex more than once would not add performance to the system, instead it might cause bottlenecks. And as a duplicated vertex is just running for boosting the performance of another vertex it is not beneficial to duplicate it, even in case of a failure.

Once a vertex is duplicated it runs until the job is finished, to avoid additional overhead and possible data loss. However, for the sake of recovery performance boosting, it would be sufficient if the duplicate task is only running as long as it takes the failed vertex to finish recovery and rollback to its pre-failure state.

5.1.4 Evaluation

For the evaluation the first question to answer is, how beneficial is it to use one or more duplicates for a failed task, considering a well-balanced job. The job used was the OCR-Job described in section 3.5.4. To find the right degree of parallelization the job was run with different DoP settings a number of times to compare the runtime. Figure 5.8 shows the results of the experiment, it started with 16 *OCR* tasks and 4 *PDFCreator* tasks. The number of *PDFCreator* tasks was increased by one step by step. It shows the lowest runtime at 16 *OCR* tasks and 7 *PDFCreator* tasks. Thus, this degree of parallelization was used for the next evaluations.

For the next step of evaluation, the job was run and a *PDFCreator* task was killed shortly after it started running. The system was configured to use different numbers of duplicates for the failed vertex. Figure 5.9 show the results for runtime of the setups.

The results show that adding a duplicate on recovery, reduces the runtime compared to the recovery scenario without a duplicate. The average runtime with one

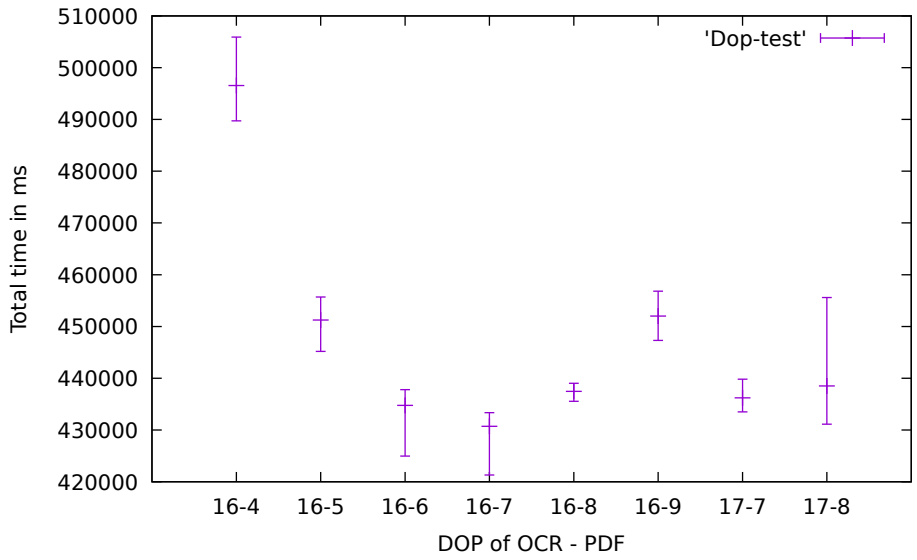

Figure 5.8: DoP Testing

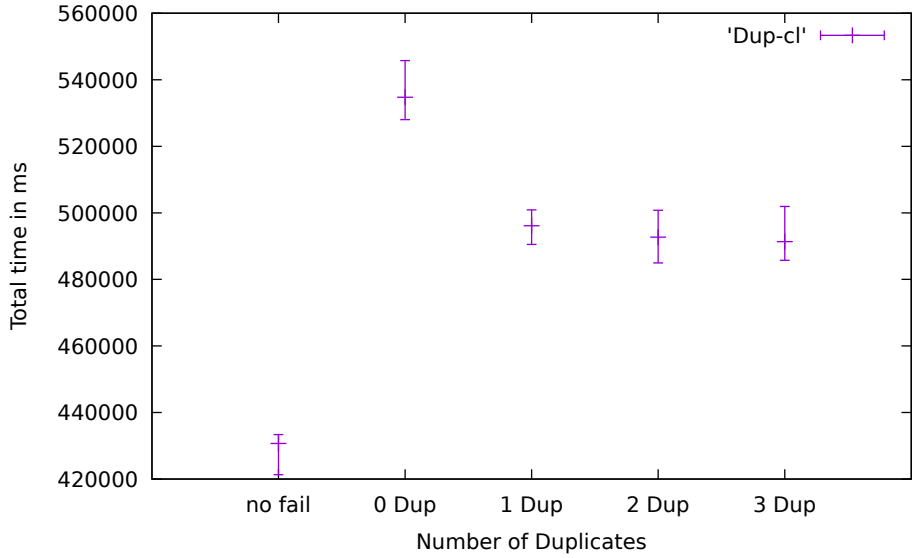

Figure 5.9: Duplicate Testing

duplicate vertex is 92% of the non-duplicate runtime. However, adding more than one duplicate does not improve the runtime, the runtimes stays at 92% of the runtime without duplicate. That confirms the hypothesis discussed in the beginning of the section. As can be seen, adding a duplicate vertex reduces the runtime. Adding more than one additional vertex does not reduce the runtime of the job further. The measurements also show, that the runtime in case of a recovery, is still noticeably longer then in a fail-free run. The recovery process and the re-execution of the input of the failed vertex add overhead to the job runtime, even with an additional vertex.

5.2 Offset Logging

Instead of using additional resources to optimize the recovery process further, the most intuitive approach would be to avoid the reprocessing of data that has already been sent to the consumers. As described before any task has to be considered to be stateful if not otherwise indicated. The only state to which a task can be rolled back is its initial state, and any restarted task has to be re-executed from the first record, to ensure it reaches the same inner state. Nevertheless, this is unnecessary for stateless tasks.

If a task is indicated to be stateless, it could go on to process the data not yet sent to the consumers and thus ignore any previously treated data. This method requires the matching of the output to the input.

After a restart of a failed task, the successors of the task can inform the task about the next expected envelope. With consumption logging, the failed task can avoid to send any data over the network that the successors have already received. Nevertheless, it still reproduces the data. In order to avoid replicating that data unnecessarily, it would be necessary to know the position in the input stream to start processing to produce the next needed envelope. As there is no 1:1 relation between records or the in- and output envelopes it is not as easy as skipping the same amount of input envelopes. Instead, the processing must restart at the record after the last record fitting entirely in the previously filled envelope.

This record or a part of it is the first amount of data that was not processed by the following tasks. There are two scenarios to differentiate: The last processed record's output could fit entirely in the previously sent envelope, or it did split into several envelopes. In the first case, the skipping would be relatively easy if the task keeps track of the last record processed: As the envelope fills up, it can go on handling the record after this, and it is not necessary to think of any order, as no task has ever received the reproduced data.

5.2.1 Channel state

If the output data did not fit entirely into one envelope, skipping is not as simple. If the output of a record splits between envelopes, the output channels have a state. The channel keeps the output in the serialization buffer until it is able to write it into a buffer of an envelope. The recovery process has to reproduce that state. It must guarantee that the remaining data is in the next envelope that the task sends to the corresponding consumer. The consuming task's input channel has already started the deserialization with the first bytes that contained in the last envelope. The deserialization only works correctly if the missing bytes are the next bytes received.

The state of the channels depends on the record that the task had processed as it has sent the last envelope. Therefore, this record has to be processed once more. The recovery logic must discard all output bytes that the task has already sent over the network previously to the crash. It is thus not enough to log the input record number to skip the head of the stream correctly. The log must also contain the offset of the output that the channel could fit in the last envelope. Note that the recovery has to take care of each channel individually. Each of the multiple output channels could have contained data during the crash. During recovery, the task thus has to reprocess any record, that produced an output which stayed in the channel buffers. The recovery can, however, skip any record in between.

If the log holds all this information, it is possible for a stateless task to skip the previously processed data after a restart. This skipping may lead to a significantly reduced recovery time, as the failed task does not have to reprocess the head of the input stream. Furthermore, this offset logging method opens the door for using ephemeral materialization points in stream processing. One main assumption for the materialization points so far is that the engine does batch processing. The system gets a finite input. If the engine processes an infinite stream of data, this would have the consequence that the materialization points grow infinitely. This is, of course unacceptable.

