On Evaluating Performance of Balanced Optimization of ETL Processes for Streaming Data Sources

Michał Bodziony
IBM Poland, Software Lab Kraków
Kraków, Poland
michal.bodziony@pl.ibm.com

Szymon Roszyk
Poznan University of Technology
Poznań, Poland
szymon.roszyk@student.put.poznan.pl

Robert Wrembel
Poznan University of Technology
Poznań, Poland
robert.wrembel@cs.put.poznan.pl

ABSTRACT

A core component of each data warehouse architecture is the Extract-Transform-Load (ETL) layer. The processes in this layer integrate, transform, and move large volumes of data from data sources into a data warehouse (or data lake). For this reason, efficient execution of ETL processes is of high practical importance. Research approaches propose task re-ordering and parallel processing. In practice, companies apply scale-up or scale-out techniques to increase performance of the ETL layer. ETL tools existing on the market support mainly parallel processing of ETL tasks. Additionally, the IBM DataStage ETL engine applies the so-called *balanced optimization* that allows to execute ETL tasks either in a data source or in a data warehouse, thus optimizing overall resource consumption. With the widespread of non-relational technologies for storing big data, it is essential to apply ETL processes to this type of data. In this paper, we present an initial assessment of the *balanced optimization* applied to IBM DataStage, ingesting data from Kafka streaming data source.

1 INTRODUCTION

A core component of a data warehouse (DW) [25] or a data lake (DL) [22] is an Extract-Transform-Load (ETL) software. It is responsible for ingesting data from data sources (DSs), integrating, and cleaning data as well as uploading them into a DW/DL. ETL is implemented as a workflow (process) composed of tasks, managed by an ETL engine, frequently run on a dedicated server or on a cluster. An ETL process moves large volumes of data between DSs and a DW. Therefore, its execution may take hours to complete. For example, simple movement of 1TB of data between a DS and DW (which use magnetic disks with a typical 200MB/s throughput) takes 2.7 hours by an ETL process. Resource consumption and the overall processing time are further increased by complex tasks executed within the process, including integrating, cleaning, and de-duplicating data. For this reason, providing means for optimization of ETL processes is of high practical importance. Big data add to the problem more complexity that results from: (1) bigger data volumes, and (2) much more complex and diverse data models and formats that need to be processed by ETL. These characteristics call for yet more efficient execution of ETL processes.

Unfortunately, this problem has been only partially addressed by technology and research, cf. Section 2. In practice, companies apply scale-up or scale-out for their ETL servers, to reduce processing time. ETL tools existing on the market support mainly parallel processing of ETL tasks. Some of them, on top of that, apply task movement within an ETL process. One of such techniques is called *balanced optimization* [17]. Research approaches propose task re-ordering and parallel processing.

This paper presents first findings from an R&D project, jointly done by IBM Software Lab and Poznan University of Technology, on assessing the performance of the *balanced optimization* on a streaming data source. In particular, we were interested in assessing: (1) if the *balanced optimization* may increase the performance of ETL processes - run by IBM DataStage and ingesting data from a stream DS - run by Kafka, (2) which ETL tasks may benefit from the *balanced optimization*, and (3) what parameters of the DS may affect performance. The findings are summarized in Section 4.

2 RELATED WORK

In this section we overview the solutions to optimizing ETL processes: (1) applied in practice by companies, (2) offered by ETL engines existing on the market, and (3) developed by researchers.

2.1 Industrial approaches

Industrial projects utilizing ETL, reduce processing time of ETL processes by increasing computational power of an ETL server and by applying parallel processing of data. This can be achieved by: (1) scaling-up an ETL server, i.e., by means of increasing the number of CPU and the size of memory, (2) scaling-out an ETL architecture, i.e., by means of increasing the number of workstations in a cluster running the ETL engine. All the commercial ETL engines, cf. [8], support this kind of optimization. Parallel processing is also applicable to uploading data into a data warehouse, e.g., command *nzload* (IBM Pure Data for Analytics, a.k.a. Netezza) or *import* (Oracle).

On top of this, an ETL task re-ordering may be used to produce a more efficient processes. In the simplest case, called *push down*, the most selective tasks are moved towards the beginning of an ETL process (towards data sources) to reduce a data volume (I/O) as soon as possible [26]. IBM extended this technique into *balanced optimization*, where some tasks are moved into DSs and some are moved into a DW, to be executed there, cf. Section 3.

