Cost-based Sharing and Recycling of (Intermediate) Results in Dataflow Programs

Stefan Hagedorn[⋆] and Kai-Uwe Sattler

Databases & Information Systems Group, TU Ilmenau,
Ilmenau, Germany,
{stefan.hagedorn,kus}@tu-ilmenau.de

Abstract. In data analytics, researchers often work on the same datasets investigating different aspects and moreover develop their programs in an incremental manner. This opens opportunities to share and recycle results from previously executed jobs if they contain identical operations, e.g., restructuring, filtering and other kinds of data preparation.
In this paper, we present an approach to accelerate processing of such dataflow programs by materializing and recycling (intermediate) results in Apache Spark. We have implemented this idea in our Pig Latin compiler for Spark called *Piglet* which transparently supports both, merging of multiple jobs as well as rewriting jobs to reuse intermediate results. We discuss the opportunities for recycling, present a profiling-based cost model as well as a decision model to identify potentially beneficial materialization points. Finally, we report results of our experimental evaluation showing the validity of the cost model and the benefit of recycling.

1 Introduction

Today, scalable distributed analytics platforms like Hadoop, Spark or Flink are typically used together with higher-level languages, either a domain-specific language (DSL) often integrated with a programming language (e.g. Scala or Python) or dedicated high-level dataflow languages such as Pig Latin or Jaql which can be extended by UDFs. Particularly, the latter ones offer the advantage of allowing to write complex analytical pipelines in an easy and incremental manner. Furthermore, this interactive and incremental formulation is also supported by notebook-like interfaces such as Jupyter or Zeppelin.

For discussing the opportunities for sharing and recycling intermediate results of dataflow programs in Apache Spark we consider the following exemplified use case: A (small) group of data scientists has to analyze several datasets containing sensor data from weather and environmental observation stations. These datasets contain spatio-temporal data but also metadata describing e.g. sensors. By analyzing this data, models are constructed to represent phenomena which can be used for forecasting, classification etc. Main tasks in this process are

[⋆] This work was partially funded by the German Research Foundation (DFG) under grant no. SA782/22.

integrating and preparing data (e.g. cleaning, transformation, and reduction), selecting relevant subsets (e.g. by time or region of interest), and applying machine learning and other analytical operations. Typically, this follows a rather incremental and explorative approach where the dataflow jobs are specified and executed step by step to inspect and validate results, test different parameters, and decide about subsequent steps to add further necessary operators. Furthermore, multiple scientists might use the same datasets in parallel or simply run multiple jobs to analyze different aspects. Based on these assumptions, there exist two obvious opportunities for sharing work:

merge: Merge a batch of submitted dataflow programs into a single job so that common parts are executed only once.

materialization: Explicitly (by user) or implicitly (i.e. automatically) insert save/load actions to recycle intermediate results across jobs, similar to materialized views.

Recycling results is especially useful when computation power is expensive or limited, but storage is cheap and available to a large extent. This is often the case for rented cluster resources on cloud providers like Amazon, Google, or Microsoft Azure. For example, an Amazon Elastic MapReduce cluster has a per-hour pricing of around \$ 2 (c4.8xlarge, Region Frankfurt) but storage is currently only \$ 0.024 per GB (S3, Region Frankfurt). If the execution times of the jobs running on such clusters can be reduced significantly, users could save a considerable amount of money. Inspired by the concept of materialized views and adaptive indexing, we present in this paper an approach on materializing and recycling (intermediate) results in Apache Spark-based dataflow programs. The goal of our work is a transparent materialization and reuse of intermediate results to unburden the data scientist from decisions about costs and benefits of materializing results, aggregations, and potentially also indexes. For this purpose, we propose a cost-based decision model which relies on profiling information obtained by instrumenting the Spark[1] runtime environment.

The contribution of our work is twofold: (1) We present and evaluate strategies for transparently materializing and reusing intermediate results of dataflow programs for platforms like Spark. (2) We present a cost and decision model for materialization points leveraging platform features and code injection for runtime profiling.

2 Related Work

Caching and reusing intermediate query results has been extensively studied for relational databases and data warehouses. Selecting partial results, or views respectively, for materialization [4, 7] as well as rewriting queries using these views [6] are two closely related problems that are used in database systems to improve query response time.

