Build your own SQL-on-Hadoop Query Engine

A Report on a Term Project in a Master-level Database Course

Stefanie Scherzinger
Ostbayerische Technische Hochschule Regensburg
Regensburg, Germany
stefanie.scherzinger@oth-regensburg.de

ABSTRACT

This is a report on a course taught at OTH Regensburg in the summer term of 2018. The students in this course built their own SQL-on-Hadoop engine as a term project in just 8 weeks. *miniHive* is written in Python and compiles SQL queries into MapReduce workflows. These are then executed on Hadoop. *miniHive* performs generic query optimizations (selection and projection pushdown, or cost-based join reordering), as well as MapReduce-specific optimizations.

The course was taught in English, using a flipped classroom model. The course material was mainly compiled from third-party teaching videos. This report describes the course setup, the *miniHive* milestones, and gives a short review of the most successful student projects.

1. MOTIVATION

When taking a big data course at an applied university of sciences such as OTH Regensburg, students expect a hands-on, coding-intensive experience. Since the majority of our students pursues a career in industry, they expect to get in touch with technology that will be an immediate asset for their CVs. Currently, this seem to be the Apache projects HDFS, Hadoop, Hive, and Spark.

Yet to the first-time user, interacting with HDFS may just feel like a simple file system. "Teaching HiveQL" is tricky, too: For the student fluent in SQL, writing first HiveQL queries seems unspectacular. Of course, these first impressions are treacherous, as there are language features in HiveQL that require a deeper understanding of the MapReduce data flow (e.g., SORT-BY versus ORDER-BY).

In designing her Master-level course titled "Modern Database Concepts", the author of this report wanted to teach the ideas behind engines like Hive, as well as the design decisions regarding query language constructs.

The students were therefore asked to build *miniHive*, an SQL-on-Hadoop engine for compiling SQL queries into MapReduce workflows. *miniHive* was designed according to the original presentation of Hive as a VLDB demo [14] in 2009: This version of Hive supported no updates, used an internal algebra to represent query plans, and could perform common logical optimizations (in particular, selection and projection pushdown). Back then, all physical operators were implemented as MapReduce jobs. As a MapReduce-specific optimization, Hive merges jobs using a technique called chain folding [11].

Among the 60 students taking the final exam, 25 students built a working SQL-on-Hadoop engine, which compiles SQL queries, performs generic logical optimizations, and executes them on Apache Hadoop. This is impressive insofar as the term project was optional, and stretched over just 8 weeks. Moreover, 11 submissions of *miniHive* also implemented chain folding among further optimizations.

Structure. In the following, we describe the course and its term project. Section 2 outlines the development of *miniHive* in four milestones, as well as how the students then perceived working with Apache Hive and Apache Spark. Section 3 describes the testbed and evaluation. Section 4 concludes.

2. THE FOUR MILESTONES

In "flipping" the course, the instructor relied on students to prepare the required theory on their own. Each week, they were assigned videos or book chapters. While studying the material, they answered a set of questions and submitted their answers online, prior to class. Since the students were allowed to take these notes into the final exam as reference material, they were motivated to diligently compile their answers.

During the weekly classroom sessions, the instructor and the students revised the prepared notes together, and worked on paper-based exercises to practice and apply the material.

During the lab sessions and in the students' own time, *miniHive* was built with Python 3.6 in four successive milestones. The deadlines for submitting

the milestones were spaced two weeks apart, which admittedly, is a sporty pace. Milestone specifications came with unit tests that submissions had to pass. Successful submissions were awarded bonus points that counted towards the exam.[1]

We now describe the scope of each milestone, the required material for self-study, and the coding challenges, in turn. We then report on our observations how students interacted with Apache Hive and Spark, having already built *miniHive*.