2.2 Research approaches

Research approaches to optimizing ETL execution can be categorized as: (1) optimization by task re-ordering augmented with: estimation of execution costs and parallelization and (2) optimization of ETL processes with user defined functions (UDFs).

2.2.1 *Cost-based, task re-ordering, and parallelization.*

In [19, 20] the authors apply the MapReduce framework to process ETL workflows to feed in parallel dimensional schemas (star, snowflake, and SCDs). The authors provide some processing skeletons for MapReduce for dimensions of different sizes and structures. A system for optimizing MapReduce workloads was presented in [11]. The optimization uses: sharing of data flows, materialization, and physical data layouts. However, neither an optimization technique nor a cost function were explained.

[18] proposes to parallelize ETL process P in three steps. First, P is partitioned into linear sub-processes LP_i. Second, in each LP_i, input data are partitioned horizontally into n disjoint partitions (n is parameterized) and each partition is processed in parallel by a separate thread. Third, for tasks with a heavy computational load, an internal multi-threading parallelization may be applied, if needed.

[10] assumes that an ETL process is assigned an estimated execution cost and a more efficient variant of this process is produced by task re-ordering, but the authors do not provide any method for the re-ordering. The main goal of [10] is to identify a set of statistics to collect for this kind of optimization. The constraint is that this set must be: minimal and applicable to estimate costs of all possible task re-orderings. Moreover, the cost of collecting statistics must be minimal. The statistics include: a cardinality of table T_i, attribute histograms of T_i, and a number of distinct values of attributes of T_i. A proposed cost function includes: data statistics, CPU speed, disk-access speed, and memory usage. The authors proposed cost functions for the following ETL tasks (all expressed via SQL): select, project, join, group-by, and transform. To find the set of statistics, which is an NP-hard problem, the authors use linear programming.

The approach presented in [24] goes one step further, as it proposes a specific performance optimization technique by task re-ordering. To this end, each workload gets assigned an execution cost. The main components of the cost formula are time and data volume. The five following workflow transformation operations were proposed: swap - changing the order of two tasks, factorize - combining tasks that execute the same operation on different flows, distribute - splitting the execution of a task into n parallel tasks, merge - merging two consecutive tasks, and its reverse operation - split [23]. For each of these operations, criteria for correct workflow transformations were proposed. Finally, a heuristic for pruning the search space of all possible workflow transformations (by task re-ordering) was proposed, with the main goal to filter data as soon as possible. In [15], the re-ordering of operators is based on their semantics, e.g., a highly selective operator would be placed (re-ordered) at the beginning of a workflow.

In the same spirit, workflow transformations were proposed in [14] for the purpose of being able to reuse existing data processing workflows and integrate them into other workflows.

The approach proposed in [16] draws upon the contributions of [24]. In [16], possible tasks re-orderings are constrained by means of a dedicated structure called a dependency graph. This approach optimizes only linear workflows. To this end, a non-linear workflow is split into linear ones (called groups), by means of pre-defined split rules. Next, parallel groups are optimized independently by task re-ordering - tasks can be moved between adjacent groups, and adjacent groups can be merged. The drawback of this approach is however, that the most selective tasks can be moved towards the end of a workflow as the result of a re-ordering.

2.2.2 *ETL with UDFs.*

None of the approaches described in Sections 2.2.1 and 2.1 supports the optimization of ETL processes with user-defined functions. In the techniques based on the re-ordering of operators, the semantics of the operators must be known. Such a semantics is known and understood for traditional operators based on the relational algebra. However, the semantics of user defined functions is typically unknown. For this reason, UDFs have been handled by solutions that require: either (1) manual annotation of UDFs or (2) perform an analysis of an UDF code to explore some options for optimization or (3) use explicitly defined parallel skeletons.

In [12, 13], UDFs are treated as black boxes. The framework consists of the Nephele execution engine and the PACT compiler to execute UDFs, based on the PACT programming model [5] (PACT is a generalization of MapReduce). PACT can use hints to execute a given task in parallel. Such hints are later exploited by a cost-based optimizer to generate parallel execution plans. The optimization is based on: (1) a re-ordering of UDFs in a workflow and (2) an execution of UDFs in a parallel environment. The re-ordering takes into account properties of UDFs. The properties are discovered by means of a static code analysis. A cost-based optimizer (model) is used to construct some possible re-orderings. Also in [9], UDFs annotations are used to generate an optimized query plan, but only for relational operators.