With the offset logging technique, it is possible to develop a sliding window materialization point. As each task has a translation table of its output envelopes to a record number, an additional log with translation between record number and input envelope would enable a task to push acknowledgments up the stream. A task that writes a materialization point could acknowledge all input records which have their output fully represented in the materialization point. This acknowledgment can then be translated to input records at the next task and so on. A materialization point up the stream, which get acknowledgments for an entire envelope, can remove it from the materialization point.

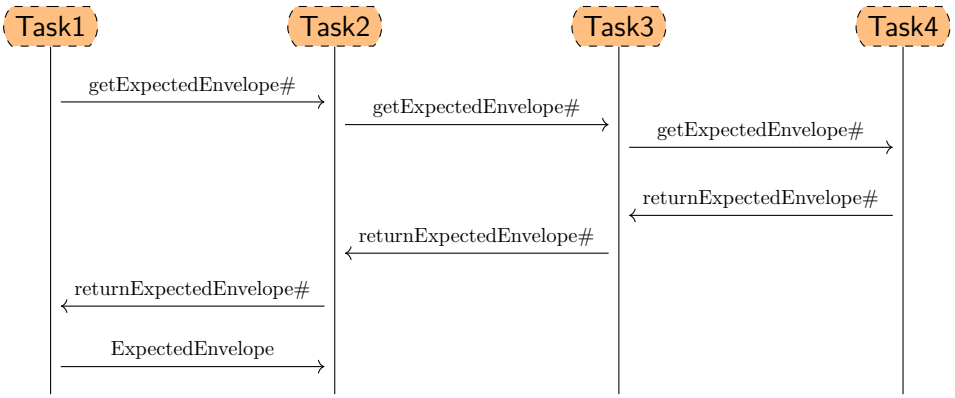

Figure 5.10: Expected Envelope Interaction

5.2.2 Implementation

As described above, the offset logging requires a stateless UDF. However, the system has a state even with stateless UDFs. For one task, The necessary information to log is

- The sequence number of the outgoing envelope
- The `ChannelID` of the log
- The byte offset of a spanning record
- The record offset for multiple output record for one input
- The input record that is currently processed
- The `InputGateID` for the input

The sequence number of the envelope is provided by the `OutputChannelBroker`, which creates new envelopes. The byte offset of a record is necessary in case a record spans over envelopes. If a record does not fit entirely into one envelope it is split. The consumer receives the first part of the record in one envelope and waits for the next envelope to contain the second part of the record. To create the next expected envelope correctly, the recovery must ensure that the next envelope contains exactly the second part of the record. Therefore, the log has to save the number of bytes that have already been shipped in the previous envelope. This information is received

from the serializer. The Nephele system provides a `SpanningRecordSerializer` that takes care of serialization of the record and writes it into the envelope's buffer. If a record does not fit entirely into the buffer of the envelope the offset is logged, when a new buffer is requested.

The record offset for the number of output records is fetched from the `OutputGate`, it counts the number of records that are emitted. The `InputGate` resets the counter every time it pushes a new record into the UDF. For each log entry, the current counter at the output gate is saved. The `InputGate` provides the information about the currently processed record and the `GateID`.

A new log entry is created every time the serialization requests a new buffer. Requesting a new buffer is equivalent to creating a new envelope, as each envelope contains exactly one buffer. At the creation of a new envelope, all described information about the previously filled envelope are available and can be logged. To do so a log entry is created as described above. The entry is passed to a writing thread, that has a log file opened for each execution vertex, and appends each newly incoming entry.

With this information it is possible to set up the failed vertex and its environment in the same state as it was before the failure without the necessity to re-execute the entire input. Instead the recovery logic is able to skip all input that is not needed for the creation of the next expected `TransferEnvelopes`.

As each output channel creates envelopes independently, based on the amount of data to transmit, each successor of the failed vertex could have received a different number of envelopes. Consider an input stream with records that produce drastically different output sizes. An output for one record might have to span over several output envelopes, whereas for another channel the output for several records may fit in one envelope. The first channel will then have a higher sequence number, but the output of the second channel depends on input data that comes in the stream before the big record. Recovering the state of the vertex is thus not as easy as using the log entry with the highest sequence number. Instead, the last log entries of all output channels have to be checked for the input record number. And the stream can only be skipped up to the smallest input record number in the logs.

Reading the Log

After the failed vertex has restarted, it tries to find an output log file. If a log file exists the data is loaded. The vertex, then asks its successors for the sequence number of the next envelope it is expecting from the vertex. The successors then answer the request with an `UnexpectedEnvelopeEvent`. This event would usually

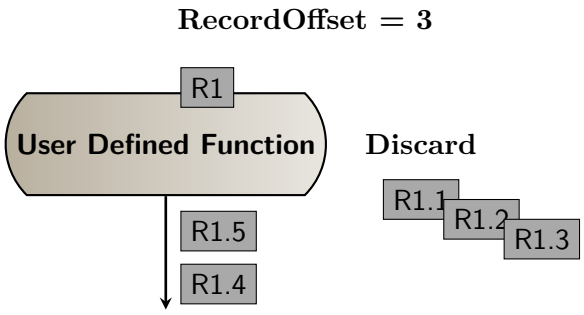

Figure 5.11: Record offset

be triggered during restart, to skip the output envelopes, as described above. With the offset logging, the event is used to hand the information about the next expected envelope to the offset logging functionality. The logger then finds the input record number, which indicates the first record that is needed to recover the previous state. This record is the first one fed to the UDF. To ensure that the output data is distributed in the same manner as before, the `ChannelSelector` has to be set to the channel that received the first output record produced by the input record, in the first run. Afterwards the channel selector runs in the same manner, as before, which is a round robin selection of output channels.

The recovery of the state is achieved by just running the UDF from the found record. The consumer may have already received parts of the record's outputs or even the entire output of some records. Therefore, the records have to be processed, but the recovery logic must keep only the parts of output that are necessary to rebuild the state.

For case that the UDF produces several output records for one input record, the `RecordOffsetTotal` keeps the information if some of the output records have to be skipped. After the vertex skipped not needed input records the output gate skips the number of output records that the log entry saved to be skipped. The output channels then receive the output records, and may skip parts of the received output record if it has sent parts of the record to the consumer before failing. Figure 5.11 shows the discarding of records for a record offset of 3. The first three records of the output for record $R1$ are discarded, the task starts sending records from the fourth record on.

After all channels have finished the recovery according to their log entries, the recovery is finished and the processing of data continues normally.

5.2.3 Evaluation

The evaluation of the offset-logging is based on runtime measuring. It is supposed to show that the logging is functioning correctly, and whether it yields performance increase in case of a failure. On the other hand the evaluation has to prove the overhead of the logging technique in case of a failure free run. This is supposed to answer the question whether it is beneficial to use the offset-logging. Similar to the other evaluations it tries to answer the question: Does the technique gain a runtime reduction in the failure case that is big enough to compensate for possible runtime increases during non-failure runs? This evaluation is run with a job that uses integers, to create fixed-size records. This allows to fine tune the position of failure and recovery within the stream.

To be able to test the different scenarios described above the evaluation used a simple job that processes integers and lists of integers. The job uses a file of random integers as input. The integer values are stored in lists of 2048 integers, and sent to the consuming vertex. The *ListFilter* vertex takes an integer list, sorts it and removes every integer that is bigger than 750000 from the list. This removal of integers creates output records of different sizes. This is necessary to test the behavior of the output channels described above.

The next task takes a list and splits it into single integer records. This covers the case that one input record produces several output records. Note that the number of output records for one input record is not fixed either, as the filtering of the list causes lists with different numbers of output records. The successor of the *ListSplitting* task takes the integer records, doubles the integer and sends the result as an integer record to the *OutputWriter* which writes all integers to a file in persistent storage.

In contrast to previous evaluations, the failure for the evaluations are triggered from the tasks. This is necessary to produce the different described scenarios reliably. Instead of killing a task after a number of milliseconds of runtime, the tasks count the number of records they have received, and if they are marked to fail after a particular record, the tasks throw a runtime exception. This allows to produce all scenarios that have to be tested. Each marked task only fails once during one runtime of a job, to allow the recovery logic to recover.