2.1 Compiling SQL to Relational Algebra

Scope: In the first milestone, the students implemented the canonical translation of conjunctive SQL queries into relational algebra. In particular, we support the fragment of queries of the form

```
SELECT DISTINCT ⟨ list of attributes to select ⟩
FROM   ⟨ list of relation names ⟩
[ WHERE  ⟨ condition ⟩ ]
```

where the condition is a conjunction of atomic equality conditions. Different from Hive, *miniHive* does not support nested relations. The translation into an equivalent relational algebra expression was intended as a warm-up exercise for students new to Python. For instance, the following query over Jennifer Widom's pizza scenario [7] produces the ages of all persons who eat mushroom pizza:

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

The compilation of this query into relational algebra is by the book. Below, we make use of the straightforward syntax of the `radb` interpreter for relational algebra [15]. This interactive interpreter was written by Jun Yang from *Duke University* and is a great teaching tool.

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

Several MapReduce-specific algebras have been proposed that provide powerful operators, e.g. [13]. However, this author chose to settle with traditional relational algebra, which is taught as part of the undergraduate database course at OTH Regensburg.

Independent Study: In advance, students taught themselves Python with a free course offered on the Udacity MOOC platform [10]. Moreover, they watched Jennifer Widom's video lectures for a refresher on relational algebra, offered on Stanford's MOOC platform Lagunita [7]. This teaching unit comes with interactive exercises that use the very same `radb` syntax as above.

The video lectures were to be completed over the first four weeks of the semester. The author took great care in choosing appealing material. Indeed, in the course evaluation, several students stated that they very much enjoyed taking these altogether excellent online courses.

Coding: The students then set out to compile SQL. As a query parser, they used the Python module `sqlparse` [1]. With over 50 contributors and over 1,600 stars on GitHub, this is a popular SQL parser.

As datastructures to represent relational algebra queries, the students simply used the Python classes declared within the `radb` source code [15].

2.2 Selection Pushdown

Scope: In the second milestone, the students performed selection pushing on the relational algebra queries, and translated cross products into joins, where possible. For this milestone, a data dictionary was provided. Selection pushing is also a key feature in first public release of Hive, as described in [14], whereas projection pushing, also included in the first release of Hive, was left as an optional feature for the final milestone.

In the example from before, this yields the following equivalent query in `radb` syntax:

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

Independent Study: For the theory on logical query optimization (selection and projection pushdown, as well as cost-based join reordering), the students followed parts of Jens Dittrich's flipped database course, which comes complete with in-class quizzes and exercises [3]. This material covers more than what is necessary for milestone 3, but is also a basis for the final milestone, where students could choose which optimizations to implement.

Coding: Coding for this milestone mainly involved recursive rewriting of the relational algebra trees. This gave the students the opportunity to familiarize themselves with the `radb` module.

2.3 A First Physical Query Plan

Scope: In the third milestone, logical operators were mapped to physical, MapReduce-based operators. The output is a tree-shaped workflow of

[1]Examination regulations at OTH Regensburg for this course require that the final grade is determined by a written exam. They further prohibit that the final grade is earned in part by an assignment. Thus, bonus points are our incentive for students to write code.

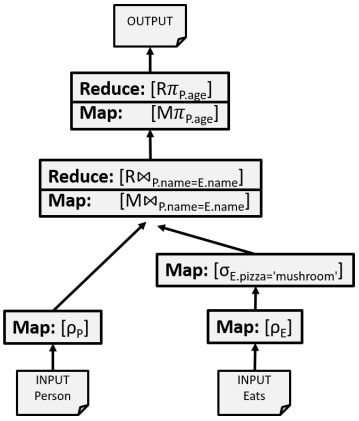

Figure 1: Naive physical query plan: Each node implements a single operator.

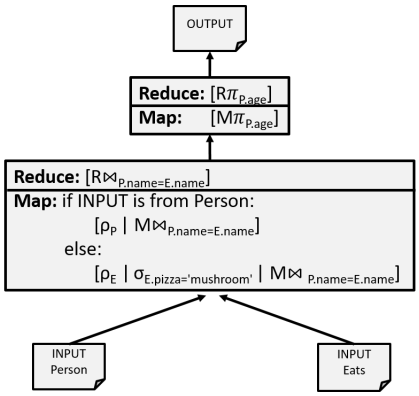

Figure 2: Plan after chain folding.