A framework proposed in [7], called SQL/MR, enables a parallelization of UDFs in a massively-parallel shared-nothing database. To achieve a parallelism, SQL/MR requires a definition of the Row and Partition functions and corresponding execution models for the SQL/MR function instances. The Row function is described as an equivalent to a Map function in MapReduce. Row functions perform row-level transformation and processing. The execution model of the Row function allows independent processing of each input row by exactly one instance of the SQL/MR function. The Partition function is similar to the Reduce function in MapReduce. Exactly one instance of a SQL/MR function is used to independently process each group of rows defined by the PARTITION BY clause in a query. Independent processing of each partition allows the execution engine to achieve parallelism at the level of a partition. The dynamic cost-based optimization is enabled for re-orderings of UDFs.

An optimizer proposed in [21] rewrites an execution plan based on: (1) automatically inferring the semantics of a MapReduce style UDF and (2) a small set of rewrite rules. The semantics can be provided by means of manual UDF annotations or an automatic discovery. The manual annotations include: a cost function, resource consumption, a number of input rows and output rows. The automatically discovered annotations include: parallelization function of a given operator, a schema information, and read/write operations on attributes.

The aforementioned approaches require understanding the semantics of an UDF (a black-box) by means of either parsing an UDF code, or applying certain coding style, or using certain keywords, or using parallelization hints. Moreover, the approaches do not provide means of analyzing an optimal architecture configuration (e.g., the number of nodes in a cluster) or a degree of parallelism.

In [2] the authors developed a framework for using pre-defined generic and specific code skeletons for writing programs to be executed in a parallel environment. The code is then generated automatically and the process is guided by configuration parameters that define the number of nodes a program is executed on.

In [1, 3] the authors proposed an overall architecture for executing UDFs in a parallel environment. The architecture was further extended in [4] with a model for optimizing a cluster configuration, to provide a sub-optimal performance of a given ETL process with UDFs. The applied model is based on the multiple choice knapsack problem and the *lp_solve* library is used to solve the problem (its implementation is available on GitHub[1]).

The work described in this paper applies: (1) the *balanced optimization*, which uses task re-orderings to move some ETL tasks into a data source, (2) parallel processing of ETL tasks, by standard parallelization mechanisms of IBM DataStage run in a micro-cluster, and (3) parallel processing of some ETL tasks moved into and executed in a data source, by means of standard parallelization mechanisms available in Kafka.

3 BALANCED OPTIMIZATION

The *balanced optimization* [17, 27] is implemented in IBM InfoSphere DataStage. Its overall goals are to: (1) minimize data movement, i.e., I/O, (2) use optimization techniques available in a source or target data servers, and (3) maximize parallel processing. The principle of this optimization is to balance processing between a data source, an ETL server, and a destination (typically a data warehouse). To this end, a given ETL task can be moved into a data source (operation *push down*), to be executed there. It is typically applied to tasks at the beginning of an ETL process, i.e., ingesting, filtering, or pre-processing data. ETL tasks that cannot be moved into a data source are executed in the DataStage engine. Finally, tasks that profit from processing in a data warehouse are moved and executed there (operation *push up*). This way, specialized features of a DW management system can be applied to processing these tasks, e.g., dedicated indexes, multidimensional clusters, partitioning schemes, and materialized views.

Moving tasks is controlled by DataStage preferences [17] that guide the DataStage engine to move certain pre-defined tasks into either a DS, or a DW, or execute them in the engine. The following tasks are controlled by the preferences: transformation, filtering, aggregation, join, look-up, merge, project, bulk I/O, and temporary staging tables. Once a given ETL process has been designed, a user specifies the preferences. Then the ETL process gets optimized by DataStage, taking into account the preferences. An executable version of the process is generated and deployed.

So far, the *balanced optimization* has been proven to be profitable when applied to structured data sources. Since numerous NoSQL and stream DSs become first class citizens, assessing the applicability of this type of optimization to such data sources may have a real business value. The work presented in this paper is the first one to assess the *balanced optimization* on a stream data source.