The size of an envelope determines the number of envelopes that are produced for a given input data size and thus the number of offset logs that have to be written. In the job described above that mostly exchanges integer records that have a fixed size of 8 bytes the size of the envelope also decides, how often records have to be split between two different envelopes. Any envelope size that is a multiple of 8 bytes, eliminates record splitting at channels that exchange pure integer records.

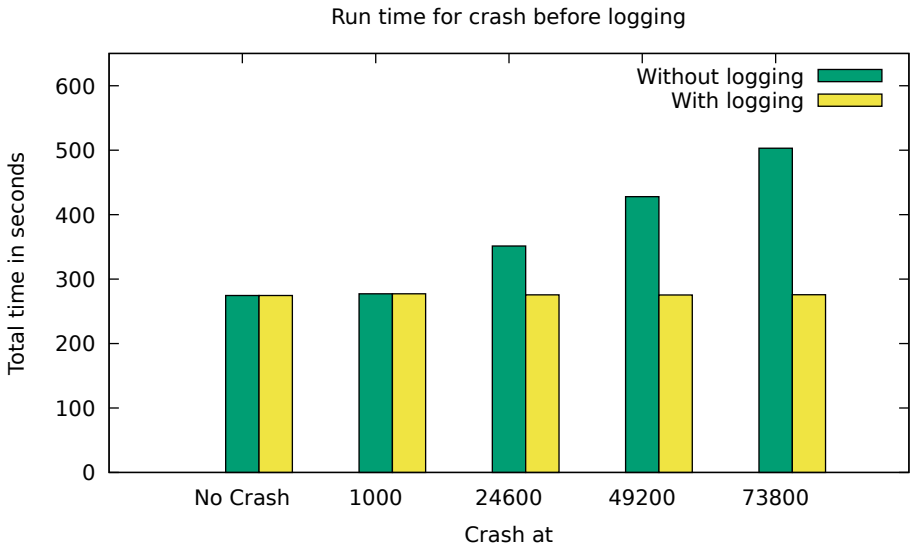

Figure 5.12: Runtimes for crashes before logging

As described above, the system must replay the state of the channels in case of split records, a lot of split records could therefore increase the recovery runtime noticeably. Thus, the size of the envelopes is also a factor that the evaluation has to cover.

Job Runtime

For the comparison, the evaluation runs with two test setups. In the first setup, the system crashes directly after writing a log entry. In the second configuration, the system crashes just before it writes the log entry. These two setups are supposed to observe the difference in runtime decrease for a recovery for the last processed envelope and the fast-forward only up to the previous envelope. Both scenarios are compared with the runtime of the initial roll-back recovery.

Additionally, the evaluation has to cover the overhead of the method. The runtime of a job run without failure is compared to one with the logging and without. An increase in runtime for the logging method indicates the overhead of the offset logging. The logging approach aims to be a lightweight solution and the prediction is that the logging overhead is negligible in comparison.

Figure 5.12 shows the arithmetical mean of job runtimes for the job runs for a crash before logging. The “No crash" bars show the difference in job runtime if no failure occurs. These bars are, on the one hand, an indicator for the overhead

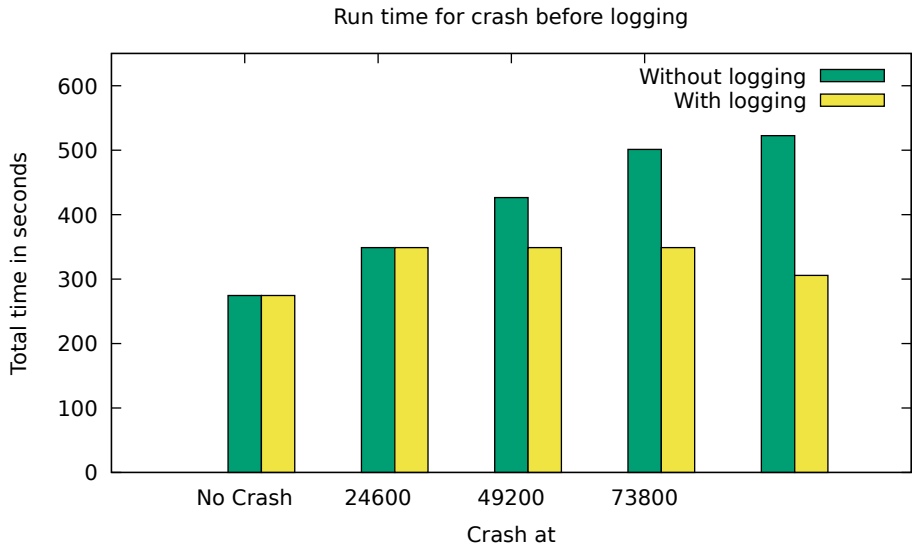

Figure 5.13: Runtimes for crashes after logging

of the logging technique in general, and a reference value for the increased runtime during recovery. As one can see the average runtimes for non-failure runtimes do not differ. The overhead for the offset logging is negligibly small. The next bars show the different runtimes for a fault at `ListSplittingTask` after the stated number of records. The measurements show that a crash in the earlier stage of the processing, in this case after 1000 records, cannot be faster recovered with the offset logging. However, it also shows that the approach does not add additional runtime. As expected the fast-forwarding with offset logging is especially beneficial if the fault occurs late in the execution. The recovery without offset logging has a constant increase in recovery time with each step. The recovery time for offset logging is almost constant, no matter when the failure occurs. The offset logging shows a speedup of 82% compared to the classic rollback recovery.

Figure 5.13 shows the measurements for a worst-case crash, right before the system writes the log entry. As the result of the reprocessing of an entire envelope, the runtime for the offset logging has now increased in comparison to the previous test. However, the runtimes are also constant between the first steps. The offset-logging runtime for the crash at the last record is slightly smaller; this is because the last envelope is only partially filled, and the system does not reprocess a full envelope. Considering the crash at record 73000, the offset logging shows a runtime decrease of 43% compared to the classic rollback recovery, even though the fault occurred right before the log entry was written.

As mentioned before, the size of the `TransferEnvelopes` may have a significant

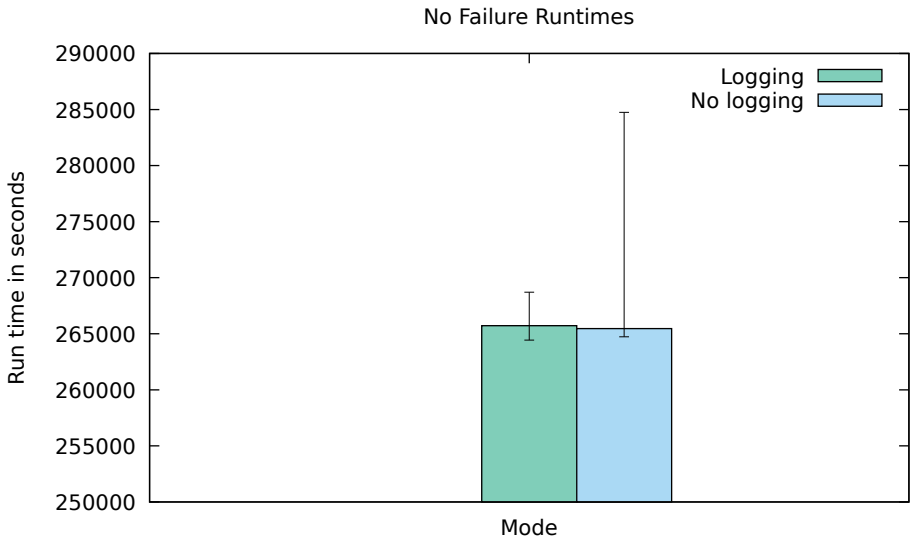

Figure 5.14: No-fault runtimes

impact on the overhead of the logging technique. The smaller the user defines the buffer size, the more log entries the system has to write. More I/O operations can slow down the logging and add additional runtime. To see the effect of the `TransferEnvelope` size, the test runs without a crash run with significantly smaller buffers of 128 bytes. Figure 5.14 shows the arithmetic mean of 150 executions, including failure bars. The measurements show that the slow down, caused by the offset logging is about 0.1%. However, the deviation between single runs is much higher with offset logging. This is most likely caused by the I/O operations, and inconsistent hardware access time.