MapReduce jobs. Figure 1 shows the physical query plan for our running example. The data flow is from bottom to top. Renaming and selection can be realized as Map-only jobs. In the syntax used in this figure, this is denoted as "**Map:** [ρ_{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 relational algebra operator implemented in brackets.

In contrast, join and relational projection (due to duplicate elimination) require a full MapReduce job. Let us consider the final MapReduce job implementing the projection. The implementation of the Map-job, which we denote as "**Map:** [$\mathsf{M}\pi_{\mathsf{P.age}}$]", will emit key-value pairs where the key is the person's age. The Reduce-job "**Reduce:** [$\mathsf{R}\pi_{\mathsf{P.age}}$]" simply outputs the unique key from its input.[2]

The resulting workflow can be immediately executed on Hadoop, to naively evaluate SQL queries. Like in Hive, the intermediary results of the physical operators in the query plan are stored in HDFS.

Independent Study: The students familiarized themselves with MapReduce processing by taking an online course offered by Cloudera [9]. This course also uses Python and contains a set of introductory MapReduce coding exercises. MapReduce jobs can be run on Hadoop on a Linux-based virtual machine, the Cloudera Quickstart VM[3].

Implementing relational algebra operators with MapReduce is comprehensively described in the book "Mining Massive Datasets" [5].

Coding: The students were started off with skeleton code that translates `radb`-encoded relational algebra expressions to a `luigi`-managed workflow of MapReduce jobs. `luigi` [8] is a Python package for building pipelines of long-running batch processes. `luigi` supports several execution platforms, and among them, Hadoop. The students were already provided with this code. Thus, they could focus on implementing the code stubs for realizing selection, projection, renaming, and join.

Figure 3 shows the skeleton code for a `luigi`-task that implements the relational selection-operator as a Map-only job on Hadoop. All that remains for the students to do is to flesh out lines 25 through 29, having ready access to the selection predicate (see line 22) represented with `radb` datastructures. The `mapper`-function is invoked once for each line of input, which is then parsed into a key-value pair.

The input is formatted as shown in Figure 4, consisting of the relation name as the key and the JSON-encoded tuple as its value. Like in the original Hive, all intermediate results are stored in HDFS before they are processed by the next operator.

The code skeleton supports three mode of operandi: (1) Reading and writing to main memory only, and merely mocking a cluster-based execution environment. This makes unit testing easy, hermetic, and fast. (2) Reading and writing to local disk, rather than HDFS, and again mocking a cluster-based environment. This is the development mode, with quick turnaround times and the option to inspect all intermediate data in local files. Moreover, the development mode does not require an HDFS or Hadoop installation. Finally, (3) reading and writing to HDFS, and running on a Hadoop cluster in the Cloudera Quickstart VM. This was the intended production mode.

Switching between these modes with a runtime flag, the students experienced the pain of debug-

[2]We chose this involved syntax in preparation for chain folding, as shown in Figure 2, where code from different jobs is merged into stages. This allows us to track which parts of the code go where.

[3]Available at `https://www.cloudera.com/downloads/quickstart_vms/5-13.html`.

```
# SelectTask implements selection as a Map-only job.
class SelectTask(luigi.contrib.hadoop.JobTask):

  # The radb-encoded relational algebra expression
  # that this operator evaluates, serialized as a string
  # of the form "\select_{cond}(R)".
  qs = luigi.StringParameter()

  # Omitting some luigi/workflow-specific code.
  ...

  def mapper(self, line):
    # Parses the input line into a key-value pair,
    # where the key is the relation name and the
    # tab-separated value is a JSON-encoded tuple.
    relation, tuple = line.split('\t')
    json_tuple = json.loads(tuple)

    # Deserializes the query string into an radb expression
    # to gain access to the selection condition.
    query = radb.parse.one_statement_from_string(self.qs)
    condition = query.cond


    ''' .... fill in your code below ....'''

    yield("foo", "bar") # To be replaced with your code.

    ''' .... fill in your code above .... '''
```