4 EXPERIMENTAL EVALUATION

The goal of the experiments was to assess if the *balanced optimization* in IBM DataStage may increase the performance of an ETL process ingesting data from a stream data source, run by Kafka. Such a software architecture is frequently used by IBM customers. In particular, we were interested in figuring out: (1) what parameters of Kafka may affect performance, cf. Sections 4.1 and 4.2 as well as (2) which ETL tasks may benefit from the *balanced optimization*. In this phase of the project we evaluated filtering (Section 4.3), dataflow split (Section 4.4), and aggregation (Section 4.5).

[1]https://github.com/fawadali/MCKPCostModel

To this end a pico-cluster composed of four physical workstations was built. Each workstation included: a 4-core CPU 3GHz, 16GB RAM, 256GB HDD, and was run under Linux RedHat. Two workstations run Kafka and the other two run the *IBM InfoSphere DataStage* ETL server. The ETL processes were designed using DataStage Designer. The system was fed with rows from table Store_Sales (1.5GB), from the TPC-D benchmark. The performance statistics were measured on each node by iostat. In each scenario 12 experiments were run. The lowest and the highest measurements were discarded and the average of the remaining 10 was computed and is shown in all the figures. Due to the space constraints, we present only selected experimental scenarios.

Notice that the goal of these experiments was to assess the behaviour of the system only at the border between Kafka and DataStage. For this reason, the ETL processes used in the experiments included elementary tasks/components, like: Kafka connector, column import, filtering, aggregation, and storing output in a file. An example process splitting a data flow is shown in Figure 1.

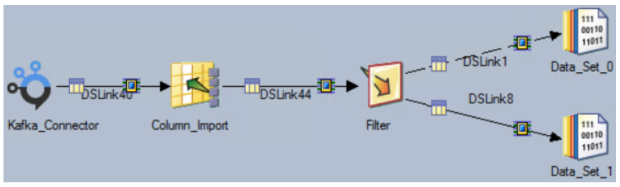

Figure 1: An example ETL process splitting a data flow

4.1 Parameter *RecordCount*

RecordCount is a parameter of a connector from DataStage to Kafka. It defines a number of rows (a rowset) that are read from a topic. After a rowset is read, the ETL connector outputs the rowset to the ETL process for further processing.

The value of *RecordCount* in the experiments ranged from 1 to 'ALL'. In the latter case, the whole test data set was read before starting the ETL process (in the experiment 11 million of rows). The elapsed times of processing the whole data set are shown in Figure 2. The value of the standard deviation ranges from 0.5 to 2.5. From the chart we observe that: (1) the performance strongly depends on the value of the parameter and (2) the performance does not improve for values of *RecordCount* greater than 50. Having analyzed detailed CPU and I/O statistics (not shown in this paper) we conclude that an optimal value of *RecordCount* is within the range (400, 600).

4.2 The number of Kafka partitions

Kafka allows to split streaming data into partitions, each of which is read by a consumer. As the number of partitions may strongly influence the performance of the whole ETL process[2], figuring out how performance is affected by the number of partitions is of great interest.

To assess the impact of the number of partitions on processing time, in our experiments the number of partitions ranged from 1 to 8. The number of partitions equaled to the number of Kafka consumers. The performance results are shown in Figure 3. Here we show only the results for *RecordCount* = ALL. As we can observe, the elapsed processing time of the ETL process varies and

[2]https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster

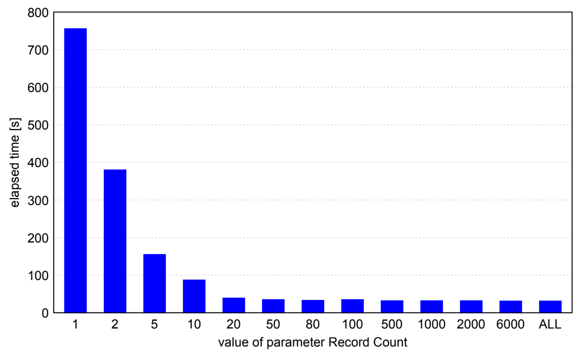

Figure 2: The dependency of the elapsed processing time on the value of parameter *RecordCount*

the best performance was achieved for the number of partitions equaled to 2 and 3. The standard deviation ranges from 0.5 to 1.3.