[1] `https://spark.apache.org/`

Early works on reusing materialized views (or derived relations) were done by Larson and Yang in [9, 18]. Other research results on the view-matching problems for SQL queries were published in [1] or [15]. In [11] the Hawc architecture is introduced that extends the logical optimizer of an SQL system and considers the query history in order to decide which intermediate result may be worth materializing to speed up further executions – even if this would create a more expensive plan which, however, is executed only once. A related problem is automatic index tuning which has been studied extensively [8, 12, 13, 16]. Here, recommenders analyze the given workload and underlying data and recommend to or autonomously create and drop indexes.

For Hadoop MapReduce the MRShare framework [10] merges a batch of jobs into a new batch of jobs so that groups of jobs can share scans over input files and the Map output. This is similar to our merging strategy described earlier. Other projects such as ReStore [5], PigReuse [2], or [17] are similar to MRShare in the sense that they all merge a batch of scripts into a single plan or share the intermediate results after a map phase. In PigReuse, the optimization goal is to minimize the number of operators and the number of generated MapReduce jobs - but they do not analyze the total cost of the generated plans.

For Spark several additional frameworks were created to support data analysts with their tasks. KeystoneML [14] is able to identify expensive operations in machine learning pipelines on Big Data platforms like Apache Spark. They employ a cost model using cluster costs (such as network bandwidth, CPU speed, etc.) and operator costs to estimate total execution costs. From this physical operators for a logical plan are chosen and materialization points are determined. RDDShare [3] is also based on Spark and simply identifies common operators in a batch of Spark programs and merges them into a single program.

Our work differs from the mentioned approaches in a way that they either only focus on merging a batch of submitted scripts into a single job or they do not use a cost model for their algorithms. Furthermore, most related work is based on Hadoop MapReduce, except KeystoneML and RDDShare, which differs significantly from Spark's characteristics. However, KeystoneML focuses on choosing the best physical implementation of a logical operator and RDDShare only tries to merge a batch of scripts without reusing intermediate results over different runs.

3 Architecture overview

Figure 1 shows the general architecture overview of our approach. Independently from the used language (Pig Latin, Scala, Pyhton) and platform (Spark, Flink, Hadoop MapReduce), a dataflow program can be represented as a directed acyclic graph (DAG), where operators are the nodes and the directed edges are the links between these nodes that represent the dataflow. An optimizer component receives the DAG for the current job and after applying general rule-based optimizations, the DAG will be modified for recycling. We employ a cache that will store the materialized results. Ideally, the cache should have access to

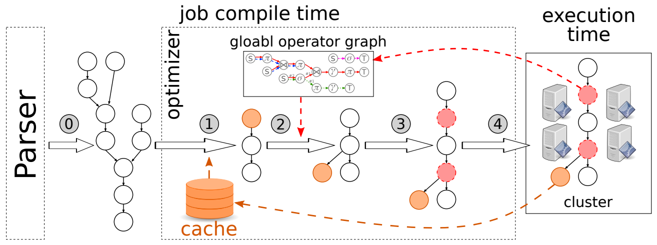


Fig. 1. Architecture overview: (0) Transform script to DAG, (1) Insert `LOAD` for existing data, (2) Insert `STORE` (based on statistics), (3) code instrumentation for profiling, (4) execute as Spark job.

HDFS for persistent storage. If materialized data exists for a part of the current DAG, we first will replace that part with a `LOAD` operator that reads the cached data. In the next step, the global operator graph (see Section 4.3) that stores profiling information is checked to determine if operators of the current DAG should be materialized and respective `STORE` operators are inserted. After that, we insert profiling operators to collect runtime statistics of the operators in the graph. During execution on a cluster, those operators will send information to the optimizer which will update its statistics. If intermediate results are to be materialized, those will be written to the cache by the inserted `STORE` operators.

We implemented the described cost-based decision model and profiler into our Piglet project: a parser and code generator from Pig Latin to Spark and other platforms. The code including the implementation of the cost model is available at our GitHub repository[2]. However, we would like to emphasize that the model described in this paper is neither restricted to Piglet nor Spark and could easily be adopted into other platforms. The details of the decision model that uses the information in the global operator graph as well as the cache will be explained in more detail in the next sections.