Figure 3: The `luigi` skeleton code for the selection operator from relational algebra.

```
Person  {"name": "Amy", "age": 16, "gender": "female"}
Person  {"name": "Ben", "age": 21, "gender": "male"}
...     ...
```

```
Eats    {"name": "Amy", "pizza": "pepperoni"}
Eats    {"name": "Amy", "pizza": "mushroom"}
Eats    {"name": "Ben", "pizza": "pepperoni"}
Eats    {"name": "Ben", "pizza": "cheese"}
...     ...
```

Figure 4: The pizza data instance [7] encoded as key-value pairs. The key is the relation name, the value is a JSON-encoded tuple.

ging in a distributed environment: Just because the unit tests passed and everything worked fine in development mode is no guarantee that their implementation succeeds in the production environment (often due to careless use of global variables, or proprietary packages not available in the production environment). Digging through the logs and troubleshooting Hadoop turned out to be cumbersome, which in itself is a good learning experience.

2.4 Beyond Selection Pushdown

Scope: With the third milestone, the students had already built a working SQL-on-Hadoop engine. Yet since each MapReduce stage only evaluates a single relational algebra operator, and reads its input from HDFS, the runtimes are unnecessarily high.

Thus, in the fourth milestone, the students were asked to optimize their query engine. While the earlier milestones came with tight specifications, the students could now decide for themselves which optimizations to implement. As an incentive, the top ranking solutions would receive extra points.

It was stated as a requirement that for a predefined set of queries over TPC-H data, the students had to beat their milestone 3 implementation in 75% of the cases. These queries had been engineered such that the students would see the benefits of the optimization techniques discussed in class. The students were further provided with the cardinality estimates from the official TPC-H benchmark. For instance, execution of the following query

```
SELECT DISTINCT CUSTOMER.C_CUSTKEY
FROM CUSTOMER, NATION, REGION
WHERE CUSTOMER.C_NATIONKEY = NATION.N_NATIONKEY
  AND NATION.N_REGIONKEY = REGION.R_REGIONKEY
```

benefits from projection pushdown (provided the pushed projections do not remove duplicates), in combination with chain folding. Moreover, reordering the joins, so that the relations with smaller cardinalities (`REGION` and `NATION` have only 5 and 25 tuples respectively, whereas the `CUSTOMER` relation contains a multiple of 150K tuples, depending on the scale factor chosen when generating the data) are joined first. This effectively reduces the costs for storing intermediary results in HDFS.

As a practical means for capturing the effects of optimization, we measured the amount of intermediate data stored in HDFS. Of course, it would have been great to actually benchmark the students' solutions on a Hadoop cluster. Yet since the course was taught without any supporting staff, the instructor had to find a way to limit the administrative overhead for validation: In the development mode of *miniHive*, where all data is stored as local files on disk, measuring the data temporarily stored in HDFS can be realized with basic shell script commands. Also, the metric chosen is roughly aligned with the communication costs introduced in [5].

Independent Study: In preparation to the final milestone, several optimizations, some of them MapReduce-specific, were addressed:

(1) *Chain Folding.* The most "bang" for one's money was to be gained with rewriting the workflow of MapReduce jobs by merging several jobs into multi-functional stages. This approach is sketched in the original Hive paper [14], and has meanwhile been explored systematically in academic research, e.g. also motivated and described in [13] and benchmarked in [6]. This is considered a generic MapReduce design pattern also among practitioners [11].

We go by the terminology of [11] and refer to this strategy as *chain folding*. By chain folding, which can be as simple as collapsing sequences of Map-only jobs, we need to store fewer temporary files in HDFS. This evidently reduces the overall communication costs, and accordingly, the elapsed wall-clock time. In Figure 2, we show the physical query plan for our running example after chain folding. Now, renaming, selection and join are evaluated within a single stage (symbolized by the Unix pipe operator).

