Implementing the TPC-H benchmark in a dataflow platform

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 306b

Systems Group, Department of Computer Science, ETH Zurich

Implementing the TPC-H benchmark in a dataflow platform

by
Goerens Olivier

Supervised by
Müller Ingo, Koutsoukos Dimitrios

18 / 03 / 2020 - 10 / 07 / 2020

inf | Informatik
Computer Science

Abstract

An increasing number of hardware platforms and processing tools arose during the last years to handle the ever increasing amount of data. However, the vast majority of data systems only achieve their best performance for specific hardware platforms. Here, the “Collection Virtual Machine" comes into play. It works across frontend languages and hardware platforms. In this thesis, we implement the TPC-H benchmark in CVM to analyse its performance and compare it to other database systems. During this process, we determine CVM's limitations which hinder the benchmark implementation and try to overcome these limitations by extending CVM.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Data nowadays originates in various formats and sizes from tons of different sources. Analysing this data has become an essential task and over the years many distinct data processing tools and software accumulated to handle and process these diverse workloads. Most of these systems are specialized for a specific combination of analytics and dataset size. Concurrently, the diversity and amount of hardware platforms has increased longing for ever increasing performance. However, the vast majority of the data systems achieve their best possible performance only for specific hardware platforms. This fixed setup leads to a lot of reimplementation effort when switching the underlying hardware to use a different processing tool.

To overcome this situation, the Systems Group of ETH Zurich proposes the "Collection Virtual Machine" (CVM), a generic data processing platform that can work across frontend languages and hardware platforms. "The framework relies on the hypothesis that most hardware platforms and types of data analysis are similar enough to be expressed in intermediate representations (IRs) based on the common abstraction of (nested) transformations of (nested) collections." [1] Throughout the thesis, we work on top of this existing platform.

The goal of the thesis is to evaluate CVM's performance and determine if it can compete with leading database systems such as HyPer. Therefore, we implement the TPC-H benchmark in CVM. The workload of the TPC-H benchmark consists of 22 database queries, which we implement in the CVM frontend. During this process, we determine CVM's limitations. The limitations hinder the execution of some TPC-H queries or worsen their performance. In general, the limitations mainly consist of missing data processing operations which we cannot execute in the current version of CVM. Consequently, to overcome these limitations, we extend the CVM framework with additional operators which increase CVM's processing width. Amongst others we implement different join variants and sorting. Moreover, we extend some operators to make them more powerful. For the query implementation, we rely on implementation patterns which occur frequently over the 22 TPC-H queries. These implementation patterns achieve high-level data operations for which no specific operator exists. Calculating the maximum, the average or counting are examples of such implementation patterns. In the end, we run two versions of the queries, one with the newly implemented features and one without, representing the state of CVM at the start of the thesis. Based on the execution times obtained from these runs, we determine if we were able to improve CVM with the newly implemented features. Furthermore, we compare the times with execution times from other database systems to see how competitive CVM is and if it can compete with current state-of-the-art systems.

Chapter 2

TPC-H

We rely on a benchmark to evaluate the performance of the “Collection Virtual Machine" and to compare the platform to other database systems. We chose the TPC-H benchmark. The reason why we use this benchmark and how we have to adapt the TPC-H schema is explained in this section.

2.1 Overview

The TPC-H benchmark [2] is the most prominent benchmark for evaluating decision support systems and is widely used for performance evaluation in database research. The main reason behind TPC-H's popularity is its schema and the breadth that its queries cover. TPC-H models a worldwide product supplying enterprise, which is subdivided into eight separate tables.The schema is depicted in [Figure 2.1]. The arrows point into the direction of the one-to-many relationships between tables. The formula below each table name represents the cardinality (number of rows) of the table. The cardinality for the lineitem table is approximate. The benchmark comprises twentytwo real-world business-oriented queries of high complexity requiring most of the functionality a relational engine should provide.

To adjust the database size and run the benchmark on bigger scales, the benchmark comes with a set of fixed scale factors (SF) ranging from 1 to 100000. The database size for SF 1 equals approximately 1GB. The individual table sizes increase linearly with the scale factor, except for the nation and region table sizes which remain constant. However, as these two tables only correspond to a tiny fraction of the database size, the sum of all the table sizes in GB equals the SF at all times.