4 A Cost-based Decision Model

For our work we assume that in the DAG each operator o has an unique lineage identifier $\mathsf{lid}(o)$. This lid consists of the operator name (`LOAD`, `FILTER`, `GROUPBY`, ...) and parameter values (e.g. filter predicates) together with the lid of its direct predecessor(s). This lineage identifiers provide a simple way to decide if two operators are identical, i.e. read the same input and produce the same output. Currently, we use this approach as a simple replacement for query containment check which is hard or even undecidable in case of user-defined code. The goal of our cost model is to identify those operators in the DAG where materializing their intermediate results speeds up subsequent executions most. Figure 2 shows a DAG where the width of a node's box represents its processing time. If, e.g., the result of the second join operator is materialized, subsequent executions of

[2] `https://github.com/dbis-ilm/piglet`

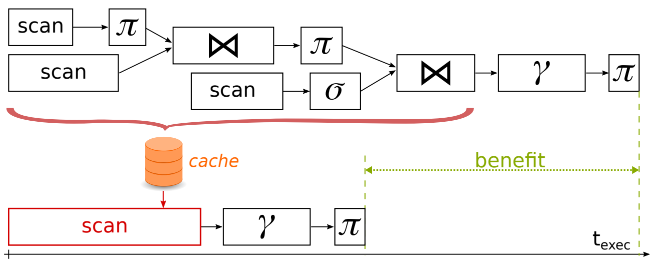


Fig. 2. Runtime difference for loading materialized result.

dataflow programs that also contain this part in their respective DAG will benefit by only having to load the already present result from disk. This leads to two basic questions that need to be answered:

(1) If multiple materialized results are applicable for reuse in a job, which of them should be loaded?
(2) The intermediate result of which operators in the current job are worth materializing so that a subsequent execution will benefit most?

In order to support these decisions, our model introduces materialization points, for which the benefits are calculated.

4.1 Materialization Point

A materialization point M is a logical marker in a DAG denoting a position for the decision model to write or load the materialized results. Here, we distinguish between *candidate materialization points* and *materialization points*. Candidate materialization points are those potential places in a DAG, where the intermediate result should either be materialized or could be loaded from storage. We denote the (candidate) materialization point representing the output of operator o_i by M_i, meaning that M_i is an alias for the output of operator o_i in the optimizer component. Every non-sink operator can be regarded as a candidate materialization point. Obviously, one cannot achieve any benefit from materializing the result of a source operator and thus, these materialization points need not be considered. We denote the set of all materialization points $M_1, M_2, \ldots, M_n$ which are currently kept in the cache as the *materialization configuration* $\mathcal{M} = \{M_1, M_2, \ldots, M_n\}$.

The decision model has to determine if the result of the respective operator is worth materializing or already materialized results can be loaded. Thus, materialization points are always a subset of these candidates.

4.2 Benefit

The decisions are based on the *benefit* regarding the execution time of the complete job with the main goal to minimize the overall execution time. Hence, the benefit is the amount of time saved when intermediate results can be loaded instead of executing the complete job, as depicted in Figure 2. Alternatively, one can also regard the benefit as the amount of money saved by needing to rent fewer machines in a public cloud.

To calculate the benefit, the decision model is based on the actual costs of operators which are measured during execution of a job. For an operator o_i uniquely identified by its $\mathsf{lid}(o_i)$ the following statistics are collected:

- **cardinality $\mathsf{card}(o_i)$:** Number of result tuples of o_i.
- **tuple width $\mathsf{width}(o_i)$:** The average number of bytes per result tuple of o_i.
- **execution time $t_{\mathsf{exec}}(o_i)$:** Duration it takes the operator to completely process its input data.

The benefit of a materialization point M_i can be expressed as in Equation (1).

$$t_{\mathsf{benefit}}(M_i) = t_{\mathsf{total}}(M_i) - t_{\mathsf{read}}(M_i) \tag{1}$$

$$t_{\mathsf{total}}(M_i) = \sum_{o \in \mathsf{prefix}(M_i)} t_{\mathsf{exec}}(o) \tag{2}$$