(2) *Projection Pushdown.* Projection pushdown only makes sense in combination with chain folding, provided that the pushed projections do not eliminate duplicates and therefore can be implemented as Map-only jobs. These can be merged in subsequent chain folding. Otherwise, adding blocking Reduce jobs drives up the communication costs.

(3) *Multi-way Joins.* Besides Reduce-side joins, we further discussed multi-way joins, covered in [5].

Coding: For the final milestone, the students were provided with a list of queries over TPC-H data, together with the cardinality estimates for this data model. The most successful student submission implemented optimizations (1) through (3) from above, as well as cost-based join reordering, heuristically joining the relations with lower cardinalities first.

2.5 Moving from *miniHive* to Apache Hive

Included in Cloudera Quickstart VM is an installation of Apache Hive, as well as Spark. Towards the end of the term, students interacted with these systems. By then, they had gained an appreciation for the scalability of Hive. Moreover, the students could now make sense of the output of Hive's EXPLAIN statements, and had an easier time understanding certain design decisions, such as Hive trying to avoid MapReduce jobs in query compilation. For instance, a simple exploratory query like "`SELECT * FROM Person LIMIT 10`" can be evaluated without spinning up MapReduce jobs, just by scanning a single chunk of the input file on HDFS.

Having implemented eager query evaluation in *miniHive*, the students now understood how lazy evaluation, as implemented in Spark, can make for a great interactive user experience.

3. TESTBED AND EVALUATION

Figure 5 summarizes the number of submissions. In total, 60 students participated in the final exam. The majority of these students also submitted a solution to milestone 1. A Python script was used to unpack the submitted `zip` files, run the unit tests, and to cross-check solutions with `pycode-similar`,

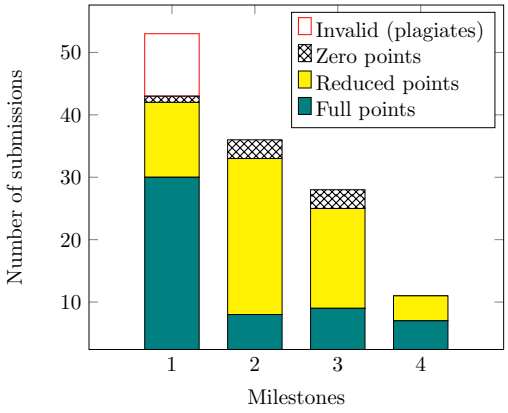

Figure 5: Submissions and points earned for the milestones in a class of 60 students.

Rank	Improvement
Top 1	4.6×
Top 2	2.9×
Top 3	2.8×

Figure 6: Improvements in milestone 4 over milestone 3 by the top-3 solutions.

a simple plagiarism checker.[4] This immediately revealed 10 submissions as plagiates, which their authors also admitted to.[5] In consequence, these students received no points for their submissions.

One milestone 1 submission was awarded no points, since the majority of unit tests had failed. A total of 30 submissions received full points, having passed all unit tests. 12 submissions further received reduced points, since non-public unit tests had failed. These tests checked for simple syntactic variations of the provided queries, and revealed submissions where students had not tested their code diligently. The students had been made aware that their submissions would undergo non-public unit tests.

Fewer students made submissions for the later milestones, which is owed to the fact that submissions were time-intensive, optional, and awarded only with bonus points. Nevertheless, the enthusiasm of the participating students remained high, and with milestone 3, still 40% of the students submitted a working SQL-on-Hadoop engine.

Figure 6 lists the improvements achieved by the top 3 submissions to milestone 4. As described earlier, we compare the size of data written into tem-

[4] `https://github.com/fyrestone/pycode_similar`.

