PARALLELIZATION OF USER-DEFINED FUNCTIONS IN AN ETL WORKFLOW

Author:
Syed Muhammad Fawad Ali

Supervisor:
Prof. Robert Wrembel

April 13, 2021

POZNAN UNIVERSITY OF TECHNOLOGY

Abstract

ETL workflows are important components in all data integration architectures, including data warehouses (DWs), data lakes, and data science applications. They are responsible for: 1) ingesting data from data sources (DSs), 2) transforming heterogeneous data into a common data model and schema, 3) cleaning, normalizing, and eliminating data duplicates, 4) loading data into a central repository - a DW.

An ETL workflow moves large volumes of data between DSs and a DW, and executes complex cleaning and data transformation tasks. Its execution is time consuming and typically takes hours to complete. Moreover, the volume, variety, and velocity of data are growing at a record rate, making data very large and complex to the point that storage and analysis of massive and complex data sets exceeds the capacity of conventional computer systems and algorithms. Especially the variety of today's data but also the wide range of different analytical use cases increasingly exceed the expressiveness of ETL tools. Therefore, most ETL tools provide the functionality to write custom code as the so-called User-Defined Functions (UDFs), which may be written by the ETL developer. A UDF can be used to perform aggregations or any kind of run-time intensive computations on a data set that may be necessary before loading it into a DW. However, a UDF written by the ETL developer may be more prone to errors and less efficient due to its poor code and high computational complexity. As a result, a poorly coded UDF may result in a performance bottleneck for an overall ETL workflow.

To this end, the focus of this study is on the optimization of a computing-intensive UDF or a set of UDFs in an ETL workflow by means of parallelization. To reach our goal, first we carried out a comprehensive analysis of the state-of-the-art methods and techniques intended for each stage in the ETL development life cycle, i.e., from the conceptual and the logical design of an ETL workflow to its semantically equivalent physical implementation. Then we further extended our analysis to the state-of-the-art methodologies for the optimization of ETL workflows. We evaluated the existing methodologies for each stage with respect to their pros, cons, and challenges based

on selected metrics, such as, 1) autonomous behavior, 2) support for various data formats, 3) supports for UDFs, 4) consideration for quality metrics, 5) capability to monitor ETL workflows and to recommend an efficient alternative ETL workflow in case of a poor efficiency.

Based on our analysis we drew the following conclusions.

- Most of the methods are not fully autonomous and require ETL developers to provide a lot of input at each stage of the ETL life cycle, thus it can be error prone, time consuming, and inefficient.
- Most of the methods are designed only for structured data and the support for semi-structured and unstructured data is very limited. Since the variety of data formats is growing rapidly and most of data are unstructured, it is important to extend the support for unstructured data in an ETL workflow.
- Almost all of the discussed design methods are based on the traditional ETL tasks (e.g., join, union, sort, aggregate, lookup, and convert) and they mostly do not consider UDFs as ETL tasks. Moreover, not much research has been done on the optimization of UDFs as ETL tasks. As UDFs are commonly used in an ETL workflow to overcome the limitations of traditional ETL tasks, it is important to optimize UDFs along with traditional ETL tasks. Since a UDF is typically considered as a black box task and its semantics is unknown, it is very difficult to optimize its execution.
- A few methods put emphasis on the issues of efficient, reliable, and improved execution of an ETL workflow using quality metrics, however, most of the methods in practice do not capture or track these quality metrics.
- Currently, there is no such framework that autonomously monitors an ETL workflow to find out which ETL tasks hinder its performance and gives recommendations to the ETL developer how to increase its performance.

The outcome of the comprehensive analysis of the state-of-the-art methods was used to formulate the following research challenges, which we have addressed in this thesis:

- **The first research challenge** is to design an ETL framework to facilitate ETL developers to write efficient UDFs by separating parallelization concerns from the code and reducing potential error sources in the otherwise manual and cumbersome parallelization process.
- **The second research challenge** is to construct a cost model that generates parallelizable UDFs by first determining the feasibility and degree of parallelism for a UDF to be executed in a parallel distributed environment.

In response to the aforementioned challenges, we proposed a framework as a first step towards the fully autonomous ETL framework, which consists of four modules namely: 1) the UDFs Component, 2) the Cost Model, 3) the Recommender, and 4) the Monitoring Agent.

The **UDFs Component** is designed and developed specifically addressing the first challenge. The idea behind implementing this component is to assist the ETL developer to implement parallelizable UDFs without worrying about the complexities and technicalities of the implementation details for parallelizing and optimizing the code. The component provides a library of parallelizable code templates a.k.a Parallel Algorithmic Skeletons (PASs) designed to be executed in a distributed (or parallel) environment. The library of PASs consist of two types of parallelizable code templates: 1) the generic PASs (e.g., worker-farm model, divide and conquer, branch and bound, systolic, MapReduce) and 2) the case-based PASs i.e., already parallelizable code for the list of commonly used big data tasks (e.g., sentiment analysis, de-duplication of rows, outlier detection). In case of the generic PASs, the ETL developer has to provide the basic program for the chosen PAS, an execution time constraint to run the ETL workflow, and distributed machine specifications. For example, for the MapReduce paradigm as a PAS, only Map and Reduce functions would be required. The MapReduce configurations (i.e., partitioning parameters, number of nodes) will be provided by the **UDFs Component**. In case of the case-based reasoning, the ETL developer only has to provide the input and output data formats, execution time constraint to run the ETL workflow (e.g., the ETL job must complete execution within x number of hours. We used the so-called Orchestration Style Sheet (OSS) processor that generates the parallelizable code and multiple possible distributed machine configurations (variants) for the parallelizable code to be executed in a distributed environment.

We showed with experiments that the proposed ETL framework reduces about 50-65% of the effort required by the ETL developer to write parallel code and ensures an error free code by replacing otherwise manual steps in writing parallelizable UDFs. We also carried out experiments to understand the impact of parallelizing computing-intensive UDFs on the performance of an ETL workflow. We found out that the non-computing-intensive code normally does not affect the overall performance of an ETL workflow when it is executed either in a distributed or a non-distributed environment. However, the computing-intensive tasks may become a bottleneck in an ETL workflow and must be optimized, because even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow.

Once the multiple variants are generated, the framework hands over the generated variants to the **Cost Model** component. The **Cost Model** along with the **Recommender** and the **Monitoring Agent** addresses the solution to our second challenge. The **Cost Model** is based on the Decision Optimization and the Machine

Learning techniques in order to generate a (sub-) optimal configuration to optimize the execution of parallelizable UDFs in an ETL workflow. The Decision Optimization technique is used to find an optimized configuration for the case-based PASs, where we mapped our problem to Multiple Choice Knapsack Problem (MCKP) in order to select the optimal PAS from a set of multiple PASs (variants). For example, in this case, suppose an ETL workflow consists of m different computationally intensive UDFs, and the **UDFs Component** may generate n parallel variants of each UDF, which results in n^m combinations of code variants. Therefore, finding an optimal UDF is mapped to MCKP. The Machine Learning technique (one of the future works) is designed to handle the generic PASs. The Machine Learning models will be trained on historical data and fine-tuned, and then will be applied to find the (sub-) optimal machine configuration to execute UDFs based on the generic PASs. Moreover, if the Decision Optimization technique of the cost model fails to find the (sub-)optimal solution, the Machine Learning technique will then be used to find the solution. The **Recommender** module utilizes a library of cost models and retrieves information from the **Monitoring Agent** in order to provide recommendations to the ETL developer. The **Monitoring Agent** module is proposed to assist the **Recommender** as well as an end-to-end monitoring of ETL workflows.

To showcase how the **Cost Model** works, we carried out an experimental evaluation, where we implemented the Set-Similarity Join use case as an ETL workflow, to detect similar records in a large data set. The ETL workflow consisted of three computing-intensive UDFs, that needed to be parallelized to be executed in a distributed environment. We used a cluster of 2,4,8, and 10 nodes on Amazon Web Services to calculate the execution time of the computing-intensive UDFs. Our experiments showed that the **Cost Model** selected the best possible configuration for a set of ETL tasks (implemented as UDFs) to be executed in a distributed environment. We executed the **Cost Model** on a local 2,6 GHz 6-Core Intel Core i7 machine to further prove that it was capable to provide the optimal solution in a fraction of a second. Moreover, we proposed the extension of the **Cost Model** to enable data scientists to choose the best possible machine learning model in a Machine Learning pipeline, based on the user-defined performance metrics, e.g., model accuracy, precision, or recall, along with the maximum execution time and maximum monetary execution costs.

Streszczenie

Przepływy ETL (zwane także procesami) są jednym z najważniejszych komponentów wszystkich architektur integracji danych, m.in. systemów hurtowni danych (ang. data warehouse - DW), jezior danych (ang. data lake) i aplikacji przetwarzania danych przez data science. Przepływy te są odpowiedzialne za: 1) pobieranie danych z różnorodnych źródeł (ang. data sources - DSs), 2) transformowanie heterogenicznych danych do wspólnego modelu i schematu, 3) czyszczenie danych, normalizowanie wartości i eliminowanie duplikatów, 4) wczytywanie danych do centralnego repozytorium, jakim jest DW.

Procesy ETL transportują duże wolumeny danych pomiędzy źródłami a DW i realizują złożone zadania transformacji danych. Z tych powodów, czasy ich wykonania są długie - typowo procesy ETL wykonują się w ciągu kilku godzin. Ponadto, wolumen, różnorodność i szybkość napływania danych do systemu stale wzrastają. Jest to przypadek tzw. gigadanych (ang. big data), co powoduje problemy z wydajnością standardowych architektur przetwarzania danych. W szczególności, różnorodność gigadanych i sposoby ich przygotowania do analizy przez różnego rodzaju aplikacje (m.in. uczenia maszynowego) wymagają rozszerzenia funkcjonalności rozwiązań ETL. Tego typu rozszerzenia w praktyce są realizowane za pomocą tzw. funkcji użytkownika (ang. user-defined functions - UDFs). UDFs są implementowane przez projektanta procesu ETL w językach dostępnych w środowisku projektowym ETL. UDF może realizować dowolną operację na danych, np. eliminowanie duplikatów zestawem algorytmów adekwatnym do przetwarzanych danych, analizę sentymentu. Projektowanie złożonych UDF nie jest łatwe, a powstały kod może być niewydajny lub niewystarczająco dobrze przetestowany.

Z tego powodu, w niniejszej rozprawie adresujemy problem wydajnego przetwarzania kosztownych obliczeniowo UDF poprzez zastosowanie przetwarzania równoległego. Prace badawcze rozprawy zostały poprzedzone szczegółową analizą istniejących rozwiązań w zakresie budowania procesów ETL, tj. modelowania konceptualnego i logicznego procesów oraz ich implementowania, a także w zakresie optymalizowania

wykonania tych procesów.

Na podstawie powyższej analizy wyciągnęliśmy następujące wnioski.

- Większość zaproponowanych metod projektowania procesów ETL wymaga dużego nakładu pracy ze strony projektanta, co czyni je nieodpornymi na błędy, czasochłonnymi i niewydajnymi.
- Większość metod umożliwia projektowanie procesów ETL dla danych o dobrze określonych strukturach (np. relacyjnych), przy niewielkim wsparciu przetwarzania danych częściowo ustrukturyzowanych (ang. semi-structured) lub nieustrukturyzowanych (ang. unstructured). Ponieważ różnorodność formatów integrowanych danych się zwiększa (szczególnie w przypadku gigadanych), więc zachodzi konieczność wspierania projektowania procesów ETL także dla nowych typów danych.
- Zdecydowana większość metod wykorzystuje standardowe zadania ETL, tj. m.in. operacje łączenia tabel, sumy zbiorów, sortowanie, agregowanie, selektywne sięgnięcie do innej tabeli (ang. lookup), konwersję danych. Nie wspierają one jednak efektywnego wykonania procesów ETL z UDF. Tego typu funkcjonalność wydaje się konieczna, ponieważ UDF są w praktyce wykorzystywane bardzo często do implementowania niestandardowych zadań. Optymalizowanie procesów ETL z UDF jest bardzo trudnym wyzwaniem badawczym i nierozwiązanym do tej pory, ponieważ UDF są najczęściej traktowane jako tzw. czarne skrzynki, tj. ich semantyka nie jest znana.
- Większość metod projektowych ETL w ogóle nie uwzględnia w procesie projektowania jakości danych produkowanych przez procesy ETL.
- Wśród analizowanych rozwiązań brakuje takich, które umożliwiałyby automatyczne monitorowanie wydajności poszczególnych komponentów procesu ETL i wskazywałyby w jaki sposób zwiększyć wydajność zadań takiego procesu.

Powyższe wnioski posłużyły do sformułowania następujących wyzwań badawczych zaadresowanych w niniejszej rozprawie.

- **Pierwszym wyzwaniem badawczym** jest zaprojektowanie podejścia wspierającego projektanta procesu ETL w implementowaniu wydajnych UDF poprzez przetwarzanie równoległe. Jest to możliwe dzięki odseparowaniu właściwego kodu UDF od części definiującej sposób jego równoległego przetwarzania. Zmniejsza się w ten sposób prawdopodobieństwo postania błędów w oprogramowaniu i zwiększa produktywność projektanta.
- **Drugim wyzwaniem badawczym** jest konstruowanie modelu kosztów, który umożliwi zdefiniowanie (sub-)optymalnych parametrów architektury przetwarzania równoległego, dla zadanej UDF.

W podpowiedzi na ww. wyzwania badawcze, w ramach rozprawy zaproponowaliśmy architekturę i techniki umożliwiające zbudowanie w pełni autonomicznego systemu do zarządzania procesami ETL. W ramach tej architektury opracowaliśmy cztery rozwiązania, tj. 1) Komponent UDF (the UDF Component), 2) Model Kosztów (the Cost Model), 3) Rekomender (the Recommender), 4) Monitor (the Monitoring Agent) (w dalszej części będziemy stosowali nazwy anglielskie).

UDF Component adresuje pierwsze wyzwanie badawcze. Głównym zadaniem tego komponentu jest wspieranie projektanta w implementowaniu wykonywanych równolegle UDF, w taki sposób, aby zapewnić wydajne uruchamianie kodu. W tym celu, **UDF Component** dostarcza predefiniowaną bibliotekę szablonów przetwarzania równoległego (ang. Parallel Algorithmic Skeletons - PASs). Dostępne są dwa rodzaje szablonów, tj. 1) generyczny PASs (ang. generic PASs), np. worker-farm model, divide and conquer, branch and bound, systolic, MapReduce i 2) case-based PASs. Oba rodzaje szablonów zawierają gotowe zrównoleglone kody częstych zadań wykonywanych w procesach ETL dla gigadanych, np. analiza sentymentu (ang. sentiment analysis), deduplikowanie danych (ang. deduplication, entity resolution, entity matching), wykrywanie odchyleń (ang. outlier detection). Dla generycznego PAS, projektant pisze tylko właściwy program, specyfikuje ograniczenie czasowe na jego wykonanie i specyfikacje komputerów w środowisku rozproszonym. Przykładowo, dla szablonu MapReduce, wymagane jest podanie jedynie funkcji Map i funkcji Reduce, a konfiguracja (m.in., parametry partycjonowania, liczba węzłów) zostaną automatycznie uzupełnione przez UDF Component. Implementacja tego mechanizmu bazuje na tzw. procesorze Orchestration Style Sheet, który generuje kod w wersji do wykonania równoległego i alternatywne konfiguracje komputerów w architekturze rozproszonej dla wykonania tego kodu.

Ocena eksperymentalna omówionego rozwiązania umożliwia skrócenie od 50% do około 65% czasu koniecznego do zaimplementowania kodu równoległego. Ponadto, przeprowadziliśmy eksperymenty pozwalające zrozumieć wpływ zrównoleglania wykonania kosztownych obliczeniowo UDF na wydajność całego procesu ETL zawierającego takie UDF. Stwierdziliśmy, że zrównoleglenie tanich obliczeniowo UDF nie wpływa na zwiększenie wydajności procesu ETL, w porównaniu ze scenariuszem, w którym ta sama UDF jest wykonywana bez zrównoleglenia. Natomiast, zrównoleglenie kosztownych obliczeniowo UDF może znacząco zwiększyć wydajność całego procesu ETL.

Wygenerowane warianty kodu i konfiguracji równoległego środowiska uruchomieniowego są wykorzystywane przez komponent **Cost Model**, który wraz z komponentami **Recommender** i **Monitoring Agent** rozwiązuje drugie zadanie badawcze. **Cost Model** wykorzystuje techniki optymalizacji kombinatorycznej i uczenia maszynowego w procesie generowania (sub-)optymalych konfiguracji środowiska uruchomieniowego dla ETL z UDF. Optymalizacja kombinatoryczna jest wykorzystywana dla

case-based PAS. Wykorzystujemy tu Multiple Choice Knapsack Problem (MCKP) do wyboru optymalnego PAS spośród wielu jego wariantów. Przykładowo, rozważmy proces ETL złożony z m kosztownych obliczeniowo UDF. Dla każdej UDF, **UDFs Component** może wygenerować n jej równoległych wariantów, dając w wyniku n^m możliwych kombinacji wariantów kodu.

Z tego powodu, problem znalezienia optymalnego sposobu wykonania UDF został odwzorowany w MCKP. Jako dalszy rozwój opracowanej tu koncepcji, proponujemy zastosowanie technik uczenia maszynowego (ang. machine learning - ML) w celu optymalizacji wykonania UDF opartych o generic PASs. Model uczenia maszynowego zostanie zbudowany w oparciu o historyczne dane z wykonania UDF, a następnie wykorzystany do znalezienia (sub-)optymalnych konfiguracji komputerów w środowisku uruchomieniowym, w sytuacji gdy techniki optymalizacji kombinatorycznej nie znajdą zadowalającego rozwiązania. Moduł **Recommender** bazując na bibliotece modeli kosztów i danych otrzymanych z **Monitoring Agent**, rekomenduje rozwiązania projektowe procesu ETL. **Monitoring Agent** umożliwia gromadzenie danych z monitorowania wykonania procesów ETL.

Zaproponowane w rozprawie rozwiązanie bazujące na modelu kosztów, zostało ocenione eksperymentalnie. Jako przypadek użycia zastosowano znany i kosztowy algorytm Set-Similarity Join, służący do eliminowania duplikatów, zaimplementowany jako proces ETL. Proces zawierał m.in. 3 kosztowne obliczeniowo UDF, podlegające optymalizacji poprzez zrównoleglenie ich wykonania. W eksperymentach wykorzystano klaster Amazon Web Services z 2, 4, 8 i 10-cioma węzłami. Eksperymenty pokazały, że zaproponowany model kosztów umożliwił znalezienie najbardziej wydajnej konfiguracji klastra dla testowanego procesu ETL, przy ograniczeniu budżetu monetarnego na obliczenia. Ten sam model kosztów zaimplementowany na komputerze PC (2,6 GHz 6-Core Intel Core i7) także umożliwił znalezienie tego samego rozwiązania w czasie ułamków sekundy. Na koniec, zaproponowaliśmy rozszerzenie modelu kosztów o zadane miary, tj. dokładność modelu, prcesion, recall i monetarny koszt wykonania procesu.

Contents

Chapter 1
Introduction

1.1. Background and Motivation

An industry accepted architecture for integrating data sources (DSs) is a data warehouse (DW) architecture [97]. The integration is implemented by means of the Extract-Transform-Load (ETL) layer where the so-called ETL workflows (processes) are run. ETL workflows (a.k.a. data processing workflows, data processing pipelines, or data wrangling) are important components in all data integration architectures, including data warehouses [48, 97], data lakes [70, 93, 18, 77], and data science applications [47, 56, 75, 93]. They are responsible for: 1) ingesting data from data sources, 2) transforming heterogeneous data into a common data model and schema, 3) cleaning, normalizing, and eliminating data duplicates, 4) loading data into a central repository - a DW. An ETL workflow has to finish its work within a given time window. Since, 1) such a process moves large volumes of data between DSs and a DW, 2) executes complex cleaning and de-duplication algorithms, its execution is time consuming and typically takes hours to complete.

Since early 2000s, the volume of produced, collected, and stockpiled digital data has been continuously growing exponentially [32] to the point that storage and analysis of massive and complex data sets exceeds the capacity of conventional computer systems and algorithms. Moreover, the *velocity* (the speed at which the data are generated and analyzed), *veracity* (quality and uncertainty), and *value* (potential business value of the data) [92] of this big data provides great opportunities and harnessing that leads to great benefits in science and business. Big data technology adoption is an imperative need for most organisations to gain competitive advantage or even survive in today's world. Thus, having the right technological basis to exploit the potential of big data, to harness it, and to extract the value out of it is essential for the companies. Hence, business intelligence and analytics and the related field of big data analytics has become increasingly important areas of research over the past decade.

Big data itself requires a great scientific contribution to deal with it. For example, the volume, variety, and velocity of data are growing at a record rate making data very large and complex. Thus, single machine and existing technologies can no longer effectively store and process such an unprecedented data. This results in more and more sophisticated methods and technologies for data storage and processing. Especially the variety of today's data but also the wide range of different analytical use cases increasingly exceed the expressiveness of ETL tools [33].

As an example from the data cleansing side, the messy and noisy nature of big data demands new types of cleansing tasks (a.k.a activities or operators), such as outlier detection or de-duplication that specifically fit the ever-changing characteristics of the data. The same applies to the data analytics side where we find a zoo of algorithms such as classification, regression, clustering, collaborative filtering, and many more.

To overcome the limited expressive power provided by the standard ETL tasks, most ETL tools provide the functionality to write custom code as User-Defined Functions (UDFs). A UDF is a software program written in any programming or scripting language. These UDFs allow the ETL developer to extend the functionality of an ETL tool that is outside a scope of the already provided built-in ETL tasks. For example, an UDF can be used to perform aggregations or any kind of run-time intensive computations on a data set that may be necessary before loading it into a DW. A UDF may be written by the ETL developer who is developing a data pipeline (ETL) or by any third-party. Therefore, it may be more prone to errors and less efficient, which may result in a performance bottleneck due to its poor code and high computational complexity.

There are a few techniques to optimize the performance of an ETL workflow by reducing its execution time and/or monetary costs. State-of-the-art for optimizing ETL workflow starts from modeling ETL workflows at a conceptual, logical, and physical level. The research on conceptual modeling of ETL workflows focuses on easy and semi-automated design of ETL workflows e.g., ontology-based [89, 90] and UML-based [95] conceptual design of ETL workflows among others. There exists a work based on scheduling the ETL tasks in an ETL workflow using different scheduling policies e.g., 'fair scheduling policy', 'empty the largest input queue of the workflow first' and 'the activity with maximum tuple' [50]. More extensive research is based on modeling ETL workflows as a 'State Space Search' problem [85] and then applying different algorithms such as 'Exhaustive Search', 'Greedy Search' and 'Heuristic Search' to construct the search space in order to find the optimal execution of an ETL workflow [80, 84, 96] . Moreover, there exist multiple methods that revolve around data flow parallelism e.g., the most prominent one is the MapReduce framework [24], PACT [12] based on the Parallelization Contracts, and Selinger-style SQL optimizer [79]. These techniques are further discussed in Chapter 2 and 3. Another approach is to scale the computing systems vertically or horizontally. Scaling up vertically means

to raise single machine's (single node's) performance level by adding more resources (typically they are CPUs and memory). Scaling up horizontally means to add nodes to the system, such as adding a new computer to distributed software application. However, it requires an army of technical resources and huge amount of investment in terms of cost, time, and energy for a company to build or possess its own computational resources. Moreover, it is not a permanent fix, because at some point in time the hardware will stop supporting the processing of a compute-intensive workflows or tasks.

From the industry point of view, IBM InfoSphere DataStage [59] and Informatica PowerCenter [2], provide some simple means of optimizing ETL workflows based on *balanced optimization* and *pushdown optimization* respectively, further discussed in Chapter 2. Other tools, including AbInitio, Microsoft SQL Server Integration Services, and Oracle Data Integrator support only parallelization of ETL tasks, with a parameterized level of parallelism.

Besides introducing parallelism in ETL Workflows, there exist an unaddressed problem of the parallelization of UDFs, which is the basis and motivation of this thesis.

1.2. Scope

The focal point of this study is on the parallelization of a compute-intensive UDF or a set of UDFs (as user defined ETL tasks) in an ETL workflow. A few use cases may include a set of UDFs in an ETL workflow, for example 1) computing sentiments on a real-time news feed, 2) implementing entity matching algorithms, 3) performing deduplication of a large natural language data-set [6]. In such scenarios, the UDFs may consume a large portion of execution time, which deteriorate the overall performance of an ETL workflow. Hence, optimizing the compute-intensive UDFs is required to optimize the execution of an ETL workflow.

The scope of the thesis is two fold:

- First, designing and developing a framework that enables developers to exploit parallelization of UDFs in an ETL workflow by either writing parallelizable code with the help of parallel algorithmic templates or by choosing already parallelizable UDFs for a given use-case.
- Second, developing a cost function to check the feasibility to exploit parallelism in user-defined ETL tasks. The feasibility for parallelism is described as follows:
  - Is an UDF parallelizable?
  - If an UDF is parallelizable, will it profit from parallel processing to satisfy user-defined performance metrics (execution time, monetary cost)?

– If an UDF profits from parallel processing, what will be the adequate (sub-optimal, optimal) parallelization parameters e.g., data partitions and configuration of a distributed framework.

1.3. Research Problems and Challenges

In an ETL workflow, an UDF is normally considered as a black-box activity i.e., it is difficult to assess the run-time and space complexity of an UDF. However, if an UDF is already optimized in terms of execution time, or is configurable to be optimized by an ETL framework without changing its code, it may help ETL developers to optimize the compute-intensive UDF activities within an ETL workflow. As mentioned above, one of the techniques to optimize execution performance of UDFs is to parallelize compute-intensive UDFs in an ETL workflow.

To exploit parallelism, the ETL developer has to explicitly write UDFs that can be executed in a parallel manner e.g., in a distributed environment. And since writing efficient parallelizable programs require training and expertise in distributed and parallel programming, there is a possibility that the developer may not be able to leverage the benefit of parallelism and therefore miss performance opportunities [87].

At this point, the **first research challenge** is *to design an ETL framework to facilitate ETL developers to write efficient UDFs by separating parallelization concerns from the code and reducing potential error sources in the otherwise manual and cumbersome parallelization process.*

To optimize the execution of parallelizable UDFs in an ETL workflow, one has to consider two core aspects of parallelization:

- to determine whether it is feasible to parallelize an UDF to optimize the execution time and monetary cost;
- if it is feasible, then determine the right degree of parallelism, e.g., choosing the appropriate number of partitions to distribute the data to be transformed in parallel.

Therefore, the **second research challenge** is *to construct a cost model that generates parallelizable UDFs by first determining the feasibility and degree of parallelism for an UDF to be executed in a parallel distributed environment.*

1.4. Contributions

The first aforementioned research challenge consists in designing an ETL framework that would help ETL developers to write efficient parallelizable UDFs without worrying about the technicalities of parallelization in order to execute them in a distributed environment leveraging the full benefits of parallelism.

As a response to this challenge, we designed and developed configurable parallelizable UDF generator *the UDFs Component* to provide the ETL developer with out-of-the-box functionality in an ETL framework to write efficient parallel custom programs. The UDFs Component converts a non-parallelized code into a parallel code and can easily be integrated into any open-source ETL framework e.g., Pentaho Data Integration (Spoon) as a third-party tool. To assist the ETL developer to write parallelizable code of an UDF, we provided different Parallel Algorithmic Skeletons (PAS) or code skeletons that can be executed in a distributed environment. For example, worker-farm, divide and conquer, branch and bound, systolic, MapReduce or Spark code skeleton, where the ETL developer has to insert his/her code for a particular UDF into the provided skeleton. Currently, the UDFs Component supports Hadoop as a distributed framework to execute UDFs in a parallel environment. However, it is extensible and can be integrated with other parallel and distributed frameworks, e.g., Flink and HPC clusters.

Our second research challenge is specifically related to the parallelization concerns. As a response to this challenge, we created a cost-model that enables the optimization of already parallelizable (e.g., MapReduce-based or Spark-based) UDFs in an ETL workflow. Our optimization approach draws upon determining the right degree of parallelism for an UDF (or a set of UDFs) to satisfy performance metrics such as execution time and monetary cost of an UDF in an ETL workflow.

1.5. Thesis Overview

The main focus of the thesis is to help future researchers to effectively use methods for the parallelization of user-defined tasks in an ETL workflow to achieve maximum performance. The methodology proposed in this thesis may also be useful for the ETL industry working on parallelization of ETL tasks and ETL tools.

The key methods to conduct this thesis consisted of theoretical study as well as empirical research. A thorough literature review was conducted to help finding the open issues in this field and to develop solutions of the problems addressed in this dissertation. The following steps were followed as a key research methods of this thesis:

- A thorough systematic literature review was conducted to understand the state-of-the-art for the topic of this thesis and open issues in the related studies.
- A prototype of the proposed solution for the parallelization of user-defined tasks was developed as a proof of concept.
- Empirical assessment of the developed prototype was conducted by testing it for several scenarios.