With the characteristics mentioned above, the TPC-H benchmark is a logical choice to evaluate the performance of the CVM dataflow framework and compare it to a leading database system such as HyPer.

2.2 Modified TPC-H schema

The fields in the original TPC-H schema contain values of several data types, but they can all be represented by either integers or strings. However, as CVM cannot yet handle string literals in its current state, we modified the TPC-H schema such that CVM can handle these fields and be able to run the majority of the queries. The majority of these modifications consist of conversions from string to integer values. Fields containing strings are converted to integer values. The conversion procedure depends on the fields' set of possible values.

Many fields are converted by domain encoding. This concerns all fields which take a value defined by a finite set of strings in the original schema. We order the strings alphabetically and take their rank order as the new value. This is the case for the following fields:

- `c_mktsegment`
- `l_linestatus`
- `l_shipmode`
- `o_orderstatus`
- `p_type`
- `l_returnflag`
- `l_shipinstruct`
- `n_name`
- `p_container`
- `r_name`

We reduce the fields containing a string prefix accompanied with an integer to the integer. This is the case for the `name` fields in the supplier and customer tables, which take the value `[‘‘Supplier#’’, S_SUPPKEY]` and `[‘‘Customer#’’, C_CUSTKEY]` respectively. We convert them to an integer by dropping the string part including the ‘#’ symbol and converting the remaining digits to an integer. The conversion of `p_mfgr` happens in the

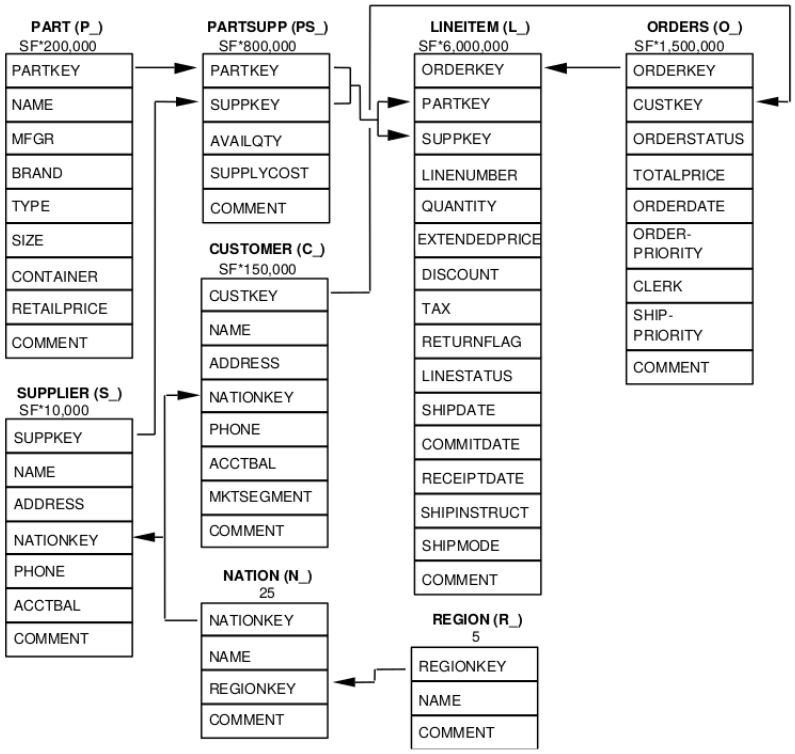

Figure 2.1: TPC-H schema [3]

same way and results to `['Manufacturer#'M]` being transformed to `[M]` where M is a single digit in the range from 1 to 5.

The phone attribute in the supplier and customer tables is converted to an integer by removing the three hyphens which split the number in 4 parts.

We convert the date from a string containing in numeric characters the date in the format 'Year-Month-Day' to an Unix timestamp [4]. We achieve this using the timetuple and timestamp function included in the datetime Python library. Furthermore, we convert all fixed-point numbers in the original schema to integers by multiplying by 100 as they have at most two digits to the right of the decimal point.