$t_{\mathsf{total}}(M_i)$ denotes the cumulative execution time of operators in the prefix of o_i from the source to M_i and $t_{\mathsf{read}}(M_i)$ is the time required to read the materialized data of M_i. If the prefix of M_i does not contain a join (or cross, etc.) $t_{\mathsf{total}}(M_i)$ can be calculated as in Equation (2). If the prefix of M_i does contain a join (or similar) operator j, with $k_1(j), \ldots, k_n(j)$ as the direct inputs to j, only the longest (concerning execution time) of those branches is considered:

$$t_{\mathsf{total}}(M_i) = \max\{t_{\mathsf{total}}(k_1(j)), \ldots, t_{\mathsf{total}}(k_n(j))\} + \sum_{\substack{o \in \mathsf{prefix}(M_i) \\ \wedge o \notin \mathsf{prefix}(j)}} t_{\mathsf{exec}}(o) \tag{3}$$

This means to take the maximum time of all input branches of the join operator and add the execution times of all other operators in the prefix of M_i which are not part of the join input, i.e. are located between j and o_i. The time to read the materialized result of M_i is calculated as:

$$t_{\mathsf{read}}(M_i) = \frac{\mathsf{card}(o_i) \cdot \mathsf{width}(o_i)}{bps} \tag{4}$$

The factor *bps* stands for the number of bytes that can be read per second and depends on the cluster setup as well as the underlying hardware and thus, is an installation-specific calibration factor.

4.3 Global Operator Graph

The benefit of each candidate materialization point is estimated based on execution of multiple jobs containing the corresponding operator instances. For this purpose, the statistics are maintained in a *global operators graph*, a DAG that was created by merging the DAGs of all ever submitted jobs. The graph is persistently stored and, therefore, available across multiple jobs.

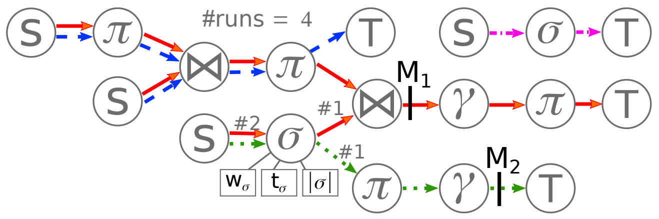


Fig. 3. The global operator graph for four jobs. Stored information on nodes and edges is exemplarily shown for a single operator node.

With each operator instance, identified by its lid, the collected runtime statistics are associated while with each edge its frequency of occurrence in all executed jobs is stored. Additionally, the global operator graph also contains the materialization points, i.e., statistics about the already executed operators and materialized results. Figure 3 shows an example of such a global operator graph for four jobs J_1 (solid red edges), J_2 (dashed blue), J_3 (dotted green), and J_4 (dashed purple)[3].

After the execution of a job finished and all statistics (runtimes and result sizes of the operators) have been collected (cf. Section 5), they are added to the respective nodes in the graph. If the operator is executed for the first time, no statistics are present for this operator and the collected values are simply added to the node in the graph. On the other hand, if the operator was already executed before as part of another job, present statistics are merged with the newly collected ones by averaging them.

The graph serves as input for the decision model and is used to calculate the benefit based on the statistics and materialization points.

4.4 Decision Model

The decision model is used to answer the two question posed in the beginning of this section.

Loading Existing Materialized Data Answering the first question is straightforward. From the list of candidate materialization points for a given job, only those are selected for which materialized results are present. Then, from these candidates, the one materialization point that will result in the highest benefit is chosen to achieve the greatest speedup. If the job contains multiple paths, which are then combined in a join (or similar), selection of multiple materialization point, one for each path, is possible.

Materializing Intermediate Results The decision model has three dimensions to consider when choosing a materialization point to actually write to persistent storage.

[3] In reality, in the model there is only one edge between the nodes. The multiple edges are just for illustrating the different jobs.

(1) Which candidate materialization points should be selected for further investigation?
(2) From the list of candidate materialization points resulting from (1), which of those should be materialized?
(3) If the persistent storage is limited in space, decide which existing materialized result has to be deleted. Note, that due to space limitations, we do not consider the dimension of cache eviction strategies here and assume an infinite cache.

Selection of Candidate Materialization Points. Obviously a sink operator is not a candidate for materialization as the result is either written to persistent storage anyway or printed to screen. In the latter case, the materialization point before the that sink operator would be one whose result could be re-used by another job. A source operator will only read data from storage and pass it to the next operator without modification. Hence, materializing the output of a source operator will write the same data back to disk and no benefit can be gained from this. Therefore, subsequent operations only need to consider candidate materialization points that do not belong to a source or sink operator.