Chapters (2-6) of this dissertations are based on the results reported in the following publications:

- **P1:** Ali, Syed Muhammad Fawad, and Wrembel, Robert: From conceptual design to performance optimization of ETL workflows: current state of research and open problems. The VLDB Journal 26(6) (2017), pp. 777-801.
- **P2:** Ali, Syed Muhammad Fawad: Next-generation ETL Framework to Address the Challenges Posed by Big Data In International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP) (2018).
- **P3:** Ali, Syed Muhammad Fawad, and Mey, Johannes, and Maik Thiele: Parallelizing user-defined functions in the ETL workflow using orchestration style sheets. International Jounral of Applied Mathematics and Computer Science (AMCS) 29(1)(2019), pp. 69-79.
- **P4:** Ali, Syed Muhammad Fawad, and Wrembel, Robert: Towards a Cost Model to Optimize User-Defined Functions in an ETL Workflow Based on User-Defined Performance Metrics In European Conference on Advances in Databases and Information Systems (ADBIS) (2019), LNCS 11695, pp. 441-456.
- **P5:** Ali, Syed Muhammad Fawad, and Wrembel, Robert: Framework to Optimize Data Processing Pipelines Using Performance Metrics In International Conference on Big Data Analytics and Knowledge Discovery (DaWaK) (2020), LNCS 12393, pp. 131-140.

Chapter 2 and 3 correspond to **P1**, where in Chapter 2 we presented the overview of DW architecture and in Chapter 3, we presented the the state-of-the-art and current trends in ETL, with a special focus on optimization. Chapter 4 corresponds to **P2**, where we proposed a next-generation ETL framework to address the challenges posed by big data. Chapter 5 corresponds to **P3**, where we extended our discussion on the proposed solution for the parallelization of user-defined tasks. Chapter 6 corresponds to **P4** and **P5**, where we presented the cost model to generate parallelizable UDFs after determining the right-degree of parallelism for the user-defined programs. Chapter 7 discusses the conclusion and the future work.

Chapter 2
State-of-the-Art in ETL Workflow Design

In this chapter, we discuss the architecture of Data Warehouse and a state of the art and current trends in designing ETL workflows. We briefly explain the existing techniques for: constructing a conceptual and a logical model of an ETL workflow, and its corresponding physical implementation, illustrated by examples. The discussed techniques are analyzed w.r.t. their advantages, disadvantages, and challenges in the context of metrics such as: autonomous behavior, support for quality metrics, and support for ETL tasks as user-defined functions. We draw conclusions on still open research and technological issues in the field of ETL.

2.1. Introduction

A traditional Data Warehouse (DW), while allows the integration of various types of data usually from transactional and operational data sources, it ensures the availability of business-specific, nonvolatile, time-variant, and not normalized data for analysis and decision making process [35]. The data originating from different data sources, may require treatment for its poor quality ranging from simple spelling errors, missing or inconsistent values, to conflicting or redundant data. The aforementioned process is called data integration process a.k.a Extract-transform-Load (ETL), which takes place between data sources and a DW, as shown in Figure 2.1. The first task of an ETL workflow is to extract data from multiple heterogeneous Data Sources (DSs), the second phase is to perform data quality checks and transformations in order to make data clean and consistent with the structure of a target DW. Finally, the third phase is to load data into a DW, which is eventually used for further analysis and reporting by mean of the dimensional modeling technique - a widely accepted technique for a DW presentation [55].

The variety of big data in terms of structured, unstructured, or semi-structured data from multiple heterogeneous data sources, as well as the volume and velocity of big data, exceeds the ability of traditional tools to extract, process, store, manage

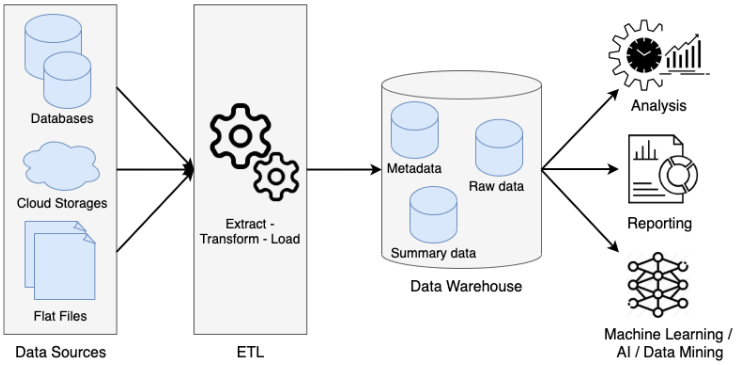

Figure 2.1: A traditional data warehouse (DW) architecture

and analyse the data [78]. This gives birth to a different DW architecture to support large volumes structured, semi-structured, and unstructured data in parallel and distributed fashion, which may or may not complement the traditional DW. To face the challenges of big data and to overcome the limitations of traditional DW, one of the new concepts was introduced, called data lakes [68]. A data lake is designed as a massively scalable storage to hold large amount of unstructured data which could further be used to derive insights.

However, simply pushing the data into a data lake cannot be readily used for analysis or further explorations e.g., machine learning. There is a need to build workflows to transform the data resided in a data lake when it is required. Unlike ETL, where data is transformed before it is loaded into a data warehouse, Extract-Load-Transform (ELT) process is typically implemented as a workflow. The motivation behind the ELT is that data lakes are not dependent on the structure of the incoming data, therefore there is no need for the data transformation before the data is stored into a data lake. The transformations are applied on the data on a need basis [74].

An ETL/ELT process is typically implemented as a workflow, where various tasks (a.k.a. activities or operators), which process data, are connected by data flows [7, 73] (note: in this thesis we refer to ETL and ELT as a same process). The tasks executed in an ETL workflow include among others: 1) extracting and filtering data from data sources, 2) transforming data into a common data model, 3) cleaning data in order to remove errors and null values, 4) standardizing values, 5) integrating cleaned data into one common consistent data set, 6) removing duplicates, 7) sorting and computing summaries, and 8) loading data into a DW. These tasks can be implemented by means of SQL commands, predefined components, or user-defined functions (UDFs) written in multiple programming languages.

There are several proprietary, cf., [4] and open source, cf., [3] ETL tools available in a business sector for designing and developing ETL workflows. The tools provide

proper documentation and graphical user interfaces to design, visualize, implement, deploy, and monitor execution of an entire ETL workflow. However, these tools have a very limited support for designing and developing efficient workflows, since automatic optimization and fine-tuning of an ETL workflow is not available. Hence, the ETL developer him/herself is responsible for producing an efficient workflow. This is one of a few reasons that make numerous organizations incline towards in-house development of such ETL tools that best suit their business needs [9, 101].

The ETL research community has proposed several methods for designing a conceptual model of an ETL workflow, which led to its semantically equivalent logical model, physical implementation, and its optimized run-time version. The set of guidelines formulated for the design of a conceptual and a logical model of an ETL workflow [83, 101], prompts the automation of a design process, in order to facilitate the development life cycle of the whole DW architecture.

Since there exist multiple methods and techniques for conceptual, logical, and physical design of an ETL workflow, there is a need of developing a uniform ETL framework, which would: 1) facilitate the ETL developer designing an efficient ETL workflow, by providing hints for optimizing the workflow and 2) allow the ETL developer to validate and benchmark some alternative workflow designs for given quality objectives.

In this chapter, we discuss the state of the art and current trends in designing an ETL workflow. First, we discuss the techniques to construct a conceptual and a logical model, and its corresponding physical implementation of an ETL workflow as well as to evaluate them on the basis of some metrics that we proposed (cf. Section 2.2.5). Second, we identify open research and technological issues in the field of designing and implementing an ETL workflow.

This chapter is divided into five sections, each of which starts with an introduction and concludes with the summary of open research and technological issues. Section 2.2 discusses the findings on a conceptual modeling of an ETL workflow. Section 2.3 overviews works carried out on designing a logical model of an ETL workflow and approaches to convert a conceptual model into its corresponding logical design. Section 2.4 introduces techniques for the physical implementation of an ETL workflow. Finally, Section 2.5 concludes this chapter with a summary of open research and technological issues.

2.2. Conceptual Model

A Conceptual model of a DW serves a purpose of representing business requirements and clearly identifies all business entities participating in a DW. A Conceptual model and its documentation helps in understanding and identifying data schema and facilitating the ETL developer in transformation and maintenance phase of an ETL

workflow.

Until 2002, design, development, and deployment of an ETL workflow was done in an ad-hoc manner due to the non-existence of specific design and development guidelines and standards. The ETL research community has put a lot of effort in formulating the required guidelines, methods, and standards.

This section highlights these methods existing in literature for a conceptual model of an ETL workflow including graph, UML, ontology, and BPMN-based conceptual models.

2.2.1. Graph-based conceptual model

A graph-based customizable and extensible conceptual model [101] is among the first approaches in providing formal foundations for the conceptual design of an ETL workflow. The proposed model focuses on the internal structure of the elements involved, inter-relationships among sources, target attributes of the elements, and transformations required during loading a DW. The idea behind the proposed framework is to provide the ETL developer with different kind of transformations required for different ETL scenarios, which cannot be anticipated. Therefore, instead of providing a limited set of transformations, an extensible framework is developed so that the ETL developer can define transformations as required. The paper presents a three-layer architecture for a conceptual model of an ETL workflow that consists of Schema Layer (SL), Meta-model Layer (ML), and Template Layer (TL).

SL contains a specific ETL scenario and all the elements in this layer are instances of ML. ML is a set of generic entities that are able to represent any ETL scenario. Finally, TL enables the generic behavior of the framework, by providing the ETL developer with customizable ETL templates, which he/she can enrich according to different business requirements.

The work in [101] describes only the formal foundations for a conceptual model of an ETL workflow. However, much needed design methods and standards were not addressed until [83] proposes a set of steps in order to construct a conceptual model in a managed, customizable, and extensible manner. The set of steps are as follows:

1. Step 1: identify participating data stores and relationships between them.
2. Step 2: identify candidates and active candidates for the involved data stores. The idea is to include only active candidates in order to keep a conceptual model simple.
3. Step 3: identify the need of data transformation. If a transformation is required, define mapping rules between source and target concepts. A transformation can be a surrogate key assignment, conversion of units, or just a simple mapping of attributes.

4. Step 4: annotate the model with run-time constraints. Annotation can be done using notes, which correspond to a particular operation, concept, or a relationship in a conceptual model.

2.2.2. UML-based conceptual model

The Unified Modeling Language (UML) [76] is a standard modeling language in the field of software engineering in order to visualize the design of systems in a standardized way. [95] points out that methods [101] and [83], discussed in Section 2.2.1, may result in a complex ETL workflow design due to the absence of a standard modeling language and treating attributes as "first-class citizens" in the model. Therefore, UML is used as a standard modeling language for defining the most common ETL tasks (including among others: data integration, transformation of attributes between source and target data stores, as well as generation of surrogate keys). The ETL tasks are represented by UML packages to model a large ETL workflow as multiple packages, thus, simplifying the complexity of an ETL workflow for the ETL developer.

2.2.3. Ontology-based conceptual model

The approaches discussed in Sections 2.2.1 and 2.2.2 require the ETL developer to manually derive the ETL transformations and inter-attribute mappings at a conceptual level of an ETL workflow. The work in [89] proposes a semi-automatic method for designing a conceptual model of an ETL workflow, leveraging an ontology-based approach. The proposed approach uses ontology instead of UML because ontology is capable of deriving ETL transformations automatically using ontology 'reasoners'. The proposed solution facilitates the construction of an ETL workflow at a conceptual level and deals with the problem of semantic and structural heterogeneity.

As the first step towards creating an ontology, a common vocabulary is constructed. To this end, the ETL developer has to provide the information about the application domain and user requirements about a DW. For example, primitive concepts and their attributes, possible values of the attributes and relationship among the concepts and attributes. Using the information provided by the ETL developer, the vocabulary is formulated.

After the common vocabulary is formulated, the second step is to annotate the data sources-based on the constructed vocabulary.

Once the application vocabulary and the annotations are described, the third step is to construct an application ontology. It describes the application domain, relationships, as well as mappings between sources and a target. The application ontology consists of: 1) a set of primitive classes similar to the specified concepts, 2) representation formats, and 3) ranges or sets of values as defined in the vocabulary. Finally, the constructed ontology is used to generate a conceptual model automatically

using the OWL 'reasoner'. This solution enables the ETL developer to explicitly and formally represent a conceptual model using Ontology Web Language-Description Logic (OWL-DL).

OWL [66] helps in creating a flexible model that can be redefined and reused during different stages of a DW design. The well-defined semantics allows automated reasoning. This solution is applicable to the relational databases only, however, a DW may also contain semi-structured and unstructured data.

To provide the support for semi-structured data, [90] proposes a solution, which is an extension of the work presented in [89] to cater both structured and semi-structured data sources. The proposed approach uses graphs to represent a conceptual model for data sources in order to handle both structured and semi-structured data in a uniform way. The schema presented as a graph is called a Data-Store Graph (DSG). For a relational schema a DSG is designed as follows: 1) nodes represent elements of a schema, 2) edges represent relationships among the elements, 3) labels on each edge represent min and max cardinalities of a reference, and 4) leaf nodes represent elements containing data. The ontology graph and a DSG are then used to annotate each data store by defining mappings between these two graphs. Finally, the semantic annotations are used along with an application ontology to infer a set of generic transformations to construct a conceptual model of an ETL workflow.

In the preceding approaches [89, 90] related to an ontology-based conceptual model, the ETL developer is responsible for manually sketching the required mappings and transformations of schema from a source to a target data store.

To reduce the manual work required by the ETL developer, [91] proposes a semi-automatic approach to build an ETL workflow in a step-by-step manner through a series of customizable and extensible set of graph transformations rules. These rules are based on the already provided ontology in order to determine which ETL tasks are applicable in the initial graph i.e., a graph generated after converting a semi-structured data. The final graph i.e., a graph generated after applying transformation rules depicts a conceptual model of an ETL workflow with an appropriate choice of ETL tasks.

Another approach presented in [14] proposes a semi-automatic approach to extract and transform data in an ETL workflow, as opposed to manually creating an ontology to generate a conceptual ETL workflow model. For a semi-automatic extraction of data from different sources, the proposed method exploits and extends the already existing systems, Mediator EnvirOnment for Multiple Information Sources (MOMIS) [13] and RELEVANT [15].

MOMIS is a semi-automatic data integration system. Data, whether structured or semi-structured, are extracted and then annotated in a semi-automatic environment i.e., facilitated by a tool with the consent of the ETL developer. Then, data are logically converted into a common language in order to generate a vocabulary using a

knowledge engine in MOMIS. The vocabulary is then forwarded to the RELEVANT data analysis system to perform data pre-processing, similarity computations, clustering, and decision making to generate mappings between source data stores and a DW.

The RELEVANT computes similarity between a source and a DW attribute values and forms clusters using clustering algorithm defined in a system. Finally, clusters are validated by developers in order to find the matching data source and target attributes. To automate the transformation process, a set of common transformation functions such as 'Retrieve', 'Project', 'Union', 'Join', 'Convert', 'Filter', and 'Aggregate' and a set of rules are formulated. First, a transformation function checks for the compatibility of data source values with target values using a similarity measure. Then, if they are compatible, the function directly maps the source values to the target values. Otherwise, the ETL developer needs to define the transformation of non-compatible attribute values.

2.2.4. BPMN-based conceptual model

Besides the aforementioned approaches proposing conceptual models (cf. Sections 2.2.1, 2.2.2, and 2.2.3), the work presented in [28] proposes a Business Process Model Notation (BPMN) to create a platform independent conceptual model of an ETL workflow. This paper points out that existing tools for designing an ETL workflow use vendor specific models. As a result, the ETL developer needs to be cautious about the implementation details while implementing an ETL workflow. Therefore, Business Process Model Notation (BPMN) can be used to construct a platform independent conceptual model. This approach implements the BPMN conceptual model into Business Process Execution Language (BPEL) that is an XML-based language and is a standard executable language for specifying interactions with web services.

The paper discusses several BPMN tasks to represent various ETL tasks. For example, BPMN gateways represent the sequence of tasks in an ETL workflow, based on conditions and constraints; BPMN events represent start, end, and error handling events; BPMN connection objects represent the flow of tasks; BPMN artifacts describe the semantics of an ETL task.

After the BPMN design is completed, the model is translated into a Business Process Execution Language (BPEL), which we will discuss in Section 2.4. This work enables the design developed in BPMN to be compatible across multiple tools and easy to extend to fit the requirements of a particular application. The BPMN approach was then adapted by [29, 72, 106] to construct a conceptual model of an ETL workflow.

The work in [106] proposes a layered method that starts with business requirements and systematically converts a conceptual model into its semantically equivalent

physical implementation. The entire method is based on the QoX - suit of quality metrics [87] to construct an optimal ETL workflow. The QoX metrics are considered during the design and development of ETL workflows ranging from quantitative to qualitative metrics (e.g., performance, recoverability, and freshness). [106] fills the gap between different stages (conceptual, logical, and physical) of an ETL workflow design. Once the conceptual design is expressed in BPMN, it is converted into XML to translate a conceptual design into its semantically equivalent logical model. The logical model is then used to optimize an ETL workflow design and for creating the corresponding physical model.

The approach presented in [29] uses BPMN and model driven development to specify an entire ETL workflow in a vendor-independent way and to automatically generate the corresponding code in different commercial tools. Once an ETL workflow code is generated, a 4GL grammar is used in order to generate a vendor-specific code. This approach is so far the only one in the literature to specify the design and implementation phases of the ETL workflow in a vendor-independent manner and the automatic code generation for specific commercial ETL tools.

The work in [72] complements and extends the work of [29, 106] by incorporating specific conceptual model constructs as BPMN patterns for ETL tasks like 'change data capture', 'slowly changing dimensions', 'surrogate key pipelining', or 'data quality coverage'. This foundation can be extended to build more BPMN patterns covering all the tasks of an ETL workflow, which results in helping develop high quality, error free, and an efficient ETL workflow.

2.2.5. Summary

In Section 2.2 we have discussed different techniques and methods for representing a conceptual model of an ETL workflow, which include graphs, UML, ontology, and BPMN. Below, we summarize the approaches on the basis of the following criteria:

- **Autonomous behavior** - whether a design is manual, automatic, or semi-automatic (i.e., how much input it requires from the ETL developer);
- **DSs format** - what kind of data sources are supported, i.e., structured, unstructured, or semi-structured;
- **UDF support** - whether user-defined functions are supported;
- **Quality metrics** - whether quality metrics guide the design of an ETL workflow;
- **Unified model** - whether an ETL design is easily translated from a conceptual model into its semantically equivalent logical model and whether it can be implemented using any ETL framework.

1. **Graph-based approaches** [83, 101]
   Pros:

   - Widely accepted graph-based models are used, which help the ETL developer to outline a conceptual model of an ETL workflow in a standardized way.
   - They present the first steps towards translating a conceptual model of an ETL workflow into its semantically equivalent logical model.

   Cons:

   - **No autonomous behavior** - the ETL developer has to manually derive ETL transformations and inter-attribute mappings at a conceptual level.
   - **Structured data only** - only the structured input data sources are supported, and there is no discussion on how to handle unstructured or semi-structured data sources.
   - **No UDF support and quality metrics** - the tasks and templates for traditional ETL tasks are proposed; there is no support for UDFs; quality metrics are not taken into consideration while constructing a conceptual model.
   - **Challenges** - an ETL design may become complex due to the absence of a standard modeling language and treating attributes as 'first-class citizens' in the model.

2. **UML-based approach** [95]
   Pros:

   - **Unified model** - a method is proposed to standardize the conceptual design of an ETL workflow (UML is a standard modeling language).

   Cons:

   - **No autonomous behavior** - the ETL developer has to provide input at each step of the conceptual design.
   - **Structured data only** - only the structured input data sources are supported.
   - **No UDF support and quality metrics** - no support for user-defined functions and quality metrics.

3. **Ontology-based approaches** [82, 89, 90, 91]
   Pros:

- **Semi-autonomous behaviour** - the ontology-based models propose semi-automatic methods to design a conceptual model of an ETL workflow in a step-by-step manner (based on reasoners on ontologies, it is possible to derive ETL transformations automatically).
- **Structured & semi-structured data** - [89, 91] focus on structured data only and [90] focuses also on semi-structured data.

Cons:

- **No UDF support and quality metrics** - UDFs are not supported; quality metrics are not taken into consideration while constructing a conceptual ETL model.
- **Unified model** - an ontology-based conceptual design cannot be directly translated into its semantically equivalent logical model. It requires a fair amount of effort from the ETL developer to translate the design.
- **Challenges** - manually creating an ontology and defining the relationships among the ontology elements is a difficult and time-consuming task. Constructing an ontology manually requires high correctness and detailed description of data sources, thus if an ontology is created manually it becomes more prone to errors.

4. **BPMN-based approaches** [28, 29, 72, 106]
Pros:

- **Semi-autonomous behaviour** - [72] introduces various BPMN patterns as constructs for frequently used ETL tasks and activities.
- **Quality metrics & unified model** - [106] proposes a systematic method to translate business requirements into a conceptual model and conceptual model into its semantically equivalent logical model, based on quality metrics. [29] uses BPMN and model driven development approach to develop vendor independent design of an ETL workflow.

Cons:

- **No UDF support**.
- **Challenges** - converting conceptual model into its equivalent logical and physical implementation requires ETL developers to have specific knowledge and hands-on experience in BPMN and BPEL.

To conclude, the graph-based models can be used to represent a complex conceptual design of an ETL workflow by using standard notations, whereas, for simpler ETL workflows, approaches based on UML, ontology, and BPMN are well suited.

Such models reflect business requirements as well as provide technical perspective of the problem. Nonetheless, there is a need for a single agreed unified model, easy to validate and benchmark an ETL design for its quality objectives. Also all the discussed approaches require the ETL developer to extensively provide some input during the design phase of an ETL workflow, as well as require technical knowledge from business users to understand and validate an ETL design. Furthermore, despite the fact that multiple approaches have been proposed, the research community has not yet agreed upon the standard notation for representing a conceptual model of an ETL workflow.

2.3. Logical Model

The next step in ETL development life cycle is a logical design of an ETL workflow. A logical design describes detailed description of an ETL workflow such as, relationships among the involved processes with participating data sources, a description of primary data flow from source data stores into a DW, including an execution order of ETL tasks as well as an execution schedule of an entire ETL workflow. A recovery plan and a sequence of steps in case of recovery from a failure are also devised during a logical design of an ETL workflow.

In this section we will discuss approaches to a logical design of an ETL workflow.

2.3.1. Graph-based logical model

Initial approaches to designing a logical model of an ETL workflow are based on graphs. The work presented in [102], proposes a formal logical model as a graph, where an ETL workflow is designed such that the nodes in the graph represent ETL tasks, record sets, and attributes, whereas the edges represent different types of relationships among ETL tasks.

[102] proposes the following steps to construct a logical model of an ETL workflow as a Graph:

1. Incorporate structured entities i.e., activities and record sets in a graph along with all the attributes. For example, Figure 2.2 illustrates S2.PARTSUPP as a structured entity along with its attributes i.e., DEPT, COST, QTY, DATE, SUPPKEY, and PKEY.

2. Connect 'Activity nodes' with their respective attributes through a part-of relationship. For example, in Figure 2.2 'Activity nodes' 'SK_T' and '$2€' are connected with their respective attributes as the part-of relationship that is depicted as a connector with a diamond. The 'IN' and 'OUT' labels on an ac-

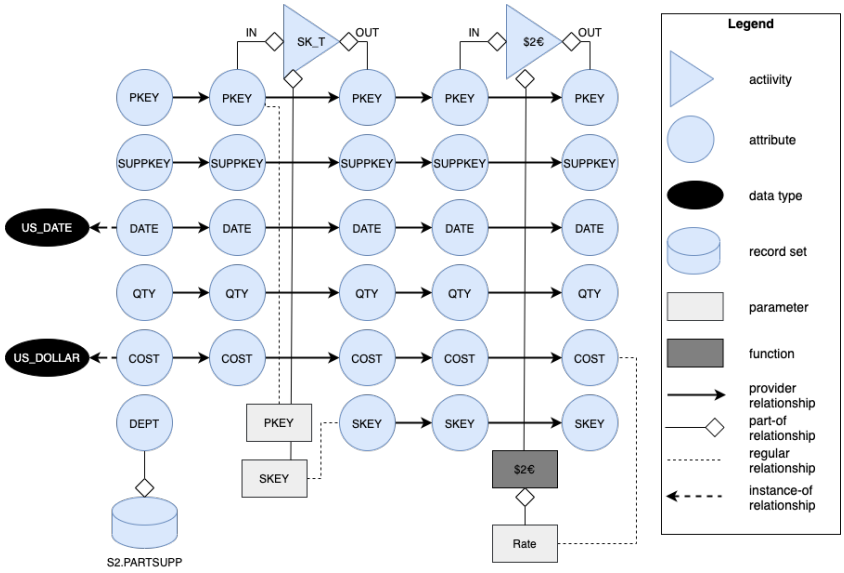

Figure 2.2: Graph of an ETL workflow

tivity represent that the attributes belong to the input and output schema of an activity, respectively.

3. Incorporate data and function types using instance-of relationships. For example, data types 'US_Date' and 'US_Dollar' are connected to their respective attributes 'DATE' and 'COST' as instance-of relationships, depicted in Figure 2.2.

4. Construct the regulator relationships. For example, in Figure 2.2, regulator relationships among the parameters of the activities and attributes are depicted with simple dotted edges.

5. Establish provider relationships that capture the flow of data from source to target. For example, the data flow from source attributes towards target attributes are depicted as bold solid arrows in Figure 2.2.

The proposed graph is considered as a formal logical model of an ETL workflow. In [100], the authors used the aforementioned model along with a formal Logical Data Language (LDL) to define the semantics of each ETL task. [100] also mentions the re-usability framework to compliment the generic behavior of the proposed model, which is achieved through a Meta-model Layer (ML) and a Template Layer (TL) as discussed in [101] (cf. Section 2.2.1). The authors also propose a user friendly graphical tool to facilitate the ETL developer to design an ETL workflow.

[106] proposes to construct a logical model of an ETL workflow using a method to parameterize a Directed Acyclic Graph (DAG), called here *Parameterized DAG* (DAG-P). The DAG is created by translating a BPMN-based conceptual model, as mentioned in Section 2.2.4. DAG-P operations, transformations, and data stores are represented as vertices of the graph. Edges represent data flows from a source data store to a target data store. The parameters in DAG-P are used to incorporate business requirements [87], physical resources (needed for an ETL workflow execution), and other generic characteristics (such as visualization) of an ETL workflow.

2.3.2. From conceptual to logical model

The work presented in [84] proposes a set of steps to transform a conceptual model of an ETL workflow to its corresponding logical model. The models are represented by graphs called *Conceptual Graph* and *Architecture Graph*, respectively. The following steps map a conceptual model into a logical model:

1. Identify data stores and transformations required in an ETL workflow, and describe inter-attribute mappings between source and target data stores.
2. Determine 'Stages' to identify the proper order of tasks in a conceptual model to assure a proper placement of tasks in a logical model.
3. Follow the following five-step method in order to translate a conceptual model into its corresponding logical model:
   (a) Simplify a conceptual model such that only required elements are present in the model.
   (b) Map the concepts of a conceptual model into data sources in a logical model such that part-of relationships do not change. The part-of relationship denotes the relationship between attributes and tasks, record set, or function.
   (c) Map transformations defined in a conceptual model to logical tasks and then determine the order of execution of the ETL tasks.
   (d) Represent ETL constraints with separate tasks in a logical model and determine their execution order.
   (e) Generate a schema involved in a logical model using the algorithm proposed in [86] in order to assure that semantics of the involved concepts does not change even after changing the execution order of tasks in an ETL workflow.

As discussed in Section 2.2.4, the work in [106] proposes a method that covers all stages of an ETL workflow design, i.e., from gathering business requirements to

designing a conceptual model and finally translating it into an XML-based logical model as a DAG-P. The reason behind choosing XML is its ability to easily transform one XML model (conceptual) into another XML model (logical).

In Section 2.2.4, we presented an ETL conceptual design based on BPMN [28]. The authors then extended their work to transform BPMN based conceptual model into its corresponding logical format using Relational Algebra (RA) [10], which essentially translates data tasks automatically into semantically equivalent SQL queries to be executed over a Relational Database Management Systems. The authors provided a mapping of different BPMN data tasks e.g., data input, column functions, join, lookup, sort, to be expressed as a RA expression. For example, the 'Aggregation' task is translated as an 'Aggregate' operation, and a 'Drop Column' operation is expressed in RA as a projection off all columns except the ones to be dropped. The approach showcased an easy way to translate the BPMN based conceptual model into its SQL based implementation.