However not all fields which originally contain string values can be converted to a supported data type. Among others, this is the case for all `comment` fields in the tables. This prevents the execution of two from the twenty-two queries as a major part of the workload of these queries consists of filtering on strings.

After all these modifications, the modified TPC-H schema now only contains integer values in all the fields manipulated by the queries with the exception of the `comment`, `address` fields, and `p_name` field.

2.3 Value ranges

Every field in the TPC-H schema has a predefined value range, which increases as the scale factor grows larger. As the ranges are known, not all fields occupy the width of a 64-bit integer and therefore they can be compressed. We compress multiple fields into one to bypass operator limitations and gain performance. Each compressed field inside the integer occupies a specific continuous bit range. The [table 2.1] contains all the fields which are compressed at certain stages during the queries' implementation. For each of these the range and the number of bits required to represent the entire range are indicated. The table corresponds to scale factor one. The number of bits required is equal to the number of bits necessary to represent the high end minus the low end of the range. We can

Field	Range	#Bits
suppkey	1 – 10000	14
partkey	1 – 200000	18
orderkey	1 – 6000000	23
n_name	0 – 24	5
l_linestatus	0 – 1	1
l_returnflag	0 – 2	2
l_extendedprice	90100 – 10494950	24
l_quantity	1 – 50	6
l_shipdate	694310400 – 912470400	12
o_shippriority	0	1
o_orderdate	694224000 – 901929600	12
o_totalprice	85771 – 55528516	26
s_acctbal	-99822 – 999972	21

Table 2.1: Value ranges for SF=1

compress the date fields even further. The date does not include the precise time, but only expresses whole days. Therefore the range can be divided by 84600 which equals the number of seconds per day. This procedure decreases the range drastically and only 2525 values need to be represented.

Chapter 3

CVM framework

The “Collection Virtual Machine" (CVM) is a data processing platform that works across frontend languages and hardware platforms.

CVM can be separated into two parts, the backend and the frontend. Along this thesis, the frontend language in use is a Spark-like Python dialect. In the frontend, the user specifies a data flow program in form of a directed acyclic graph (DAG). We form the DAG by sequencing operators, complementing them with user-defined functions (UDF) and indicating the input sources on which the DAG is executed. The frontend materializes the DAG in JSON format by specifying a list of upstream sources for every operator. The backend, which is written in C++, then generates the code for the DAG and emits it into a shared library object. Once this is done, the control flow returns to the frontend which calls the shared library object with the input and receives the result of the data flow program.

3.1 Operators

CVM implements a set of dataflow operators necessary to perform common database queries. We implement the operators' individual logic in the C++ backend. Let's take a look at the operators which were implemented before the beginning of the thesis.

We can divide the operators into two separate categories: transformations and actions. For both categories, the operators work on collections of data elements, which are either tuples or records. The output is different for each type of operator. Whereas transformations create a new collection of data elements, actions return either a single value, a tuple or a Numpy array to the Python frontend. We call an action operator at the end of a sequence of transformations. It triggers the execution of the chained sequence.

Depending on the operator, we have to provide a first-order user-defined function (UDF) as a parameter to obtain the desired output. We pass the UDF to the operator during the query implementation in the Python frontend.

3.1.1 Transformations

alias The `alias` operator transforms the tuples of the input collection to records. Each field of the tuple gets associated to a name in a string list which we provide as an argument.

map The `map` operator applies a user-defined function to each data element in the collection. We often use it to reorder the columns of the collection before executing a different operator on the collection or perform a calculation on data elements. Structure, size and data type of the output collection may change depending on the supplied user-defined function.

filter The `filter` operator evaluates every data element against a user-defined boolean function and includes the data elements in its output if the test is successful.

cartesian The `cartesian` operator performs a cross-product of two input collections.

join The `join` operator equijoins two input collections on their leftmost data element.

reducebykey The `reducebykey` operator computes for each key in the leftmost column an individual aggregation and returns a collection containing all the keys and their respective aggregation. The

aggregation is defined by a user-defined function provided during the query implementation in the frontend.

range The `range` operator differs from the other operators as it does not work on a collection. In fact, it takes two arguments and creates a new collection containing data elements with values ranging from the small argument to the large argument.