Ranking Materialization Points. To decide which materialization point to really materialize, different strategies exist, which might be suitable for different use cases:

– **latest:** A naïve but intuitive strategy for selecting a materialization point is to always choose the last possible one. This is the one materialization point that is closest to a sink operator. If a job contains n sinks, the materialization point before each sink is selected, which means to write n intermediate results. This is a simple caching strategy and might work well during the incremental development of scripts, described earlier. However, this bears the "risk" that the materialized result will not be needed again, e.g., if subsequently executed scripts are not the next step of the incremental development, but branch off at another operator so that an earlier materialization point would have been a better choice. Furthermore, the last materialization point may only bring a small (or even no) benefit for reusing.
– **maxbenefit:** Therefore, another option is to choose that one materialization point with the highest benefit. Compared to the previous strategy, where the last one is selected, it is guaranteed to bring the best possible benefit when the result is needed again. Like in the previous strategy, if the result is not needed again, materialization was pointless.
– **markov:** Thus, selection of the materialization points should consider the probability for reuse – for which a Markov chain can be applied. In fact, we should regard this as a two dimensional (probabilities and benefits) optimization problem to maximize the benefit as well as the probability for reuse of the selected materialization points. This optimization problem is known as the Skyline or Pareto efficiency. The result of this optimization problem are all points where there exists no other point with both a higher benefit and higher probability. All points in the Pareto front mark materialization points with either

a high probability and/or high benefit, thus being worth materializing. If only one materialization point should be selected, it has to be chosen from the Pareto front. For this, the probability of re-occurrence of a materialization point can be considered as the weight for the benefit, so that the materialization point with the highest product of probability and benefit should be selected:

$$\begin{aligned}\{M_i \in \mathcal{M} \mid &\nexists M_j \in \mathcal{M}, i \neq j : \\ &P_{total}(M_j) * t_{benefit}(M_j) > P_{total}(M_i) * t_{benefit}(M_i)\}\end{aligned} \tag{5}$$

If this set contains multiple elements, one can be chosen arbitrarily or user specified weights can be applied to express a favor of one dimension over the other. $P_{total}(o_i)$ denotes the minimum probability found on the path in the DAG from the source operators to o_i:

$$P_{total}(o_i) = min\{P_{o_k,o_l} | o_k, o_l \in prefix(o_i), o_k \rightarrow o_l\} \tag{6}$$

where $o_k \rightarrow o_l$ means that o_l is a direct successor of o_k in the DAG. P_{o_k,o_l} describes the probability that o_k will be followed by operator o_l and can be calculated in different ways. One approach is to put the frequency into relation of the total number of executions, with respect to some time window $\mathcal{W}$. The probability P_{o_k,o_l} then would be as in Equation (7)

$$P_{o_k,o_l} = \frac{f^{\mathcal{W}}_{o_k,o_l}}{min(\mathcal{W}, runs)} \tag{7}$$

$$P_{o_k,o_l} = \frac{f^{\mathcal{W}}_{o_k,o_l}}{deg^{+}_{\mathcal{W}}(o_k)} \tag{8}$$

A second approach is to put the frequency in relation to the possible other operators that follow o_k, as shown in Equation (8). Here, $f^{\mathcal{W}}_{o_k,o_l}$ is the plain frequency count for the transition stored on the edges, that lie within the considered window $\mathcal{W}$, $runs$ is the total number of jobs that are executed by the system, and $deg^{+}_{\mathcal{W}}(o_k)$ is the outdegree of a node o_k.

5 Profiling Dataflow Programs

Result Size. Unlike traditional DBMS, in the Big Data field one often works with plain text files (log files, csv, ...) and no central data management system that has access to detailed statistics. Thus, in order to gain the desired information, there are two options: (1) The optimizer is started separately to analyze the input file and create a profile. Before executing a job, the optimizer then tries to come to a decision based on the statistics and selectivity estimations of the involved operators. (2) The job is instrumented with code that collects the necessary statistics during execution and the runtime or execution platform has to be extended by such an optimizer that manages and utilizes the collected data. The first approach is more or less what traditional DBMS do and is currently also being implemented in Apache Spark for Spark SQL. However, the second approach has the advantage that execution time is measured as well as the result size, instead of relying on estimations that are based on assumptions. In our evaluation we will show that the instrumentation does not incur in any significant overhead. To determine the total number of bytes in the result of an operator, and thus the size of the materialization point, we use Spark's `SizeEstimator` that estimates the number of bytes for a given object. We sample the result of the operator and pass each result tuple individually into the estimator. The information for each partition is accumulated using Spark's accumulator mechanism and send to the optimizer. From the received information the optimize can calculate the average tuple size as well as the total number of tuples in the result.