2.3.3. Summary

In this section we have discussed a graph-based logical representation of an ETL workflow and steps to translate a conceptual model into its semantically equivalent logical model. We have outlined step-by-step methods to formulate a graph-based logical model of an ETL workflow. The discussed methods are not trivial to adopt and require a substantial amount of manual effort and background knowledge from the ETL developers. Below, we summarize the approaches on the basis of the criteria described in Section 2.2.5.

1. **Graph-based approaches** [99, 100, 102, 106]
   Pros:

   - **Quality metrics** - [106] proposes annotations to incorporate quality metrics in an ETL workflow for its efficient and reliable execution.
   - **Unified model** - [100] proposes a reusable framework that supplements a generic behavior of a logical model by defining semantics of each ETL task in a graph. [106] proposes a logical model as a graph, which is implemented in XML, and can be used in any XML-based framework.

   Cons:

   - **No autonomous behavior** - the discussed methods either require the ETL developer to manually construct a logical model from a given conceptual model or to provide an extensive amount of input to the system, to generate a logical model from a conceptual model.
   - **No UDF support**.

- **Challenges** - the discussed approaches demand a substantial amount of input from the ETL developer. For example, such an input is required in: 1) the task of identifying stages (c.f. Section 2.3.2) to make sure the ETL tasks are in a proper order, and 2) the task of defining the mappings to translate a conceptual model to its equivalent logical model. Such tasks are not trivial in nature and are prone to errors.

From the above discussion we can conclude that there still exists a need to develop a fully- or semi-automatic intelligent system that would guide the ETL developer to produce a logical design of an ETL workflow satisfying some predefined quality criteria. One such work to take inspiration from is *Consolidation Algorithm* [49], where data-intensive workflows are represented as DAG to generate executable data-intensive workflows. The proposed solution works at the logical level and is therefore applicable to a variety of approaches that generate logical data flows from information requirements expressed either as high level business objects or in engine specific languages.

2.4. Physical Implementation

Having developed a conceptual and a logical model of an ETL workflow, its physical model has to be produced. A physical model describes and implements the specifications and requirements presented in a conceptual and a logical model.

2.4.1. Reusable templates-based implementation

The work in [96] proposes a method for mapping a logical model of an ETL workflow into its corresponding physical model. A logical model is formulated as a state-space problem, where states are a set of physical level scenarios and each state has a different cost. An initial state of the state-space problem is generated by converting each logical task to its corresponding physical task using a library of re-usable templates. The library consists of both logical and physical level templates. The templates include a set of properties and require some input parameters, thus are capable to be customized according to a particular ETL scenario.

Multiple versions of the solution can be generated by selecting different physical templates, provided all constraints and conditions of the selected physical template are satisfied.

Another non-traditional ETL approach [71] uses high-level ETL constructs to integrate semantic data sources. The integration process is designed in two layers namely *design layer* and *execution layer*. The *design layer* serves as an overview of the entire integration process in the form of a mapping file which, consists of

transformations from source to target. The mapping file is then fed to the ETL operations orchestrated in the execution layer to automatically generate ETL data flows. This approach not only supports physical integration but may be extended to support virtual data integration as well as can be extended to create integration process in a big data and data lake environment.

2.4.2. BPEL-based implementation

A BPMN approach [28] discussed in Section 2.2.4 implements a BPMN-based conceptual model into Business Process Execution Language (BPEL). BPEL is a standard executable language for specifying interactions with Web services, based on XML. BPEL has four main sections: 'partnerLinks', 'variables', 'faultHandlers', and 'process'. In order to translate BPMN into BPEL, first the basic attributes are mapped such as business process name and related namespaces are mapped to the 'process name' in BPEL. Then, ETL tasks in a BPMN are represented as the type of 'services' and are mapped into the 'partnerLinks' in BPEL.

2.4.3. XML-based implementation

[106] proposes to model an ETL workflow as an ETL graph encoded in XML representation (as described in Sections 2.2 and 2.3). To translate an XML-based logical model to its corresponding physical implementation the authors use an appropriate parser. For example, an XML encoded logical model can be translated into a physical implementation format understandable by Pentaho Data Integration (PDI), as follows:

1. element $< design/ >$ maps into 'job' task, and if $< design/ >$ element is nested, then it maps into 'transformation' task in PDI,
2. element $< node/ >$ maps into 'step',
3. elements $< name/ >$ and $< optype/ >$ map into 'name' and 'type' of 'step', respectively,
4. element $< type/ >$ of node describes the type of an ETL task, e.g., a data store or an ETL task in 'step',
5. element $< edge/ >$ specifies the order and interconnection of 'step',
6. element $< properties/ >$ specifies the physical properties of the 'step'.

'job', 'step', 'name and type of step' are artifacts in PDI. Hence, using the aforementioned mapping rules and the appropriate parser, the physical implementation of a XML encoded logical model is easily generated.

Another approach presented in [11], extends the work of designing the ETL conceptual model as BPMN [28], by introducing an XML based interchange format (BEXF) to translate BPMN 2.0 model information across tools. BEXF provides mapping of BPMN attributes to the corresponding BEXF representation, e.g., start/end event, process, task and supports common tasks within an ETL workflow e.g., Data Input, Lookup, Union, Column Operations, Aggregation. Hence, BEXF is an XML interchange format and can be translated into various ETL tools e.g., Microsoft SISS, Pentaho Data Integration and as well as to different SQL dialects.

2.4.4. Summary

In this section we have examined the methods and techniques to translate logical models of an ETL workflow into their corresponding physical implementations using reusable templates, engine specific XML parser, and BPEL. The advantages and disadvantages of the discussed approaches can be summarized based on the metrics described in Section 2.2.5 as follows.

1. **Re-usable templates approach** [96]
   Pros:

   - **Semi-autonomous behavior** - the techniques that physically implement a graph-based logical model use a library of reusable (possibly error free and efficient) templates; a number of an ETL workflow variants may be generated by selecting different physical templates.
   - **UDF support** - UDFs are supported as black-box ETL tasks and are considered during implementation and performance optimization.

   Cons:

   - **Structured data** - only structured data sources are supported.
   - **Quality metrics** - only execution and performance cost as quality metrics are considered.
   - **Unified model** - the ETL developer has to manually translate a logical model into its corresponding physical implementation if a template is not already provided for a certain ETL task. Furthermore, the templates are platform dependent, which limits their application.
   - **Challenges** - a limited set of logical and physical templates is provided.

2. **BPEL-based approach** [28]
   Pros:

- **Semi-autonomous behaviour** -BPEL is used to physically implement a BPMN-based logical model; mapping rules are required to implement a BPMN-based model into BPEL.
- **Unified model** - the physical implementation is done using BPEL, thus is platform independent as an ETL workflow can be exposed as a Web service.

Cons:

- **Structured data** - only structured data sources are supported.
- **No UDF support**.
- **Challenges** - the ETL developer must have prior knowledge of BPEL and tools that support BPEL-based ETL workflows.

3. **XML-based approach** [106]
Pros:

- **Semi-autonomous behavior** - a step-by-step method is proposed to generate a physical implementation of an XML-based logical model using engine specific XML parser, but the ETL developer has to provide the mapping rules to implement an ETL workflow.
- **Quality metrics** - the performance, freshness, recoverability, and reliability quality metrics are addressed.
- **Unified model** - the logical design is created in the XML format. Since most of the ETL tools support XML, it is easy to generate a corresponding physical implementation using existing tools.

Cons:

- **Structured data** - only structured data sources are supported.
- **No UDF support**.
- **Challenges** - generating a physical implementation of a workflow from its XML-based logical model requires a set of carefully defined rules.

To conclude, the discussed methods require extensive amount of input from the ETL developer to execute a translation of a logical model into its physical representation. Although a few approaches have been proposed in this field, there is a need for a framework that automatically or semi-automatically translates a logical model into its physical implementation with minimum or no human support.

2.5. Conclusions

An ETL workflow comprises numerous tasks, e.g., data extraction, validation, transformation, cleaning, conversion, de-duplication, and loading. All these tasks can be represented and executed in many distinctive ways, e.g., using relational operators or user-defined functions (implemented in various programming languages). Workflows deployed in business architectures typically includes hundreds of tasks, this designing an error-free and efficient ETL workflow is a complex and expensive task. For this reason, the research community has contributed a significant amount methods for the design, development, deployment, and optimization of an ETL workflow.

2.5.1. ETL workflow development: summary

There exist multiple techniques for an ETL workflow development. At the conceptual level there are methods involving graphs, semantic Web ontology, UML notation, and BPMN. All these methods propose diverse approaches to design an ETL conceptual model in a precise and productive way. Then, at a logical level, there exist approaches using graphs, which supplement a generic behavior of a logical design. Additionally, there are methods to incorporate quality metrics in an ETL workflow for its efficient and reliable execution. For a physical implementation of a logical design of an ETL workflow, there exist methods that give detailed account on translating a logical model into its corresponding physical implementation. Table 2.1 summarizes the discussed methods for each development stage, i.e., conceptual modeling, logical modeling, and physical implementation. Based on the analysis of the presented methods for each development stage, we draw the following conclusions.

1. There are diverse methods for constructing a conceptual model of an ETL workflow e.g., graph-based [83, 101], UML-based [95], ontology-based [14, 82, 89, 90, 91], and BPMN [28, 29, 72, 106], from which we can conclude that the research community has not yet concurred upon the standard notation and model for representing an ETL conceptual design. As a consequence, it is difficult to develop guidelines for validating an ETL design.

2. The discussed graph-based [83, 101] and UML-based [95] methods require the ETL developer to extensively provide input during the modeling and design of an ETL workflow, thus it can be error prone, time consuming, and inefficient.

3. Most of the methods are designed only for structured data [28, 83, 95, 101, 106] and the support for semi-structured and unstructured data is very limited. Since the variety of data formats is growing rapidly and most of data are unstructured, it is important to extend the support for unstructured data in an ETL workflow.

Table 2.1: Summary of an ETL workflow design methods

Developmer Stage	Method Used	Autonomo Behavior	Data Source Format	UDF Supp.	Quality Metrics
Conceptual Model	**Graph** [83, 101]	Manual	Str.	No	No
	UML [95]	Manual	Str.	No	No
	Ontology [82, 89, 90, 91]	Semi-Auto	Str. & Semi-Str.	No	No
	BPMN [28, 29, 72, 106]	Semi-Auto	Str.	No	Few
Logical Model	**Graph** [99, 100, 102, 106]	Manual	Str. & Semi-Str.	No	Few
Physical Implementation	**Reusable templates** [96]	Manual	Str.	No	Few
	XML represent. using Parser [106]	Semi-Auto	Str.	No	Few
	BPEL [28]	Manual	Str. & Semi-Str.	No	No

4. Almost all of the discussed design methods are based on the traditional ETL tasks (e.g., join, union, sort, aggregate, lookup, and convert) and they mostly do not consider UDFs as ETL tasks.

5. Few methods [88, 106] put emphasis on the issues of efficient, reliable, and improved execution of an ETL workflow using quality metrics, as described in [87]. However, most of the methods in practice do not capture or track these quality metrics.

2.5.2. Open issues

On the basis of this literature review, we can conclude this chapter with the following open issues.

1. There is a need for a unified model for an ETL workflow, so that it is easier to validate an ETL design for its quality metrics.

2. There is a need to consolidate and fully support UDFs in an ETL workflow along with traditional ETL tasks.

Chapter 3
State-of-the-Art and Current Trends in ETL Optimization

In this chapter, we discuss the state of the art and current trends in optimizing ETL workflows. We explain the existing techniques for its optimization, illustrated by examples. The discussed techniques are analyzed w.r.t. their advantages, disadvantages, and challenges in the context of metrics such as: autonomous behavior, support for quality metrics, and support for ETL tasks as user-defined functions. We draw conclusions on still open research and technological issues in the field of ETL.

3.1. Introduction

As discussed in Chapter 2, the design of an ETL workflow may become complex, as it consists of multiple tasks and each of the ETL task has its execution cost, which increases with the increase of the volume of data being processed. As a result of a varying cost and a complex design, an ETL workflow may fail amid execution or may not be able to finish its execution within a specified time window. In consequence, a DW becomes outdated and cannot be utilized by its stakeholders. In order to increase the productivity, quality, and performance of ETL workflows some ETL design methods and optimization methods have been proposed e.g., the research community has been focusing on techniques for optimizing the execution of an ETL workflow [51]. The most common techniques are based on tasks rearranging and moving more selective tasks towards the beginning of a workflow, e.g., [38, 57, 86]. On top of that, the existing research proves that applying processing parallelism at a data level or at task level, or both, is a known approach to attain better execution of an ETL workflow.

In this chapter, we discuss the state of the art and current trends in optimizing ETL workflows. First, we study and understand the existing techniques to optimize ETL workflows as well as to evaluate them on the basis of some metrics that we proposed (cf. Section 3.10). Second, we identify open research and technological issues in the field of optimizing ETL workflows.

The next sections of this chapter discusses different optimization techniques. Sections 3.2, 3.3, 3.4, and 3.5 discusses the State-space, the Dependency graph, the Scheduling strategies, and the Reusable patterns approaches for ETL workflow optimization, respectively. Section 3.6 discusses the Parallelism for ETL workflow optimization. Section 3.7 discusses the Quality metrics for ETL workflow optimization. Section 3.8 discusses the statistics for ETL workflow optimization, Section 3.9 talks about different commercial ETL tools, and Sections 3.10 and 3.11 concludes this chapter with the summary and conclusion of open research and technological issues in optimizing ETL workflow, respectively.

3.1.1. Running example

To begin our discussion on the existing literature, we will be using the example described in [101], which involves two data sources S1.PARTSUPP (PKEY, SUPPKEY, QTY, COST) and S2.PARTSUPP (PKEY, SUPPKEY, DATE, DEPT, QTY, COST) and a central DW.PARTSUPP (PKEY, SUPPKEY, DATE, QTY, COST). PKEY is the part number, SUPPKEY is the supplier of the part, QTY and COSTas are the available quantity and cost of parts per supplier, respectively.

Data are propagated from S1.PARTSUPP and S2.PARTSUPP into a DW table DW.PARTSUPP, as shown in Figure 3.1.

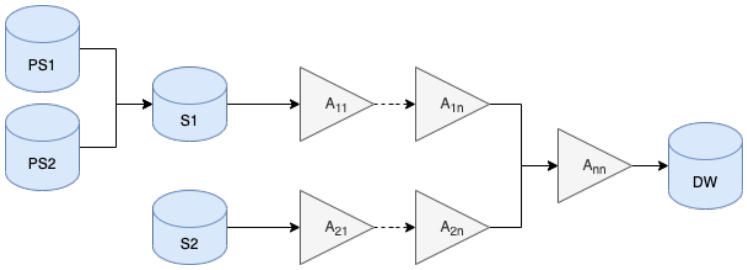

Figure 3.1: An ETL workflow for the running example

The example assumes that source S1 stores everyday data about supplies in the European format and source S2 stores the month to month data about supplies in the American format. The DW stores monthly data on the available quantity of parts per supplier in the European format, which means that data coming from S2 need to be converted into the European format and data coming from S1 need to be rolled-up at the month level in order to be accepted by the DW. S1 joins data from two separate sources PS1 and PS2, and later the data are transformed into a format accepted by the DW. Data from sources S1 and S2 undergo several transformations, denoted as $A_{11}, \ldots, A_{1n}$ and $A_{21}, \ldots, A_{2n}$, respectively. Finally, data from S1 and S2 are merged at task A_{nn} to be finally loaded into the DW.

3.2. State-space Approach for ETL Workflow optimization

In order to reduce the execution cost, [86] presents a concept of optimizing ETL workflows by decreasing the total number of tasks or by changing the order of tasks in an ETL workflow. To this end, a state-space search problem is defined, where each state in a search space is a Directed Acyclic Graph (DAG). In a DAG, tasks are represented as graph nodes and relationships among nodes are represented as directed graph edges. To find an optimal ETL workflow, new states are generated that are semantically equivalent to the original state. A transition from an original state to a new state may involve *swapping* two tasks, *factorizing/distributing* two tasks, *merging*, or *splitting* tasks.

To illustrate the generation of a semantically equivalent new state using operation *distribute* (i.e., to distribute the data flow in an task into different data flows rather than operating over a single data flow), consider a conceptual model of the running example as depicted in Figure 3.2. The data is propagated into a DW in two parallel flows passing through different tasks and is finally unified at task 10. Then, in task 11, the flow checks for the value of attribute QTY before loading data into a DW. This task is highly selective, therefore it is beneficial to push the task to the beginning of the flow and distribute the task into two parallel flows. Figure 3.3 shows Task 11_1 and Task 11_2 after applying the *distribute* operation to Task 11 in Figure 3.2. This approach reduces the total cost of the flow without changing the semantics of the ETL workflow.

[88] extends [86] w.r.t. generating an optimal ETL workflow in terms of performance, fault-tolerance, and freshness, as described in [87]. In order to achieve quality objectives, the approach applies 3 new transitions, namely: *partition*, *add_recovery_point*, and *replicate*. *partition* is used to parallelize an ETL workflow to achieve better performance. *add_recovery_point* and *replicate* are used to

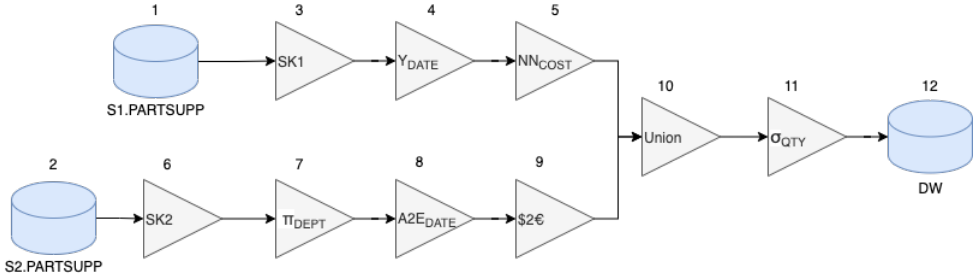

Figure 3.2: ETL workflow before applying operation *distribute*

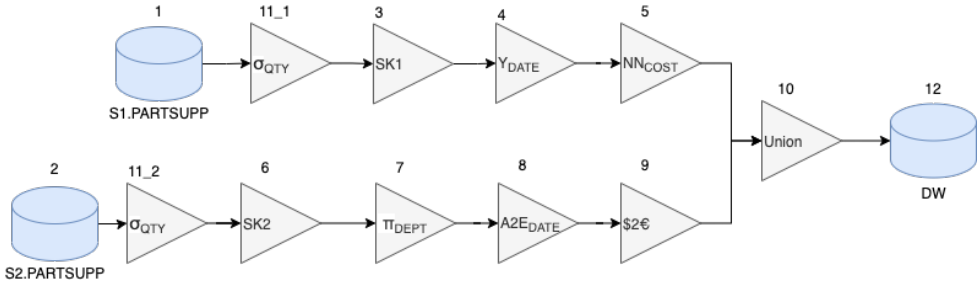

Figure 3.3: ETL workflow after applying operation *distribute*

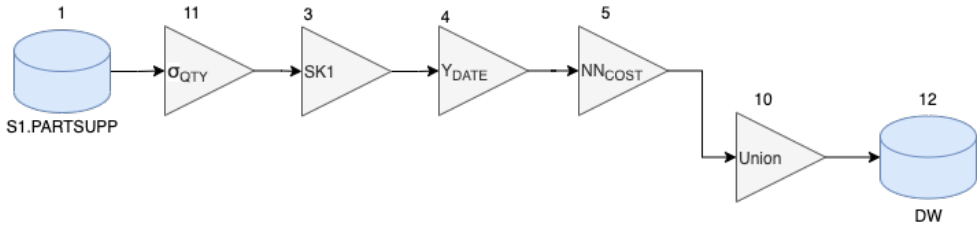

Figure 3.4: Part of the ETL workflow after applying 'Performance Heuristic'

provide a workflow persistence and recovery in case of a failure.

To generate a search space, the 'Exhaustive Search algorithm (ES)' is used [86]. Next, the search space is pruned by using a cost model and different heuristics for performance, reliability, and recoverability metrics. Once the state-space search problem is constructed using the ES algorithm, heuristics and greedy algorithms are used to reduce and explore the search space to get an optimal ETL workflow.

For example, we want to apply the 'Performance Heuristic' (i.e., more restrictive tasks should be placed at the beginning of the flow) on the upper level flow of Figure 3.2. Task 11, which filters the data on the basis of attribute QTY should be pushed before Task 3, as shown in Figure 3.4.

Consider another example, one of the 'Recoverability Heuristic' states 'add recovery at the end of an ETL phase, e.g., after extraction or transformation phase'. Figure 3.5 shows recovery point (RP) after data is extracted from S1.PARTSUPP.

[96] also models an ETL workflow as a state-space search problem and applies *sorters* in a graph node. Sorters change the order of input tuples, because in some cases the order plays an important role in an improved execution of an ETL task. In order to obtain the optimal solution, the 'Exhaustive Ordering (EO)' algorithm is used. EO takes a logical design of an ETL workflow as an input in the form of a DAG and computes its *signature* and its computing cost. The signature is a string representation of a physical design of an ETL workflow. To represent a workflow

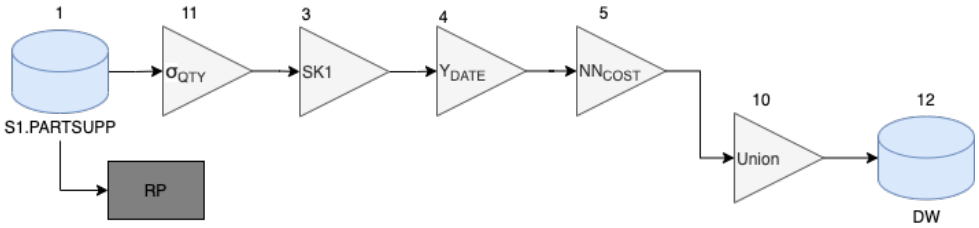

Figure 3.5: Addition of Recovery Point (RP) according to 'Recoverability Heuristic'

(i.e., a graph) as a string, the following rules are proposed: **a@p** - the physical implementation of 'p' of logical task 'a', **.** (dot) - names of tasks forming a linear path separated by dot (.), // - concurrent tasks delimited by '//' and each path enclosed in parenthesis, **a_b(A,B)** - a sorter placed among tasks 'a' and 'b' based on attributes 'A' and 'B', **V!A** - a sorter on table 'V' based on attribute 'A'. Based on the rules, an ETL workflow shown in Figure 3.2 can be represented in terms of the following signature:

((1.3.4@DT.5@NN) // (2.6.7@PO.8.9)) . 10@NL.11.12

In the signature, tasks 1, 3, 4, and 5 (in Figure 3.2) form a linear path in the upper level flow and tasks 2, 6, 7, 8, and 9 form a linear path in the lower level flow. Both the upper and the lower level flows are concurrent and therefore are separated by '//'. *((1.3.4@DT.5@NN)* denotes that task 4 is aggregated based on the date function 'DT' and task 5 performs a not null check 'NN'. *(2.6.7@PO.8.9)).10@NL.11.12* denotes that task 7 applies projection 'PO' on attribute DEPT as it is not required by a DW. Finally, both flows are merged based on nested loop 'NL' at task 10.

Once the signature is computed, the EO algorithm generates all possible states by placing sorters at all possible positions. The EO algorithm then uses all possible combinations of different physical implementations for each task. Finally, it chooses a state with minimum execution cost as the optimal physical implementation.

3.3. Dependency Graph for ETL Workflow Optimization

The optimization concept contributed in [57] draws upon the idea of rearranging tasks in an ETL workflow (as proposed in [86]), in order to construct a more efficient variant of this workflow. The following assumptions are made in [57]:

- an ETL workflow is represented as a DAG,

- every task has associated a selectivity (defined as a ratio: output/input data volume),
- every task has associated a cost, which is a function of an input data size,
- a workflow is rearranged by means of operations: swap, factorize/distribute, merge/un-merge (as in [86]),
- a workflow rearrangement is guided by an optimization rule that moves (if possible) more selective tasks to the beginning of the workflow.

[57] introduced a dependency graph - a structure that represents dependencies between tasks in a workflow. The graph is constructed by applying the 'swappability test', proposed in [86]. Two given tasks are considered independent of each other if they are swappable, i.e., if they conform to the following four rules:

1. the tasks to swap must be adjacent to each other in the dependency graph,
2. the tasks to swap must have a single in/output schema, and must have exactly one consumer of the output schema,
3. the tasks must have the same name for attributes in their in/output schema,
4. the tasks must generate the same schema before or after applying the *swap* operation.

As an example, let us consider a linear flow shown in Figure 3.6. Recall, that: 1) data source S2 stores costs and dates in the US format, 2) function \$2€ (task 9) converts USD into EUR, and 3) task 9_1 selects some rows based on the value of attribute 'cost' (expressed in EUR). Since task 9_1 cannot be executed before task 9, they are non-swappable. Therefore, tasks 9 and 9_1 are dependent on each other. On the contrary, tasks 7 and 8 are swappable because they have non-intersecting schemas and they are independent on each other. Task 6 is also independent. Such checks are performed for each task in an ETL workflow and the dependency graph is created.

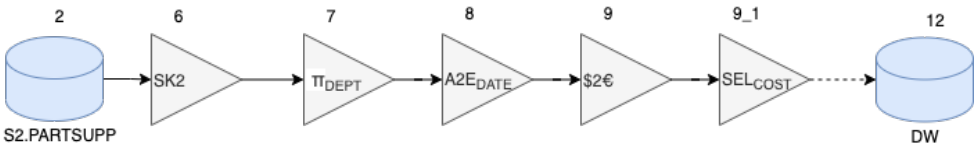

Figure 3.6: Swappable & non-swappable tasks in an ETL workflow

The dependency graph is used for narrowing possible space of allowed rearrangements of tasks within a given workflow. To this end, the authors proposed a greedy

heuristic that is applicable only to linear flows. For this reason, a given complex ETL workflow, with multiple splits and merges (joins) is divided into n linear flows.

As an example, let us consider a complex ETL workflow, as shown in Figure 3.7. The workflow is divided into the three following linear flows: LFI with tasks (3,4,5), LFII with tasks (6,7,8,9,10), and LFIII with tasks (12,13).

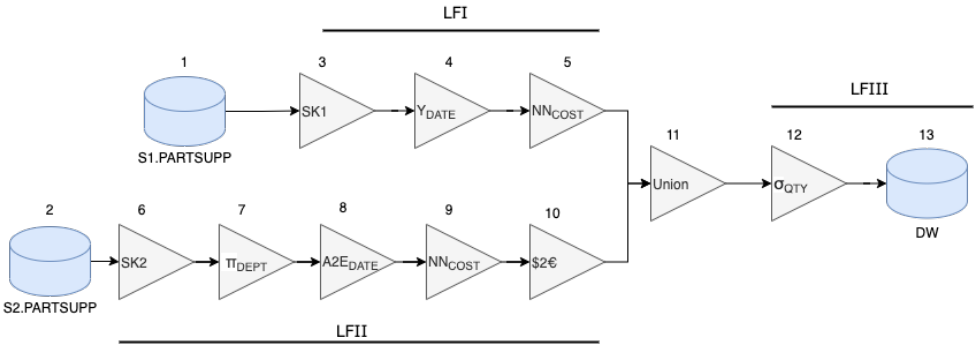

Figure 3.7: Logical linear division of a complex ETL workflow

Having divided a complex workflow into linear flows, each linear flow is optimized by rearranging its tasks. To this end, an algorithm was proposed whose intuition is as follows. Nodes of the dependency graph are ordered in a linear workflow by their selectivities. Less selective tasks are placed closer to the end of the flow (the target). This way, more selective tasks are moved towards the beginning of the flow (towards a data source). Tasks that depend on another tasks T must be placed to the right of T. This way, the dependencies between tasks represented in the dependency graph, are respected.

Having optimized the linear flows, the final step is to combine the flows into larger linear flows that include all the tasks processing data from the source to the destination. For example, the linear flows from Figure 3.7 are combined into two larger linear flows - the first one is composed of [LFI, LFIII] and the second one is composed of [LFII, LFIII].

Next, each combined linear flow is optimized by rearranging its tasks, as described above. There are tasks that may be moved from a given linear flow LF_m to the next linear flow LF_n, i.e., in the direction towards the end of a workflow. Such tasks are called forward-transferrable. There are also tasks that may be moved in the opposite direction, i.e. towards the beginning of a workflow. They are called backward-transferrable. An execution order within a combined linear flow is determined by the order implied by the dependency graph.

For example, in Figure 3.7 tasks 5 and 9 have the same semantics (selecting rows with not null costs). Therefore, they can be moved forward to the beginning of linear

flow III, such that the linear flow will be composed of tasks (5_9, 12, 13). Task 5_9 is a new task created by the factorize task, having the same semantics as 5 and 9. Similarly, task 12 (selecting rows based on the value of attribute QTY) can be moved to LFI and LFII by applying the distribute task.