3.1.2 Actions

collect The collect operator transforms a collection to a Numpy Array.

reduce The `reduce` operator aggregates all data elements in the collection using a user-defined function. If the collection is empty, it returns `None`. Otherwise it returns an integer (or a tuple) containing the aggregated value(s).

count The `count` operator counts the number of data elements in a collection and returns it. The operator is a sequence of a `map` and a `reduce` operator. The user-defined functions are predefined for both operators.

3.2 Limitations

Several of the operators come with limitations which hinder the TPC-H query implementation and degrade the performance. Furthermore, the set of operators is not sufficient to perform all the SQL queries in the TPC-H benchmark.

- Join
  It is common in database queries that we join two collections on multiple arguments. However, the `join` operator can only join on a single argument and we have to check the equality on the remaining arguments with an additional `filter` operator. This leads to a supplementary pass through the output collection and consequently worsens performance. Supplementary, the column on which we execute the join is by implementation always the leftmost column. The input collections have to take this into account and we may need to reorder the columns in advance with the map operator such that the column that is the relation of the key is located at the leftmost position.

```
res = RDD \
  .join(RDD2)\
  .filter( lambda x:( x[k] == x[l] ))\
  .collect()
```

Listing 1: Workaround: Join on multiple fields

- Semi-Join
  An inclusion test returns a data element from one input collection if there is at least one matching key on the second input collection. We cannot perform such an inclusion test efficiently with the current operators. The workaround consists of removing redundant columns from the second collection with the map operator, making the inner side unique on the key with a reduce_by_ key operation and then performing a join on the two collections.

```
Inner = RDD2 \
  .map(lambda x:( x[k], 1))\
  .reduce_by_key(lambda x1, x2:( x1 + x2))\
  .map(lambda x: x[0])
res = RDD \
  .join(Inner)
```

Listing 2: Workaround: Semi-Join

- Anti-Join
  An exclusion test returns a row from one input collection if there is no matching key on the second input collection. We can only perform this test in very specific cases with the current set of operators. If the second collection contains additional columns to those of the first collection, no workaround is possible and we are unable to execute the query.
- Outer join
  We cannot execute a right, left or full outer join with any sequence of operators.
- Reducebykey
  Grouping operations in database queries often include multiple fields. Due to two different workarounds, we are able to do this with the `reduce_by_key` operator despite that it only groups on the leftmost field. The first option is to omit fields out of the grouping operation which are functionally dependent on a field that is part of the grouping fields. The second workaround fits multiple fields into a single field. This is only feasible if we can store all the relevant fields in a 64 bit integer. We describe how this works in Section 5.2.
- Sort
  Most database queries are sorted by a column to return the collection in a sensible order. This is not possible with the current operators. We have to rely on the sort function of the Python Pandas library to sort the collections after the query execution.
- TopK
  Rather than sorting the whole collection, topK only returns a certain amount of rows of a collection in a sorted order. Obviously, this cannot be achieved with the current set of operators and again we have to rely on the Python Pandas library.

Query	Limitation								
	Join		Outer-Join	Semi-Join	Anti-Join	R_by_key		Sorting	String
	single col.	first col.				single key	first col.		
1						W(bits)	W	E(O(1))	
2	W	W					W	E(O(top100))	
3		W				W(bits)	W	E(O(top10))	
4		W		**W**				E(O(1))	
5	W	W					W	E(O(1))	
6									
7	W	W				W(bits)	W	E(O(1))	
8		W					W	E(O(1))	
9	W	W				W(bits)	W	E(O(1))	E
10		W				W(FD)	W	E(O(top20))	
11		W						E(O(SF))	
12								E(O(1))	
13			**X**						**X**
14		W							
15								E(O(SF))	
16									**X**
17									
18		W				W(bits)/W(FD)	W	E(O(top100))	
19									
20	W	W		**W**				E(O(SF))	E
21		W		**W**	W		W	E(O(top100))	
22					**X**				

Table 3.1: Limitations for every Query
W: workaround - E: externally done - X: prevents execution

3.2.1 Discussion

We discuss the consequences of the mentioned limitations in more depth during this Section. The presented limitations refer to the CVM state at the beginning of thesis. The objective of the thesis is to execute as many queries as possible to run without using external libraries and to improve the execution time. With this objective in mind, we analyse with what limitations we should deal first. The [table 3.1] provides an overview of the limitations' impact on the implementation of the TPC-H queries. The queries are highlighted with different colors. Whereas CVM can run green queries entirely, for orange queries it needs the assistance of external libraries and it cannot run red queries at all. Fields marked in blue color present opportunities for important performance boosts.