5.3 Related Work

There is some related work in the context of elastic scaling, that tries to find the sweet spot in the degree of parallelization. This work is usually placed in the scenario with changing workload over time and scaling during runtime. Even though they aim to optimize the DoP during runtime not during failure they head in a similar direction.

As described earlier, Warneke et al. implemented a bottleneck detection algorithm for parallel data flow system. It is based on measurements of the network throughput and the CPU usage. They differentiate between CPU and Network I/O bottlenecks and adapt the degree of parallelization manually to find the sweet spot with the

lowest number of bottlenecks. They also show that adding resources beyond that sweet spot is not beneficial.

There are several approaches for elastic scaling in stream processing environments, where the degree of parallelization is adapted based on the workload. Gedik et al[60] introduce a dynamic scaling of the number of channels depending on the workload and the availability of resources. They handle stateful tasks using state migration.

Lohrmann et al[61] use elastic scaling and scale out to enforce latency guarantees in stream processing engines. However, those techniques are run to optimize the engine during runtime, based on the workload. They could be used for optimization of the engine, which is out of the scope of this thesis. The changing workload, due to the recovery of a node is described above and does not have to be calculated dynamically.

5.4 Summary

This chapter introduces two methods for recovery optimization in case of stateless tasks. If the system consists of stateless tasks, it is possible to use optimization techniques that reduce the job runtime in case of recovery. If all tasks have to be considered to be stateful, the fault tolerance technique of ephemeral materialization points includes rolling back the task to the last known state and reading all necessary input to come back to the state in which the fault occurred. It is not possible to change the degree of parallelization or to skip input data. This is different for stateless tasks.

The degree of parallelization of a stateless task can be changed without problems. This fact leads to the idea of adaptive recovery. Adding resources during recovery can reduce the runtime. However, the results in section 5.1 show that it is only beneficial to add one additional resource during recovery (considering a well-balanced degree of parallelization for the tasks).

Another optimization technique is the introduced offset logging. Offset logging enables the system to fast-forward the stream during recovery, avoiding the recomputation of unneeded records. The reprocessing of the entire input stream is necessary for stateful tasks, as the task has to reach the same state again. A stateless task can continue processing anywhere in the stream and produce the same output data. It is thus beneficial to skip the head of the stream, if the successors of a failed task have already seen the output.

To be able to skip the right amount of input data, it is necessary for the task to know which of its input data is already represented in the data that its successors have

received. The task has to save information about the input records and which output they produce. The offset logging mechanism takes care of this. It logs the outgoing transfer envelopes and the input record that is processed when a new envelope is created. This way, the system does not write log entries for every record, but for a transfer envelope that may include several records. A failed task can then ask its successors for the next expected envelope and skip an input that is not needed to process the expected data. The evaluation shows that this lightweight logging solution offers a great decrease in recovery time.

CHAPTER 6

Conclusion

Contents

This thesis covers fault tolerance mechanisms in parallel data flow execution engines. The main focus of this thesis is to use intermediate data that the system has to produce during runtime to achieve a fast recovery rollback in case of failure.

Failures are especially common in cloud environments, as they consist of many components that may fail. Faults lead to the re-execution of jobs, with additional cost for the user and additional stress to the environment for using additional computing resources. A fault tolerant data flow system should be able to reduce the

reprocessing to a minimum to avoid those extra costs. This thesis introduces three main challenges for the fault tolerance: Covering multiple failure types, low cost, and transparency. The following sections recapitulate the mechanisms, introduces future work and discusses how they approach reaches the defined goals.

6.1 Recapitulation

This section recapitulates all described methods for fault tolerance in parallel data flow systems. The fault tolerance mechanism in this thesis aims to offer a broad fault tolerance with small overhead, and high transparency to the user. Each subsection covers a previous chapter and summarizes the idea and findings of the approaches.

6.1.1 Hardware-Faults

The term Hardware-fault stands for faults that are transient. A transient fault occurs just for an endless period of time. This makes these faults hard to detect but easy to recover from. A system can recover from a transient fault by restarting the system on the same or on different hardware. As the fault does not persist over time, the restart can enable the system to run smoothly afterwards.

In the context of data flow processing, a restart of a component of a system causes the entire running job to restart, because all other components in the system depend directly or indirectly on data of the restarted node. In the data flow processing system, where the nodes pipeline data between them and not saved to persistent storage, all data has to be reprocessed once one node has to restart. Unless the system runs in an at-least-once semantic any node that has received data from the failed node has to restart to avoid duplicates. This leads to a rollback propagation, where one failing node causes all other nodes to restart. Obviously, this is not an ideal behavior for fault tolerance. Even though the system might be able to restart a failed job automatically, it is a poor fault tolerance mechanism regarding transparency and supplementary cost.

Therefore chapter 3 introduces the idea of ephemeral materialization points. Those ephemeral materialization points take advantage of the intermediate data the tasks in the job produce. They can save the intermediate data to persistent storage, to use it for recovery in case of a failure. The intermediate data serves as input for the inner task of the job and can thus avoid the restart of the entire job. However, there is a reason why data flow systems usually do not save intermediate data, as it slows down the system and takes up disk space. Saving all intermediate data for a job is insufficient. The key to economic fault tolerance is to decide where to

save intermediate data, and where to avoid it. This means to create materialization points at a spot, where the amount of data is small or at tasks that are important for recovery (e.g., for a task that splits the data flow).

One main challenge is to find those spots, without any knowledge of the job before it is running. As the tasks of the job are black boxes for the engine, it is not possible to make assumptions about the behavior before the job is actually running. Instead of using a sample run to detect the tasks that should materialize data (which would add runtime), the ephemeral materialization points hold the output data in memory and monitor the task during the production of this output. Once the memory is full, the materialization point decides either to write all data from memory to persistent storage and materialize all upcoming output or discards the data and does not materialize anything else. During materialization and the monitoring phase, the task still pipelines data to the data consumer. The materialization point writes data and afterward sends it over the network immediately.

This ephemeral materialization point technique offers a fault tolerance approach that is fast in failure-free case, as the tasks still pipeline the data and materialization points do not occur at all positions in the graph. As the algorithm aims to save data only at locations where the intermediate data is small, the approach is also space saving compared to another method, where the system would collect all intermediate data.

Ephemeral materialization points enable the system to recover from hardware failures without restarting the entire job, while being fast and space-saving. The evaluation shows that the usage of ephemeral materialization points does add runtime overhead in a neglectable amount while enabling a fast recovery in case of a failure.

6.1.2 Software- and Data-Faults

After the introduction of a fault tolerance mechanism for Hardware-faults, the next chapter covered software and Data-faults. Software- and Data-faults are persistent faults, in contrast to the previously described Hardware-faults. A persistent fault does not vanish over time, it occurs every time the job is started in the same configuration. It is therefore not suitable to just restart the whole or parts of the system, in the manner described before. The restarted job runs into the same fault every time it is restarted. This only leads to additional runtime, while the system tries several restarts, until the job finally fails.

The two types of faults deeply associate with each other. A Data-fault is a caused by a flaw in the data, which causes the user-defined function to fail. These faults occur apparently because the programmer made assumptions about the data. Accordingly,

the user code does not handle exceptions in the data that do not comply with those assumptions. This is indeed a bug, which is the definition of a Software-fault. However, the two types of faults appear differently to the system.

A Software-fault is a fault that the system is not able to detect. A Software-fault does not crash the UDF or machine and does not lead to an exception. From the viewpoint of the system, a job with a Software-fault runs smoothly and it can thus not mark the job as faulty. The user, however, may detect the fault in the job, based on wrong output data. In contrast to that, the Data-faults do cause exceptions and are therefore detectable by the system. Nevertheless, the Data-faults are still persistent. The system cannot recover from a Data-fault by a simple restart.