Having constructed the combined linear flows, each combined flow is optimized by and algorithm whose intuition is as follows. All possible rearrangements of backward- and forward-transferrable tasks are analyzed and for each of them an execution cost of the combined linear flow is computed. Next, the rearrangement with the lowest cost is selected.

3.4. Scheduling Strategies for ETL workflow Optimization

[51] proposes a solution to optimize the performance of an offline batch ETL workflow in terms of execution time and memory consumption without the loss of data. To this end, a multi-threaded framework with incorporated ETL scheduler is presented, where each node of an ETL workflow is implemented as a thread. The proposed framework monitors, schedules, and guarantees the correct execution of an ETL workflow based on the proposed scheduling strategies such as, 'Minimum Cost Prediction (MCP)', 'Minimum Memory Prediction (MMP)', and 'Mixed Policy (MxP)'.

The MCP scheduling strategy is proposed to improve the performance of an ETL workflow by reducing its execution time. The ETL scheduler prioritizes tasks to be scheduled at each step that have the largest volume of input data to process at that time. As a result, the task with the largest input queue is able to process all data without any interruption from the ETL scheduler.

The MMP scheduling strategy is proposed to improve memory consumption by scheduling tasks at each step that have the highest consumption rate (consumption rate = number of rows consumed / processing time of input data). As a result, the ETL scheduler maintains a data volume low in a system by scheduling the flow of tasks whenever an input queue is exhausted due to higher memory consumption of the task.

The MxP strategy is proposed to combine the benefits of MCP and MMP including operating system's default scheduling strategy (i.e., Round-Robin) by exploiting parallelism within an ETL workflow.

A set of scheduling policies is assessed for the execution of an ETL workflow. The results of incorporating scheduling policies are as follows: 1) MCP outperforms other scheduling strategies w.r.t. execution time, 2) MMP is better w.r.t. average memory consumption, 3) MxP, which incorporates multiple scheduling strategies (e.g., MCP, MMP, or Round-Robin) by splitting an ETL workflow, achieves better time

performance. The better time performance is achieved by either prioritizing memory intensive tasks or by scheduling an ETL workflow to avoid blocking ETL operations.

3.5. Reusable Patterns for ETL Workflow Optimization

[98] presents reusable patterns, as a mean to characterize and standardize the representation of ETL tasks along with the strategy to improve the efficiency an ETL workflow execution. The paper proposes to standardize the representation of frequently used ETL tasks that involve a single transformation (e.g., surrogate key transformation, checking null values, and primary key violations), called *ETL Particles*. ETL tasks that perform exactly one job and involve exactly one transformation (e.g., \$2€ conversion) are called *ETL Atoms*. ETL Atoms that involve a linear flow of ETL particles are called *ETL Molecules* and an ETL workflow is called *ETL Compound*.

The paper then presents a normal form of ETL tasks, e.g., the normal form of task 8 in Figure 3.7 can be represented as follows:

$$\boldsymbol{I}(\pi_{DEPT}),\boldsymbol{A2E}(DATE),\boldsymbol{O}(NN_{COST})$$

I is the input schema coming from task π_{DEPT}, **A2E** is the specific template task that converts the format of attribute DATE from American to European. **O** represents the output of task 8 as an input to the next task NN_{COST}. Similarly, tasks 3 to 12 are ETL Molecules and the entire flow in Figure 3.7 including tasks and record sets represent the ETL Compound.

3.6. Parallelism for ETL Workflow Optimization

Besides the aforementioned optimization strategies, incorporating parallelism in an ETL workflow is another popular strategy to achieve better execution performance. Parallelism can be achieved either by partitioning the data into N subsets and process each subset in parallel sub-flows (data parallelism) or by using pipeline parallelism (task parallelism) as shown in Figure 3.8.

For a simple ETL workflow, the data parallelism can work well. However, for a complex ETL workflow (e.g., non-linear flows or flows with data and compute-intensive tasks), combination of data and task parallelism is required. Hence, a mixed parallelism is beneficial when either communication in a distributed environment is slow or the number of processors is large [21].

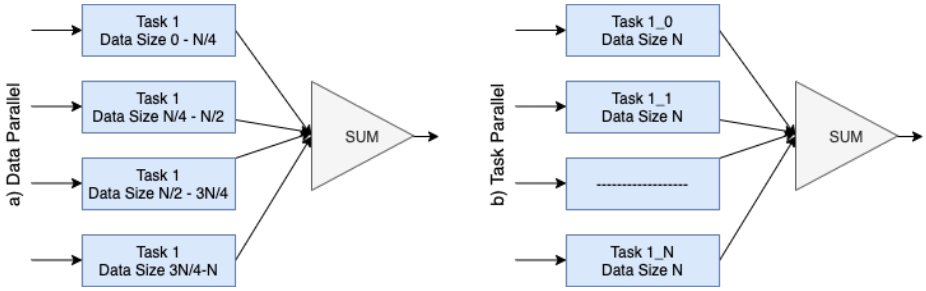

Figure 3.8: Data parallelism and task parallelism

3.6.1. Parallelism in traditional dataflow

Most of research on ETL workflow parallelism has focused on a traditional dataflow parallelism, from which the most prominent one is the MapReduce framework [24]. It uses a generic key/value data model to process large scale data in a parallel environment. MapReduce provides two functions, Map and Reduce, both having two input parameters i.e., 1) a set of input data set in a key/value format and 2) an user defined function (UDF). Map assigns a key/value pair to its input data using an UDF and produces a set of output key/value pairs. The Reduce function then groups the key/value pairs of its input data on the basis of keys and finally each group is processed using an UDF. Storage for MapReduce is based on distributed file system called Hadoop Distributed File System (HDFS) [81].

The MapReduce framework has its limitations i.e., 1) it accepts a single set of input data set at a time in a key/value format and 2) always executes in a strict order, i.e., first Map and then Reduce, 3) the output data from Map have to be stored into an intermediate file system, which makes data processing slow due to data partitioning and shuffling, and 4) developers have to write a custom code for the Map and the Reduce functions, which is hard to maintain and reuse.

Another parallel processing framework is PACT [12]. It is based on the Parallelization Contracts (PACTs), which consists of an Input Contract and an optional Output Contract. The Input Contract is the generalization of the Map and Reduce functions and takes an UDF and one or more data sets as an input. It also provides an extended set of Input Contracts i.e., 'Cross', 'Math', and 'CoGroup' functions, which complements Map and Reduce functionality and overcomes the limitations of MapReduce model i.e., The Input Contract does not need to be executed in a fixed order and allows multiple inputs of key/value pairs. The Output Contract denotes different properties of the output data, which are relevant to parallelization. These properties can be either 1) preserving a partitioning property or 2) an ordering property on data that is generated by an UDF. An input UDF contains the strategy (such

as partitioning, re-partitioning, or broadcasting) for parallelizing in PACT.

For example, let us introduce another concept in our running example, REVIEWS (RID, PKEY, DEPT, REVIEW, SCORE, ENTRYDATE), which stores user reviews along with its computed score for each 'part' stored in a department. Now consider analytical query Q, which provides 'parts' from S1.PARTSUPP having COST greater than some amount, let us say 'c'. Query Q joins S1.PARTSUPP with REVIEWS, keeping only 'parts' where the review score is greater than some threshold 's', and reduces the result set to the 'parts', which are not present in S2.PARTSUPP for some department 'd'.

Figure 3.9 shows query Q as: (a) the MapReduce implementation and (b) the PACT implementation. The MapReduce implementation requires two stages of the MapReduce job. The first stage performs a join on the basis of PKEY in S1.PARTSUPP (S1) and REVIEWS (R), and carries out the specific selection based on S1. COST > 'c' and R.SCORE > 's'. The result from the first stage joins with the selection on S2.PARTSUPP (S2) on the basis of S2.DEPT = 'd' in the Map function of the second stage. The reducer in this stage performs an anti-join on the result set (when no S2.PARTSUPP row with an equal key is found).

The PACT implementation for the same query Q requires three separate user-defined functions (UDFs) attached to the Map contract to perform a selection on data sources S1, R, and S2 instead of a single user-defined function in a MapReduce scenario, such that each UDF is executed in parallel. The Match contract replaces the Reduce contract in the original MapReduce implementation. It matches the incoming key/value pairs from data sets with the same key and forms an independent set based on the similar keys. The Match contract guarantees that the independent set of key/value pairs is supplied to exactly one parallel instance of the user-defined function so that each set is processed independently. Finally, the CoGroup contract implements the anti-join by releasing the rows coming from the Match contract. The CoGroup contract assigns the independent subsets with equal keys to the same group. The PACT implementation allows the flexibility to parallelize tasks by giving parallel hints using output contracts such as the SameKey contract. SameKey is the output contract attached to the Map, Match, and Co-Group contracts, which assures that the value and type of a key will not change even after applying the user-defied functions. Such hints are later exploited by an optimizer that generates parallel execution plans.

The strategy for the generation of an efficient parallel data flow is implemented in the Selinger-style SQL optimizer [79]. This kind of optimizer selects a globally efficient plan by generating multiple plans, starting from data sources, and pruning the costly ones based on partitioning and sort order properties. The plan with the lowest cost is selected as an optimized query plan. A PACT program is executed on a three-tier architecture [5] composed of: a PACT compiler, engine Nephele [105], and a distributed file system. There are two steps to create a parallel data flow for the

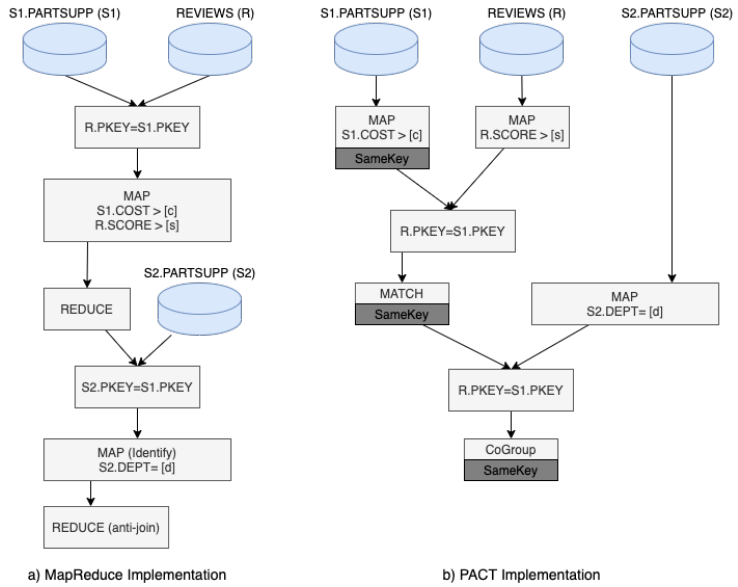

Figure 3.9: MapReduce vs PACT implementation

PACT program, which are as follows:

1. Once the execution plan for a PACT program is chosen, a PACT compiler transforms the PACT program into a Nephele DAG where vertices represent UDFs wrapped in a PACT code and edges show data flows between those vertices.

2. A DAG is then parallelized by creating multiple instances of each vertex depending on the required degree of parallelism for each PACT. That is, a number of parallel instances of each vertex varies hence different parts of a PACT program may have different degree of parallelism. This property of a parallelized DAG can be exploited in order to get an optimal parallelized data flow.

[41] proposes a method to optimize a PACT program consisting of UDFs to process input data into multiple subsets, as follows:

1. use a static code analysis of an UDF (UDFs are considered as black-box tasks with unknown semantics) to obtain relevant properties required to order UDFs;

2. enumerate all valid re-orderings for a given PACT program [42];

3. compute all possible alternative re-orderings using a cost-based optimizer to generate the execution plan by selecting execution strategies;

4. select and submit the plan with a minimum estimated cost for a parallel execution [12].

To overcome the aforementioned limitations of the MapReduce framework, an in-memory parallel computing framework Spark [1] uses multi-pass computation, i.e., computing components several times, using a Direct Acyclic Graph pattern. It also supports in-memory data sharing between multiple tasks. Spark allows the developers to create applications using API based on Resilient Distributed Dataset (RDD). RDD is a read-only multiset of data items distributed over a cluster of machines. RDD is placed on top of a distributed file system (typically HDFS) to provide multi-pass computations on data by rearranging the computations and optimizing data processing.

Another approach towards parallelizing a data flow is Structured Computations Optimized for Parallel Execution (SCOPE) [20]. It supports analyzing massive amount of data residing on clusters of hundreds or thousands of machines by means of and extensible scripting language similar to SQL. SCOPE uses a transformation-based optimizer which is a part of Microsoft's distributed computing platform called Cosmos. Cosmos accepts SCOPE scripts, translates them using SCOPE compiler, and finally invokes the SCOPE optimizer. The SCOPE optimizer introduces considerable parallelism based on a cost function. As presented in [109], the SCOPE optimizer generates a large number of execution plans by taking into account structural properties of data (e.g., partitioning, sorting, or grouping). The generated execution plans are then pruned using a cost model. However, this approach is restricted to relational operators (ROs) only, whereas in practice, it is important to optimize both ROs and UDFs.

The work presented in [37] acknowledges the importance of optimizing both ROs and UDFs. It extends the work presented in [109] by introducing parallelization techniques for UDFs. An UDF is treated as a black box operation. In order to describe an UDF behavior and provide means for its parallelization, a set of UDF annotations were proposed. They describe pre- and post-conditions for partitioning and hints for an optimizer. For example, annotations BEGIN and END keywords, enclosed within annotations BEGIN PARALLEL and END PARALLEL mark the beginning and end point of the user-defined code. A script (with annotations) similar to SQLScript [16] is used to express complex data-flows containing ROs and UDFs together. The main goal is to parallelize ROs and UDFs together, which is achieved by directly translating a RO into the internal representation of the proposed cost-based optimizer as described in [109] and by applying the 'Worker-Farm' pattern [36] on an UDF. A complete set of annotations is described in [37]. The proposed approach has two main limitations: 1) it supports UDFs implemented only as table functions and 2) it requires the ETL developer to annotate the custom code manually, i.e., in fact optimize the code manually.

3.6.2. Parallelism in an ETL workflow

Introducing parallelism into an ETL workflow is not a trivial task. The ETL developer has to decide which tasks to parallelize, how much to parallelize, and when to parallelize before incorporating parallelism into an ETL workflow.

Task and code parallelism

[94] proposes a method to exploit parallelism at a code level by introducing strategies for both task and data parallelism. A set of constructs is proposed in order to enable the ETL developer to convert a linear ETL workflow into its corresponding parallel flow. The constructs are easy to use and do not require complicated modification of a non-parallel ETL workflow. However, the proposed solution is code-based and requires the ETL developer to configure the degree of parallelization. Furthermore, this solution does not provide any cost model to estimate a performance gain.

Parallelizing by means of MapReduce

[61, 62] present a parallel dimensional ETL framework based on MapReduce called ETLMR. The focus of ETLMR is on star schema, snowflake schema, slowly changing dimensions (SCDs), and data intensive dimensions. The ETLMR processes an ETL workflow in two stages. In the first stage, dimensions are processed using MapReduce tasks. In the second stage, facts are processed using another MapReduce task. Dimensions can be processed by using either of the following strategies: *One Dimension One Task* (ODOT) and *One Dimension All Tasks* (ODAT). In ODOT, dimensions are processed by the Map task using an UDF and then the processed data are propagated into a single Reduce task. In the Reduce task, user-defined transformations are performed on rows and then loaded into a DW. In ODAT, an output of the Map task is partitioned in a 'round-robin' fashion i.e., the output is processed by all the Reduce tasks such that each Reduce task receives equal number of rows. The uniqueness of dimension key/values is maintained by using a global ID generator and a 'post-fixing' method, which merges rows with the same values but different keys into a single row. To optimize ODOT, keys with the same values are combined together in the Combiner task, to reduce the communication cost between the Map and Reduce tasks. Using the single Reduce task can become a bottleneck in case of a data intensive dimension, therefore ODAT is used to overcome the bottleneck.

Fact processing in ETLMR requires looking up of dimension keys and aggregation (if required). If aggregations are not applicable, only the Map task is used and the Reduce task is dropped. Otherwise, only the Reduce task is used since aggregations must be completed from all the data in the Reduce task. Once the fact data is processed, it is loaded into a temporary buffer where it resides until the buffer is fully loaded. The Map and Reduce tasks then perform bulk loading in parallel into a DW.

For example, consider DATE and PARTS dimension in a DW. The Map task generates key/value pairs for each dimension, such that, a key is the name of a dimension (i.e., key = DATE for dimension DATE, and key = PARTS for dimension PARTS).

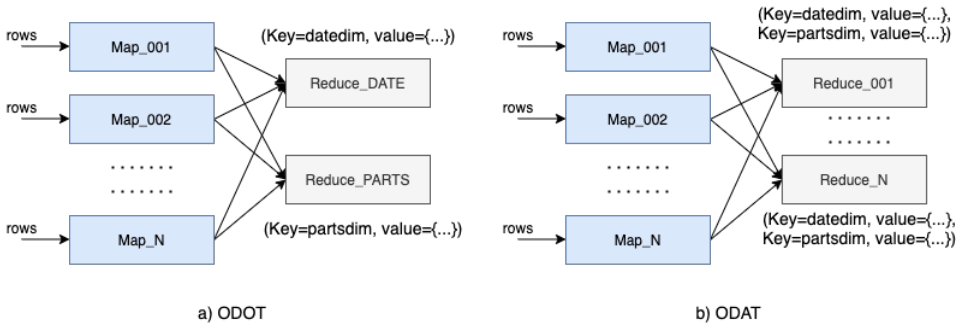

Figure 3.10: ETLMR - Dimension processing methods

All keys with the same name go to a single Reduce task i.e., a record with key= DATE propagates to Reduce_DATE and record with key= PARTS propagates to Reduce_PARTS, as shown in Figure 3.10(a).

For example, consider two Reduce tasks (REDUCE_001 and REDUCE_002) and dimensions (DATE and PARTS). The DATE dimension has 50,000 rows and the PARTS dimension has 150,000 rows (i.e., 200,000 rows in total). Each Reduce task receives an equal number of rows (i.e., 100,000 rows) from the Map tasks to process, as shown in Figure 3.10(b).

However, ODAT causes duplication of key values as the data for the dimension table is processed by all tasks. ETLMR handles this situation by using one of two methods, i.e., 1) a global ID generator and a post-fix method, which fixes duplicate values in lookup attribute or by using private ID generator, 2) a post-fix method to fix duplicate key values, which is applied after all the tasks of dimension processing have finished. To optimize the processing of dimensions, ETLMR offers offline processing of dimensions where dimensions are stored offline physically on all nodes. Thus, tasks do not have to communicate with a DW to process online dimensions, which reduces the communication cost. ETLMR also uses a hybrid of ODOT and ODAT. In this approach, data intensive dimensions are partitioned based on business keys such that rows with the same keys are sent to one mapper task. Finally, the global ID generator is used to generate the dimension key values therefore, no post-fixing is required.

[63] presents a Hadoop based scalable dimensional ETL framework for Hive, called *CloudETL*. The idea behind CloudETL is to allow ETL developers to write MapReduce jobs to be executed on Hadoop without concerning the specific details of the MapReduce configuration. The framework exposes a Java based library of commonly used ETL constructs to the ETL developers to easily implement parallel ETL programs as well as increasing the productivity of developers.

The CloudETL's workflow is designed in two sequential steps: 1) dimension processing and 2) fact processing. For dimension processing, source data are first split

and are assigned to the *map* tasks. A mapper processes source data that will go to different dimensions. Data are then sent to different *reducers* to be written to the HDFs. The dimension processing takes special care of Slowly Changing Dimensions (SCDs) [54] by collecting different versions of the dimension values from both the incremental incoming data and the existing data in the dimensions. For fact processing, the CLoudETL reads and transforms the source data to retrieve the surrogate keys from the referenced dimension table, called as *lookup* operations. Since, the lookup operation is very slow in Hive, the CloudETL uses *multi-way lookups* to retrieve the dimension key values through lookup indices, which consists of business key values, dimension key values, and the SCD dates.

The experimental evaluation shows that the CloudETL outperforms ETLMR when processing different dimension schemas as well as dimensional capabilities of Hive.

Partitioning and parallelization

[60] proposes ETL workflow partitioning and parallelization, as an optimization method. Vertical and horizontal partitioning is suggested. Vertical partitioning is impacted by tasks in an ETL workflow and the following tasks are distinguished: *row-synchronized* - it processes row by row (e.g., filter, lookup, split, data format conversion) and uses a shared cache to move data from task to task, *block* - processing cannot start until all rows are received by the task (e.g., aggregation), *semi-block* - receives rows from multiple tasks and merges them (e.g., join, set operators); processing of the task starts no sooner than all expected rows are received. The authors propose to partition and parallelize an ETL workflow at 3 levels. First, the whole ETL workflow is vertically partitioned into multiple sub-workflows - called execution trees. Second, execution trees are partitioned horizontally and each partition is run in parallel. Third, single tasks in an execution tree are parallelized by multi-threading.

Vertical partitioning is executed as follows. An ETL workflow analysis is run depth-first and it starts from a data source (the root of an ETL graph). All row-synchronized components (i.e., the ones that use a shared cache) are added into a new sub-workflow, so that their original order is preserved. If a block or semi-block task is found, then it becomes a root of a new sub-workflow.

Figure 3.11 explains the workflow partitioning method. Analyzing the ETL graph starts from data sources. For example S1 and S2 create two separate execution trees T1 and T3, respectively. Task 3 is row-synchronized and it is added into T1, whereas row-synchronized tasks 6,7,8, and 9 are added into T3. Since task 4 is a block task, it becomes the root of a new execution tree T2 having task 5 as its only child. Task 10 is a semi-block task, which forms execution tree T4, composed of tasks 11 and 12.

Once the execution trees are constructed, internal parallelization is carried out inside each of the execution trees. To this end, input data are partitioned horizontally into n disjoint partitions (n is parameterized), where each partition is processed by a separate thread. Finally, internal parallelization is carried out for tasks with a

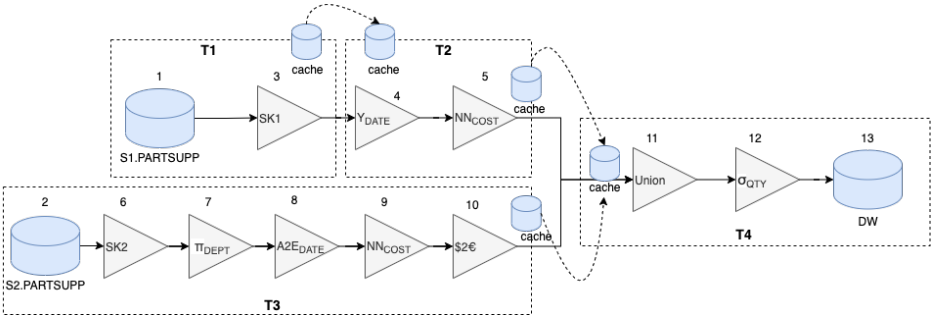

Figure 3.11: ETL workflow partitioning

heavy computational load. To find such a task, time is measured during which a task does not produce any output. If the time is greater than a given threshold then the task becomes a candidate for internal parallelization. To parallelize a single task, multi-threading is applied, i.e. an input of the task is divided into n equal splits, each of which is run by a separate thread. Moving data between tasks within the same execution tree is implemented by means of a shared cache, whereas moving data between adjacent execution trees is implemented by means of coping data between separate cache (cf. dotted arrows in Figure 3.11).

3.7. Quality Metrics for ETL Workflow Optimization

The goal of [87] is to reduce time and cost of ETL design, implementation, and maintenance (DIM) by incorporating some quality metrics into an ETL workflow design. A layered approach is proposed for an ETL DIM, where each layer represents a logical design, implementation, optimization, and maintenance. At each layer some metrics are introduced (or refined from higher levels) that guide the ETL developer to produce a high quality workflow. Furthermore, dependencies among metrics that impact DIM are identified and discussed. The following metrics are proposed in the so called *QoX metric suite*: performance, recoverability, reliability, and maintainability - which characterize an ETL workflow as well as freshness - which characterizes processed data.

Performance is represented by an elapsed execution time. The following techniques for increasing performance are considered: 1) increasing the number of CPUs and main memory, 2) reordering of tasks so that the most selective ones are placed at the beginning of a workflow, as proposed in [57, 86], 3) data partitioning and parallelization, as discussed in Section 3.6. The authors stress the importance of an ETL engine parameters (features) to tune the engine, like: the number of CPUs assigned to ETL processing, the size of main memory, a set of data partitioning algorithms, a

set of workflow partitioning algorithms, detecting tasks to parallelize, a method for determining where to split and merge parallel flows, and a method for determining an optimal number of parallel flows.

Recoverability is defined as the ability of an ETL workflow to resume its execution after an interruption and to restore the process to the point at which a failure occurred. Recoverability can be achieved by: 1) restarting the ETL workflow from scratch or 2) inserting recovery points (RPs) to make a persistent copy of partial results of the process. Implementing RPs requires thorough analysis to figure out where and how many recovery points to set up, i.e., RP after extraction, RP after transformations, RP after merging multiple flows, or RP after an operation that is costly or difficult to undo (e.g., sort).

Reliability is defined as the resistance of an ETL workflow to execution errors. It can be achieved by: 1) running in parallel multiple identical instances of an ETL workflow (replication), 2) providing a pool of multiple identical and ready to use instances of an ETL workflow (redundancy), e.g., in case of a failure of one instance, its execution taken over by an instance from the pool, or 3) providing a pool of multiple different implementations of an ETL workflow (diversity), e.g., in case of a failure, the execution is taken over by another alternative workflow implementation from the pool.

The authors identify the most important reliability measures, namely: a computation availability (i.e., an expected computation capacity of an ETL engine at a given time), a computation reliability (i.e., a failure-free probability that an ETL engine will execute a given task within a certain time window, without errors), a mean time to failure, a mean computation before failure, a capacity threshold, and a computation threshold.

Maintainability is defined as an effort to modify an ETL workflow. A workflow needs to be modified as the result of: source structural changes, source software changes, data warehouse structural changes, DW software changes, connecting or disconnecting data sources.

Freshness is defined as a time difference between data appearance in a data source and a corresponding data appearance in a data warehouse. It can be achieved by increasing performance, improving an ETL design, or providing alternative implementation techniques for real-time processing (streaming).

The aforementioned QoX metrics are interrelated and interdependent, which may lead to a contradictory behavior of a workflow. For example, on the one hand, increasing performance by partitioning and parallelization may increase freshness but on the other hand, it decreases maintainability due to a more complex workflow design. The authors stress that some metrics can be used only at certain levels of DIM. For example, freshness and reliability can be handled at the physical level but at the conceptual level their usefulness is questionable; conversely, performance is a perti-

nent metric at the conceptual, logical, and physical level of an ETL workflow design. The interdependencies between the quality metrics have been confirmed by some experiments. The authors report that: 1) increasing recoverability by means of recovery points decreases performance, as additional disk operations are required to store RPs and 2) the impact of some optimization techniques on the performance varies, e.g., increasing processing power does not improve performance linearly - there are parts for an ETL workload whose parallelization impacts performance stronger than the other parts.

As it requires a complete implementation and execution of the ETL workflow to assess the aforementioned quality metrics, the authors in [27], propose a set of internal metrics (a.k.a measures), that provide an ETL developer a set of guidelines to design efficient ETL workflows even before writing a single line of programming code.

To define and devise the internal measures, the authors first depicted the ETL workflows as directed graphs G(N,E), where N denotes the set of nodes representing the data tasks, and E denotes the set of edges between the nodes. A node can be of one of the several types e.g., 'data input', 'data output', 'filter', 'field lookup','join','union', 'aggregation', 'sort', 'pivot', 'script'. Moreover, there are actions a node is able to perform e.g., 'field manipulation', 'field generation', 'join', 'lookup', 'branching', 'extraction', 'load'. The proposed internal measures are branched into two families: *1) Size Family* and *2) Cohesion Family*.

The *Size Family* is based on the assumption that the efficiency of the ETL workflow is based on the graph size (number of nodes), i.e., the larger the graph size, the lower the efficiency, where the definition of efficiency comprises the execution time and/or resource usage. Furthermore, different type of nodes have different type of impact on the overall efficiency of the ETL workflow. For example, different node measures are defined as follows:

- **Branchin Nodes** measure: cardinality of the nodes in the 'branching' (filter) category,
- **Joining Nodes** measure: cardinality of the nodes in the 'joining' (join, union) category,
- **Lookup Nodes** measure: cardinality of the nodes in the 'lookup' (fieldlookup) category,
- **Script Nodes** measure: cardinality of the nodes having a number of actions greater than 2.

Moreover, the paper proposed further metrics depending on the type of incoming data handled by a specific node e.g., Row-Set, Row-by-Row, and Input-output measures.