Execution Time per Operator. A more difficult task is to measure the execution time of an operator. In Spark, a job is divided into stages, where each stage contains a sequence of operators that can be executed without data shuffling. Shuffling happens when an operation needs data from several partitions, e.g., `COGROUP`. Operators in the same stage can be executed in one scan over the partition. Spark comes with a `SparkListener` interface that provides information about status of the current execution including start and completion time of the stages that form the job. However, relying on the execution times of the stages is too coarse for our goal as possible materialization points for recycling would be after a stage only. Thus, there would not be many of such materialization points and more importantly we would lose most operators that are shared between different jobs, because they are hidden inside a stage and thereby reducing the usefulness of the idea. We therefore implemented our own approach, based on code instrumentation, to measure the execution duration of an operator per partition. On the logical level, *timing* operators are inserted between all other operators in the plan, as depicted in Figure 4.

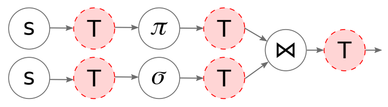


Fig. 4. Timing operators inserted into dataflow plan.

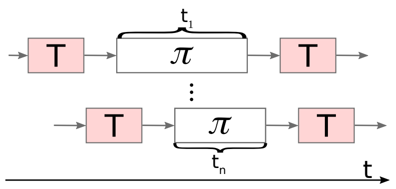
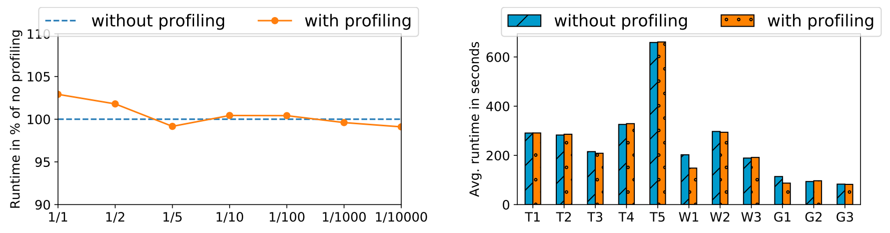


Fig. 5. Parallel execution of operations.

The task of the timing operators is to send a message to the profiling manager component of the optimizer with the current system time, when they are executed[4]. The profiling manager will receive a timestamp and the `lid` of the according operator for each partition and calculates the average execution time of the operators based on this information.

The realization of this concept needs to deal with Spark's lazy evaluation as well as the data parallelism. Thus, for each timing operator we inject code to report the current time when a partition is processed (using `mapPartitions`).

[4] This requires that the clocks on all nodes are synchronized, of course. For example via NTP.



(a) Varying sample rates.

(b) Overhead per script

Fig. 6. Overhead of code instrumentation.

The partitions that each operator instance processes can be of different sizes, although the platforms try to keep them balanced to avoid skewed workload on the nodes. Thus, the operator instances require different times to process their input. For n partitions, it results in n different execution time information, from which we have to derive an overall execution time for an operator (cf. Figure 5). In our approach we use the average of these n collected times. Other strageties (min, max, median), however, are also possible and *min* or *max* could be used to implement an optimistic or pessimistic behavior.

Since an RDD has information about its parents, it is enough to only insert this code after the operator and let the profiling manager calculate the execution duration, based on the received times of the respective parent operator.

6 Evaluation

To evaluate the proposed decision model we make the following three hypotheses:(1) Injected profiling code only introduces a negligible overhead. (2) The materialization decision always improves query execution time and avoids bad decisions. (3) The best strategy for selecting a materialization point for writing depends on the workload setting.