If the system wants to avoid the failure caused by a Data-fault, it can only make sure that the UDF does not get the deficient data in its input. If the data contains only a few flawed data sets, it may be possible to run the job, even though on a reduced input. Section 4.1 introduces the record skipping technique. This approach identifies the record that the UDF processed during a failure. If the UDF fails at the same record after the restart, the system skips the record on the next restart. This changes the output, and this technique might not be appropriate for any job. It is, however, an input checking, which the programmer of the UDF may have implemented the same way if he anticipated the possible flaws in the data. With the record skipping method, it can be possible to finish a job run that would otherwise fail. As mentioned, it is not a suitable technique for any job, and the user has to explicitly configure that the system is allowed to use the record skipping method.

For Software-faults, it is impossible to offer classic fault tolerance. As the system is not aware of the fault, it can naturally not recover from it. Only the user can correct the error in the code send an updated job to the system, that does not contain the error anymore and leads to a fault-free job run, with the correct output. The system is not able to provide fault tolerance in this scenario, but it may speed up the re-execution. Section 4.2 introduces a possibility to re-use materialization points of previous job runs, if the user indicates the similarities of two jobs. If the programmer updates parts of a previously executed job, and give the information that the new job is an updated version, the system may use materialization points from previous job runs.

This permits to speed up execution time. If the execution engine reuses materialization point, it is equal to skipping any execution previous to the materialization point. The job may not have to execute every task, but the task after the materialization point the system found to be suitable for memoization. This technique can also be useful in a nonfailure case, where the job changed programmatically. It might even be helpful for upper layer optimizations if different job versions are committed.

The evaluations show that the approach can enable the system to recover from faults, that would otherwise cause a fault of the entire job. The overhead of the memoization technique is negligibly small. The overhead of the record skipping approach however cannot be measured, as the job would not finish without the technique.

6.1.3 Recovery Optimization

The introduced ephemeral materialization points offer a fast fault tolerance for transient faults. It is fast in comparison to a complete restart of a job, and it is faster than saving all intermediate data. However, it is still slower than a job run without failure. This is because the restarted tasks must reprocess all input data, even though the successors might have already received the outputs of the head of the input stream. This reprocessing is necessary because the system allows stateful tasks. If a UDF has an internal state, it is required to reprocess the entire input to reach the same inner state again, before processing the next portion of input data.

If a job contains stateless task, it may be possible to speed up the recovery process. Chapter 5 introduces two techniques that can reduce the time the system needs to recover from a failure: *Adaptive Recovery* and *Offset Logging*. The former introduces new resources to the execution, which take up work, that would pile up during recovery. The latter keeps track of what parts of the input a task has already processed and send to the consumer.

Chapter 5.1 discusses the adaptive recovery technique. Assuming the job consists of stateless tasks it is possible to add additional resources and spread the work of a task to another task instance. As described before, the recovery always restarts the failed task (and if necessary, its predecessors that did not materialize) from the first record of the stream. The failed task has to reprocess all its input data even if its successors have already seen the corresponding output. It processes the data, and discards every output that the successors have already seen. This is even necessary if the task is stateless because the task cannot set the output into proportion to the input, the consuming task may be able to tell the failed task, which output it expects next, but the task cannot decide which part of the input it can skip from this.

Therefore, the failed task redoes previous work, during recovery. At the same time, the new work piles up in the materialization point. Adding additional resources during recovery, that steal the work form the recovering task can speed up the job's recovery time. If the system adds another task instance, it does work which the failed task should do but cannot because it is busy recovering. The measurements show that the adaptive recovery method is only suitable for one additional vertex. If

the system adds more than one vertex, it does not speed up recovery more, because it exceeds the optimal degree of parallelization.

The recovery logic adds resources if it has an idling resource available. If the user indicates the system is allowed to provision additional resources, it provisions resources if the estimated work the additional vertex takes at least one lease period. This estimation is done with a technique called *progress forwarding*, that is introduced in section 5.1.2.

Even though the adaptive recover technique can reduce the recovery time for stateless tasks, it is dissatisfactory that the failed task has to reprocess data, just to discard the output afterwards. Section 5.2 introduces a method that aims to handle this drawback.

Considering the failed task is stateless, it is per se not necessary to redo previous work. The task does not hold an internal state, and must not reproduce it with the reprocessing of all input data. However, as described above the task cannot set the output in relation to the input. If a successor informs the task, that it expects the 120th envelope next, the task does not know which input records it can skip.

With the *offset logging* technique, this becomes possible. It logs the current input record, for every sent output envelope. Additionally, it has to save information about the state of the output connections. Even though the task may be stateless, the wrapping environment does have a state. With this information, a task that restarts is able to calculate which parts of the input it can skip. Then the restarted task can jump over the input without reprocessing it.

This technique is a lightweight recovery optimization for stateless tasks. The measurements show that the offset logging adds a low overhead to the system. At the same time it provides a great opportunity for a fast recovery.

6.2 Future Work

The work presented in this thesis provides a lightweight fault tolerance technique for pipelined data flow systems. However, there is still room for improvement and future work. Two open issues are stateful tasks and stream processing.

Even though the general ephemeral materialization point fault tolerance works for stateful tasks, some key features are not usable for them. The system cannot monitor them like stateless tasks, it cannot scale unconditionally, and it always has to rollback entirely to reproduce their internal state. Stateful tasks are a major challenge for further optimization of the recovery process.

One solution for stateful tasks is for the upper layer or programmer to provide additional information about the internals of the task. The PACT layer could provide information about the estimated projection or selection ratio; or the type of task (Map, Join, etc.) This can help during the materialization decision process. Information about the current record could enable the system to use record skipping for stateful tasks.

Another approach to handle stateful tasks could be to change the materialization technique in general for those tasks. To avoid rolling back a stateful task to its initial state, the system would have to save the state of the task, with the materialized data. Unfortunately saving the state of the task is not a trivial thing. As the programmer can do whatever he wants in the UDF Java code, the state could, for example, include files written to disk, and the Nephele engine could not just make a snapshot of the JVM. Thus, writing the state of a stateful task would either mean to make a snapshot of the virtual machine it runs on, or to force the programmer to provide a method that does the snapshot and includes all necessary information.

The second issue, stream processing, depends on this partial rollback for a stateless task, if a stream processing job contains one. To provide fault tolerance for stream processing, the system must ensure the materialization points do not grow infinitely. With the current ephemeral materialization points, the materialization point files would grow endlessly as long as the system receives stream data. Furthermore, any task that restarts would reprocess the entire stream. To adapt the ephemeral materialization point technique for stream processing, the materialization points have to change from a materialization of the entire output to a sliding window. A materialization point should only contain the data, that is necessary to reproduce any unsaved data down the stream. If a succeeding materialization point contains all data that come from a portion of data in the first materialization point, the first one can discard it.

The first step in this direction is already done with the offset logging. It enables the task to set output data in ratio to input data. If it writes its output to disk, it could be able to acknowledge the input that produced this output. The task receiving the acknowledgment for output, can ack the corresponding input and so on. Another materializing task can then remove acknowledged data from its materialization point. If a job, however, contains a stateful task, it can only work with that sliding window if it can rollback to a state other than the initial state.

Additionally to that this acknowledgment technique could probably also be used to indicate a flawed job input record, for a record that causes an inner task to fail. Or at least curtail the candidate input records.

6.3 Discussion

The thesis aims to answer one central question.

> *Given the restrains of parallel data flow systems in IaaS Clouds, how can fault tolerance be achieved in a transparent, fast and space-saving manner for several types of faults?*

This section discusses how this question was addressed in the thesis, and if the provided techniques can fulfill the requirements that the initial problem raises. It starts with the design decisions that build the basis of the implementations and discusses each of the three requirements, fault tolerance, transparency, and cost for each provided approach in this work.

6.3.1 Design decisions

The goal for the design of the fault tolerance approaches was to avoid changes in the internals of the general system design wherever possible. If the fault tolerance mechanisms are not enabled, the Nepehele execution engine works in the same way as it used to before the fault tolerance technique was implemented.

The implementation of the approaches focuses on the fault tolerance aspects. It does not cover optimizations or improvement of the system itself.