- **Row-Set Nodes** measure: cardinality of the nodes type [sort, pivot, aggregation].
- **Row-by-Row Nodes** measure: cardinality of the nodes type [fieldderivation, fieldgeneration]
- **Input-output Nodes** measure: cardinality of the nodes type [datainput, dataoutput]

The *Cohesion Family* is related to the amount of work processed by the graph nodes. That is, the nodes performing more actions (low cohesion nodes) consume more time and resources than noes performing less actions (high cohesion nodes), which can be calculated as the proportion of low cohesion nodes in a graph i.e., cardinality of actions of low cohesion nodes divided by the sum of number of all actions performed by the nodes in a graph.

The calculation and analysis of the aforementioned measures are helpful in pointing out to the ETL developers, which tasks within an ETL workflow may be time and resource consuming in an ETL workflow.

3.8. Statistics for ETL Workflow Optimization

Most of ETL workload optimization methods rely on various statistics. In [38] the authors provide a framework for gathering statistics for cost-based workload optimization. To this end, a workload must be divided into parts, called sub-expressions (SEs). The authors proposed to divide a workflow into SEs based on division points, which are ETL tasks (activities). The following tasks are used as division points: 1) materialization of intermediate results, 2) transformation of values of an attribute that is derived from the join of multiple relations and that is further used in another join, 3) an UDF. Then, each SE is optimized independently. It must be stressed that the proposed framework does not deal with generating execution plans or estimating their costs, i.e., it is assumed that the set of SEs and their optimized execution plans exists and are delivered by an ETL optimizer module.

Finding an optimal execution plan is based on: 1) identifying different possible re-orderings of tasks (operators or activities) in a given SE and 2) estimating their execution costs, in the spirit of [57, 86]. Each task has a cost function that is based among others on: cardinalities of input relations (based on histograms), CPU and disk-access speeds, memory availability. The following tasks are supported: select, project, join, group-by, and transform.

There are multiple sets of statistics (called candidate statistics set - CSS) suitable for optimizing a given SE. Some statistics can be computed from others, based on the following computation rules:

- the cardinality of the select task can be estimated if the data distribution on a filtering attribute is known,
- for the project task, output cardinalities and distributions are identical to the input ones,
- the cardinality of a join can be determined from the distributions of the input tables on a join attribute,
- the cardinality of group by is identical to the number of distinct values of grouping attributes,
- for the transform task, output cardinalities and distributions are identical to the input ones,
- if there exists a histogram on any set of attributes of table T, then the cardinality of T can be computed by adding the values of buckets,
- if there exists a detailed histogram on attributes A and B, then a histogram for A can be computed by aggregating buckets on B.

Each CSS may have a different cost of collecting its statistics (e.g., CPU and memory usage). For this reason, a challenging task is to identify and generate a set of statistics to be collected by an ETL engine during its execution, such that: 1) the set of statistics can be used to estimate costs of all possible re-orderings of tasks within a given SE and 2) time and resource overhead of collecting the statistics is minimal. The authors identified that this is an NP-hard problem and to solve it they proposed a linear programming algorithm. Finally, the authors suggested that the whole ETL optimization method is divided into the 7 following steps: 1) identifying optimizable blocks by dividing a workflow into sub-expressions, 2) generating optimized sub-expressions by means of task reordering, 3) generating candidate statistics sets, 4) determining a minimal set of statistics, 5) augmenting an optimized SE by injecting into it a special component for collecting statistics, 6) running a SE and gathering statistics, 7) optimizing the whole ETL workflow by means of cost-based techniques.

3.9. Commercial ETL Tools

To the best of our knowledge, only 2 commercial ETL tools, namely IBM InfoSphere DataStage [59] and Informatica PowerCenter [2], provide some simple means of optimizing ETL workflows.

In IBM InfoSphere DataStage, the so-called *balanced optimization* is used. The optimization is included in the following design scenario: 1) an ETL workflow is designed manually (each task should be elementary, e.g., simple select from one table instead of a join), 2) the workflow is compiled into an internal representation, 3) optimization options are defined by the ETL developer, 4) the balanced optimization method is applied, and 5) the optimized workflow is produced to be run. The optimization process is guided by some parameters/options/hints, including: 1) reduce a data volume retrieved from a data source (if possible), i.e., move data transformations, aggregations, sorting, duplicate removal, joins, and lookups into a data source, 2) alternatively, if possible, move processing into a data target, 3) use bulk loading to target, (4) maximize parallelism, and 4) use not more than a given number of nested joins. Balanced optimization is supported for relational data sources, but its performance for other data sources still needs to be assessed. In [17], the authors evaluated effects of applying the *balanced optimization* to a streaming data source implemented in Kafka Streams.

Informatica PowerCenter implements the so-called *pushdown optimization*. In this optimization, some ETL tasks that can be implemeted as SQL commands are first identified. Second, these tasks are converted into SQL and executed either at an appropriate source or target database (depending on the semantics of the tasks). This approach leverages the processing power of a database in which data reside.

Other tools, including AbInitio, Microsoft SQL Server Integration Services, and Oracle Data Integrator support only parallelization of ETL tasks, with a parameterized level of parallelism.

3.10. Summary

To sum up, we discussed various approaches to the performance optimization of an ETL workflow, i.e., state-space search, dependency graphs, scheduling policies, and reusable patterns to optimize and ETL workflow execution. We also presented strategies that use parallelism in order to achieve execution performance in an ETL workflow. Finally, we presented optimization support available in commercial and open-source ETL tools. Below, we summarize the approaches on the basis of the following criteria:

1. **Autonomous Behavior** - whether an optimization method is automatic, semi-automatic (i.e., requires input from the ETL developer), or manual;

2. **UDF Optimization** - whether a method supports the optimization of user-defined functions;

3. **Monitoring** - whether a method or framework supports monitoring an ETL workflow in order to identify performance bottlenecks;

4. **Recommendation** - whether a method or framework provides recommendations to improve the implementation of an ETL workflow.

1. **State-Space-based approaches** [38, 57, 86, 88, 96]
text

   - Optimization techniques based on execution costs and tasks reordering (similarly as query optimization techniques) are applied.
   - **Semi-autonomous behavior** - an input is an ETL workflow in the form of a graph. A given workflow is then transformed by an algorithm into a more efficient but semantically equivalent workflow.

   Cons:

   - **No UDF support** - the proposed optimization algorithms support only basic ETL tasks.
   - **No monitoring and recommendation** - the proposed frameworks neither monitor ETL workflows for the performance bottlenecks nor provide hints on how to improve the performance of a workflow.
   - **Challenges** - one of the biggest challenges is the generation of an optimal ETL workflow using the proposed algorithms. If an ETL workflow is large and complex, the generation of an optimal workflow may take longer than the actual time of execution of the ETL workflow itself. Moreover, [86, 88] are limited to a few transition techniques from one state to another and also do not give an account to translate an optimized logical model to its semantically equivalent physical implementation.

2. **Scheduling-based approach** [51]
   Pros:

   - Scheduling policies are applied to optimize an ETL workflow execution time and memory consumption.
   - **Semi-autonomous behavior** - it is mainly focused on scheduling of ETL tasks; the scheduling algorithm requires an ETL workflow as an input; the scheduling is based on pre-defined policies.
   - **Monitoring** - it monitors the entire ETL workflow, but only for the purpose of scheduling ETL tasks at the right time; it does not monitor the tasks to find out performance bottlenecks.

Cons:

- **No UDF optimization and recommendation** - it does not specifically optimize the behavior of an UDF task if it tends to be a bottleneck in an ETL workflow; the scheduling algorithm does not provide any recommendations to improve an input ETL workflow.
- **Challenges** - there is a possibility of losing data during scheduling, therefore, the approach will not be applicable to most of the traditional ETL processing.

3. **Parallelism-based approaches** [60, 61, 62, 94]
Pros:

- **Semi-autonomous behavior** - the proposed approaches give a reasonable account for parallelizing the ETL tasks, but the methods require a considerable amount of input from the ETL developer to decide which parts of an ETL workflow need to be parallelized, how much to parallelize, and where to put split points in order to enable parallelism.
- **UDF support** - the methods proposed in [61, 62] support the ETL tasks as UDFs but do not support their optimization.

Cons:

- **No monitoring and recommendation** - neither mechanisms for monitoring ETL workflows for bottlenecks nor recommendations to improve an input ETL workflow are supported.
- **Challenges** - the proposed solutions do not provide any cost model to identify the required degree of parallelism. Therefore, the ETL developer has to either perform trial and error method or execute the ETL transformations using test data to figure out the required degree of parallelism.

4. **QoX Suite** [87]
Pros:

- Analyzes various quality metrics and their inter dependencies for an ETL design (at a conceptual and logical level), implementation, optimization, and maintenance. The metrics are used for assessing the quality of a workflow.

Cons:

- **No autonomous behaviour** - the approach provides only a theoretical framework.

- **UDF support** - not discussed.
- **No monitoring and recommendation** - neither methods for monitoring the performance of an ETL engine nor the functionality of improving an ETL design based on analyzing quality metrics has been discussed.
- **Challenges** - the biggest challenge is how to efficiently guide the ETL developer through subsequent stages of an ETL design, taking into account current values of the proposed metrics.

To conclude, the state-space search approaches [86, 88] are considered among the first ones towards the logical optimization of an ETL workflow. [86] models the problem as a state-space search problem, where each state in a search space is a DAG. An optimal ETL workflow is achieved by choosing the optimal state from the number of generated states that are semantically equivalent to the original state. [88] focuses on optimizing an ETL workflow for fault tolerance, performance, and freshness. These approaches served as the premise for the optimization of an ETL workflow and were later utilized by various researchers and specialists in this particular topic. [57] proposes the dependency graph that is used for narrowing the space of allowed rearrangements of tasks within a given workflow. Most of ETL workload optimization methods presented in this survey rely on various statistics. In [38] the authors provide a framework for gathering statistics for cost-based workload optimization.

The second group of approaches focuses on strategies that use parallelism as a mean for increasing ETL execution performance, but most of them focus on data flow parallelism. [94] introduces a method to parallelize an ETL workflow (developed in some programming language) by introducing both task parallel and data parallel strategies. [61, 62] present an ETL framework based on MapReduce. Although [61, 62, 60, 94] give a reasonable account for parallelization, these methods require a considerable amount of input from the ETL developer to decide which parts of an ETL workflow need to be parallelized, how much to parallelize, and where to put the split points in order to enable parallelism.

3.11. Conclusions

This chapter focused on the methods for the optimization of an ETL workflow and afterwards narrowed down our point of interest to parallelization techniques. The following ETL optimization approaches have been proposed so far: state-space search, dependency graph, and scheduling policies. The state-space search approach serves as the foundation for the optimization of an ETL workflow for other research. The dependency graph approach focuses on the optimization of a linear and a non-linear logical ETL workflow. The scheduling policies are proposed to optimize the ETL workflow with respect to execution time and memory consumption.

3.11.1. ETL workflow optimization: summary

Table 3.1: Summary of an ETL workflow optimization techniques

Optimization Technique	Autonomous Behavior	UDF Optimization	Monitoring	Recommendation
State-Space (DAG) [38],[86],[88],[96]	Semi-Auto	No	No	No
Dependency graph [57]	Semi-Auto	No	No	No
Scheduling [51]	Semi-Auto	No	No	No
Reusable Patterns [98]	Semi-Auto	No	No	No
Parallelism [60],[61],[62],[94]	Semi-Auto	Yes	No	No

In the literature, there exist multiple methods that revolve around data flow parallelism [12, 20, 24, 37, 41, 109]. However, research on an ETL workflow parallelism has not appealed much consideration.

Table 3.1 summarizes methods on optimizing ETL workflows w.r.t. the criteria described in Section 3.10 i.e., Autonomous Behavior, UDF Optimization, Monitoring, and Recommendation.

The summary of Table 3.1 is as follows:

1. [51, 57, 60, 61, 62, 86, 88, 94, 96, 98] require extensive amount of input from the ETL developer to optimize an ETL workflow, which makes his/her job very complicated and time consuming. Furthermore, the proposed methods require the ETL developer to be highly technical in programming as well as cautious in order to understand the quality metrics and their impact on the performance.

2. The methods do not consider the optimization of UDFs in a comprehensive manner. As UDFs are commonly used in an ETL workflow to overcome the limitations of traditional ETL tasks, it is important to optimize UDFs along with traditional ETL tasks. Since an UDF is typically considered as a black box task and its semantics is unknown, it is very difficult to optimize its execution.

3. Currently, there is no such framework that autonomously monitors an ETL workflow to find out which ETL tasks hinder its performance and gives recommendations to the ETL developer how to increase its performance.

3.11.2. Open issues

On the basis of this literature review, we can conclude this chapter with the following open issues.

1. There is a need for an ETL framework that shall reduce the work of the ETL developer from a design and performance optimization perspective. The framework should provide recommendations on: 1) an efficient design an ETL workflow according to the business requirements, 2) how and when to improve the performance of an ETL workflow without conceding other quality metrics.

2. To improve the execution performance of an entire ETL workflow, techniques based on task parallelism, data parallelism, and a combination of both for traditional ETL tasks as well as UDFs are required.

A new and yet almost unexplored area is handling structural changes in data sources at an ETL layer. In practice, data sources change their structures (schema) frequently. Typically after such changes, an ETL workflow cannot be executed and must be repaired (cf. the maintainability metric in Section 3.7). Such a repair is done manually by the ETL designer, as neither of commercial and open source ETL tools supports (semi-)automatic repairs of ETL workflows. The tools support only impact analysis.

So far, only two research approaches have been proposed that address this problem, namely *Hecataeus* [65] and *E-ETL* [107]. In [65], an ETL workflow is manually annotated with rules that define the behavior of the workflow in response to a data source change. In [107] a case-based reasoning is used to semi-automatically (or if possible - automatically) repair an ETL workflow. Since both of the approaches do not provide comprehensive solutions, this problem still needs substantial research.

A rapidly growing need for analyzing big data calls for novel architectures for warehousing data, such as a *data lake* [93] or a *polystore* [25]. In both of the architectures, ETL workflows serve similar purposes as in traditional DW architectures. Since big data exist in a multitude of formats and the relationships between data often are very complex, ETL workflows are much complex than in traditional DW architectures. Such architectures also need data transformations and cleaning (often on-the-fly, i.e., when a query is executed). For these reasons, designing ETL workflows for big data challenging. Multiple, off-the-box ETL tasks are not suitable for processing big data and such tasks have to be implemented by UDFs. Since a big data ETL engine processes much complex ETL workflows and much larger data volumes, the performance of the engine becomes vital.

The consequence of the aforementioned observation is that designing and optimizing ETL workflows for big data is much more difficult than for traditional data. Therefore, the open issues identified for traditional ETL workflows become even more difficult to solve in the context of big data, which leads us to a development of the next-generation ETL framework presented in next chapter.

Chapter 4
The Next-Gen ETL Framework

In this chapter, we discuss the architecture of the proposed extendable ETL framework that addresses the challenges posed by big data discussed in Chapters 2 & 3. The proposed consists of four modules named as: the UDFs Component, the Recommender, the Cost Model, and the Monitoring Agent. The UDFs Component module addresses the issue of no or minimal support for UDFs and their optimization in currently existing ETL frameworks, which is an integral part to develop ETL transformations for big data. The Recommendation module utilizes the Cost Model component and retrieves information from the Monitoring Agent in order to provide recommendations to the ETL developer. The Monitoring Agent module is proposed to assist the Recommendation module as well as an end-to-end monitoring of ETL workflows.

4.1. Introduction

As discussed in Chapter 2 and 3, we carried out an intensive study on the existing methods for designing, implementing, and optimizing ETL workflows. We analyzed several techniques w.r.t their pros, cons, and challenges in the context of metrics such as: autonomous behavior, support for quality metrics, and support for ETL tasks as user-defined functions.

From the design and implementation point of view, we highlighted that most of the design methods require ETL developers to extensively provide input during the modeling and design phase of an ETL workflow, thus it can be error prone, time consuming, and inefficient. Hence, there is a need for an ETL framework that shall reduce the work of the ETL developer from a design and performance optimization perspective. The framework should provide recommendations on: 1) an efficient design for an ETL workflow according to the business requirements, 2) how and when to improve the performance of an ETL workflow without conceding other quality metrics. Moreover, there is a need to overcome the complexity of writing parallelizable UDFs by incorporating the functionality in an ETL framework to allow ETL developers to

write parallelizable UDFs that tackle the range of different analytical use-cases.

From the optimization point of view, we discovered only a few methods emphasized on the issues of efficient, reliable, and improved execution of an ETL workflow. Whereas, today's need of real-time availability of data requires efficient ETL workflows that can quickly process and analyze huge amount of data. Therefore, to improve the execution performance of an entire ETL workflow, techniques based on task parallelism, data parallelism, and a combination of both for traditional ETL tasks as well as UDFs are required.

In this chapter, we proposed a three-layered architecture of the ETL Framework that allows easy implementation of parallelizable UDFs to execute as an ETL task powered by a distributed framework. Moreover, the proposed ETL framework uses a cost-model to identify how and when to improve the performance of an ETL workflow without conceding other quality metrics.

4.2. The Extendable ETL Framework

The extendable ETL framework is designed keeping in mind the limitations and shortcomings in the currently existing ETL methodologies and tools, and the challenges posed by big data. The three-layered architecture of the proposed ETL Framework is shown in Figure 4.1.

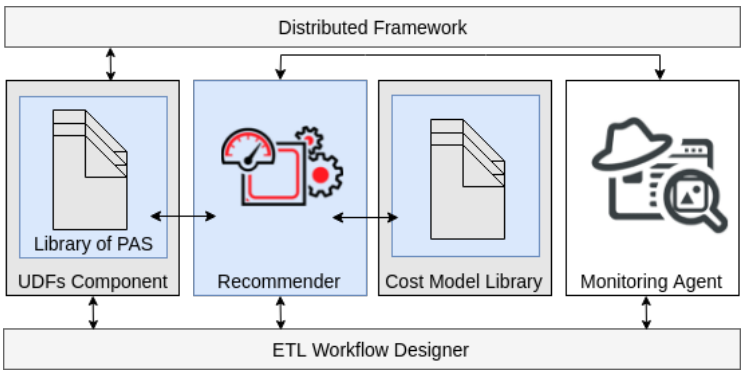

Figure 4.1: The overall architecture of the *ETL Framework*

The bottom layer is an *ETL Workflow Designer*, which may be any standard open source ETL tool for designing ETL workflows. This layer communicates with the middle layer, which is extendable and consists of the four components: 1) the *UDFs Component*, 2) the *Recommender*, 3) the *Cost Model*, and 4) the *Monitoring Agent*, described in detail in the following sub-sections.

The top layer in the architecture is the *Distributed Framework*. Its task is to

execute parallel codes of UDFs in a distributed environment, in order to improve the overall execution performance of an ETL workflow.

4.2.1. The UDFs Component

The idea behind introducing this component is to assist the ETL developer in writing a parallelizable UDF by separating parallelization concerns from the code.

The *UDFs component* contains a library of *Parallel Algorithmic Skeletons* (PASs) or parallelizable code templates. These PASs are designed to be executed in a distributed environment, (e.g., a template for MapReduce or Spark to be executed in Hadoop). The UDFs component requires a basic knowledge of distributed computing and parallelization aspects from the ETL developer.

Figure 4.2 shows the working of the *UDFs Component*. The component provides the already parallelizable code for the list of commonly used big data tasks (case-based PASs) to the ETL developer (e.g., sentiment analysis, de-duplication of rows, outlier detection) and a list of generic PASs (e.g., worker-farm model, divide and conquer, branch and bound, systolic, MapReduce). The ETL developer either chooses the Case-based PASs or the Generic PASs based on his/her requirements.

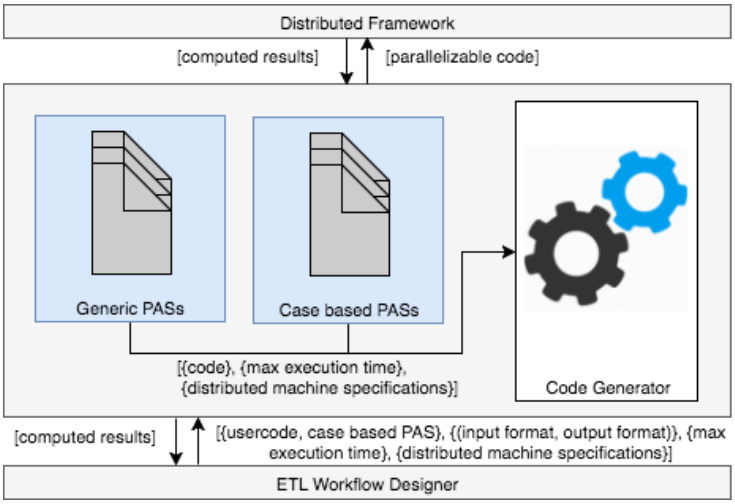

Figure 4.2: Extendable UDFs Component Architecture

As shown in Figure 4.2, a generic input to the *UDFs Component* is depicted as *[{usercode, case based PAS}, {(input format, output format)}, {max execution time constraint}, {distributed machine specifications}]*. For example, in case of the case-based reasoning the ETL developer only has to provide the input and output data formats *{(input format, output format)}*, execution time constraint to run the ETL

workflow (e.g., the ETL job must complete execution with in 'x' number of hours *{max execution time}*), and distributed machine specifications *{distributed machine specifications}*, if known. In case of the generic PASs, the ETL developer has to provide the basic program for the chosen PAS *{usercode}*, an execution time constraint to run the ETL workflow *{max execution time}*, and distributed machine specifications *{distributed machine specifications}*. That is, for the MapReduce paradigm as a PAS, only Map and Reduce functions would be required. The MapReduce configurations (i.e., partitioning parameters, number of nodes) will be provided by the *UDFs Component*. The *Code Generator* then generates the configuration and a paralellizable code based on the ETL developer's input to the component about the distributed machine specifications, time constraints on the completion of the ETL workflow, and by the recommendation of the *Recommender* component in the proposed ETL framework. The specific configurations provided by this component are very critical to achieve the right degree of parallelism.

Once the configurations are generated, the code provided by the ETL developer and the distributed environment configurations will be executed in *Distributed Framework*. The computed results are then returned to the ETL workflow for the next steps in the workflow.

4.2.2. The Recommender

The Recommender includes an extendable set of machine learning algorithms to optimize a given ETL workflow (based on metadata collected during past ETL executions) and to generate a more efficient version of the workflow. Metadata may be collected with the help of the *Monitoring Agent*, where it collects various performance statistics of different ETL workflows and provide them to the Recommender. Since, there are a few algorithms that can be applied to optimizing a workflow (e.g., *Dependency Graph* approach) [88, 86], *Scheduling Strategies* [51], the ETL developer would then be able to experiment with alternative algorithms and compare their optimization outcomes.

The *Recommender* component also helps the ETL developer to choose the best possible PAS from the *UDFs Component* based on the developer's input (c.f. Section 4.2.1) to the Recommender. To provide the optimal PAS to the *UDFs Component*, it uses the *Cost Model* component.

4.2.3. The Cost Model

The algorithms used by the *Recommender* need cost models. The Recommender can choose the appropriate cost model from a library of cost models in order to make optimal decisions based on the ETL developer's input to it.

The library of cost models may include cost models for monetary cost, performance cost, and both cost and execution performance optimization. Since most of the big

data ETL workflows or UDFs for big data are executed in a cloud or a distributed framework, there would be cost models to evaluate the performance of workflows in a cloud computing environment [44, 46] and also to determine the best possible configuration of virtual machines both in terms of execution time and monetary cost [104].

Since the Recommender uses the Cost Model component to provide the optimal PAS to the UDFs Component, the cost model would be able to select the optimal PAS based on the *Multiple Choice Knapsack Problem* (MCKP) [43]. For example, suppose an ETL workflow consists of n different computationally intensive UDFs and UDFs component may generate m parallel variants of each UDF, there are m^n combinations of code variants. Therefore, finding an optimal UDF may be mapped to MCKP.

4.2.4. The Monitoring Agent

The *Monitoring Agent* allows to:

- monitor ETL workflow executions - e.g., number of input rows, number of output rows, execution time of each step, number of rows processed per second.
- identify performance bottlenecks - e.g., which tasks are being delayed or aborted, which tasks need to be optimized.
- report errors - e.g., task or workflow failures and the possible reasons.
- schedule executions - e.g., execution time of ETL workflows and creating a dependency chart for ETL tasks and workflows.
- gather various performance statistics - execution time of each ETL task w.r.t rows processed per second, execution time of the entire ETL workflow w.r.t rows processed per second, memory consumption by each ETL task.

This is a standard component of any ETL engine. However, we would store all of the aforementioned collected information in an ETL framework repository to be later utilized by the *Recommender* and the *Cost Model* in order to make recommendations to the ETL developer and to generate optimal ETL workflows.

4.3. Conclusions

Few research work has been proposed in literature on ETL frameworks specifically for big data besides some cloud based distributed frameworks (e.g., *Amazon Web Services*[1] stack, *Google Cloud Platform*[2], and *Microsoft Azure*[3]). These cloud based

[1] https://aws.amazon.com
[2] https://cloud.google.com/products
[3] https://azure.microsoft.com

distributed platforms provide several products that help in creating big data ETL data pipelines and solutions. However, the provided products are not fully autonomous as well as does not provide recommendations to the ETL developer for creating optimized data pipelines at run-time.

In our proposed ETL framework, we addressed the issues like complex ETL workflows due to 3Vs of big data and how to solve the issues of compute-intensive UDFs. Finally we also provided a fully automated framework to create ETL workflows.

In the next chapters, we discuss UDFs component and Cost Model module of the proposed ETL framework to parallelize UDFs in an ETL workflow in detail with experimental results.

Chapter 5
Parallelizing User-defined Functions in an ETL Framework

In this chapter we present the UDFs Component of the ETL framework, which allows the ETL developer to choose a design pattern in order to write a parallelizable code and to generate a configuration for the UDFs to be executed in a distributed environment. This enables ETL developers with minimum expertise of distributed and parallel computing to develop UDFs without taking care of parallelization configurations and complexities. We performed experiments on large-scale data sets based on TPC-DS and BigBench. The results show that our approach significantly reduces the effort of ETL developers and at the same time generates efficient parallel configurations to support complex and data-intensive ETL tasks.

5.1. Introduction

In this chapter, we present a novel approach to incorporate a functionality in an ETL framework, which assists the ETL developer in writing parallelizable UDFs to be executed in a distributed environment e.g., Hadoop, Flink, etc. To achieve our goal, we leverage the so-called *Orchestration Style Sheet (OSS) processor* that encapsulates and separates the parallelization concern from the development of UDFs. This processor generates a parallelizable code and a set of configurations to execute a generated UDF in a distributed environment.

In particular, our contributions highlighted in this chapter are the following:

- We provide a software application for the integration of any open source ETL framework to facilitate the ETL developer in writing a UDF by separating parallelization concerns from the code, thus reducing potential error sources in the otherwise manual and cumbersome parallelization process. The choice of open source ETL tool was made to use it as a sandbox for our approach, because it provides more flexibility and control to the developers in order to develop custom ETL tasks and alter currently existing ETL tasks. For example, we

used Pentaho Data Integration, which allows the developers to program user-defined functions and generate them as any other built-in ETL task.

- We provide code skeletons or design patterns to be used by our application to reduce the amount of effort required by the ETL developer in writing complex and efficient programs.
- We present experiments on a large-scale data set based on BigBench [34], proving that optimizing the computing-intensive user-defined task improves the overall performance of an ETL workflow.

In Section 5.2 we describe the use case scenario of our running example, which we will use throughout the chapter. We will then introduce Orchestration Style Sheet (OSS) processor in Section 5.3. Our proposed framework and approach towards generating parallelizable UDFs using OSS are described in Section 5.4. In Section 5.5, we discussed the use of Map-Reduce OSS for the sentiment analysis use case. Experimental evaluation for the feasibility of our approach is discussed in Section 5.6. Conclusion is included in Section 5.7.

5.2. Running Example

To motivate our discussion, we borrowed the product retailer use case of BigBench [34] that covers 3Vs of big data. The data model consists of structured, semi-structured, and unstructured components. The structured part of data is adopted from the TPC-DS benchmark [69]. The semi-structured portion of data consists of user's clicks on the product retailer's website. The unstructured portion of data comprises product reviews submitted online in the English language.
As a use case for the remainder of the chapter we focus on an analytical scenario that compromises the sentiment analysis of the product reviews. The data set contains one or more reviews for each product submitted online by the users of the products. Since it would be not acceptable to perform sentiment analysis over huge amount of data every time during query processing, we want to pre-compute the sentiments. The respective ETL workflow developed in Pentaho Data Integration (PDI) is depicted in Figure 5.1. The ETL workflow sources data from tables *INVENTORY*, *DATE DIM*, *WAREHOUSE*, *ITEM*, and *PRODUCT_REVIEWS*. The process fetches the products that are in the inventory from year 2000 onward and are available in all the warehouses located in the US. Then it applies the sentiment analysis algorithm on the incoming data set in order to classify the unstructured product reviews as *Negative* or *Positive*. This functionality is implemented as a UDF called *CPUDF_DWH_OUT*. Finally, the computed result is exposed for analysis as a table that can be queried later by the analyst.