We use real world data from three use case scenarios: `weather` contains sensor data from the SRBench [19] benchmark for hurricane Katrina (180 mio. tuples, 34.7 GiB), New York `taxi` trip data[5] (Yellow Cabs, 2013 – 2016, 550 mio. tuples, 90 GiB) together with New York `block` data[6] (38,000 tuples, 18 MiB), and the `GDELT`[7] data from 2013 to 2016 (127 mio. tuples, 48 GiB). We created several scripts for each use case scenario: `T1` – `T5` for `taxi` scenario, `W1` – `W3` for `weather` scenario, and `G1` – `G3` for `GDELT`. The scripts and their DAG visualizations can be found in our GitHub repository[2]. Our Spark cluster consists of 16 nodes with: Intel Core i5 2.90 GHz, 16 GB DDR3 RAM, 1 TB disk, 1 GBit/s LAN. The cluster runs Hadoop 2.7, Spark 2.0.1, and Java 8u102. All experiments were repeated several times to remove outliers. To avoid caching effects, we executed a word count program between all executions of test scripts.

[5] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`
[6] `http://www1.nyc.gov/site/planning/data-maps/open-data.page`
[7] `https://www.gdeltproject.org/data.html`

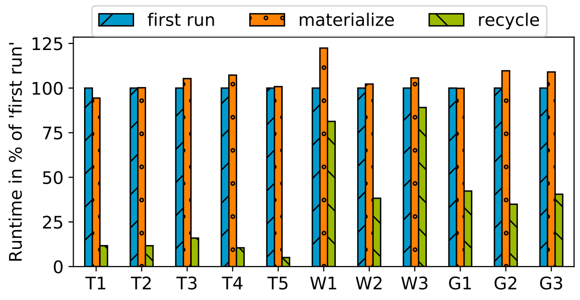
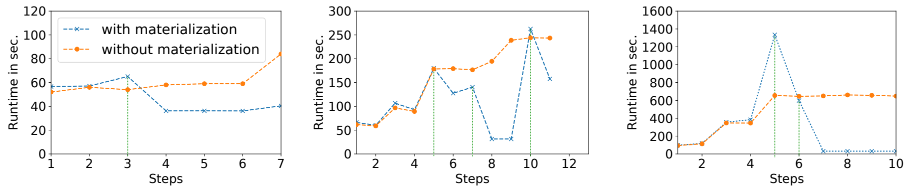


Fig. 7. Three executions times for each script showing the benefit of loading materialized results. Runtime in percent of the first execution.

Figure 6 shows the execution time without profiling as well as with code instrumentation for profiling for each of our test scripts. It can be seen that profiling incurs only a small overhead for a sample rate of 1/1 and 1/2 (meaning 100% or 50% respectively are selected) and runtimes only differ in less than 10 seconds or 5%. For our other experiments we selected a sample rate of 10%.

Figure 7 shows runtimes for each script for three executions. Prior to the first execution no statistics were available and thus are collected during this execution (blue bars). In the second execution, previously generated statistics were used to decide which operator to materialize (orange bars). Hence, this execution includes also the writing of the intermediate result. For the last execution the materialized results were loaded (green bars) and thereby reduced the overall execution time of the job.

As we argued in previous sections, scripts are often developed incrementally. In this experiment we show the runtime differences for one script of each scenario for incremental execution (Figure 8). We first ran the according script without materialization support (dashed orange lines) and compared the runtimes to an execution with materialization enabled (dotted blue lines). On the x-axis, step 0 is the initial execution with a `LOAD` and a first `FILTER` operator and subsequent steps add one or more operators. For the first few steps, both execution times are equal except for some minimal discrepancies. At some point however, the optimizer recognizes a materialization point for which a benefit will be achieved and writes the respective result to disk (indicated by vertical lines). In this step, the execution time for *with materialization* rises above the reference time *without materialization*. However, since this is executed only once, the additional costs (clearly visible in Figure 8(c)) will easily be amortized in subsequent executions that benefit from loading the materialized data. In fact, materialization reduced the cumulative execution time for `G1` from 7 to 5 minutes, for `W2` from 30 to 20 minutes, and for `T5` from 80 to 30 minutes. The estimation of the benefit is an important aspect of our approach. In our experiments we saw that the estimated benefits often were close to real measured speedups, but sometimes also deviated to some extent. We observed that in almost all cases when a benefit could be achieved, we underestimated it – meaning that a subsequent execution was even shorter than calculated. In our tests we never encountered the situation that



(a) Script G1 (b) Script W2 (c) Script T5

Fig. 8. Incremental execution of one script for each scenario (different y scales).

the optimizer calculated a benefit for a candidate materialization point which actually did not bring any benefit during execution. From this we conclude that our cost model calculates executions costs well enough to select an appropriate materialization point and avoid candidate materialization points that would cause longer execution times. Deviations are caused by our current implementation of the time measurement which depends on the information of a partition's parent(s). If the parent partitions could not be determined precisely the computation of the respective operators execution time may assume a shorter or longer time.

