Have your Students Build their own mini Hive in just eight Weeks

Stefanie Scherzinger

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

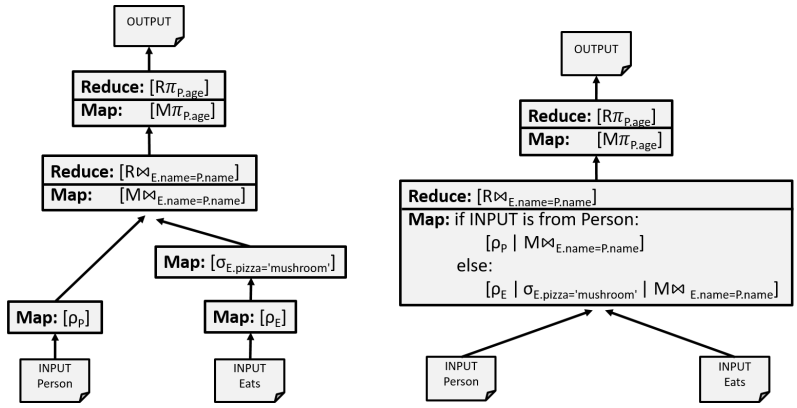
Abstract. This paper summarizes a published report on miniHive, a Python-based prototype implementation of the SQL-on-Hadoop engine Hive. Master-level students at OTH Regensburg have built miniHive over the course of just eight weeks. miniHive compiles basic SQL queries into MapReduce workflows. These 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 [10], 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 actually designed along Facebook's original demo of Hive at VLDB'09 [11].

In the following, we sketch the milestones in building miniHive. We refer to [9] for the extended version of this report.



(a) Naive plan. (b) Plan after chain folding.

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

2 Query Compilation and Optimization

miniHive is written in Python 3.6. Simple SQL statements (conjunctive queries and only equality comparisons in predicates, as discussed in the typical textbook chapter on query compilation [7]) are parsed using the popular 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 open source Python module `radb` [12] (an interactive relational algebra interpreter). The students then wrote code for carrying out selection pushing and translating cross products into joins, where possible (projection pushing was left as an optional task for later).

Next, each relational algebra operator is encoded in MapReduce function code. The algorithms are comprehensively described in the textbook "Mining of Massive Datasets" [3]. The result is a workflow of Map-only and MapReduce jobs, and managed using the popular Python module `luigi` [5]. This first version of a physical query plan can already be executed on Hadoop, which makes for a great sense of achievement. Like in Hive, the intermediary results of the physical operators are stored in HDFS. Reducing the amount of data stored temporarily is the main optimization goal, as described next.

For optimization, it was recommended that students at least implement chain folding [6] and thus merge sequences of Map-only jobs into the next MapReduce job upstream. This effectively reduces the number of temporary files stored in HDFS between each MapReduce stage (to use Hive terminology). Some students additionally implemented projection pushing, cost-based join reordering, and even MapReduce n-way joins (as described in [3]).

We next examplify the four milestones towards a fully functional miniHive.

Example 1. We consider a query over Jennifer Widom's pizza scenario[1]:

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

The milestone 1 code 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))
```

Pushing selections and introducing joins 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, logical operators are mapped to physical, MapReduce-based operators. The output is a tree-shaped workflow of MapReduce jobs, as shown in Figure 1a. 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 state the operator implemented in brackets.

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.

In the fourth milestone, 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 with rewriting the workflow of MapReduce jobs, merging jobs into multi-functional stages. Chain folding is considered a generic MapReduce design pattern [6]: it is described in the original Hive paper [11], and has also been motivated in [8], and benchmarked in [4]. Chain folding can be as simple as folding sequences of Map-only jobs into a single stage. As an immediate result, we store fewer temporary files in HDFS. This evidently reduces the overall communication costs (along the notion defined in [3]), and accordingly, the elapsed wall-clock time. In Figure 1b, we show 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 On Effort and Success

Among the 60 students taking the final exam in 2018, 25 students built a working SQL-on-Hadoop engine (milestone 3). This is impressive, as the term project was

[1] `https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about`

not mandatory and students could only earn bonus points towards their exam. Also, the miniHive project stretched over just eight weeks, a sporty pace.

Since effort goes two ways and does not only concern students, some words on the effort for the instructor: The author had no supporting staff available in teaching this class. Thus, prototyping miniHive in Python, finding suitable libraries and writing skeleton code as well as unit tests and auotmated testing scripts, was a one-person job. While conducting the project was indeed time-intensive, it was also very rewarding, and overall, surprisingly feasible. The miniHive project is therefore offered again this summer term.

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.

Acknowledgements: miniHive is greatly inspired by my experience from an earlier database implementation term project [2]. Of course, I also owe my students at OTH Regensburg. Their enthusiasm, cooperation, and ambition made miniHive a success.

References

1. Albrecht, A.: python-sqlparse - Parse SQL statements (2019), available at `https://github.com/andialbrecht/sqlparse`.
2. Koch, C., Olteanu, D., Scherzinger, S.: Building a native XML-DBMS as a term project in a database systems course. In: Proceedings of XIME-P'06 (2006)
3. Leskovec, J., Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets, 2nd Ed. Cambridge University Press (2014)
4. Lim, H., Herodotou, H., Babu, S.: Stubby: A Transformation-based Optimizer for MapReduce Workflows. Proc. VLDB Endow. **5**(11), 1196–1207 (Jul 2012)
5. Spotify AB: luigi (2018), a Python package, available as open source at `https://github.com/spotify/luigi`.
6. Miner, D., Shook, A.: MapReduce Design Patterns. O'Reilly Media, Inc. (2012)
7. 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)
8. Sauer, C., Härder, T.: Compilation of Query Languages into MapReduce. Datenbank-Spektrum **13**(1), 5–15 (2013)
9. 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)
10. 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)
11. 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)
12. Yang, J.: RA (radb): A relational algebra interpreter over relational databases (2019), available at `https://github.com/junyang/radb`.