Currently, as the Table shows we are unable to run 3 of 22 queries. For 2 of these 3 queries the reason is CVM's inability to handle strings. Unfortunately, getting rid of this deficiency exceeds the scope of this thesis. Consequently, we will leave the concerned queries, Q13 and Q16, out of our discussion. We explain how they differ from Q9 and Q20 in Section 5.3.

We are also unable to run Q22 because we are missing an `antijoin` operator. The implementation of an `antijoin` operator should not pose any problems as its logic is close to that of a `join` operator. An operator that differs in its logic only marginally from the `antijoin` operator is the `semijoin` operator. The `semijoin` operator would speed up two queries, Q4 and Q21, as the current workaround is time consuming.

Looking back at the Table, we see that currently an external library executes the sorting of the result at the end of a query. This helps us to run the queries and check their correctness even without a sorting operator. However, as the majority of the queries need sorting, we should implement our own sorting operator. With this addition, we are no longer dependent from external libraries and performance comparisons with other systems become more meaningful.

The `reduce_by_key` operator and the `join` operator operate currently only on a single field. As the Table shows, workarounds do exist. However, while the workaround for the `reduce_by_key` operator does not increase the execution time, the workaround for the `join` operator is a slowdown factor for the queries that it is used in. Consequently, we should consider an extension to the `join` operator allowing to join on multiple fields to gain performance.

The two remaining columns of the table which we have not yet talked about are those showing which queries have to change the column order of a collection before using the `reduce_by_key` operator or the `join` operator. We notice that a lot of queries are affected by this. However, changing the order of the columns during the query execution is not a big performance issue and we will focus on the other limitations.

To conclude, we note that the implementation of the `antijoin` operator will allow the execution of an additional query. Furthermore, with the implementation of the sort operator we will be able to run the majority of queries without external libraries. Once these are done, we concentrate on improving the performance with the implementation of the `semijoin` operator and the extension of the `join` operator.

Chapter 4

New Features

In this chapter, we present our contributions made after analysing Table 3.1. We implement numerous new features to overcome the current *Limitations* and improve CVM's performance. The new features are a combination of implementing new operators and upgrading existing ones.

4.1 Overview

Operators' logic resides in the C++ backend and is divided into three functions: the `open`, `next` and `close` functions. The `open` function opens all the upstream operators, which provide data to the operator. The operator either consumes data from one or two upstreams. Whenever the operator consumes data from two upstreams, the `open` function consumes one of the two inputs in its entirety and we store its data in a data structure of our choice. For the different join variants and the `reduce_by_key` operator, we use the unordered maps and sets from the parallel hash map library from gitlab [1] as they outperform those from the C++ standard library by a large factor. The `next` function consumes the other upstream. If a single upstream provides data to the operator, the function that consumes the data depends on the operator. If the operator has to know all the data elements to perform its operation, the `open` function consumes the data, otherwise the `next` function consumes the data. The `next` function returns the data elements after the operator has processed them. Finally, the `close` function closes the upstream(s).

4.2 New Operators

Sort operator The `sort` operator sorts the data elements of a collection in ascending order with regards to its leftmost argument. To sort in a descending order, we have to multiply the leftmost argument with -1 before sorting. To get back to the original values once sorting is done, we multiply again with -1. We can perform these multiplications with two separate `map` operators. Whenever we want to order on multiple fields, we have to compress and store them in a single integer. We place the field with the highest order priority to the left. The other fields follow by descending order priority to the right. If the order direction is the same for every field, the compression happens as described in Section 5.2. However, if the order direction varies for the different fields, we have to invert the bits for certain fields before compressing them into a single field. If the field with the highest order priority is ordered in ascending order, we have to invert the bits for all the fields which are ordered in descending order. The same applies if we change the order direction. With this approach, a single sort of the compressed field is enough to sort multiple fields with mixed order direction.

```
res = RDD \
        .map( lambda x:( (x[k] << y + (pow(2,y-1)-1) ^ x[1], ... ))) \
        .sort() \
        .map( lambda x:( x[0] >> y,  (pow(2,y-1)-1) ^ (x[0] % pow(2,32)) ))
```

