Have your Students build their own Mini Hive in just Eight Weeks

Stefanie Scherzinger*

OTH Regensburg, Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

Abstract. This paper summarizes a published report on *miniHive*, a Python-based prototype implementation of the SQL-on-Hadoop engine Hive. Master-level students at OTH Regensburg have built *miniHive* over the course of just eight weeks. *miniHive* compiles basic SQL queries into MapReduce workflows. These can then be executed directly on Hadoop. Like the original Hive, *miniHive* performs generic logical query optimizations (selection and projection pushdown, or cost-based join reordering), as well as MapReduce-specific optimizations. By building the query engine, the students learn about database systems implementation and gain an appreciation for the power of query optimizers. We share our experience as well as our code for building *miniHive* with the academic database community, and hope to inspire engaging discussions.

Keywords: Teaching database systems architecture · Hadoop · Hive.

1 Introduction

As for coming up with instructive coding exercises when teaching cloud database technologies, writing MapReduce jobs seems to be the state-of-the-art: In a survey conducted within the German-speaking academic database community [8], the majority of the participating lecturers reported to not only teach the theory of MapReduce processing (90%), but to also have students code MapReduce jobs (about 70%). Yet teaching how to write MapReduce jobs is a two-edged sword: One the one hand it makes for straightforward exercises and exam questions. On the other hand, the trend in big data processing is clearly towards declarative query languages. While it is crucial that students understand the principles behind MapReduce, it is unlikely that they will be making a living writing MapReduce functions. Accordingly, more than half of the survey participants reported that they also include declarative query languages such as HiveQL or Pig Latin in their syllabi. Inspired by this study, the author of this paper re-designed her Masters-level course “Modern Database Concepts” for the summer term of 2018. To provide her students with hands-on experience, they were asked to build *miniHive*, a prototypical SQL-on-Hadoop engine for compiling a fragment of SQL into MapReduce workflows. *miniHive* is written in Python and designed along Facebook's original demo of Hive at VLDB'09 [9].

In the following, we sketch the milestones in building *miniHive*. We refer to [7] for the extended version of this report.

2 Query Compilation and Optimization

miniHive is written in Python 3.6, Figure 1a shows the architecture, adapted from the generic architecture in [4]. Simple SQL statements (conjunctive queries and only equality comparisons in predicates, as discussed in the typical textbook chapter on query compilation, c.f. [4]) are parsed using the Python module `sqlparse`[1]. The query compiler then performs the canonical translation to relational algebra. To programmatically handle relational algebra queries, we instantiate the classes representing the operators selection, projection, cross product, join, and renaming from the interactive relational algebra interpreter `radb`, an open source Python module[2]. A first rewrite phase performs selection pushing, supported by a data dictionary, and further translates cross products into joins, where possible. (Projection pushing was left as an optional later task.)

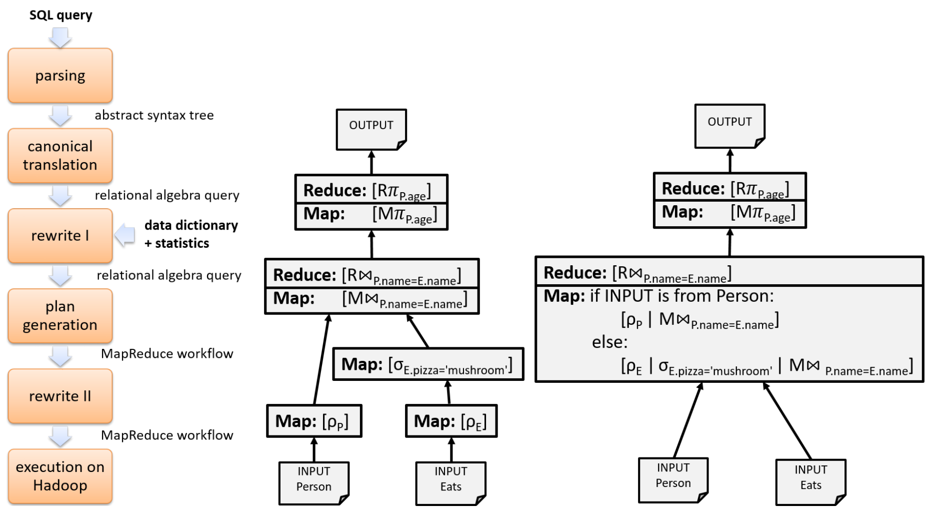
In plan generation, each relational algebra operator is encoded in MapReduce function code. The algorithms are comprehensively described in the textbook "Mining of Massive Datasets" [1]. The result is a workflow of Map-only and MapReduce jobs, managed using the popular Python module `luigi`[3]. This first version of a physical query plan can be immediately executed on Hadoop, which makes for a great sense of achievement. Like in Hive, the intermediary results of physical operators are stored in HDFS. Reducing the amount of intermediary data is the main optimization goal, as described next.

For optimization, it was recommended that the second rewrite phase performs chain folding, a generic and well-explored MapReduce design pattern [2, 3, 6, 9]. Chain folding can be as simple as folding sequences of Map-only jobs into a single stage. Fewer stages means fewer temporary files in HDFS, which reduces the overall communication costs (c.f. [1]). Again, the resulting MapReduce workflow can be directly executed on Hadoop.