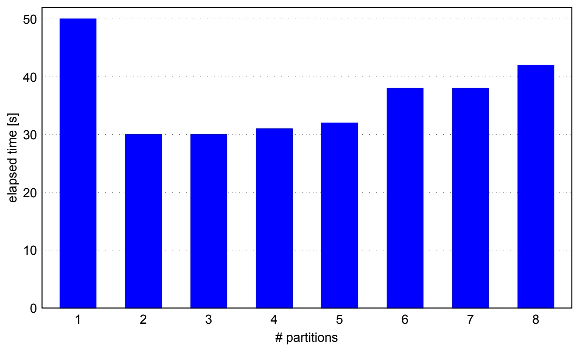

Figure 3: The dependency of the elapsed processing time on the number of partitions (consumers); parameter *RecordCount*=ALL

A detailed analysis of I/O (not shown here due to space constraints) reveals that the number of I/O increases with the increasing number of partitions. Similar increase was observed also in a CPU time. That may be some of the factors that influence the characteristics shown in Figure 3.

4.3 Selectivity of data ingest

In this experiment we assessed the elapsed ETL processing time w.r.t. the selectivity of a data ingest from Kafka (the selectivity is defined as: the number of rows ingested / the total number of rows). Two scenarios were implemented. In the first one, operation *push down* was applied to move data filtering into Kafka (available in library Kafka Streams). In the second one, data were filtered by a standard pre-defined task in DataStage. The results for *RecordCount*=10 and *RecordCount*=ALL are shown in Figure 4. The results labeled with suffix *PD* denote the scenario with operation *push down* applied. The standard deviation ranges from 0.1 to 2.8.

The results clearly show that a wrong combination of *RecordCount* with applied *push down* can decrease the overall performance of an ETL process, cf. *RecordCount=ALL PD* (with *push down*) vs. *RecordCount=ALL* (without *push down*), for selectivity=90%. From other experiments (not presented in this paper) we observed that values of *RecordCount* $\leq$ 20, caused that *push down* offered stable better performance for the whole range of selectivities.

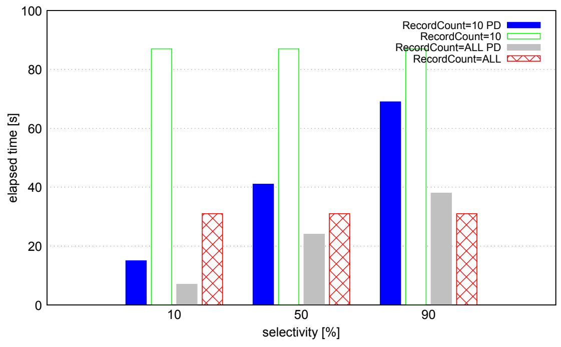

Figure 4: The dependency of the elapsed processing time on the selectivity; parameter *RecordCount* in {10, ALL}

4.4 Flow split

The purpose of a flow split is to divide a data flow into multiple flows. In our case, we split the input flow into two, using a parameterized split ratio: from 10% of data in one flow to an even split. The flow split was executed in: (1) Kafka, as the result of *push down*, and (2) DataStage. The obtained elapsed execution times for *RecordCount*=10 and *RecordCount*=ALL are shown in Figure 5. The standard deviation ranged from 1.3 to 3.3.

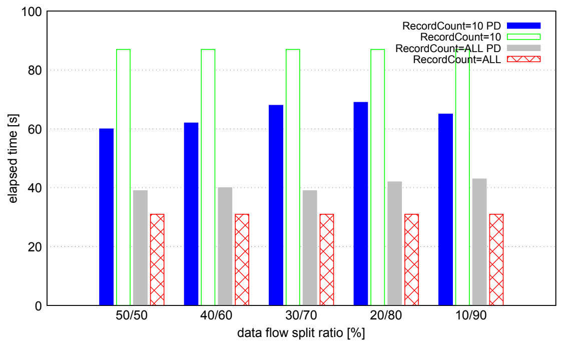

Figure 5: The dependency of the elapsed processing time on the data flow split ratio; parameter *RecordCount* in {10, ALL}

The characteristics show the performance improvement while applied *push down* for *RecordCount*=10, cf. bars labeled *RecordCount=10 PD* vs. *RecordCount=10*. From other experiments (not discussed in this paper) we observed that the maximum value of *RecordCount* that improved the performance is 10. From the analysis of detailed data on CPU and I/O, we tentatively draw a conclusion that the observed behaviour is caused by Kafka that was not able to deliver data on time into DataStage.