Listing 3: Inverting bits for sorting ascending on the first field and descending on the second field

[1] https://github.com/greg7mdp/parallel-hashmap

We delve into details about how the operator works. In the `open` function, we build a `std::vector` containing the same data elements as the input data collection. We sort the vector with the sort function provided by the C++ standard library [5]. The sort function takes a comparator as an argument. We implement this comparator so that it compares the first field of the input tuples. The `next` function then iterates through the vector and returns the tuples one by one.

Semijoin operator The `semijoin` operator replaces the time-inefficient workaround which we used to perform an inclusion test on a collection. The operator takes as input two collections, namely an inner and outer one. Afterwards, it checks if the leftmost field of a tuple in the outer collection exists in the inner collection. If we want to check the inclusion on multiple fields, we have to compress them into a single field or use a workaround as before, if the fields do not fit into a single field after compression.

We take a closer look at the operator's implementation details. The `open` function builds an unordered hash set out of the leftmost field of the inner collection. Once this is done, the `next` function iterates through the outer collection and checks for matches in the previously build hash table. If a match is found, the `next` function returns the outer collection's tuple.

Antijoin operator The `antijoin` operator implements an exclusion test. The operator takes as input two collections, namely an inner and outer one. Afterwards, it checks if the leftmost field of a tuple in the outer collection does not exist in the inner collection. However, when using this operator we can define an additional condition and modify the exclusion test. We provide this condition as a boolean function to the operator during the implementation of the query. This condition allows us to handle a variety of special cases. If the condition is redundant, because we want to perform a simple exclusion test on a single field, we provide a function which is always true.

We focus now on the implementation details of the operator. The `open` function builds an unordered hash map out of the data from the inner collection. The leftmost field is the key. We map each key to a vector containing all the data elements that the key is part of in the inner collection. The `next` function then iterates through the outer collection and checks for each leftmost column in a tuple if it matches a key in the map. If no match is found, it returns the data element. On the other hand, if a match is found, we check a boolean condition for each inner data element in the vector. If the condition fails for every data element in the vector, the `next` function returns the data element of the outer collection.

TopK operator The `topK` operator returns a given number of tuples in descending order with regards to the leftmost argument. The number of rows returned is provided by a parameter to the operator. If this number is larger than the number of data elements, the operator returns the entire collection in descending order. Whenever we want the `topK` operator to order in ascending order or on multiple fields, we follow the same approach as for the `sort` operator.

4.3 Extensions

Join on multiple keys The `join` operator previously could only join on a single field. Although a workaround existed, this worsened performance. Hence, we extended the `join` operator to join on multiple fields. The operator now checks multiple fields on equality during a single data pass. We indicate the number of fields to be checked by providing a positive number as an argument when implementing a query. Both collections must contain at least as many fields as indicated by the argument, otherwise the operator throws an exception.

With the new features added to the platform, the majority of the TPC-H queries can now be run without any external libraries.

Chapter 5

Query implementation

We implement 20 out of the 22 queries of TPC-H's workload in the Python frontend. The implementation of the queries are divided into three different functions: the `load`, `run` and `postprocess` functions. The source code containing every query implementation in its entirety is available at Gitlab [1].

5.1 Overview

The TPC-H database schema, modified to be in compliance with CVM's limitations, is accessible in a Python dictionary which contains each table as a Pandas dataframe. Out of this dictionary, the `load` function copies the relevant columns for each query into separate dataframes. The data types for each dataframe column are converted to a data type supported by the CVM platform: `Int32` or `Int64`. Once this is done, every dataframe is converted into the input form expected by CVM. Finally, the `load` function returns a dictionary containing all the relevant tables. In general, the `load` function contains only the loading of the data and it does not contain any part of the execution logic, with two exceptions. These exceptions are described in Section 5.3.