To test the impact of the selected strategy (cf. Section 4.4) on the performance, we looked at two cases: In the first case we used additional scripts that all share the first six operators and then diverge into their individual paths that all contain another five operations. Strategy `last` did not materialize data as the last candidate materialization point of a job will not be repeated and thus no job benefited from recycling and execution for each script took around 420 seconds (7 minutes). For strategies `maxbenefit` and `markov` intermediate results were recycled and execution time was around 160 secs (2:40 minutes) for both when results were loaded. In the second case we disassembled the jobs from our three use case scenarios into a total of 132 small jobs, executed them one after the other in a random sort order and captured the time it took to complete all jobs. For strategy `last` the execution time was 3:47 hours, while for `maxbenefit` and `markov` the total time was 2:52 hours and 2:48 hours, respectively. This shows that a selection strategy that takes the costs of operators into account achieves good results. In this setting `maxbenefit` and `markov` created similar results, but `markov` strategy performed slightly better in this last experiment as it sometimes chose other materialization point than `maxbenefit`, which more subsequent jobs could recycle.

7 Summary & Outlook

In this paper we presented an approach for a cost-based decision model to speed up execution of dataflow programs by merging jobs and reusing intermediate results accross multiple executions of the same or different jobs. The model was implemented in our Pig-to-Spark compiler Piglet which injects profiling code into the submitted jobs and rewrites them to materialize intermediate results or reuse existing results. In our evaluation we showed that profiling does not add any significant overhead to execution time, that jobs greatly benefit from reusing existing results, and that different strategies for choosing which intermediate result to materialize are needed.

In future work we will address the problem of cache replacement as there is no infinite space for storing the materialized results and existing data may need to be removed. Furthermore, our current implementation is based on the lineage information of operators and could be improved by implementing strategies for query containment checks. To address the fact that often machines are rented for data processing, the monetary costs of CPU cycles and storage may also be integrated into our cost model.

References

1. S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, pages 254–263, 1998.
2. Camacho-Rodríguez et al. PigReuse: A Reuse-based Optimizer for Pig Latin. Technical report, Inria Saclay, 2016.
3. H. Chao-Qiang et al. RDDShare: Reusing Results of Spark RDD. *DSC*, pages 370–375, 2016.
4. R. Chirkova, A. Y. Halevy, and D. Suciu. A Formal Perspective on the View Selection Problem. In *VLDB*, pages 59–68, 2001.
5. I. Elghandour and A. Aboulnaga. Restore: reusing results of mapreduce jobs. In *VLDB*, volume 5, pages 586–597, 2012.
6. A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 2001.
7. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing Data Cubes Efficiently. *SIGMOD Rec.*, 25(2):205–216, 1996.
8. S. Idreos et al. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):585–597, 2011.
9. P.-Å. Larson and H. Z. Yang. *Computing queries from derived relations: Theoretical foundation.* University of Waterloo. Department of Computer Science, 1987.
10. T. Nykiel et al. MRShare: sharing across multiple queries in MapReduce. *PVLDB*, 3(1-2):494–505, 2010.
11. L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531. IEEE, mar 2014.
12. K. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous Query-driven Index Tuning. In *VLDB*, pages 1129–1132, 2003.
13. K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-line Tuning. In *SIGMOD*, pages 793–795, 2006.
14. E. R. Sparks et al. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*, pages 535–546, 2017.
15. D. Srivastava, S. Dar, H. V. Jagadish, and A. Y. Levy. Answering queries with aggregation using views. In *VLDB*, volume 96, pages 318–329, 1996.
16. G. Valentin et al. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*, pages 101–110, 2000.
17. G. Wang and C.-Y. Chan. Multi-Query Optimization in MapReduce Framework. *PVLDB*, pages 145–156, 2013.
18. H. Z. Yang and P.-Å. Larson. Query Transformation for PSJ-Queries. In *PVLDB*, volume 87, pages 245–254, 1987.
19. Y. Zhang et al. SRBench: A Streaming RDF / SPARQL Benchmark. In *ISWC*, pages 641–657, 2012.