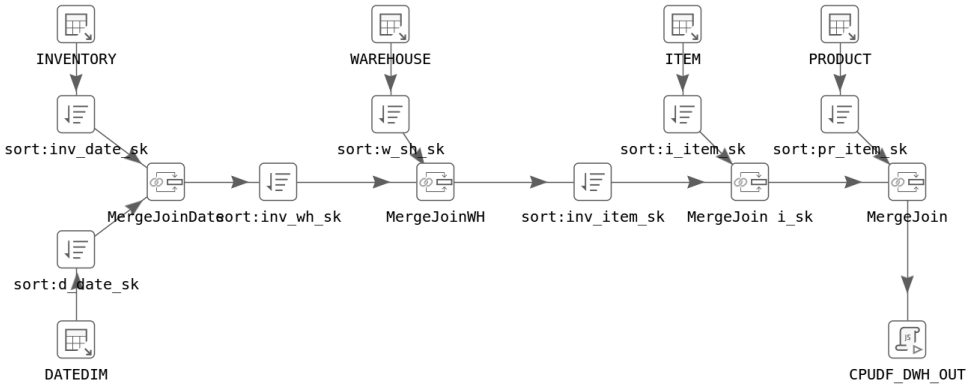

Figure 5.1: The ETL workflow for the running example

5.3. Orchestration Style Sheets (OSS)

To support multiple, potentially differently parallelized target platforms with as little implementation overhead as possible, *parallel algorithmic skeletons (PAS)* can be used. PAS are defined as algorithmic skeletons, or parallelism patterns, which are high-level parallel programming models for parallel and distributed computing. These can be provided as libraries working on specific data structures like vectors and matrices as well as algorithms like MapReduce. However, these libraries are constrained to their supported data structures and algorithms, making them unsuitable for problems not meeting these constraints.

In these cases, pragma languages offer a very flexible, yet low-level, alternative: languages like OpenMP [23] support a wide range of target platforms while offering high customizability. The price for this, however, is an increased implementation overhead.

Pragma languages are also known as directive languages. These are described as a construct that specifies how a compiler processes its input. Directives are not part of the grammar of a programming language, and may vary from compiler to compiler. They can be processed by a pre-processor to specify compiler behavior. Both in task- and loop-based parallelization, the pragmas have to be repeated for every task or loop, each time with slightly different parameters.

OSS [67] provide a way to combine the simplicity of skeleton libraries with the flexibility of pragma languages using *invasive software composition (ISC)* [8]. Hence, achieving language and platform independence. The central idea is to split the code into reusable code *fragments* that can be *woven* into different variants of the source code. Two aspects of this weaving process are particularly important.

First, the **fragment specification** and the **weaving specification** are performed declaratively in *style sheets* and *recipes*. Style sheets contain *styles* consisting of code fragments and *addressing expressions* determining the positions in the source code at which they can be inserted. The code fragments themselves can contain variability points, *slots*, that can serve as positions at which other fragments can be inserted. The weaving specification is done with *recipes* that determine the selection and order of styles to be applied to the code. Different recipes can be used to acquire different code variants. Figure 5.2 gives an overview over the involved artifacts in OSS code weaving.

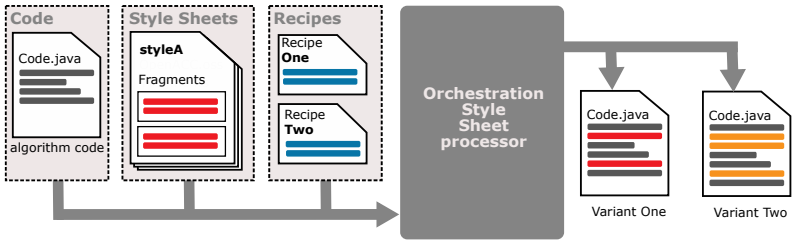

Figure 5.2: A workflow of the OSS processor showing the required and generated artifacts

The second important aspect is that fragment specifications can contain **attributes**, special code fragments that are not contained in the style definition but rather computed using the program code. Using attributes, the results of problem-specific, user-defined static analyses of the source code can be directly inserted into the code.
In the presented use case, this analysis is used to reason about and transform data types to adapt them to the interfaces provided by the distributed framework - Hadoop.

Furthermore, attributes can be used to specify application-specific variability points. This permits all OSS-specific parts of the code to be hidden from the application developer. Code fragments specified by him/her do not have to be included in style files but can also be specified implicitly, e.g., as class members at specifically defined positions. This method is used in the example as shown in Listing 5.1.

The source code composition of OSS including the fragment computation with attributes is performed utilizing *reference attribute grammars (RAGs)* [39], a well-known technology in compiler construction. OSS processor uses SkAT [52], a composition system built using the RAG tool JastAdd [26], that allows the type-safe and well-formed composition of code fragments and the utilization of an extensible Java compiler, ExtendJ. The OSS processor replaces the programmatic code composition of SkAT with the presented declarative approach. The computed fragments used by OSS are implemented as attributes of the attribute grammar, which in JastAdd are simply specifically annotated methods added to language elements with an aspect

weaver [53]. Thus, writing new, user-defined, and problem-specific attributes is a feasible task for a Java and Hadoop developer, enabling future extensible to distributed frameworks and programming paradigms other than Hadoop, e.g., Spark, Flink.

5.4. Generating Parallelizable UDFs for an ETL Workflow

In order to facilitate the ETL developer to write parallelizable UDFs without the parallelization and optimization aspects of a program, we contribute the *configurable-parallelizable UDF generator* called as the *UDFs Component*. The UDFs Component can easily be integrated into an open source ETL framework e.g., Pentaho Data Integration (Spoon) as a third-party tool. It utilizes OSS to generate parallelizable code. In this approach we used OSS because it is a lightweight and extensible approach that can be adopted to any target run-time on any High Performance Computing (HPC) cluster. For example Graphics Processing Unit (GPU) or can be adopted by parallel computing frameworks such as Hadoop and Spark.

Figure 5.3 shows the three-tier architecture of our proposed framework. The top layer depicts Pentaho Data Integration (PDI), which provides a graphical user interface to create ETL workflows. The middle layer comprises the UDFs Component and the OSS composition system to provide parallel skeletons to the ETL developer, thus enabling him/her to write parallelizable code without taking care of the critical parallelization details (i.e., degree of parallelization - specified by the number of data partitions, the number of map and reduce tasks in case of Hadoop as a distributed framework). Subsequently, OSS is used to generate configurations for the user's code, which are finally executed in a distributed environment - Hadoop, as shown in the bottom tier.

The involved artifacts and procedures are enumerated in Figure 5.3 and will be explained in the following section to illustrate the process of generating optimized UDFs to be executed in a distributed environment. We will explain the working of the UDFs Component and application of the OSS processor with the help of our running example described in Section 3.1.1.

5.5. Using Map-Reduce OSS for Sentiment Analysis UDF

The running example in Section 3.1.1 discusses the use case of sentiment analysis of product reviews as a UDF in an ETL workflow performed by a user-defined step depicted as $CPUDF\_DWH\_OUT$ in Figure 5.1 (c.f. Section 3.1.1). The idea is to

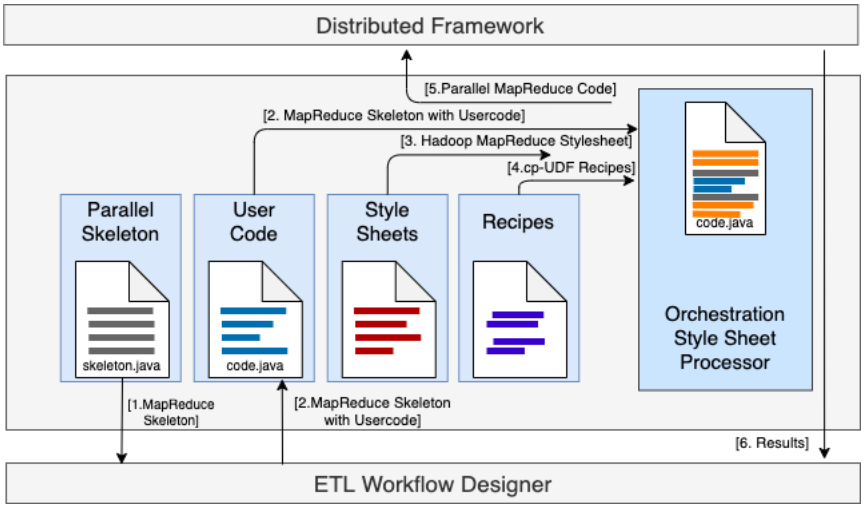

Figure 5.3: the UDFs Component high-level design for the generation of parallelizable code using OSS

pre-compute the user sentiments during the ETL phase and propagate the computed results into a data mart. This would help the data analysts to avoid executing the high latency query every time they want to make decisions based on the user sentiments.

To assist the ETL developer to write parallelizable code for sentiment analysis, we provide him/her with different *Parallel Algorithmic Skeletons (PAS)* or code skeletons that can be executed in a distributed environment. For example, worker-farm, divide and conquer, branch and bound, systolic, MapReduce or Spark code skeleton, where the ETL developer has to insert his/her code for sentiment analysis into the provided skeleton (Figure 5.3, Step 1).

Listing 5.1 shows the template for the Hadoop MapReduce use case with comments highlighting the positions at which the user can include his/her code[1].

As the template is a regular Java class, additional code like helper functions and definitions of fields can be added, if necessary. The template is filled with the required map and reduce code by the ETL developer and is sent to the UDFs Component, as depicted in Figure 5.3, Step 2. The ETL developer only has to know the theoretical detail of MapReduce paradigm so that he/she can logically divide the UDF code into the map and the reduce functions.

The critical and the most important mode of parallelization i.e., deciding the optimal number of mappers and reducers, the number of partitions to make, the processing power of virtual machine to execute the UDF in order to achieve maximum performance and all modifications to the code are denoted in style sheets (*Runner Class*

[1]In addition to the method bodies, the parameter types have to be adapted accordingly.

```
// #HADOOP_MAP_REDUCE#
public class Template {
    public static class Map {
        void setup() {
            // add setup code here
        }
        void map(MapKeyType key, MapValueType value, Context context) {
            // add map code here
        }
    }
    public static class Reduce {
        void setup() {
            // add setup code here
        }
        public void reduce(ReduceKeyType key, Iterable<ReduceValueType>
            values, Context context) {
            // add reduce code here
        }
    }
    void config() {
        // add configuration code here
    }
}
```

Listing 5.1: The empty template (PAS) provided to the user

```
style map:hadoop {
  fragment MethodSlot * if(isMapMethod) {
    slot KEYTYPE : mapKeyType
    slot VALUETYPE : mapValueType
    code:
      <Method>
        @Override
        public void map(#KEYTYPE# key, #VALUETYPE# value, Context context)
            throws IOException, InterruptedException {
          #INNER#
       }
      </Method>
  }
  // other fragments
}
```

Listing 5.2: Excerpt of the Hadoop-MapReduce OSS style

```
1 recipe cpUDF {
2   map:hadoop
3   reduce:hadoop
4   runner:benchmark
5 }
```

Listing 5.3: OSS recipe used in the UDFs Component

in case of MapReduce paradigm) automatically provided by the UDFs Component. Hence, separating the parallelization concerns from the code.

Therefore, the ETL developer does not have to provide the critical details of parallelization. The UDFs Component will provide the best possible configuration for executing the UDF in a distributed framework using *recipes* in a form of a *Runner class*.

Listing 5.2 depicts the example of a Map *style sheet*, which modifies the `map` method provided by the ETL developer in Listing 5.1, inserts the required data types and returns values.

A *recipe* is used to trigger the composition (Figure 5.3, steps 3 and 4). A recipe determines the selection and order of styles to be applied to the code. Listing 5.3 contains the used recipe, which first applies map and then reduce style sheets (cf. line 2 and 3 respectively) on to the provided UDF code as well as append the runner class to the UDF in order to specify the parallelization configuration. Finally, with the provided skeleton code including the UDFs, the UDFs Component invokes the OSS processor to build a parallelized version. The OSS processor is considered as a black-box for the UDFs Component and we assume that the output of the OSS processor will always be correct and accurate.

Using these input artifacts, OSS generates a MapReduce (parallelized) version of the user-defined function (Figure 5.3, Step 5) that is subsequently processed by Hadoop.

Currently in the proposed framework, we used a *runner* class (i.e., configuration class for optimal parallelization) best suited for the available distributed environment. However, as a future work, we will introduce the mechanism in our proposed framework to generate multiple *recipes* i.e., multiple parallel configurations, and then to choose the optimal configuration for the distributed environment based on cost and computation performance requirements. Furthermore, we will add more code skeletons (PAS) e.g., for Spark, Flink, etc. in our library of skeletons.

5.6. Experimental Evaluations

In this section we discuss the execution performance (i.e., execution time of ETL workflow) of the sentiment analysis code as a UDF generated by the UDFs Component (i.e., parallelizable code). The generated parallelized UDF is executed in a cloud-based distributed environment and a non-parallelizable UDF program executed in a cloud based non-distributed environment.

We created two versions of the sentiment analysis algorithm in order to show two different variants of the same algorithm, which are semantically equivalent.

The first exemplary variant of a sentiment analysis algorithm, called naïve, is a handwritten custom code and is taken from [22]. It counts the number of positive and negative words in a review by comparing them with positive and negative dictionaries loaded into the memory as a part of initial configuration[2]. For each positive and negative word it increments a respective counter. Finally the sentiment score is calculated by a formula: *positivity=good / (good + bad)*. To categorize the user review, each result above a *THRESHOLD* value is classified as *Positive*, otherwise as *Negative*.
The second variant of the algorithm, called CoreNLP, is a long running, computing-intensive program, which uses the Stanford CoreNLP framework [64]. It provides natural language tools to annotate sentences to indicate parts of speech, named entities, word dependencies, and sentiment.

The idea behind testing different variants in different environments (i.e., cloud-based distributed environment, pseudo-distributed environment (single node EMR Cluster), and a non-distributed and a non-EMR environment) was to prove the following two assumptions:

- the non computing-intensive code (i.e., naïve) normally does not effect the overall performance of an ETL workflow whether it is executed in a distributed or a non-distributed environment. In most of the cases, the non-compute intensive program has an extra overhead if executed in a distributed environment. Nevertheless, if execution performance is a strict requirement, it will cost a lot more resources and the improvement of execution performance would still be much lower than expected.

- The computing-intensive tasks become a bottleneck in an ETL workflow and must be optimized. Because even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow.

To evaluate our approach, we used a BigBench [34] data set on around 20 Million

[2]The configuration can also be done in the `setup` method of the `Map` class

product review tuples. Each product has around 3 reviews and each review consists of approximately 36 words.

Following are the details and learning of our experiment.

To carry out the evaluation, We used the M3.xlarge model of EC2 instances, each having the similar specifications i.e., Intel processors with 4 cores vCPU and 15 GB RAM. We evaluated the non-parallelizable version of the UDF on the non-distributed EC2 instance with the same system configurations and specifications. We evaluated our MapReduce UDFs on a single node (MR-SN) Amazon Elastic MapReduce (EMR) cluster to depict a pseudo-distributed environment and on a six node EMR cluster (MR-EMR) as a distributed environment. One node served as the master and the rest as core workers. We tested the execution time of both parallel and non-parallel UDFs with different sizes of data sets ranging from one thousand tuples to 20 million tuples of unstructured data. Each test was executed at least five times and the results presented are the average values of those test runs.

The Performance execution comparison of a non-parallelizable naïve sentiment analysis program vs. semantically equivalent parallelizable MapReduce version is shown in Figure 5.4 and Figure 5.5. The graph in Figure 5.4 shows 1) the execution time of the non-parallelizable variant of the naïve code, 2) its corresponding MapReduce version executed in a single-node (MR-SN) EMR cluster, and (3) the MapReduce version executed in the six node Amazon EMR cluster (MR-EMR).

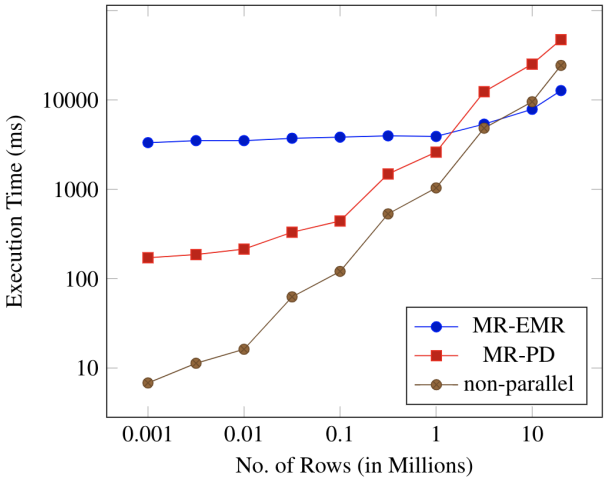

Figure 5.4: MapReduce vs. non-parallelizable naïve code exe. time

For a data set ranging from one thousand to less than one million tuples, the naïve non-parallelizable program is more efficient in terms of execution performance as compared to its MapReduce variants (MR-SN and MR-EMR). As the number of

tuples increases to more than one million, the execution time of MR-EMR reduces as compared to MR-SN as well as the non-parallelizable program.

The graph in Figure 5.5 shows that the speedup of MR-EMR is small, i.e., by a factor of two and the speedup of MR-SN is worse than the non-parallelizable variant because of the Hadoop overhead.

The speedup is determined as the execution time of a MapReduce program divided by the execution time of a non-parallelizable program.

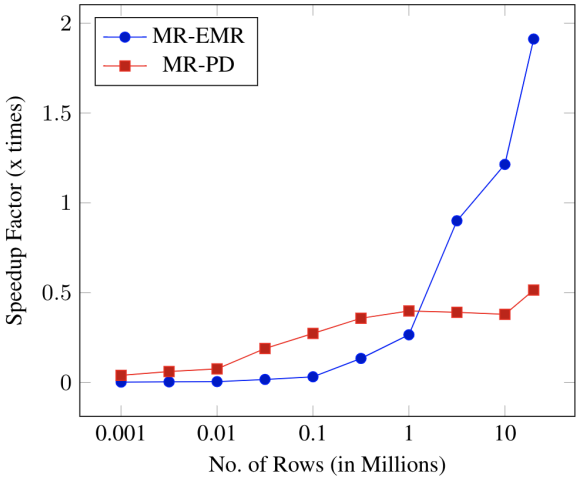

Figure 5.5: Parallelizable MapReduce speedup for naïve code

Hence proves our first assumption that the non computing-intensive code normally does not effect the overall performance of an ETL workflow whether it is executed in a distributed or a non-distributed environment.

Performance execution comparison of a non-parallelizable CoreNLP sentiment analysis program vs. semantically equivalent parallelizable MapReduce version is shown in Figures 5.6 and 5.7.

Figure 5.6 shows the execution time of the CoreNLP variant of the sentiment analysis program. The execution time for MR-SN and the non-parallelizable version are similar. The execution time difference between the MR-EMR and the non-parallelizable program is small for the number of rows less than or equal to 0.1 million. However, as the data size increases, the MR-EMR execution performance increases as the number of Map workers increases and computing intensive tasks are executed in parallel.

Figure 5.7 shows a notable speedup for MR-EMR for 0.5 million tuples and above. However, there is no speedup for MR-SN since there are only two Map workers in a pseudo-distributed environment. Hence, the results prove our second assumption

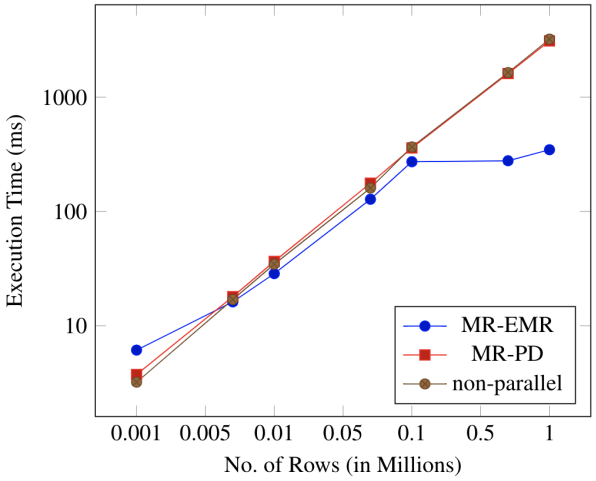

Figure 5.6: MapReduce vs. non-parallelizable CoreNLP exe. time

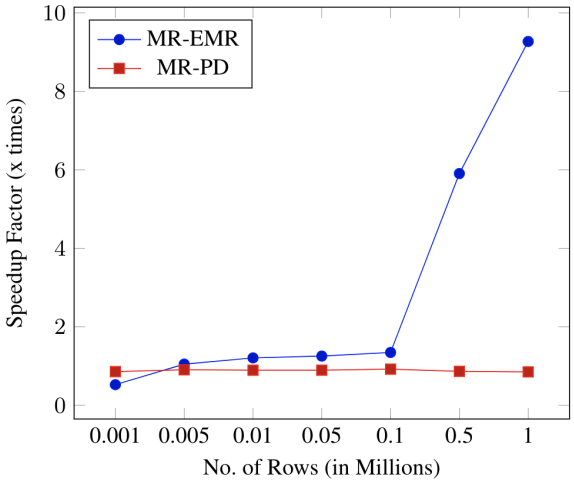

Figure 5.7: Parallelizable MapReduce speedup for coreNLP code

that for computing-intensive ETL tasks, even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow.

We also analyzed the time saved by the programmers in developing or designing such UDF's using or not using the UDFs Component to address the second contribution i.e., code skeletons or design patterns to be used in the UDFs Component helps to reduce the amount of effort required by the ETL developer in writing complex and efficient programs. Figures 5.8(a) and 5.8(b) shows the effort required by the ETL developer in terms of Line of Code (LOC) to write an efficient parallel sentiment analysis program. The black portion of the figure shows the user-defined code weaved together with the gray portion of the code, which is generated by the UDFs Component using the OSS processor. In the naïve version, almost 50% of the code and in case of the CoreNLP version almost 65% of the code is generated automatically by the UDFs Component.

Figure 5.8: Estimate of effort required by the ETL developer to write a MapReduce program using OSS in terms of LOC

Overall we can observe that the UDFs Component significantly reduces the effort required by the ETL developer to write parallel code and ensures an error-free code by replacing otherwise manual steps in the parallelization process with automatized semantic analysis. Also it hides the low level details of parallel execution of a program from the ETL developer, and in addition accomplishing considerable speedup without

worrying about controlling the costs occurred by processes, communication, and data distribution. The directive code and parameterized attributes make OSS flexible and extensible that can easily be adopted to other use cases with special, user-defined analysis attributes and even to other languages (e.g, Python, Fortran).

5.7. Conclusions

As discussed in Chapter 3, there does not exist many approaches in literature focusing on optimization of ETL workflows through parallelising UDFs. Moreover, writing parallelizable UDFs require extensive knowledge of parallel and distributed programming and therefore there lies a huge complexity of writing parallelizable UDFs as ETL tasks.

The contribution presented in this chapter is threefold. First, we address the optimization of the ETL workflow through parallelizing the UDFs. Second, we provide the ETL developer with out-of-the-box functionality in an ETL framework to write efficient parallel UDFs. Third, we additionally focus on the UDFs that support semi-structured and unstructured data sets to pre-compute the computing-intensive analytic queries. We provide a component to assist ETL developers in writing parallelizable UDFs for an ETL workflows. Although, most ETL tools provide the functionality to write custom code as UDFs but a UDF to transform a large and partially structured data can be very complex and may become a performance bottleneck if not implemented optimally. One of the possible ways to implement an efficient program is parallelizing its execution, which requires expertise and deep understanding and knowledge of parallel and distributed programming. To provide the ETL developer with out-of-the-box functionality in an ETL framework to write efficient parallel custom programs, we proposed the UDFs Component, which separates the parallelization concerns from the development of UDF tasks for data-intensive ETL workflows. Currently, the UDFs Component supports Hadoop as a distributed framework to execute UDFs in a parallel environment. However, it is extensible and can be integrated with other parallel and distributed frameworks e.g., Flink and HPC clusters. On top of that, we showed that the UDFs Component provides an easy, fast, and flexible way for the ETL developer to write efficient and error-free parallel programs for data-intensive tasks.

In the next chapter, we will present our contribution on the cost model for UDFs, which are treated as black-box tasks. The cost model enables the optimization of already parallelizable (e.g., MapReduce-based [24] or Spark-based [108]) UDFs in an ETL workflow. Our optimization approach draws upon determining the right degree of parallelism for an UDF (or a set of UDFs) to satisfy user-defined performance metrics.

Chapter 6
The Cost Model

In this chapter we address the problem of the optimization of UDFs in data-intensive workflows and presented our approach to construct a cost model to determine the degree of parallelism for both case-based and generic parallelizable UDFs. Moreover, we extend our cost model to enable data scientists to choose the best possible machine learning model based on user-defined performance metrics.

6.1. Introduction

As mentioned in Chapter 4, the extendable ETL framework consists of four modules, namely: 1) the *UDFs Component*, 2) the *Cost Model*, 3) the *Recommender*, and 4) the *Monitoring Agent*, c.f., Figure 6.1.

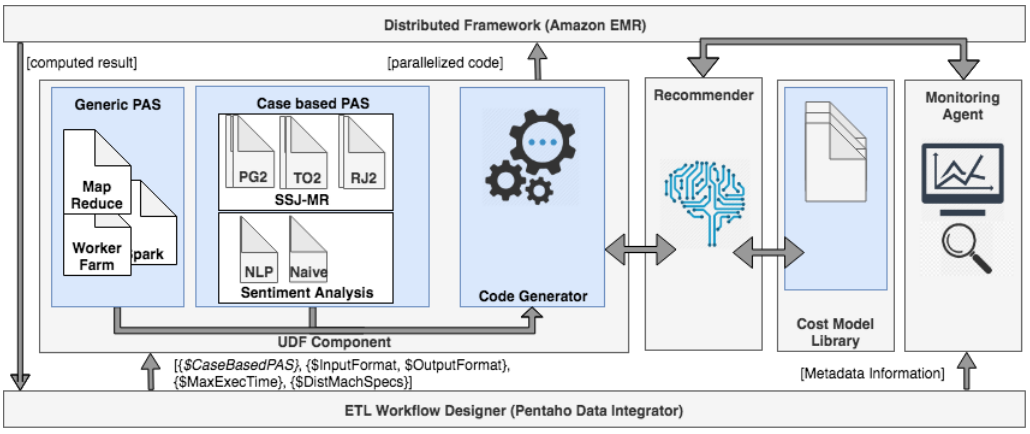

Figure 6.1: The architecture of the *Extendible ETL Framework*

The *UDFs Component* allows ETL developers to write parallelizable UDFs by separating parallelization concerns from the code. It contains a library of *Parallel*

Algorithmic Skeletons (PASs), to be executed in a distributed environment like MapReduce or Spark. The *UDFs Component* provides to the ETL developer: 1) the already parallelizable code of some commonly used big data tasks (a.k.a *Case-based PASs*) including: sentiment analysis, de-duplication of rows, outlier detection and 2) a list of *Generic PASs* (e.g., worker-farm model, divide and conquer, branch and bound, systolic, MapReduce).

In this chapter, we present a cost model for UDFs, which are treated as black-box tasks. First, we propose methods towards parallelization of UDF execution by generating an optimized distributed machine configuration. Second, we extend our cost model to enable data scientists to choose the best possible machine learning model based on user-defined performance metrics e.g., accuracy, precision, or recall of a model using decision optimization.

The *Cost Model* leverages *Decision Optimization* and *Machine Learning* techniques in order to generate a (sub-)optimal configuration to optimize the execution of parallelizable UDFs in an ETL workflow, cf., Figure 6.2. The *Decision Optimization* technique is used to find an optimized configuration for the *Case-based PASs* (c.f., Section 6.3), fulfilling user-defined performance metrics: execution time and monetary costs. The *Machine Learning* technique consists of machine learning models trained on historical data and fine-tuned. They are applied to find the (sub-)optimal machine configuration to execute UDFs based on the *Generic PASs* (c.f., Section 6.4). The dotted line indicates that if the *Decision Optimization* fails to find the (sub-)optimal solution, then the *Machine Learning* portion is used to find the solution.

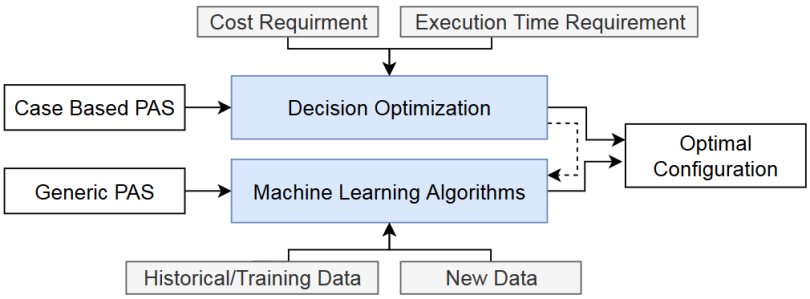

Figure 6.2: High level design of the cost model

To determine the right degree of parallelism and to generate an optimal configuration for an UDF to be executed in a distributed environment, the cost model must provide the following functionality. First, it must answer the below questions.