6.3.2 Fault tolerance

The requirement of fault tolerance for a system was indicate as the ability of the system to *react properly to various kinds of failures. That means noticing the failure and recovering from it.*

The basic idea of ephemeral materialization points enables the system to notice task and machine failures and to recover from the failure if it is a transient fault. If the fault is transient, the system can finish the job after the recovery. The ephemeral materialization points do not offer recovery for permanent faults. As the recovery with ephemeral materialization is based on a partial restart, it fails the job finally at some point if the fault is permanent.

The record skipping technique offers an extension to the materialization points, which enables the system to recover from permanent faults if a particular record causes it. The job, however, does not run on the entire input if the record skipping technique is used.

As discussed before, the memoization technique does not provide classical fault tolerance. In fact, it does not fulfill the requirement, as the system is not able to notice the faults, that the memoization technique tries to tackle.

6.3.3 Transparency

The transparency requirement was summarized as: *Ideally the user does not even notice the fact that a failure occurred. The entire failure recognition and recovery should be as transparent to the user as possible. And the user should be able to run the same jobs, he used to run on the system before it implemented fault tolerance.*

The initial ephemeral materialization points fulfill this requirement. The user runs the same job on the system with ephemeral materialization points, as he would in the basic system. It produces the same output data for the same inputs. In a nonfailure case, the user does not even recognize an additional runtime. In case of a failure, the system detects and recovers from the failure without input from the user. The entire recovery process is transparent to the user. He recognizes additional runtime at most.

Memoization and record skipping do not comply to transparency fully. For Memoization, the usage of materialization points is transparent to the user, but the user has to change the job to use the memoization technique. For record skipping the user does not have to change the job, and the recovery process is transparent to the user. It runs without interaction or input from the user. However, the job does not run on the entire input. The system informs the user, about the failure and the output may not include all expected output data.

The optimization techniques, which include consumption logging, adaptive recovery and offset logging increase the transparency regarding runtime. The lower the additional runtime for a recovery process, the less recognizable it is to the user.

6.3.4 Cost

The cost requirement includes two central cost factors, disk space, and runtime: *The fault tolerance should be achieved with as little additional costs as possible. Especially the runtime increase for saving necessary recovery data and recovery must*

be noticeably shorter than the complete restart of the system. Additionally, it should use as less disk space as possible.

The main focus of the ephemeral materialization point implementation is to reduce the disk space usage and recovery runtime. They are a hybrid option, between the two extremes: To write no data to disk, which is fast during nonfailure case, but slow during recovery. Or to write all data to disk, which is slow during nonfailure start, but has fast recovery time. However, writing all data to the disk can also be slower as not writing any data, if the writing overhead exceeds the runtime gain during recovery.

The optimization techniques reduce the supplementary cost even further. They focus on the runtime optimization and aim to reduce the time to recover. Consumption logging reduces the number of restarted nodes, and the offset logging reduces the recovery time for the failed vertex.

The supplementary cost of the adaptive recovery approach depends on the situation. If the system can use spare idle nodes, the approach can decrease the recovery time without additional cost. However, if the system has to allocate more resources or allocate them longer as previously planned, the adaptive recovery technique adds monetary cost for the reduction of runtime cost.

The memoization technique reduces the cost of an updated job run if previous materialization points can be used. At the same time, the system must write the materialization points to persistent storage which might add monetary cost for the rental of the storage space.

The record skipping does not directly reduce cost. The job runs longer with the record skipping technique, as it has to be partially related several times. However, as the job would fail without the record skipping approach, at least the time and resources of the execution previous to the fault are saved. Nevertheless, the record skipping technique adds runtime compared to the nonfailure case and does not fulfill the cost requirement.

6.4 Conclusion

If the system has to handle a transient fault in a job with stateless tasks, it can recover from it transparently with almost no additional runtime and low disk space usage. If the system has to handle permanent faults it may be able to recover from it, but with drawbacks to the stated optimal requirements. If the job consists of stateless tasks, the additional runtime can be decreased even further with offset logging. This technique also provides the first step towards adaption of ephemeral

materialization points to streaming environments.

It is possible to provide fault tolerance for multiple fault types that is transparent at low cost. However, the more advanced the fault tolerance technique is, the more kinds of faults it is trying to cover, and the lower it tries to bring the cost, the decreasingly it fulfills every requirement. Fault tolerance will always be a tradeoff between different competing goals.

APPENDIX A

Supplementary Information

A.1 Tables

property	default value
checkpoint.mode	never
checkpoint.upperbound	1.0
checkpoint.lowerbound	0.2
checkpoint.useCL	true
checkpoint.useDuplicat	false
checkpoint.numberDuplicats	1

A.1.1 Percentages for evaluation from chapter 3.8

Triangle Enumeration

One quarter of runtime

	Build Triads	Close Triads	Triangle Output
never	100	100	100
dynamic	86	91	71
always	204	271	235
network	167	212	223

Half of runtime

	Build Triads	Close Triads	Triangle Output
never	100	100	100
dynamic	87	89	60
always	224	196	163
network	138	164	152

Three quarter of runtime

	Build Triads	Close Triads	Triangle Output
never	100	100	100
dynamic	86	88	59
always	225	174	135
network	135	113	129

No failure

never	100
dynamic	101
always	243
network	170

TPCH

One quarter of runtime

	LineItems	Join
never	100	100
dynamic	86	73
always	136	120
network	92	78

Half of runtime

	LineItems	Join
never	100	100
dynamic	87	73
always	136	114
network	91	81

Three quarter of runtime

	LineItems	Join
never	100	100
dynamic	81	69
always	97	97
network	86	75

No failure

never	100
dynamic	101
always	160
network	110

A.2 List of Abbreviations

SQL Structured Query Language

VM virtual machine

UDF User Defined Function

TE TransferEnvelope

IaaS Infrastructure as a Service

DoP Degree of Parallelization

APPENDIX B

Example Code

```
public class LineReader extends AbstractFileInputTask {

        private RecordWriter<StringPathRecord> output;

        @Override
        public void registerInputOutput() {
                this.output = new RecordWriter<StringPathRecord>(this)
                    ;
        }

        @Override
        public void invoke() throws Exception {
                final Iterator<FileInputSplit> inputSplits =
                    getFileInputSplits();

                while(inputSplits.hasNext()) {
                        FileRecord fileRecord;
                        FileInputSplit split = inputSplits.next();
                        fileRecord = FileRecord.fromSplit(split);
                        BufferedReader br = new BufferedReader(new
                            InputStreamReader(fileRecord.
                            getInputStream()));

                        String line = null;
                        int lineNumber=1;
                        while ((line = br.readLine()) != null) {
                                System.out.println(line);
```

```
                                this.output.emit(new StringPathRecord(
                                    line, lineNumber, fileRecord.
                                    getPath()));
                        }

                        br.close();
                }
        }
}
```

Listing B.1: The LineReader Task

```
package examples;

import eu.stratosphere.nephele.io.RecordReader;
import eu.stratosphere.nephele.io.RecordWriter;
import eu.stratosphere.nephele.template.AbstractTask;

public class WordSplit extends AbstractTask {

        private RecordReader<StringPathRecord> input;
        private RecordWriter<StringPathRecord> output;

        @Override
        public void registerInputOutput() {
                this.output = new RecordWriter<StringPathRecord>(this)
                    ;
                this.input = new RecordReader<StringPathRecord>(this,
                    StringPathRecord.class);
        }

        @Override
        public void invoke() throws Exception {
                while (this.input.hasNext()) {
                        StringPathRecord record = input.next();
                        String line = record.getString();
                        String[] words = line.split("␣");
                        for (int i = 0; i < words.length; i++) {
                                this.output.emit(new StringPathRecord(
                                    words[i], record.getLineNumber(),
                                    record.getPath()));
                        }
                }

        }

}
```