The `run` function is the core of the query implementation. By chaining operators together and providing them with user-defined functions (UDF), we create a dataflow program in form of a directed acyclic graph (DAG) which is executed on the specified data. The last operator in the sequence is always an action operator that triggers the execution and produces the final result of the run function.

During the `postprocess` function, we convert the result to a Pandas dataframe and revert the modified TPC-H schema to the original TPC-H schema.

5.2 Implementation patterns

Certain implementation patterns repeat themselves several times over the twenty queries. We describe these patterns in this section and associate them with their respective SQL operators. Patterns already described as workaround in Section 3.2 are omitted.

Count
SQL: COUNT()
Many database queries query the count for the different groups. To obtain this count, we add an additional column containing constant 1 values to the collection. We achieve this with the map operator and by specifying a UDF, which returns its input and a one. In the following step, we reduce the collection by key and the ones are added. Finally, the collection holds the count for each group.
Pattern:

[1] Gitlab: `https://gitlab.inf.ethz.ch/OU-SYSTEMS/jitq/-/blob/tpch_tests_ogoerens/python/jitq/benchmarks/tpch/queries.py`

```
count = RDD \
        .map(lambda x:( x[0], ..., x[k], 1) \
        .reduce_by_key( lambda x1, x2:( ..., x1[k] + x2[k])) \
        .collect()
```

Listing 4: Count per individual key

Sum, Average
SQL: SUM(), AVG()
The sum and average return the total sum of a numeric column and the average value of a numeric column respectively. We can calculate these values either for the entire collection or for groups of the collection. Depending on this choice, we use the `reduce` or `reduce_by_key` operator. The average is a combination of the sum and the count pattern followed by a division. We calculate the sum and the count during the same operation. The division happens by adding an additional map operation at the end.
Pattern:

```
res = RDD \
        .map( lambda x:( x[0], x[1], 1 )) \
        .reduce_by_key( lambda x1, x2: x1[0] + x2[0], x1[1] + x2[1] )) \
        .map( lambda x:( x[0], x[0] / x[1] )) \
        .collect()
```

Listing 5: Sum of the first column and Average of the second column

Minimum, Maximum
SQL: min(), max()
The minimum and maximum value of a collection are heavily used in database queries. Depending on whether we want to obtain the maximum/minimum for the whole collection or for each group, we use the `reduce` respectively the `reduce_by_key` operator.
Pattern:

```
res = RDD \
        .map( lambda x:( x[0], x[1])) \
        .reduce(lambda x1,x2 :( x1[0] if x1[0] > x2[0] else x2[0],
                                x1[1] if x1[1] < x2[1] else x2[1]))
```

Listing 6: Maximum of first column and Minimum of second column

Extract year
SQL: EXTRACT(year from date)
The extraction of the year from a date allows grouping values by year and comparing annual statistics. We store the date in seconds since the Unix epoch, which was at the first of January 1970, and is also known as timestamp format. Timestamps do not store leap seconds and the first leap year since 1970 was in 1972. We must take this into account when we extract years from the timestamp format. Therefore to do the conversion, we proceed as following:

- We calculate the number of leap years previous to the timestamps' (ts) year. Note that // represents integer division.

$$\text{prevLeapyears} = ((\text{ts}//\text{secsPerYear}) + 1)//4$$

- For every previous leap year, we subtract an entire day in seconds from the timestamp.

$$\text{tsWoLeapdays} = ts - \text{prevLeapyears} * \text{secsPerDay}$$

- Finally, we divide by the number of seconds per year.

$$\text{year} = 1970 + \text{tsWoLeapdays}//\text{secsPerYear}$$

This formula only works for dates until 2100 as it does not take into account that years divisible by 100, but not by 400 are common years. However as the TPC-H schema only contains dates between 1992 and 1998 included, this is not a problem.
Pattern:

```
secsperday = 24 * 60 * 60
secsperyear = secsperday * 365
res = RDD \
  .map(lambda x:( ..., 1970 + (x[m]-((x[m]//secsperyear + 1)//4) * secsperday)//secsperyear) \
  .collect()
```