- Is an UDF parallelizable?
- If an UDF is parallelizable, will it profit from parallel processing to satisfy user-defined performance metrics?

- If an UDF profits from parallel processing, what will be the adequate (sub-optimal, optimal) parallelization parameters? In this research we consider the following parameters: 1) the number of data partitions, 2) the number of mapper and reducer tasks (in case of a MapReduce-based UDF), 3) the number of nodes in a cluster, 4) a physical and software configuration of the cluster.

Second, the cost model must support generating an optimized configuration for an already parallelizable UDF to be executed in a distributed framework. Notice that the configuration may also support a sub-optimal execution plan.

This chapter is organized as follows. Section 6.2 presents the general overview of the Cost Model. Sections 6.3 and 6.4 discusses the *Case-based PASs* specific explanation and the *Generic PASs* specific explanation of the cost model respectively. Section 6.5 discusses the extension to the Cost Model to enable data scientists to choose the best possible machine learning model based on user-defined performance metrics. Section 6.6 discusses the efficiency of the Cost Model by means of experimental evaluation, and finally Conclusions is included in Section 6.7.

6.2. Overview of the Cost Model

To optimize the execution of parallelizable UDFs in an ETL workflow according to some user-defined performance metrics, i.e., execution time and monetary costs, the use of the cost model is as follows.

- *Stage 1 - Feasibility*: the cost model will first determine the feasibility to parallelize UDFs, i.e., whether it makes sense to parallelize an UDF, in order to satisfy the user-defined performance metrics.
- *Stage 2 - Degree of Parallelism*: the cost model will reason on the right degree of parallelism, i.e., how much to parallelize (e.g., choosing the appropriate number of partitions to distribute the data to be transformed in parallel).
- *Stage 3 - Optimal Code Generation*: the cost model will guide the creation of an efficient configuration for distributed machines, so that the UDF is executed optimally in a distributed environment, adhering to the execution performance and monetary cost constraints defined by the developer as an input to the UDFs component.

6.2.1. Stage 1 - feasibility

The Cost Model is used by the *Simulator* to simulate an UDF execution in a non-distributed parallel environment. The simulation helps in finding out if it makes sense to execute the UDF in a non-distributed environment by comparing the actual

execution time of the UDF with the user-defined performance metrics. If the execution time is lower than or equal to the required execution time, the framework would execute the UDF in the non-distributed environment.

Further extension is planned to the *Simulator* to be able to identify the core aspects about an UDF such as: execution time of the UDF for a given data set, the number of rows processed per second, the number of bytes processed per second, the size of data, the used distributed machine configuration as well as memory, IO, and CPU usage characteristics, in the spirit of [38]. The performance data extracted from the simulator will then be used to predict the optimal configuration for an UDF execution.

6.2.2. Stage 2 - degree of parallelism

The right degree of parallelism is to assure the user-defined performance metrics and it can be achieved by tuning certain performance parameters depending on the distributed environment and programming paradigm. For example, performance tuning of a MapReduce UDF to be executed in Hadoop is dependent on 190 configurations. Optimal settings for these parameters depend upon a workflow, data characteristics, and distributed machine configurations. However, a fraction of parameters play an important role in achieving the performance optimization and a lack of knowledge of these parameters is mostly the cause of performance problems [40].

The parameters that seem to be *critical* to the optimal execution of UDFs in a distributed framework include:

- The *number of partitions/shards*: represents appropriate number of partitions (neither too few nor too many) for the MapReduce and Spark jobs executing on top of the Hadoop framework.
- *Machine configurations*: represents appropriate processing power of a distributed machine along with optimal configuration of the critical parameters.
- *Parallel processing architecture*: include a degree of parallelism, partitions shuffling scheme (for Spark jobs), the number of Mapper and Reducer tasks (for MapReduce jobs).

6.2.3. Stage 3 - optimal code generation

The critical parameters (described in Section 6.2.2), are of vital importance in order to satisfy the user-defined performance metrics. The Cost Model uses obligatory user-defined performance metrics: 1) maximum execution time for an ETL workflow (T) and 2) maximum monetary cost for an ETL workflow to be executed in the distributed environment (B), as an input from the ETL developer. Optional parameters include:

1) the size of a data set (R) in terms of the number of rows and 2) configuration of a distributed machine (M).

In order to find out the right degree of parallelism, the proposed cost model will be used as follows (also shown in Figure 6.3).

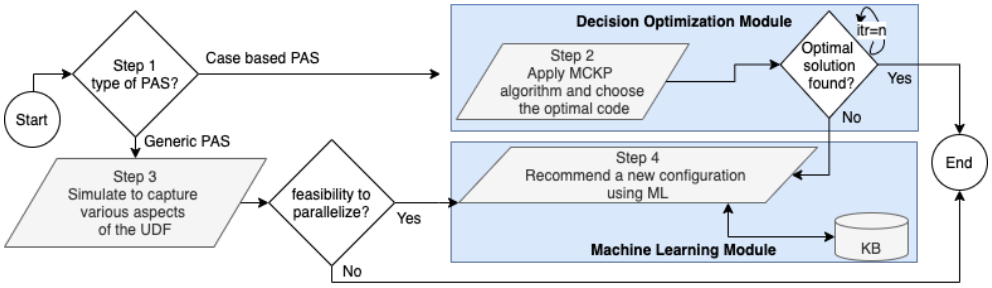

Figure 6.3: The processing workflow of the cost model

- *Step 1*: checks for a user code input either as the *Generic PASs* or *Case-based PASs*, (c.f., Section 4.1). If the ETL developer selects the *Generic PASs*, the cost model executes Step 2, otherwise it executes Step 4.

- *Step 2*: if the ETL developer selects the *Case-based PASs*, where a user selects a use case, the cost model tries to find the (sub-)optimal solution using decision optimization (c.f., Section 6.3). If the optimal configuration solution is found, the solution is handed over to the distributed machine to be executed in parallel, otherwise the best possible solution with all the relevant statistics and machine configuration is sent to Step 4 as an input, in order to generate a (sub-)optimal solution.

- *Step 3*: if the ETL developer selects the *Generic PASs*, then an UDF provided by the ETL developer is first executed in the *Simulator* to collect run-time execution statistics, e.g., execution time, estimated monetary cost, the number of rows processed per second, the number of bytes processed per second, the size of data, the current configuration of a distributed machine as well as memory, CPU, and IO usage characteristics.

  At this point, the cost model is only interested in the execution time and estimated monetary cost. If both are within the user-defined performance metrics constraints, then the processing is stopped, otherwise the processing continues to Step 4.

- *Step 4*: the output from the *Simulator* i.e., Step 3 will be used as an input to this step. The details to generate optimal code/machine configuration for *Generic PASs* are discussed in Section 6.4.

6.3. Optimal Code Generation for Case-based PASs

In order to explain the optimal code generation for case-based PASs and to evaluate the Cost Model, we leverage a Set-similarity joins using MapReduce (SSJ-MR) [103] use case. It is one of the parallel approaches using the MapReduce framework to detect similar records based on string similarity. SSJ-MR uses three components, namely: join-attribute, set-similarity function, and a similarity threshold.

Figure 6.4 illustrates the workflow for SSJ-MR approach. The workflow is divided into three stages and each stage processes data using MapReduce. Also each stage can be solved by either of the two approaches, e.g., TO1 or TO2 - for the Token Ordering stage, PG1 or PG2 - for the Pair Generation stage, and RJ1 or RJ2 - for the Record Join stage.

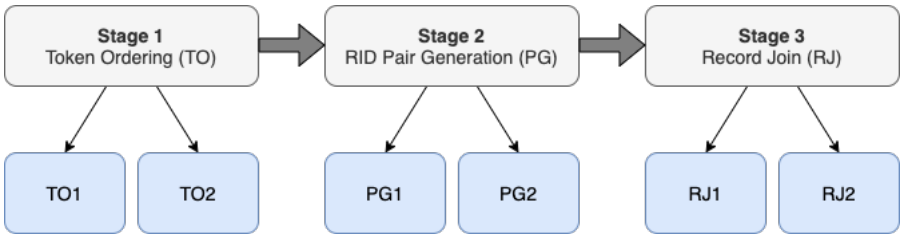

Figure 6.4: Set-similarity join workflow

Stage 1 - Token Ordering (TO). It computes data statistics using MapReduce in order to generate partitioning keys called *signatures*, which are used in Stage 2. The signatures are generated by tokenizing the incoming record into a word set. For example, the record string "Parallelizing the custom code" is tokenzied into word set ["Parallelizing", "the", "custom", "code"]. Each element in the word set is a signature, which is used as a partitioning key instead of using actual join attribute value, because partitioning records using an entire string for partitioning (e.g., hash-based partitioning) is a difficult task.

Stage 2 - RID pair Generation (PG). It extracts a record ID (RID) and join-attribute value for each record, computes the similarity of the join-attribute values, and propagates the RID pair of similar records to the next stage. The similarity is computed in the Reduce phase of MapReduce, using *signatures* from the previous stage as partitioning keys.

Stage 3 - Record Join (RJ). It uses the RID pair of similar records from the previous stage and generates actual pairs of joined records.

The explanation of the use case clearly indicates that SSJ-MR is a computationally-intensive process divided into multiple stages (considered as multiple UDFs).

6.3.1. Use case for running example

In the example, we use Pentaho Data Integration (PDI)[1] to implement the SSJ-MR algorithm as an ETL workflow to efficiently detect similar records in a large amount of datasets. The SSJ-MR algorithm is divided into three stages and executed in Amazon EMR Cluster[2] - a Hadoop [19] managed framework that makes it feasible to process vast amounts of data across dynamically scalable Amazon EC2 instances[3]. Each stage of SSJ-MR may be executed in a differently configured Amazon EMR cluster, if proposed by the cost model. Finally, data are stored in a data warehouse.

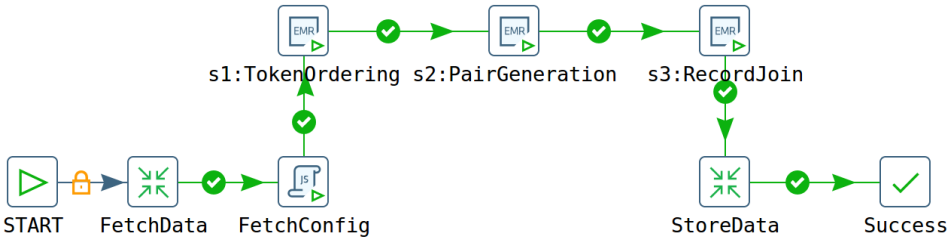

Figure 6.5: The running example scenario

Our example ETL workflow is shown in Figure 6.5. The first step - *START* indicates the start of the ETL workflow in PDI. The next step - *FetchData* fetches data from a data store. *FetchConfig* fetches the configuration for each of the Amazon EMR jobs *s1:TokenOrdering*, *s2:PairGeneration*, and *s3:RecordJoin* (generated by the cost model) to be executed in parallel in the Amazon EMR cluster. Finally, *StoreData* step stores the resulted data set a DW.

6.3.2. Optimal code generation

For the *Case-based PASs* as shown in Figure 6.6, where a user selects a use case, e.g., SSJ-MR there exists m different stages, and the **UDFs Component** may generate n parallel variants for each stage. Therefore, there are n^m possible combination of variants, which correspond to an NP-hard problem and can be mapped to Multiple Choice Knapsack Problem (MCKP) [43] as a special case of our problem. The MCKP is defined as follows:

Given m classes N_1 ... N_m of items to pack in a knapsack of capacity c, where each item $j \in N_i$, i = 1,2,....,m, has profit p_{ij} and weight w_{ij}, the problem is to choose

[1] https://github.com/pentaho/pentaho-kettle
[2] https://aws.amazon.com/emr/
[3] https://aws.amazon.com/ec2/

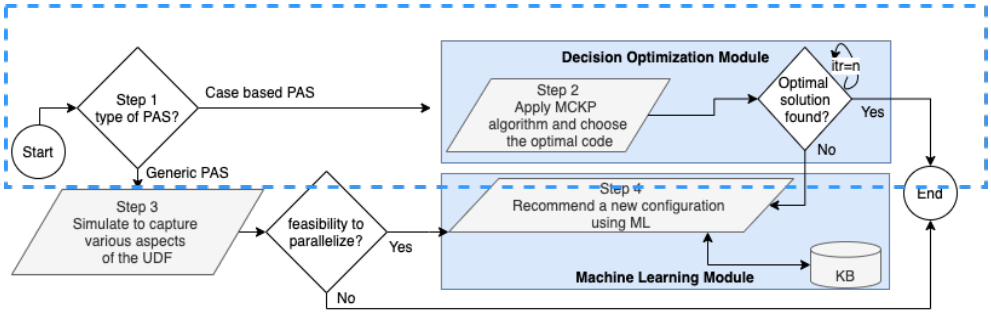

Figure 6.6: Case-based PASs scenario

exactly one item from each class such that the sum of profits is maximized while the sum of weights does not exceed capacity c.

In order to show MCKP as a special case of the running example, we introduce the following terms. Let *Minimize(Z)* be an optimal solution containing exactly one program from each stage with minimum execution time, while remaining within budget B. Note that we want to calculate the execution time T_{ij} of a program variant N_j from each stage m, such that the total cost C_{ij} of the entire ETL workflow is $\leq B$.

According to the aforementioned definition of MCKP, we can map:

- m classes in MCKP definition to m stages in the running example,
- c weight constraint to B budget constraint,
- w_{ij} cost of item for class to c_{ij} cost of variant j at stage i,
- p_{ij} profit of each item to T_{ij} execution time of each variant at each stage.

Then, we can find out the optimal solution as follows:

$$\text{Minimize(Z)} \qquad \sum_{i=1}^{m} \sum\nolimits_{j \in N_{\mathrm{i}}} T_{\mathrm{i,j}} \cdot x_{\mathrm{i,j}}$$

$$\text{subject to}$$

$$\sum_{i=1}^{m} \sum\nolimits_{j \in N_{\mathrm{i}}} C_{\mathrm{ij}} \cdot x_{\mathrm{ij}} \leq B,$$

$$\sum\nolimits_{j \in N_{\mathrm{i}}} x_{\mathrm{ij}} = 1, i = 1, \ldots, m,$$

$$x_{\mathrm{ij}} \in \{0,1\}, j \in N_i, i = 1, \ldots, m.$$

If the optimal configuration solution is found by the MCKP step, the solution is handed over to the distributed machine to be executed in parallel, otherwise the best

possible solution with all the relevant statistics and machine configuration is sent to Step 4 as an input, in order to generate a (sub-)optimal solution.

6.3.3. Experimental evaluations

As an experimental evaluation of our approach to select an optimal code variant for the Case-based PASs, we used the running example (c.f., Section 6.3.1), where each stage has two variants. For stage 1, i.e., Token Ordering, the first variant is called *Basic Token Ordering (BTO)* and the second one is *One Phase Token Ordering (OPTO)*. For stage 2, i.e., RID Pair Generation, the first variant is called *Basic Kernel (BK)* and the second one is *Indexed Kernel (PK)*. For stage 3, i.e., Record Join, the first variant is *Basic Record Join (BRJ)* and the second variant is *One Phase Record Join (OPRJ)*.

In Table 6.1, column *Stage* stores the number of the stage, cf. Sections 6.2.1, 6.2.2, and 6.2.3. Column *Algorithm* stores the aforementioned algorithm variants of each stage. Column *#Nodes [exec cost/h]* represents the per hour execution cost associated to a n-node micro-cluster configuration (Amazon Web Service applies per hour billing cycle). For example, column *2 [0.4$/h]* represents execution cost per hour for a 2-nodes micro-cluster, *4 [0.8$/h]* represents execution cost for a 4-nodes micro-cluster, etc.

The costs are estimated based on a machine type mentioned in [103] and near equivalent configuration available for Amazon EC2 instances[4]. We used the Linux machine on t3.2xlarge, 4 vCPUs, and 16 GB RAM, at a main cost of $0.164/hour plus $0.036/hour as a buffer cost, which eventually results in $0.2/hour per node.

Then, each cell under *2 [0.4$/h]*, *4 [0.8$/h]*, *8 [1.6$/h]*, and *10 [2.0$/h]* stores execution time in seconds of a given algorithm in a given micro-cluster in a given stage. Thus, Table 6.1 includes 24 variants of execution times. The execution times of each variant of each stage are taken from the already carried out evaluation in [103].

In order to evaluate the correctness of our MCKP-based cost model, we mapped our problem on to the *Linear Integer Programming Model*. The cost model is implemented in Java, which utilizes the *lp_solve*[5] library to generate the optimized (i.e., minimum execution time with respect to the allocated budget) combination of available machine configurations to execute each stage of our running example. lp_solve is a mixed integer linear programming (MILP) solver based on the revised Simplex method [30] and the Branch-and-bound method [58] for integers. The implementation of the cost model is accessible via our online git repository[6].

[4] https://calculator.s3.amazonaws.com/index.html
[5] http://lpsolve.sourceforge.net/
[6] `https://github.com/fawadali/MCKPCostModel`

Table 6.1: Execution time in seconds of each stage for self-joining the DBLP dataset on different cluster sizes

Stage	Algorithm	#Nodes [exec cost/h]			
		2 [0.4$/h]	4 [0.8$/h]	8 [1.6$/h]	10 [2.0$/h]
1	BTO	191.98	125.51	91.85	84.02
	OPTO	175.39	115.36	94.82	92.80
2	BK	753.39	371.08	198.70	164.57
	PK	682.51	330.47	178.88	145.01
3	BRJ	255.35	162.53	107.28	101.54
	OPRJ	97.11	74.32	58.35	58.11

The shaded cells in Table 6.1 represent execution costs of the selected algorithms at each stage that are minimal (optimal) for a given maximum budget of $1.6. Thus, for the first stage OPTO is suggested to be executed on 2 nodes, for the second stage PK is selected to be executed on 4 nodes, and for the third stage OPRJ is selected to be executed 2 nodes.

The blacked cells and the shaded one with value 97.11 represent execution costs of the selected algorithms at each stage that are minimal (optimal) for a given maximum budget of $4.0. That is, for the first stage BTO is suggested to be executed on 8 nodes, for the second stage PK is selected to be executed on 10 nodes, and for the third stage OPRJ is selected to be executed on 2 nodes.

The results show that the cost model provides the best possible configuration for a set of ETL tasks to be executed in a cloud based pay-as-you-go environment. The average execution time of the cost model to choose the best possible configuration was **2ms**. The cost model was executed on a machine configuration of 2.6 GHz 6-Core Intel Core i7, 16GB RAM.

In the future, we will conduct experiments with different use cases in order to fine-tune the MCKP algorithm.

6.4. Optimal Code Generation for Generic PASs

The *Cost Model* requires obligatory user-defined performance metrics: 1) maximum execution time for an ETL workflow (T) and 2) maximum monetary budget for an ETL workflow to be executed in the distributed environment (B), as an input from the ETL developer. Optional parameters include: the size of a data set (R) in terms of the number of rows and a configuration of distributed machines (M).

When the ETL developer selects the *Generic PASs* as shown in Figure 6.7, then an UDF provided by the ETL developer is first executed in a simulated environment to collect run-time execution statistics.

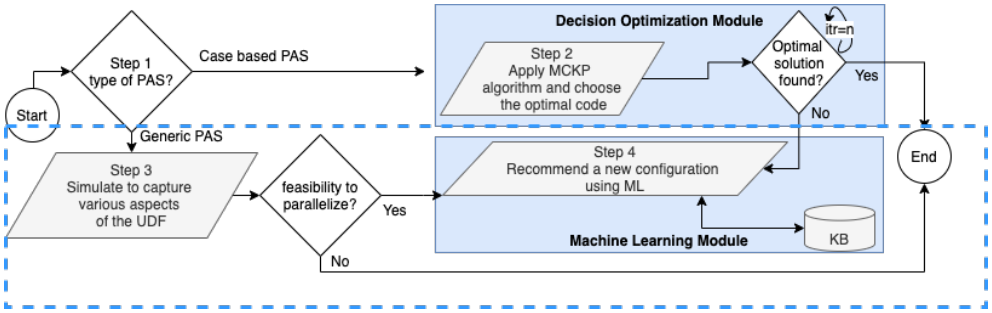

Figure 6.7: Generic PASs scenario

At this point, the *Cost Model* is only interested in the execution time and estimated monetary cost. If both are within the user-defined performance metrics constraints, the processing is stopped. Otherwise, it continues to *Step 4*. This step generates the best possible machine configuration for an UDF based on *Machine Learning* techniques using input from *Step 3* and an integrated *Knowledge Base* (KB). The KB is built on past executions of the UDFs in a distributed environment within the *ETL Framework* and is updated after every execution. This KB also serves for the training purposes of the machine learning model. The structure of KB is as follows:

ExecutionHistory(executionId, executionTime, rowsProcessed, bytesProcessed, sizeOfData, rowsRead, rowsWrite, cpuUsage, machineId)

Machine(machineId, memory, numOfCPUs, operatingSystem, provider, price, costType, otherCosts)

Attributes in table *ExecutionHistory* are the candidates for the feature set for machine learning models, except *machineId*, which serves as a class label.

Predicting *machineId*, i.e., a class, is essentially a classification problem. The configuration for the predicted value of *machineId* then can be obtained from the *Machine* table. A sample response from the machine learning model is shown below:

```
"machineId":"m5.xlarge",
"memory":16,
"numOfCPUs":6,
"operatingSystem":"Linux",
"provider":"AWS",
"price":0.192,
"costType":hourly,
"otherCosts":10
```

As of now, the implementation of the machine learning model to predict the best possible configuration is one of the future works mainly because of the unavailability of sample data. To build such a model, there is a need to have some reasonable number of samples per *machineId* (class), which are not readily available at the moment.

6.5. Extending the Cost Model for a Machine Learning Pipeline

In this section, we extend the use of the *UDFs Component* not only to the ETL developers or the data engineers but to the data scientists as well to create an end-to-end Machine Learning Pipeline (MLP). The MLP may comprise multiple steps, such as data pre-processing, feature selection, and training & testing of a given ML algorithm. The data scientist may select a MLP from the *UDFs Component* as a *Generic PASs*, which may consist of a number of stages (data pre-processing, feature selection, ML model), as shown in Figure 6.8.

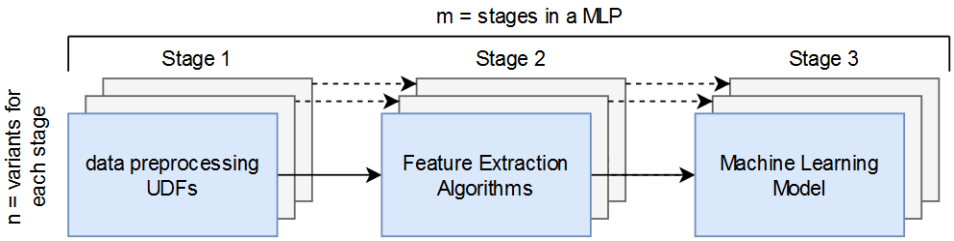

Figure 6.8: An end-to-end machine learning pipeline

Besides providing the obligatory user-defined performance metrics i.e., 1) maximum execution time for an MLP (T) and 2) maximum monetary execution cost for the MLP(B), the data scientist will need to provide the size of a data set (S) and the performance metric for the ML model i.e., accuracy, precision, or recall (P). These parameters are used by the *Decision Optimization Module* to provide the best possible ML algorithm for the said MLP.

6.5.1. Optimal selection of a machine learning model

The problem of finding an optimal Machine Learning model is modeled as the Multiple Choice Knapsack Problem that can be formulated as follows.

Let *maximize(Z)* be an optimal solution containing the best possible MLP, based on P (i.e., accuracy, precision, or recall), such that: execution time $E \leq T$ and execution cost $C \leq B$. Note that we want to provide the maximum value of accuracy, precision, or recall P_{ij} of an UDF or a ML model variant N_j from each stage m, such that the total cost $C_{ij} \leq B$ and execution time of the entire MLP $E_{ij} \leq T$.

Our problem is mapped to the MCKP as follows: 1) n classes in MCKP definition **to** m stages, 2) a weight constraint c **to**: B - budget constraint and E - execution time constraint, 3) a cost of item for a class w_{ij} **to** C_{ij} - a cost of variant j at stage i

and E_{ij} - execution time of variant j at stage i, 4) a profit of each item p_i **to** P_i - a maximum performance metric. Then, we can find out the optimal solution as follows:

$$\begin{aligned}
&\text{Maximize(Z):} && \sum_{i=1}^{m} \sum_{j \in N_{\mathrm{i}}} P_{\mathrm{i,j}} \cdot x_{\mathrm{i,j}} \\
&\text{subject to:} && \sum_{i=1}^{m} \sum_{j \in N_{\mathrm{i}}} C_{\mathrm{ij}} \cdot x_{\mathrm{ij}} \leq B \\
& && \sum_{i=1}^{m} \sum_{j \in N_{\mathrm{i}}} E_{\mathrm{ij}} \cdot x_{\mathrm{ij}} \leq T \\
& && \sum_{j \in N_{\mathrm{i}}} x_{\mathrm{ij}} = 1; i = 1, \ldots, m;\ x_{\mathrm{ij}} \in \{0,1\}; j \in N_i; i = 1, \ldots, m
\end{aligned}$$

6.5.2. Experimental evaluations

In order to evaluate the correctness of our MCKP-based cost model, we mapped our problem on to the *Linear Integer Programming Model*. The cost model is implemented in Python using Google OR-Tools[7]. The implementation of the cost model is accessible via our online git repository[8].

As a preliminary evaluation of our approach to generate the best possible ML model, we used the experimental test-bed and results on the sentiment analysis described in [45] that include: 1) three different sentiment analysis ML models, namely: *Bernoulli Naïve Bayes* (BNB), *Multinomial Naïve Bayes* (MNB), and *Support Vector Machine* (SVM), where each model has two variants, 2) accuracy of each ML model, 3) execution time of each variant of the model in seconds. To generate preliminary experimental results, we built an execution environment based on machines indicated in [103] (2.6 GHz 6-Core Intel Core i7, 16GB RAM) and near equivalent configuration available in Amazon EC2 instances[9].

Table 6.2: Experimental results on finding the best possible ML Model. The shaded row (MNB-1) indicates the model predicted by our extended cost model, with execution time of 0.05s and an accuracy of 81.34%

ML Model	Execution Time [s]	Accuracy [%]	Cost/hour [$]
BNB-1	0.24	75.21	1.6
BNB-2	0.61	65.18	2.0
MNB-1	0.05	81.34	1.6
MNB-2	0.11	72.14	2.0
SVM-1	4.22	77.16	1.6
SVM-2	12.95	69.95	2.0

Table 6.2 presents the obtained results, which we used as an input to our extended

[7] https://developers.google.com/optimization/mip/mip
[8] `https://github.com/fawadali/MCKPCostModel/blob/master/ML-CostModel/`
[9] https://calculator.s3.amazonaws.com/index.html

cost model. Column *ML Model* represents the variants of the aforementioned ML models i.e., *{BNB-1, BNB-2, MNB-1, MNB-2, SVM-1, SVM-2}*, *Execution Time [s]* is the execution time of each variant in seconds, *Accuracy [%]* represents the accuracy of the ML algorithm, and *Cost/hour [$]* represents execution cost in $ per hour, for each sentiment analysis ML model. The average execution time of the cost model to predict the best possible ML was **4ms**. The shaded row in Table 6.2 indicates the obtained result from the cost model as the best possible ML model for a given input performance metric parameters:

$$\text{Execution Time} \leq 1.0\text{s; Cost per hour} \leq \$2.0$$

At this point in time, the experiments were carried out on part of the machine learning pipeline dedicated to finding an optimal ML model. In the future, we will conduct experiments with different use cases and will also include the entire MPL ,i.e., data pre-processing, feature selection, and training & testing of a given ML algorithm, in order to fine-tune the MCKP algorithm.

6.6. Discussion on Experimental Evaluations

MCKP is an NP-complete problem, however, in reality large instances can be optimally solved in a short period of time [31]. Our problem is much smaller catering at most two constraints i.e., execution time and execution cost.

In order to prove the efficiency of our cost model, we executed the *lp_solve* based cost model (c.f. Section 6.3.3) and *Google OR-Tools* based cost model (c.f. Section 6.5.2) on a range of test data sets to measure the execution time of the cost models. We used the same machine configuration for this experiment as mentioned in Section 6.5.2, i.e., 2.6 GHz 6-Core Intel Core i7, 16GB RAM.