4.5 Aggregation

In this experiment, `count` is used to count records in groups. The number of groups ranges from 10 to 90. The aggregation is executed in two variants: (1) in Kafka (available in library Kafka Streams), as the result of *push down*, and (2) in DataStage. The results are shown in Figure 6 for *RecordCount*=10 and *RecordCount*=ALL. The standard deviation ranged from 0.5 to 2.2. As we can see, the execution time does not profit from *push down* in either of these cases, cf. bars labeled as *RecordCount=10 PD* (*push down* applied) vs. *RecordCount=10* (without *push down*) and *RecordCount=ALL PD* vs. *RecordCount=ALL*.

The analysis of CPU and I/O reveals that *push down* caused higher CPU usage times and higher I/O, which resulted in much worse performance of the aggregation in Kafka than in DataStage.

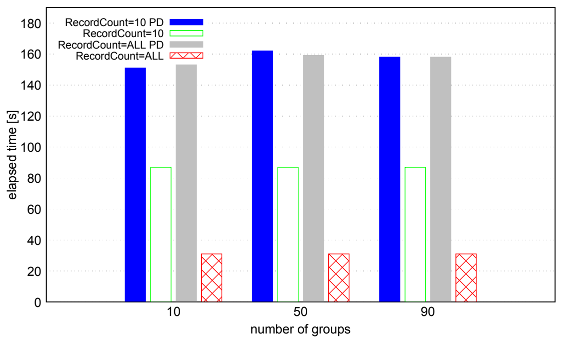

Figure 6: The dependency of the elapsed processing time on the number of groups in `select count(*) ... group by`; parameter *RecordCount* in {10, ALL}

5 SUMMARY

In practice, data integration architectures are frequently built using IBM DataStage, NoSQL (HBase, Cassandra), and stream DSs (Kafka). The *balanced optimization* available in DataStage offers performance improvements for relational data sources, however, its characteristics on NoSQL and stream DSs are unknown. For this reason, a project was launched at IBM to discover these characteristics, with a goal to build an execution optimizer for a standard and multi-cloud architectures [6], based on a learning recommender.

In this paper we presented the experimental evaluation of the *balanced optimization* applied to a stream DS run by Kafka, within a joint project of IBM and Poznan University of Technology. To the best of our knowledge, it is the first project on analyzing the possibility of using this type of ETL optimization on non-relational DSs.

From this evaluation, the most interesting observations of a real business value are as follows. First, Kafka turned out to be a bottleneck for *push down* applied to: (1) filtering, for *RecordCount=ALL* and the selectivity 50%; for other tested operations *push down* increased performance; (2) split, for *RecordCount=ALL*, for all split ratios; (3) aggregation, for *RecordCount in* {10, ALL}. Second, an overall performance of an ETL process strongly depends on specific parameters of Kafka, e.g., RecordCount and the number of partitions. Third, the characteristics of CPU and I/O usage may suggest that in order to increase the performance of Kafka, more hardware needs to be allocated for Kafka than for DataStage.

Even though, the aforementioned observations cannot be generalized (they apply to this particular experimental setting), they turned out to be of practical value. First, they were very well received by the Executive Management @IBM. Second, the observations were integrated into a knowledge base of IBM Software Lab and have already been used for multiple proofs of concept.

The next phase of this project will consist in: (1) extending the evaluation of *push down* to Kafka (other sizes of a cluster, other parameters of Kafka), (2) evaluating *push down* for HBase and Cassandra, (3) designing and building a learning recommender for DataStage. The recommender will analyze experimental metadata to discover patterns and build recommendation models, with a goal to propose configuration parameters for a given ETL process and to propose an orchestration scenarios of tasks within the *balanced optimization*.

Acknowledgements. The work of Robert Wrembel is supported by: (1) the grant of the Polish National Agency for Academic Exchange, within the Bekker Programme and (2) IBM Shared University Reward 2019.