The students were encouraged to implement further optimizations, by their own choosing. In the first rewrite phase, some added projection pushing or cost-based join reordering. Others also added MapReduce n-way joins (as described in [1]) to the plan generation phase.

Example 1. We next exemplify the four milestones towards a fully functional *miniHive* and consider a query over Jennifer Widom's pizza scenario[4]:

[1] Available at `https://github.com/andialbrecht/sqlparse`.
[2] Available at `https://github.com/junyang/radb`.
[3] Available at `https://github.com/spotify/luigi`.
[4] Online at `https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about`.



(a) Architecture. (b) Naive workflow. (c) Workflow after chain folding.

Fig. 1. In *miniHive*, SQL queries are compiled to executable MapReduce workflows.

```
SELECT DISTINCT P.age FROM Person P, Eats E
WHERE P.name = E.name AND E.pizza = 'mushroom'
```

The milestone 1 code parses and translates this query into relational algebra, as shown below using the straightforward `radb` syntax:

```
\project_{P.age} \select_{P.name = E.name and E.pizza = 'mushroom'}
  (\rename_{P:*}(Person) \cross \rename_{E:*}(Eats))
```

The first rewrite phase, implemented in milestone 2, yields:

```
\project_{P.age}
  (\rename_{P:*}(Person) \join_{P.name = E.name}
    (\select_{E.pizza = 'mushroom'} \rename_{E:*}(Eats)))
```

In the third milestone, the physical plan is generated. The output is a tree-shaped workflow of MapReduce jobs, as shown in Figure 1b. Renaming and selection can be realized as Map-only jobs. In the syntax used in this figure, this is denoted as "**Map:** $[\rho_{\mathsf{E}}]$" and "**Map:** $[\sigma_{\mathsf{E.pizza='mushroom'}}]$" respectively, where we first specify the type of the function (either **Map** or **Reduce**), and then the operator implemented. Join and relational projection (due to duplicate elimination) require a full MapReduce job. Let us consider the projection. The Map-job "**Map:** $[\mathsf{M}\pi_{\mathsf{P.age}}]$" emits key-value pairs where the key is the person's age. Then the Reduce-job "**Reduce:** $[\mathsf{R}\pi_{\mathsf{P.age}}]$" simply outputs all input keys.

The fourth milestone covers the second rewrite phase, where the students were asked to optimize their query engine. As a practical means for capturing the effects of optimization (without requiring access to a large Hadoop cluster), we measured the amount of intermediate data stored in HDFS. The most "bang" for one's money was to be gained by chain folding, as outlined previously: Figure 1c shows the physical query plan for our running example after chain folding. Now, renaming, selection and join are evaluated within a single MapReduce stage (symbolized by the Unix pipe operator). □

3 Summary and Outlook

Among the 60 students taking the final exam in 2018, 25 students built a working SQL-on-Hadoop engine (milestone 3), and 11 implemented all four milestones. This is impressive, as the term project was not mandatory. The project was again offered in 2019, with equal success.

A natural future extension to *miniHive* is to allow aggregation queries, since Hive is intended for data warehousing scenarios. As both `sqlparse` and `radb` already support aggregate queries, this does not require any customization of third-party libraries. Further, it would be instructive to add *partition pruning*: Provided that a Hive table is divided into several HDFS folders, based on partitioning attribute values (similar to building a cluster index), Hive can ignore data in irrelevant folders when evaluating selection predicates [9]. More sophisticated forms of indexing for Hadoop processing have by now been explored, e.g. [5], so this would be an opportunity to integrate more recent research results.

Access to materials: The *miniHive* material for students, including skeleton code and unit tests, is available at `https://github.com/miniHive/assignment`. To instructors, the complete course material, including a prototype solution, can be made available.

References

1. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd Ed. Cambridge University Press (2014)
2. Lim, H., Herodotou, H., Babu, S.: Stubby: A Transformation-based Optimizer for MapReduce Workflows. Proc. VLDB Endow. **5**(11), 1196–1207 (Jul 2012)
3. Miner, D., Shook, A.: MapReduce Design Patterns. O'Reilly Media, Inc. (2012)
4. Moerkotte, G.: Textbook query optimization. In: Building Query Compilers, chap. 2. Online draft from `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf` (Mar 2019)
5. Richter, S., Quiané-Ruiz, J.A., Schuh, S., Dittrich, J.: Towards Zero-overhead Static and Adaptive Indexing in Hadoop. The VLDB Journal **23**(3) (Jun 2014)
6. Sauer, C., Härder, T.: Compilation of Query Languages into MapReduce. Datenbank-Spektrum **13**(1), 5–15 (2013)
7. Scherzinger, S.: Build your own SQL-on-Hadoop Query Engine – A Report on a Term Project in a Master-level Database Course. SIGMOD RECORD, accepted for publication (Jun 2019)
8. Scherzinger, S., Thor, A.: Cloud-Technologien in der Hochschullehre - Pflicht oder Kür? - Eine Standortbestimmung innerhalb der GI-Fachgruppe Datenbanksysteme. Datenbank-Spektrum **14**(2), 131–134 (2014)
9. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., et al.: Hive: A Warehousing Solution over a Map-Reduce Framework. Proc. VLDB Endow. **2**(2), 1626–1629 (Aug 2009)