Figure 6.9 shows the time in milliseconds the *lp_solve* based cost model took to find the optimal solution from the possible number of variants starting from 24 to more than 1000. For the number of variants under 400, the execution time of the cost model is less than 100ms. As the number of variants increases to more than 1000, the execution time of the cost model becomes almost 1 second. However, for our problem we expect the number of variants to be less than 1000.

Figure 6.10 shows the time in milliseconds the *Google OR-Tools* based cost model took to find the optimal solution from the possible number of variants starting from six to more than 1000. Even if the number of variants increases to more than 1000, the execution time of the *Google OR-Tools* based cost model remains under 20ms.

Based on the evaluation and our problem size, we can confidently conclude that the proposed cost model is capable to provide the optimal solution in an acceptable time (in the case of our experiments - in a fraction of a second).

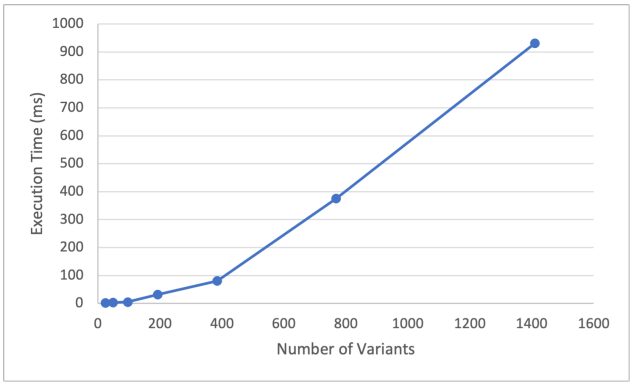

Figure 6.9: Evaluation of efficiency of the *lp_solve* based Cost Model

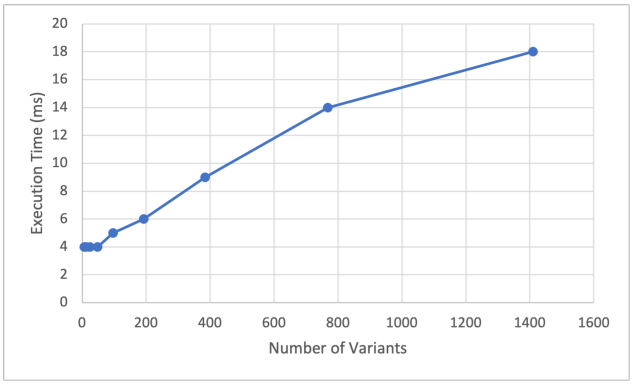

Figure 6.10: Evaluation of efficiency of the *Google OR-Tools* based Cost Model

6.7. Conclusions

In this chapter, we presented our contribution towards a cost model to determine the best possible configuration for an UDF generated either via the *Generic PASs* or the *Case-based PASs*. In particular, this chapter **contributes**: 1) the cost model for optimizing execution of an UDF (as a black-box), 2) the method for selecting a (sub-)optimal configuration of a parallel execution environment for an UDF, and 3) possibility to enable data scientists to choose the best possible machine learning model based on user-defined performance metrics. The proposed method uses simulation, recommendation, and prediction algorithms to generate the best possible configuration for an UDF generated by means of the *Generic PASs*. For the *Case-based PASs*, the cost model uses Multiple Choice Knapsack Problem (MCKP) along with the recommendation and prediction algorithms (if required) to generate a (sub-)optimal

configuration of a parallel run-time environment for an UDF.

Furthermore, We evaluated experimentally our Decision Optimization module with the final goal to find the best possible ML model. The problem of finding the model was mapped into the MCKP and implemented in Python using Google OR-tools.

Chapter 7
Conclusions and Future Directions

7.1. Conclusions

Data from different data sources require care for deprived quality, oscillating from basic spelling mistakes, missing or varying values, toward contradictory or repetitive data. Data warehouse and data lake architectures were developed to handle heterogeneous data i.e., data coming from different sources like relational data from operational databases and data from line of business applications, and non-relational data like mobile apps, IoT devices, and social media. In these architectures, data integration tasks in ETL workflows are executed through SQL directives, predefined modules, or UDFs, implemented in several programming languages.

Since the basis of the thesis was to address the problems and challenges in the field of ETL workflows, an extensive study was carried out on the state-of-the-art and current trends in ETL workflows, specifically addressing the optimization of ETL workflows. Workflows implemented in business architectures usually require hundreds of tasks. Designing a fault free and effective ETL workflows is a costly process. For this reason, a large number of approaches have been proposed by the research community. These approaches address the conceptual, logical, physical implementation, and optimization of the ETL workflows.

For the conceptual design of an ETL workflow, one of the primary methods is to design the workflow as a graph. At the abstract level, the ETL developer must physically develop the ETL transformations and inter-feature mappings. Other approaches for the conceptual design of an ETL workflow involve semantic web ontology, UML notation, and BPMN. All these methods suggest different approaches to the precise and efficient design of an ETL conceptual model. However, there does not exist any industry standard for validating an ETL design. Then, on a logical side, there are methods using graphs that complement a logical design's generic actions. Furthermore, there exists XML and BPEL based approaches to translate a logical model into its corresponding physical implementation. However, if a template has not already

been given for a specific ETL task, the ETL developer must manually translate a logical model into its corresponding physical implementation. To translate BPMN-based conceptual model into its BPEL-based equivalent physical implementation, the ETL developer is required to have a prior knowledge of BPEL and tools that support BPEL-based ETL workflows. A few approaches have been proposed in this field so far and, there is a need for an automatic or partially automatic system to translate a logical model into its physical execution with minimal or no human intervention, in the spirit of [75].

For the optimization of ETL workflows, several approaches ranging from state-space search to scheduling strategies and the use of reusable patterns for the optimization of ETL workflows were proposed. Other techniques to achieve better execution efficiency were largely based on introducing parallelism in an ETL workflow. For example, Hadoop and MapReduce based solutions focusing on data flow parallelism require code templates to allow ETL developers to write parallelizable code.

Based on our study on the state-of-the-art and current trends in the ETL workflow design, implementation, and optimization, we identified the following key issues to address in this thesis.

- Throughout the modeling, design, implementation, and optimization process of an ETL workflow, most of the approaches involve ETL developers to provide extensive amount of input to the ETL framework, which is very much error prone and inefficient. Specifically, to optimize an ETL workflow, it is expected from the ETL developers to be highly technical in programming as well as cautious in order to understand the quality metrics and their impact on the overall performance of an ETL workflow.

- The rise of big data has exposed the limited expressive power of ETL operators or tasks provided by the ETL tools to deal with the complexity of big data. Therefore, UDFs are normally used as ETL tasks in an ETL workflow for complex transformations. However, methods discussed in the state-of-the-art did not consider the optimization of UDFs in a comprehensive manner.

- Currently, there does not exist any ETL framework that autonomously monitors an ETL workflow for performance issues and suggests the ETL developer the ways to improve its performance.

In this thesis, we addressed and contributed to the aforementioned issues as follows:

- In Chapters 4 & 5, we proposed a next generation ETL framework that enables the ETL developers to exploit parallelization for UDFs in an ETL workflow. We provided a configurable-parallelizable UDF generator to facilitate the ETL

developer to implement parallelizable UDFs without concerning about the implementation details for parallelizing and optimizing the code. To parallelize UDFs in an ETL workflow, we addressed the UDF generator portion and Cost Model module of the proposed ETL framework. In the framework, we have implemented the mechanism that creates multiple variants, i.e., multiple parallel configurations, and then select the optimal configuration based on cost and computational efficiency requirements for the distributed environment.

We proved with experiments (c.f. Section 5.6) that enabling the ETL developers to assist in writing parallelizable code can help in avoiding 50-65% of the human effort in writing efficient parallelizable UDFs. Furthermore, we carried out experiments to prove (c.f. Section 5.6) that: 1) the non computing-intensive code normally does not effect the overall performance of an ETL workflow whether it is executed in a distributed or a non-distributed environment and 2) the computing-intensive tasks may become a bottleneck in an ETL workflow and must be optimized, because even a small change in the distributed factor can make a big difference in improving the execution performance of an overall ETL workflow. Thus, suggesting that it is important to first check: 1) if an UDF is parallelizable and 2) if it is parallelizable, will it profit from parallel processing or not because there is always an extra cost overhead to execute programs in a distributed environment.

- In Chapter 6, we implemented a cost model within the proposed ETL framework. Once the ETL developer writes or chooses a desired UDF, the cost model takes over the responsibility to efficiently execute the UDF based on the execution and monetary constraints provided by the ETL developer.

  Our experiments showed that our cost model provides the best possible configuration for a set of ETL tasks (implemented as UDFs) to be executed in a distributed environment (c.f. Section 6.3.3). We further showed that the proposed cost model is capable of providing the solution in a fraction of a second (c.f. Section 6.6).

7.2. Future Directions

The proposed framework is one step towards the fully autonomous ETL framework to address the challenges posed by big data. As mentioned in Section 6.1, the ETL framework consists of four modules namely: 1) the UDFs component, 2) the Cost Model, 3) the Recommender, and 4) the Monitoring Agent. In the following, we provide future directions for each module separately.

- The **UDFs Component:** currently, this module provides MapReduce based parallel algorithmic skeletons for Generic PASs and can easily be extended to

Spark or Python based parallel algorithmic skeletons. For the Case-based PASs, the library of use cases can be extended to more broader range of use cases ranging from financial services, oil and gas to health care.

- The **Cost Model:** we have devised the Cost Model as a MCKP problem implemented as an integer programming model. In the future, the cost model library can be extended with alternate implementation of MCKP e.g., using dynamic programming.

- The **Recommender:** this module is specifically used in case of generic PASs. The cost model at this point in time is capable of efficiently executing an ETL workflow based on use-case specific UDFs without the use of a Recommender. However, in order to address the generic UDFs, a machine learning based recommender is yet to be built. To this end, there is a need to have some reasonable amount of training data, which is not readily available at the moment. This is also addressed in detail in Section 6.4.

- The **Monitoring Agent:** although, this is a standard component of any ETL engine, the implementation of this module is also one of the future works. The Monitoring Agent will be able to autonomously monitor an ETL workflow for performance issues and suggests the ETL developer the ways to improve its performance with the help of the Recommender module. The monitoring agent will collect several metrics from the execution of the ETL workflows e.g., the number of input rows, the number of output rows, the execution time of each step, the number of rows processed per second, the execution time of each ETL task w.r.t rows processed per second, the execution time of the entire ETL workflow w.r.t rows processed per second, the memory consumption by each ETL task. Such statistics will be stored in the knowledge base to be used by the Recommender to make recommendations to the ETL developer to improve the execution performance of an ETL workflow.

Bibliography

[1] Apache Spark - lightning-fast cluster computing. `http://spark.apache.org/`. (Accessed on 02/22/2021).

[2] How to achieve flexible, cost-effective scalability and performance through pushdown processing. `https://www.informatica.com/downloads/pushdown_wp_6650_web.pdf`, 2007. (Accessed on 02/22/2021).

[3] 10 open source ETL tools. data science central. `www.datasciencecentral.com/profiles/blogs/10-open-source-etl-tools`, 2015. (Accessed 02/20/2021).

[4] Gartner magic quadrant for data integration tools. `https://www.talend.com/lp/gartner-2020-dimq/`, (2020). (Accessed 02/20/2021).

[5] Alexandrov, A., Heimel, M., Markl, V., Battré, D., Hueske, F., Nijkamp, E., Ewen, S., Kao, O., and Warneke, D. Massively parallel data analysis with pacts on nephele. *VLDB Endowment 3* (2010), 1625–1628.

[6] Ali, S. M. F., Mey, J., and Thiele, M. Parallelizing user - defined functions in the ETL workflow using orchestration style sheets. *International Journal of Applied Mathematics and Computer Science (AMCS) 29* (2019), 69–79.

[7] Andzic, J., Fiore, V., and Sisto, L. Extraction, transformation, and loading processes. In *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. Idea Group Incorporation, 2007, pp. 847–865.

[8] Assmann, U. Invasive software composition. In *Invasive Software Composition*. Springer, 2003, pp. 107–145.

[9] Awad, M. M., Abdullah, M. S., and Ali, A. B. M. Extending ETL framework using service oriented architecture. *Procedia Computer Science 3* (2011), 110–114.

[10] Awiti, J., Vaisman, A., and Zimányi, E. From conceptual to logical ETL design using BPMN and relational algebra. In *Proceedings of the International Conference on Big Data Analytics and Knowledge Discovery (DaWaK)* (2019), pp. 299–309.

[11] Awiti, J., and Zimányi, E. An XML interchange format for ETL models. In *Proceedings of the Advances in Databases and Information Systems (ADBIS)* (2019), pp. 427–439.

[12] Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., and Warneke, D. Nephele/pacts: A programming model and execution framework for web-scale analytical processing. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC)* (2010), pp. 119–130.

[13] Beneventano, D., Bergamaschi, S., Guerra, F., and Vincini, M. Synthesizing an integrated ontology. *IEEE Internet Computing 7* (2003), 42–51.

[14] Bergamaschi, S., Guerra, F., Orsini, M., Sartori, C., and Vincini, M. A semantic approach to ETL technologies. *Data & Knowledge Engineering 70* (2011), 717–731.

[15] Bergamaschi, S., Sartori, C., Guerra, F., and Orsini, M. Extracting relevant attribute values for improved search. *IEEE Internet Computing 11* (2007), 26–35.

[16] Binnig, C., May, N., and Mindnich, T. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In *Proceedings of the Datenbanksysteme für Business, Technologie und Web (BTW)* (2013), pp. 363–382.

[17] Bodziony, M., Roszyk, S., and Wrembel, R. On evaluating performance of balanced optimization of ETL processes for streaming data sources. In *Proceedings of the International Workshop on Design, Optimization,Languages and Analytical Processing of Big Data (DOLAP)* (2020), pp. 74–78.

[18] Bogatu, A., Fernandes, A. A. A., Paton, N. W., and Konstantinou, N. Dataset discovery in data lakes. In *Proceedings of the International Conference on Data Engineering (ICDE)* (2020), pp. 709–720.

[19] Borthakur, D. The hadoop distributed file system: Architecture and design. *Hadoop Project Website 11* (2007), 21.

[20] Chaiken, R., Jenkins, B., Larson, P.-Å., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. Scope: easy and efficient parallel processing of massive data sets. *VLDB Endowment 1* (2008), 1265–1276.

[21] Chakrabarti, S., Demmel, J., and Yelick, K. Modeling the benefits of mixed data and task parallelism. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (1995), pp. 74–83.

[22] Cloudera. Example: Sentiment analysis using MapReduce custom counters. `https://docs.cloudera.com/documentation/other/tutorial/CDH5/topics/ht_example_4_sentiment_analysis.html`. (Accessed on 03/18/2019).

[23] Dagum, L., and Menon, R. OpenMP: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering 5* (1998), 46–55.

[24] Dean, J., and Ghemawat, S. MapReduce: simplified data processing on large clusters. *Communications of the ACM 51* (2008), 107–113.

[25] Duggan, J., Elmore, A. J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., and Zdonik, S. The BigDAWG Polystore System. *SIGMOD Record 44* (2015), 11–16.

[26] Ekman, T., and Hedin, G. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming 69* (2007), 14–26.

[27] El Akkaoui, Z., Vaisman, A. A., and Zimányi, E. A quality-based ETL design evaluation framework. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)* (2019), pp. 249–257.

[28] El Akkaoui, Z., and Zimányi, E. Defining ETL worfklows using BPMN and BPEL. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2009), pp. 41–48.

[29] El Akkaoui, Z., Zimànyi, E., Mazón, J.-N., and Trujillo, J. A model-driven framework for ETL process development. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2011), pp. 45–52.

[30] Evans, J. P., and Steuer, R. E. A revised simplex method for linear multiple objective programs. *Mathematical Programming 5* (1973), 54–72.

[31] Furini, F., Monaci, M., and Traversi, E. Exact approaches for the knapsack problem with setups. *Computers & Operations Research 90* (2018), 208–220.

[32] Gantz, J. F. The expanding digital universe: A forecast of worldwide information growth through 2010. International Data Corporation (IDC) White Paper.

[33] Gartner. Magic Quadrant for Data Integration Tools. `https://blogs.bmc.com/gartner-magic-quadrant-data-integration-tools/?print=pdf`, 2020. (Accessed on 03/31/2021).

[34] Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., and Jacobsen, H.-A. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD)* (2013), pp. 1197–1208.

[35] Gour, V., Sarangdevot, S., Tanwar, G. S., and Sharma, A. Improve performance of extract, transform and load (etl) in data warehouse. *International Journal on Computer Science and Engineering 2* (2010), 786–789.

[36] Grosse, P., Lehner, W., and May, N. Advanced Analytics with the SAP HANA Database. In *Proceedings of the International Conference on DAta management Technologies and Application (DATA)* (2013), pp. 61–71.

[37] Grosse, P., May, N., and Lehner, W. A study of partitioning and parallel UDF execution with the SAP HANA database. In *Proceedings of the ACM International Conference on Scientific and Statistical Database Management (SSDBM)* (2014), p. 36.

[38] Halasipuram, R., Deshpande, P. M., and Padmanabhan, S. Determining essential statistics for cost based optimization of an ETL workflow. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2014), pp. 307–318.

[39] Hedin, G. Reference attributed grammars. *Informatica (Slovenia) 24* (2000), 301–317.

[40] Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F. B., and Babu, S. Starfish: A self-tuning system for big data analytics. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2011), vol. 11, pp. 261–272.

[41] Hueske, F., Peters, M., Krettek, A., Ringwald, M., Tzoumas, K., Markl, V., and Freytag, J.-C. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *Proceedings of the International Conference Data Engineering (ICDE)* (2013), pp. 1292–1295.

[42] Hueske, F., Peters, M., Sax, M. J., Rheinländer, A., Bergmann, R., Krettek, A., and Tzoumas, K. Opening the black boxes in data flow optimization. *VLDB Endowment 5* (2012), 1256–1267.

[43] Ibaraki, T., Hasegawa, T., Teranaka, K., and Iwase, J. The multiple choice knapsack problem. *Journal of the Operations Research Society of Japan 21* (1978), 59–93.

[44] Iosup, A., Ostermann, S., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. H. J. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Transactions on Parallel and Distributed Systems 22* (2011), 931–945.

[45] Ismail, H., Harous, S., and Belkhouche, B. A comparative analysis of machine learning classifiers for twitter sentiment analysis. *Research in Computing Science 110* (2016), 71–83.

[46] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. Performance analysis of high performance computing applications on the amazon web services cloud. In *Proceedings of the International Conference on Cloud Computing Technology and Science (CloudComm)* (2010), IEEE, pp. 159–168.

[47] Jarke, M., Jeusfeld, M. A., Quix, C., and Vassiliadis, P. Architecture and quality in data warehouses. In *Seminal Contributions to Information Systems Engineering, 25 Years of CAiSE*. Springer, 2013, pp. 161–181.

[48] Jarke, M., Lenzerini, M., Vassiliou, Y., and Vassiliadis, P. *Fundamentals of Data Warehouses*. Springer, 2003.

[49] Jovanovic, P., Romero, O., Simitsis, A., and Abelló, A. Incremental consolidation of data-intensive multi-flows. *IEEE Transactions on Knowledge and Data Engineering 28* (2016), 1203–1216.

[50] Karagiannis, A. Macro-level scheduling of ETL workflows. In *Proceedings of the International Workshop on Quality in Databases (QDB)* (2011).

[51] Karagiannis, A., Vassiliadis, P., and Simitsis, A. Scheduling strategies for efficient ETL execution. *Information Systems 38* (2013), 927–945.

[52] Karol, S. *Well-formed and scalable invasive software composition*. PhD thesis, Dresden University of Technology, 2015.

[53] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. *ECOOP Object-oriented programming* (1997), 220–242.

[54] Kimball, R. Slowly changing dimensions. *Information Management 18* (2008), 29.

[55] Kimball, R., and Ross, M. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[56] Konstantinou, N., and Paton, N. W. Feedback driven improvement of data preparation pipelines. *Information Systems 92* (2020), 101480.

[57] Kumar, N., and Kumar, P. S. An efficient heuristic for logical optimization of ETL workflows. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2011), pp. 68–83.

[58] Lawler, E. L., and Wood, D. E. Branch-and-bound methods: A survey. *Operations Research 14* (1966), 699–719.

[59] Lella, R. Optimizing BDFS jobs using InfoSphere DataStage Balanced Optimization. IBM Developer Works, (2014).

[60] Liu, X., and Iftikhar, N. An ETL optimization framework using partitioning and parallelization. In *Proceedings of the Annual Symposium on Applied Computing (SAC)* (2015), pp. 1015–1022.

[61] Liu, X., Thomsen, C., and Pedersen, T. B. ETLMR: A highly scalable dimensional ETL framework based on mapreduce. In *Proceedings of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK)* (2011), pp. 96–111.

[62] Liu, X., Thomsen, C., and Pedersen, T. B. ETLMR: A highly scalable dimensional ETL framework based on mapreduce. *Transactions on Large-Scale Data-and Knowledge-Centered Systems 8* (2013), 1–31.

[63] Liu, X., Thomsen, C., and Pedersen, T. B. CloudETL: scalable dimensional ETL for hive. In *Proceedings of the International Database Engineering & Applications Symposium (IDEAS)* (2014), pp. 195–206.

[64] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S., and McClosky, D. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of the Association for Computational Linguistics System Demonstrations (ACL)* (2014), pp. 55–60.

[65] Manousis, P., Vassiliadis, P., and Papastefanatos, G. Impact analysis and policy-conforming rewriting of evolving data-intensive ecosystems. *Journal on Data Semantics 4* (2015), 231–267.

[66] McGuinness, D. L., Van Harmelen, F., et al. Owl web ontology language overview. *W3C Recommendation 10* (2004), 2004.

[67] Mey, J., Karol, S., Assmann, U., Huismann, I., Stiller, J., and Fröhlich, J. Using semantics-aware composition and weaving for multi-variant progressive parallelization. In *Proceedings of the International Conference on Computational Science (ICCS)* (2016), pp. 1554–1565.

[68] Miloslavskaya, N., and Tolstoy, A. Big data, fast data and data lake concepts. *Procedia Computer Science 88* (2016), 63.

[69] Nambiar, R. O., and Poess, M. The making of TPC-DS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2006), pp. 1049–1058.

[70] Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., and Arocena, P. C. Data lake management: challenges and opportunities. *VLDB Endowment 12* (2019), 1986–1989.

[71] Nath, R. P. D., Romero, O., Pedersen, T. B., and Hose, K. High-level ETL for semantic data warehouses. *Computing Research Repository abs/2006.07180* (2020).

[72] Oliveira, B., and Belo, O. BPMN patterns for ETL conceptual modelling and validation. In *Proceedings of the International Symposium on Foundations of Intelligent Systems (ISMIS)* (2012), pp. 445–454.

[73] Patil, P., Rao, S., and Patil, S. Data integration problem of structural and semantic heterogeneity: data warehousing framework models for the optimization of the ETL processes. In *Proceedings of the ACM International Conference Workshop on Emerging Trends in Technology (ICWET)* (2011), pp. 500–504.

[74] Ravat, F., and Zhao, Y. Data lakes: Trends and perspectives. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)* (2019), pp. 304–313.

[75] Romero, O., and Wrembel, R. Data engineering for data science: Two sides of the same coin. In *Proceedings of the International Conference on Big Data Analytics and Knowledge Discovery (DaWaK)* (2020), pp. 157–166.

[76] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2004.

[77] Russom, P. Data lakes: Purposes, practices, patterns, and platforms. `https://info.talend.com/rs/talend/images/WP_EN_BD_TDWI_DataLakes.pdf`, 2017. (Accessed 03/31/2021).

[78] Santoso, L. W., et al. Data warehouse with big data technology for higher education. *Procedia Computer Science 124* (2017), 93–99.

[79] Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., and Price, T. G. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD)* (1979), pp. 23–34.

[80] Sellis, T. K., and Simitsis, A. ETL workflows: from formal specification to optimization. In *Proceedings of the Advances in Databases and Information Systems (ADBIS)* (2007), pp. 1–11.

[81] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The hadoop distributed file system. In *Proceedings of the IEEE Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[82] Simitsis, A., Skoutas, D., and Castellanos, M. Representation of conceptual ETL designs in natural language using semantic web technology. *Data & Knowledge Engineering 69* (2010), 96–115.

[83] Simitsis, A., and Vassiliadis, P. A methodology for the conceptual modeling of ETL processes. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)* (2003).

[84] Simitsis, A., and Vassiliadis, P. A method for the mapping of conceptual designs to logical blueprints for ETL processes. *Decision Support Systems 45* (2008), 22–40.

[85] Simitsis, A., Vassiliadis, P., and Sellis, T. Optimizing ETL processes in data warehouses. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2005), pp. 564–575.

[86] Simitsis, A., Vassiliadis, P., and Sellis, T. K. State-space optimization of ETL workflows. *IEEE Transactions on Knowledge and Data Engineering 17* (2005), 1404–1419.

[87] Simitsis, A., Wilkinson, K., Castellanos, M., and Dayal, U. Qox-driven ETL design: reducing the cost of ETL consulting engagements. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD)* (2009), pp. 953–960.

[88] Simitsis, A., Wilkinson, K., Dayal, U., and Castellanos, M. Optimizing ETL workflows for fault-tolerance. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2010), pp. 385–396.

[89] Skoutas, D., and Simitsis, A. Designing ETL processes using semantic web technologies. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2006), pp. 67–74.

[90] Skoutas, D., and Simitsis, A. Ontology-based conceptual design of ETL processes for both structured and semi-structured data. *International Journal on Semantic Web and Information Systems 3* (2007), 1–24.

[91] Skoutas, D., Simitsis, A., and Sellis, T. Ontology-driven conceptual design of ETL processes using graph transformations. *Journal on Data Semantics 13* (2009), 120–146.

[92] Stefanowski, J., Krawiec, K., and Wrembel, R. Exploring complex and big data. *International Journal of Applied Mathematics and Computer Science 27* (2017), 669–679.

[93] Terrizzano, I., Schwarz, P., Roth, M., and Colino, J. E. Data Wrangling: The Challenging Journey from the Wild to the Lake. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* (2015).

[94] Thomsen, C., and Pedersen, T. B. Easy and effective parallel programmable ETL. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2011), pp. 37–44.

[95] Trujillo, J., and Luján-Mora, S. A UML based approach for modeling ETL processes in data warehouses. In *Proceedings of the International Conference on Conceptual Modeling (ER)* (2003), pp. 307–320.

[96] Tziovara, V., Vassiliadis, P., and Simitsis, A. Deciding the physical implementation of ETL workflows. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2007), pp. 49–56.

[97] Vaisman, A. A., and Zimányi, E. *Data Warehouse Systems - Design and Implementation*. Springer, 2014.

[98] Vassiliadis, P., Simitsis, A., and Baikousi, E. A taxonomy of ETL activities. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2009), pp. 25–32.

[99] Vassiliadis, P., Simitsis, A., Georgantas, P., and Terrovitis, M. A framework for the design of ETL scenarios. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)* (2003), pp. 520–535.

[100] Vassiliadis, P., Simitsis, A., Georgantas, P., Terrovitis, M., and Skiadopoulos, S. A generic and customizable framework for the design of ETL scenarios. *Information Systems 30* (2005), 492–525.

[101] Vassiliadis, P., Simitsis, A., and Skiadopoulos, S. Conceptual modeling for ETL processes. In *Proceedings of the ACM International Workshop on Data Warehousing and OLAP (DOLAP)* (2002), pp. 14–21.

[102] Vassiliadis, P., Simitsis, A., and Skiadopoulos, S. Modeling ETL activities as graphs. In *Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW)* (2002), pp. 52–61.

[103] Vernica, R., Carey, M. J., and Li, C. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (MOD)* (2010), pp. 495–506.

[104] Viana, V., De Oliveira, D., and Mattoso, M. Towards a cost model for scheduling scientific workflows activities in cloud environments. In *Proceedings of the IEEE World Congress on Services (IEEE)* (2011), pp. 216–219.

[105] Warneke, D., and Kao, O. Nephele: efficient parallel data processing in the cloud. In *Proceedings of the ACM International Workshop on Many-task Computing on Grids and Supercomputers (MTAGS)* (2009), pp. 1–10.

[106] Wilkinson, K., Simitsis, A., Castellanos, M., and Dayal, U. Leveraging business process models for ETL design. In *Proceedings of the International Conference on Conceptual Modeling (ER)* (2010), pp. 15–30.

[107] Wojciechowski, A. ETL workflow reparation by means of case-based reasoning. *Information Systems Frontiers 20* (2017), 21–43.

[108] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. Apache spark: a unified engine for big data processing. *Communications of the ACM 59* (2016), 56–65.

[109] Zhou, J., Larson, P.-A., and Chaiken, R. Incorporating partitioning and parallel plans into the scope optimizer. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)* (2010), pp. 1060–1071.