REFERENCES

[1] Syed Muhammad Fawad Ali. 2018. Next-generation ETL Framework to Address the Challenges Posed by Big Data. In *DOLAP*.
[2] Syed Muhammad Fawad Ali, Johannes Mey, and Maik Thiele. 2019. Parallelizing user-defined functions in the ETL workflow using orchestration style sheets. *Int. J. of Applied Mathematics and Comp. Science (AMCS)* (2019), 69–79.
[3] Syed Muhammad Fawad Ali and Robert Wrembel. 2017. From conceptual design to performance optimization of ETL workflows: current state of research and open problems. *The VLDB Journal* (2017), 1–25.
[4] Syed Muhammad Fawad Ali and Robert Wrembel. 2019. Towards a Cost Model to Optimize User-Defined Functions in an ETL Workflow Based on User-Defined Performance Metrics. In *ADBIS*. LNCS 11695, 441–456.
[5] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *ACM Symposium on Cloud Computing*. 119–130.
[6] Michal Bodziony. 2019. ETL in Big Data Architectures: IBM Approach to Design and Optimization of ETL Workflows (Invited talk). In *DOLAP*.
[7] Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *PVLDB* 2, 2 (2009), 1402–1413.
[8] Gartner. 2019. Magic Quadrant for Data Integration Tools.
[9] Philipp Große, Norman May, and Wolfgang Lehner. 2014. A study of partitioning and parallel UDF execution with the SAP HANA database. In *SSDBM*. 36.
[10] Ramanujam Halasipuram, Prasad M. Deshpande, and Sriram Padmanabhan. 2014. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. In *EDBT*. 307–318.
[11] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, Vol. 11. 261–272.
[12] Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. 2013. Peeking into the optimization of data flow programs with mapreduce-style udfs. In *ICDE*. 1292–1295.
[13] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the black boxes in data flow optimization. *PVLDB* 5, 11 (2012), 1256–1267.
[14] Petar Jovanovic, Oscar Romero, Alkis Simitsis, and Alberto Abelló. 2016. Incremental Consolidation of Data-Intensive Multi-Flows. *IEEE TKDE* 28, 5 (2016), 1203–1216.
[15] Anastasios Karagiannis, Panos Vassiliadis, and Alkis Simitsis. 2013. Scheduling strategies for efficient ETL execution. *Information Syst.* 38, 6 (2013), 927–945.
[16] Nitin Kumar and P. Sreenivasa Kumar. 2010. An Efficient Heuristic for Logical Optimization of ETL Workflows. In *VLDB Workshop on Enabling Real-Time Business Intelligence*. 68–83.
[17] Rao Lella. 2014. Optimizing BDFS jobs using InfoSphere DataStage Balanced Optimization. IBM white paper: Developer Works.
[18] Xiufeng Liu and Nadeem Iftikhar. 2015. An ETL optimization framework using partitioning and parallelization. In *ACM SAC*. 1015–1022.
[19] Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen. 2012. MapReduce-based Dimensional ETL Made Easy. *PVLDB* 5, 12 (2012), 1882–1885.
[20] Xiufeng Liu, Christian Thomsen, and Torben Bach Pedersen. 2013. ETLMR: A Highly Scalable Dimensional ETL Framework Based on MapReduce. *Trans. Large-Scale Data- and Knowledge-Centered Systems* 8 (2013), 1–31.
[21] Astrid Rheinländer, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. 2015. SOFA: An extensible logical optimizer for UDF-heavy data flows. *Information Syst.* 52 (2015), 96–125.
[22] Philip Russom. 2017. Data Lakes: Purposes, Practices, Patterns, and Platforms. TDWI white paper.
[23] Alkis Simitsis, Panos Vassiliadis, and Timos Sellis. 2005. Optimizing ETL Processes in Data Warehouses. In *ICDE*. 564–575.
[24] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. State-Space Optimization of ETL Workflows. *IEEE TKDE* 17, 10 (2005), 1404–1419.
[25] Alejandro A. Vaisman and Esteban Zimányi. 2014. *Data Warehouse Systems - Design and Implementation*. Springer.
[26] Informatica white paper. 2007. How to Achieve Flexible, Cost-effective Scalability and Performance through Pushdown Processing.
[27] IBM white paper. 2008. IBM InfoSphere DataStage Balanced Optimization. Information Management Software.