Listing B.2: WordSplit Task

```
public class IndexBuilder extends AbstractTask{
        private Map<String, IndexRecord> hashmap = new HashMap<String,
             IndexRecord>();
        private RecordReader<StringPathRecord> input;
        private RecordWriter<IndexRecord> output;

        @Override
        public void invoke() throws Exception {

                while (input.hasNext()) {
                        StringPathRecord record = input.next();
                        insertIntoInvertedIndex(record.getString(),
                            record.getPath());
                }
                Iterator<String> keyIter = hashmap.keySet().iterator()
                    ;
                while(keyIter.hasNext()) {
                        final String key = keyIter.next();
                        final IndexRecord record = hashmap.get(key);
                        this.output.emit(record);
                }
        }

        private void insertIntoInvertedIndex(String keyword, Path
            document) {

                IndexRecord record = hashmap.get(keyword);
                if (record == null) {
                        record = new IndexRecord();
                        record.key = keyword;
                        hashmap.put(keyword, record);
                }
                if (!record.documents.contains(document)) {
                        record.documents.add(document);
                }
        }

        @Override
        public void registerInputOutput() {
                this.input = new RecordReader<StringPathRecord>(this,
                                StringPathRecord.class);
                this.output = new RecordWriter<IndexRecord>(this,new
                    DefaultChannelSelector<IndexRecord>());
        }

}
```

Listing B.3: Index Builder Task

```
        private static Path path;
        private byte[] buffer = null;
```

```
        private class FileRecordInputStream extends InputStream {
                private final FileRecord fileRecord;
                private int bytesReadFromStream;;
                public FileRecordInputStream(FileRecord fileRecord) {
                        this.fileRecord = fileRecord;
                        this.bytesReadFromStream = 0;
                }

                @Override
                public int read() throws IOException {
                        if (this.bytesReadFromStream >= this.
                            fileRecord.buffer.length) {
                                return -1;
                        }
                        return this.fileRecord.buffer[this.
                            bytesReadFromStream++];
                }

                @Override
                public int read(byte[] b, int off, int len) throws
                    IOException {
                        if (this.bytesReadFromStream >= this.
                            fileRecord.buffer.length) {
                                return -1;
                        }
                        len = Math.min(len, (this.fileRecord.buffer.
                            length - this.bytesReadFromStream));
                        System.arraycopy(this.fileRecord.buffer, this.
                            bytesReadFromStream, b, off, len);
                        this.bytesReadFromStream += len;
                        return len;
                }

                @Override
                public int read(byte[] b) throws IOException {
                        return read(b, 0, b.length);
                }

                @Override
                public void reset() {
                        this.bytesReadFromStream = 0;
                }

                @Override
                public void close() {
                        reset();
                }

                @Override
                public long skip(long n) {
                        final int bytesToSkip = (int) Math.min(n, (
                            this.fileRecord.buffer.length - this.
```

```
                        bytesReadFromStream));
                this.bytesReadFromStream += bytesToSkip;
                return bytesToSkip;
        }

        @Override
        public int available() {
                return (this.fileRecord.buffer.length - this.
                    bytesReadFromStream);
        }}

    private class FileRecordOutputStream extends OutputStream {

        private final FileRecord fileRecord;
        public FileRecordOutputStream(FileRecord fileRecord) {
                this.fileRecord = fileRecord;
        }

        @Override
        public void write(int b) throws IOException {
                increaseBuffer(1);
                this.fileRecord.buffer[this.fileRecord.buffer.
                    length - 1] = (byte) b;
        }

        @Override
        public void write(byte[] b) {
                write(b, 0, b.length);
        }

        @Override
        public void write(byte[] b, int off, int len) {
                increaseBuffer(len);
                System.arraycopy(b, off, this.fileRecord.
                    buffer, this.fileRecord.buffer.length -
                    len, len);
        }

        private void increaseBuffer(int size) {
                if (this.fileRecord.buffer == null) {
                        this.fileRecord.buffer = new byte[size
                            ];
                } else {
                        byte[] oldBuf = this.fileRecord.buffer
                            ;
                        this.fileRecord.buffer = new byte[
                            oldBuf.length + size];
                        System.arraycopy(oldBuf, 0, this.
                            fileRecord.buffer, 0, oldBuf.
                            length);
                }}}
    public static FileRecord fromSplit(FileInputSplit
        fileInputSplit) throws IOException {
```

```
                if (fileInputSplit.getLength() > SIZE_THRESHOLD) {
                        throw new IOException(fileInputSplit.getPath()
                             + " is too large to be processed  "
                                        + fileInputSplit.getLength() +
                                             " <> " + SIZE_THRESHOLD);
                }
                setPath(fileInputSplit.getPath());
                final FileSystem fs = FileSystem.get(fileInputSplit.
                    getPath().toUri());
                final FSDataInputStream fdis = fs.open(fileInputSplit.
                    getPath());
                final int length = (int) fileInputSplit.getLength();
                final FileRecord fileRecord = new FileRecord(length);
                int totalBytesRead = 0;
                fdis.seek(0);
                int bytesRead = fdis.read(fileRecord.buffer,
                    totalBytesRead, (int) length - totalBytesRead);
                while (bytesRead != -1) {
                        totalBytesRead += bytesRead;
                        bytesRead = fdis.read(fileRecord.buffer,
                            totalBytesRead,
                                        (int) Math.min(READ_SIZE,
                                            length - totalBytesRead));
                        if ((length - totalBytesRead) == 0) {
                                break;
                        }
                }
                fdis.close();
                return fileRecord;      }

        public static FileRecord fromFile(File file) throws
            IOException {
                if (!file.exists()) {
                        throw new IOException("File does not exist");
                }
                if (file.length() > SIZE_THRESHOLD) {
                        throw new IOException(file + " is too large to
                            be processed");
                }
                final FileInputStream fis = new FileInputStream(file);
                final FileRecord fileRecord = new FileRecord(file.
                    length());
                int totalBytesRead = 0;
                int bytesRead;
                while (true) {
                        bytesRead = fis.read(fileRecord.buffer,
                            totalBytesRead, (fileRecord.buffer.length
                            - totalBytesRead));
                        if (bytesRead == -1) {
                                break;
                        }
                        totalBytesRead += bytesRead;
                        if (totalBytesRead >= fileRecord.buffer.length
```

```
                        ) {
                            break;
                    }
            }
            fis.close();
            return fileRecord;}

    private FileRecord(int size) {
            this.buffer = new byte[size];
    }

    public FileRecord() {
    }

    @Override
    public void read(DataInput in) throws IOException {
            final int bufferSize = in.readInt();
            this.buffer = new byte[bufferSize];
            in.readFully(this.buffer);
    }

    @Override
    public void write(DataOutput out) throws IOException {
            out.writeInt(this.buffer.length);
            out.write(this.buffer);
    }

    public InputStream getInputStream() {
            return new FileRecordInputStream(this);
    }

    public OutputStream getOutputStream() {
            return new FileRecordOutputStream(this);
    }

    public byte[] getBuffer() {
            return this.buffer;
    }

    public Path getPath() {
            return path;
    }

    public static void setPath(Path path) {
            FileRecord.path = path;
    }}
```

Listing B.4: File Record

```
                private int lineNumber = 0;
                private Path path;
                public StringPathRecord(String string, int lineNumber,
                     Path path) {
                        if(string != null){
                                this.string = string;}
                        this.setLineNumber(lineNumber);
                        if(path != null){
                                this.path= path;
                        } }

                public StringPathRecord() {
                }

                public String getString() {
                        return string;
                }

                public void setString(String string) {
                        this.string = string;
                }

                public int getLineNumber() {
                        return lineNumber;
                }

                public void setLineNumber(int lineNumber) {
                        this.lineNumber = lineNumber;
                }

                public Path getPath() {
                        return path;
                }

                public void setPath(Path path) {
                        this.path = path;
                }
                @Override
                public void read(DataInput in) throws IOException {
                        string = StringRecord.readString(in);
                        lineNumber = in.readInt();
                        path= new Path(StringRecord.readString(in));
                }
                @Override
                public void write(DataOutput out) throws IOException {
                        StringRecord.writeString(out, string);
                        out.writeInt(lineNumber);
                        StringRecord.writeString(out, path.toString())
                            ;
                }}
```

Listing B.5: StringPath Record

Bibliography

[1] o. t. P. S. O. The White House, "Fact sheet: President obama's precision medicine initiative," Jan 2015, https://www.whitehouse.gov/the-press-office/2015/01/30/fact-sheet-president-obama-s-precision-medicine-initiative; last visit 20. November 2015.