[5] Apparently, code plagiarism is quite common in computer science education, a topic also noted by the press, e.g. [2]. It is a debate whether this problem is specific to our field, or merely revealed by tool-based checks.

porary files into HDFS against the milestone 3 implementation. As the specifications for milestone 3 were tight, all successful milestone 3 submissions had the same baseline regarding costs. The best *miniHive* engine achieved a 4.6× improvement, using all optimizations discussed in Section 2.4.

4. SUMMARY AND OUTLOOK

In their anonymous course feedback, the students described *miniHive* as work-intensive. When asked about the time spent on the individual milestones, the students reported anything between five and forty hours per milestone. It is up to debate whether this may be seen as evidence of the "factor 10 coder" effect, a controversial theory among developers stating that some coders are more effective by a factor 10. At the same time, the majority of students who participated in the project stated that they believed they had learned a lot. The students who completed *miniHive* where literally ecstatic about the improvements that they had achieved.

While the term project was not mandatory, the students who did well with *miniHive* also excelled in the final exam, since they had acquired a solid understanding of MapReduce processing and the associated communication costs. It is one thing to learn about the theory of query optimization, it's another thing to watch your own code perform the magic.

As a future feature for *miniHive*, it would be very instructive to add *partition pruning*, as also implemented in the first version of Hive from 2009: Provided that a Hive table is partitioned into several HDFS folders, based on attribute values (similar to building a traditional cluster index), Hive can ignore irrelevant folders in evaluating selection predicates. Indexing for Hadoop processing has, of course, also been explored in research, e.g. [12], so this would be an opportunity to integrate more recent research results into class. Experiencing the speedups achievable by indexing would be a further valuable learning experience.

The *miniHive* material for students, including the assignment descriptions as well as skeleton code and unit tests, is available at: `https://github.com/miniHive/assignment`.

To instructors, the complete course material, including a prototype and selected student solutions, can be made available upon request.

Acknowledgements: In 2006, my PhD advisor Christoph Koch taught a database systems course at *Saarland University*, where students built a native XML database that could evaluate a practical fragment of XQuery [4]. The *miniHive* term project is inspired by this experience.

5. REFERENCES

[1] A. Albrecht. python-sqlparse - Parse SQL statements, 2019. A Python package, available as open source at `https://github.com/andialbrecht/sqlparse`.

[2] J. Bidgood and J. B. Merrill. As Computer Coding Classes Swell, So Does Cheating. New York Times, May 29, 2017.

[3] J. Dittrich. *Patterns in Data Management: A Flipped Textbook*. CreateSpace Independent Publishing Platform, 2016.

[4] C. Koch, D. Olteanu, and S. Scherzinger. Building a native XML-DBMS as a term project in a database systems course. In *Proceedings of XIME-P'06*, 2006.

[5] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[6] H. Lim, H. Herodotou, and S. Babu. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *Proc. VLDB Endow.*, 5(11):1196–1207, July 2012.

[7] Jennifer Widom. Database Mini-Courses, 2014. Online course, available at `https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about`.

[8] Spotify AB. luigi, 2018. A Python package, available as open source at `https://github.com/spotify/luigi`.

[9] Udacity Inc. Intro to MapReduce and Hadoop, 2018. Online course, available at `https://classroom.udacity.com/courses/ud617`.

[10] Udacity Inc. Introduction to Python Programming, 2018. Online course, available at `https://classroom.udacity.com/courses/ud1110`.

[11] D. Miner and A. Shook. *MapReduce Design Patterns*. O'Reilly Media, Inc., 2012.

[12] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards Zero-overhead Static and Adaptive Indexing in Hadoop. *The VLDB Journal*, 23(3):469–494, June 2014.

[13] C. Sauer and T. Härder. Compilation of Query Languages into MapReduce. *Datenbank-Spektrum*, 13(1):5–15, 2013.

[14] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, et al. Hive: A Warehousing Solution over a Map-Reduce Framework. *Proc. VLDB Endow.*, 2(2):1626–1629, Aug. 2009.

[15] J. Yang. RA (radb): A relational algebra interpreter over relational databases, 2019. A Python package, available as open source at `https://github.com/junyang/radb`.