[2] M. by Gaillard and S. Pandolfi, "Cern data centre passes the 200-petabyte milestone," Jul 2017. [Online]. Available: http://cds.cern.ch/record/2276551

[3] P. Cabena, P. Hadjinian, R. Stadler, J. Verhees, and A. Zanasi, *Discovering Data Mining: From Concept to Implementation.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[4] L. Li, W.-Y. Cheng, B. S. Glicksberg, O. Gottesman, R. Tamler, R. Chen, E. P. Bottinger, and J. T. Dudley, "Identification of type 2 diabetes subgroups through topological analysis of patient similarity," *Science Translational Medicine*, vol. 7, no. 311, pp. 311ra174–311ra174, 2015.

[5] A. De Mauro, M. Greco, and M. Grimaldi, "What is big data? a consensual definition and a review of key research topics," *AIP Conference Proceedings*, vol. 1644, no. 1, pp. 97–104, 2015. [Online]. Available: http://scitation.aip.org/content/aip/proceeding/aipcp/10.1063/1.4907823

[6] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] T. White, *Hadoop: the definitive guide.* O'Reilly, 2012.

[8] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: the pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, 2009.

[9] C. Olston, B. Reed, S. Utkarsh, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD Conference*, 2008, pp. 1099–1110.

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[11] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[12] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE*, 2011, pp. 1151–1162.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 59–72.

[14] D. Warneke and O. Kao, "Nephele: efficient parallel data processing in the cloud," in *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*. New York, NY, USA: ACM, 2009, pp. 1–10.

[15] M. Giovannozzi, P. Skands, I. Zacharov, P. Jones, A. Harutyunyan, N. Høimyr, E. McIntosh, B. Segal, F. Grey, L. Rivkin *et al.*, "Lhc@ home: A volunteer computing system for massive numerical simulations of beam dynamics and high energy physics events," in *Conf. Proc.*, vol. 1205201, no. CERN-ATS-2012-159, 2012, p. MOPPD061.

[16] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[17] Amazon, "Amazon elastic compute cloud (amazon ec2)," Website, https://aws.amazon.com/ec2/; last visit 16. November 2015.

[18] Telekom, "Telekomcloud for business customers," Website, https://cloud.telekom.de/en/; last visit 31. Oktober 2017.

[19] P. Mell and T. Grance, "The nist definition of cloud computing," Website, Juli 2009, (Version 15, 10-7-09). Online verfügbar unter http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc; besucht am 28. September 2010.

[20] Amazon, "Amazon ec2 service level agreement," Website, June 2013, https://aws.amazon.com/ec2/sla/; last visit 16. November 2015.

[21] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, pp. 205–216, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/2528521.1508269

[22] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "Greencloud: A new architecture for green data center," in *Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session*, ser. ICAC-INDST '09. New York, NY, USA: ACM, 2009, pp. 29–38. [Online]. Available: http://doi.acm.org/10.1145/1555312.1555319

[23] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Computer*, vol. 22, no. 2, pp. 63–70, Feb 1989.

[24] P. Banerjee, R. Friedrich, C. Bash, P. Goldsack, B. Huberman, J. Manley, C. Patel, P. Ranganathan, and A. Veitch, "Everything as a service: Powering the new information economy," *Computer*, vol. 44, no. 3, pp. 36–43, 2011.

[25] Microsoft, "Microsoft azure: Cloud computing-plattform und -dienste," Website, https://azure.microsoft.com; last visit 31. Oktober 2017.

[26] U. Rackspace, "Inc.,the rackspace cloud,," 2010, https://www.rackspace.com; last visit 31. Oktober 2017.

[27] DataPipe, "Gogrid - a datapipe company | datapipe," Website, https://www.datapipe.com/gogrid; last visit 12. November 2017.

[28] P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour, *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[29] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[30] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras, "Asterix: towards a scalable, semistructured data platform for evolving-world models," *Distributed and Parallel Databases*, vol. 29, no. 3, pp. 185–216, 2011.

[31] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[32] R. Hanmer, *Patterns for fault tolerant software*. John Wiley & Sons, 2013.

[33] P. Jalote, *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994.

[34] J.-C. Laprie, "Dependability: Basic concepts and terminology," in *Dependability: Basic Concepts and Terminology*. Springer, 1992, pp. 3–245.

[35] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang, "Failure data analysis of a large-scale heterogeneous server environment," in *Dependable Systems and Networks, 2004 International Conference on*. IEEE, 2004, pp. 772–781.

[36] J. Dean, "Experiences with mapreduce, an abstraction for large-scale computation," in *PACT*, vol. 6, 2006, pp. 1–1.

[37] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[38] Oracle, "Mxbeans," Website, https://docs.oracle.com/javase/tutorial/jmx/mbeans/mxbeans.html; last visit 31. Oktober 2017.

[39] J. Cohen, "Graph twiddling in a mapreduce world," *Computing in Science Engineering*, vol. 11, no. 4, pp. 29–41, july-aug. 2009.

[40] A. Ghazal, R. Bhashyam, and A. Crolotte, "Block optimization in the teradata rdbms," in *Database and Expert Systems Applications*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2736, pp. 782–791.

[41] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs, "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems." *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 5, pp. 546–554, 1995.

[42] B. K. Bhargava and S.-R. Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems - an optimistic approach." in *Symposium on Reliable Distributed Systems*, 1988, pp. 3–12. [Online]. Available: http://dblp.uni-trier.de/db/conf/srds/srds88.html#BhargavaL88

[43] D. Johnson, "Efficient transparent optimistic rollback recovery for distributed application programs," in *Reliable Distributed Systems, 1993. Proceedings., 12th Symposium on*, 1993, pp. 86–95.

[44] J.-A. Quiane-Ruiz, C. Pinkel, J. Schad, and J. Dittrich, "Rafting mapreduce: Fast recovery on the raft," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, april 2011, pp. 589–600.

[45] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta, "Making cloud intermediate data fault-tolerant," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 181–192. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807160

[46] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endow.*, vol. 1, pp. 574–585, August 2008. [Online]. Available: http://dx.doi.org/10.1145/1453856.1453920

[47] L. Su and Y. Zhou, "Passive and partially active fault tolerance for massively parallel stream processing engines," *IEEE Transactions on Knowledge and Data Engineering*, 2017.

[48] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[49] D. Michie, ""memo" functions and machine learning," *Nature*, vol. 218, no. 5136, pp. 19–22, Apr. 1968. [Online]. Available: http://dx.doi.org/10.1038/218019a0

[50] W. Pugh and T. Teitelbaum, "Incremental computation via function caching," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/75277.75305

[51] I. Elghandour and A. Aboulnaga, "Restore: Reusing results of mapreduce jobs," *Proceedings of the VLDB Endowment*, vol. 5, no. 6, pp. 586–597, 2012.

[52] L. Popa, M. Budiu, Y. Yu, and M. Isard, "Dryadinc: Reusing work in large-scale computations," in *USENIX workshop on Hot Topics in Cloud Computing*, 2009.

[53] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters." in *OSDI*, vol. 10, 2010, pp. 1–8.

[54] D. Battré, M. Hovestadt, B. Lohrmann, A. Stanik, and D. Warneke, "Detecting bottlenecks in parallel dag-based data flow programs," in *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*. IEEE, 2010, pp. 1–10.

[55] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *2015 IEEE 35th International Conference on Distributed Computing Systems*, June 2015, pp. 399–410.

[56] F. Pan, W. Wang, A. K. H. Tung, and J. Yang, "Finding representative set from massive data," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, Nov 2005, pp. 8 pp.–.

[57] J. Zhang, G. Chen, and X. Tang, "Extracting representative information to enhance flexible data queries," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 6, pp. 928–941, June 2012.

[58] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling." in *CIDR*, 2017.

[59] B. Liu and H. V. Jagadish, "Datalens: Making a good first impression," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 1115–1118. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559997

[60] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[61] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 2015, pp. 399–410.