Listing 7: Year extraction on field m

Case distinction
SQL: case (when condition then val1 else val2)
Sometimes, we attribute different values to a field depending on an input condition. We achieve this using the `map` operator and adding a UDF containing a ternary operator which evaluates the given condition and returns the expected value.
Pattern:

```
res = RDD \
        .map(lambda x:( x[0], ..., u if condition else v)) \
        .collect()
```

Listing 8: Case distinction

Multiple aggregation on single column
Occasionally, we compute multiple aggregation functions on a single field. To do this, we create copies of the fields which we use in more than one aggregation function. Each aggregation function then accesses a different column of the field during the `reduce/reduce_by_key` operation. *Pattern:*

```
res = RDD \
  .map( lambda x:( x[0], ..., x[k], x[k], ... )) \
  .reduce_by_key( lambda x1, x2 :( ..., f(x1[k-1], x2[k-1]), g(x1[k], x2[k]), ... )) \
  .collect()
```

Listing 9: Double aggregation on field k

Compression
As mentioned in Section 2.3, we frequently compress fields into a single one. We use the `map` operator to assign to each compressed field its bit range in a 64 bit integer. The associated UDF shifts each field by an individually predefined amount to the left and then adds the values together. Once this combined field has to be decompressed, we extract the individual fields again from the 64 bit integer. We obtain the fields by computing different remainders of the integer and right shifting the values by the same amount they were left shifted.

Pattern:

```
res = RDD \
        .map( lambda x:( ((x[k] << y + x[l]) <<  z) + x[m], ...)) \
        ...
        .map( lambda x:( x[0] >> y + z, (x[0] >> z) % y, x[0] % z ))
        .collect()
```

Listing 10: Compression and decompression of the fields k, l and m

Data type conversion
CVM currently supports only 32 and 64 bit integers. However, due to a bug, they do not combine well when stored together in a tuple in the data collection. The values of the fields do not persist. Consequently, whenever we store a field in a tuple that contains 64 bit integers, we have to convert every field to a 64 bit integer. We achieve this with the `map` operator.

Pattern:

```
res = RDD \
        .map( lambda x:( np.int64(x[0]),  np.int64(x[1])... ))
        .collect()
```

Listing 11: Conversion to 64bit Integers

5.3 String filtering

As already mentioned before, CVM does not support strings. However, Queries 9, 13, 16 and 20 filter on a string condition. Queries 9 and 20 have a condition on the `p_name` field. As it is a string, CVM cannot load this field and we filter on the aforementioned condition during the `load` function using Pandas and return the reduced `part` table on to the rest of the query. We then perform only the remaining part of the query during the `run` function in CVM. Consequently, a comparison of the execution times of these queries with other systems is not very meaningful. We add both queries despite this issue to show that the string filtering is the only missing feature to execute both queries in its entirety. In query 13, next to the string filtering, we are also unable to perform an outer join and therefor we cannot execute the query anyway. For query 16, counting only distinctive values in a `reduce_by_key` operation makes it impossible to run the query as such an option is not yet given with the current set of CVM operators.

Chapter 6

Performance evaluation

In this chapter we analyse CVM's performance on the TPC-H benchmark. We distinguish between 2 different states of CVM. The first state represents CVM at the beginning of the thesis, whereas the second state is CVM with the newly implemented features. We refer to these states as old and new, respectively. We run the queries for scale factors 1 and 10. The execution times for scale factor 1 are displayed in Table 6.1 and for scale factor 10 in Table 6.2.

For SF=1, we compute the execution times on a workstation with an Intel Core i7-4710HQ (4th generation, 6MB Cache, up to 3,5 GHz) with 8 GB RAM on a 64-bit Ubuntu 18.04.1 operating system.

6.1 Influence of new features

Running the queries in both states allows us to observe the potential performance gains which may have been achieved. It is important to note that in the old state the sorting was executed externally while in the new state we perform all sorting operations in CVM. Consequently, this adds additional workload which influences the execution time. However, for scale factors 1 and 10 we sort rather small collections and no major differences in execution time show up. Queries with a `topK` operation may experience a speed up. Queries 6, 14, 17 and 19 have the same running time for both states and consequently no time differences will appear for these queries.

Let us take a look at the other new features and their effect on the execution time. We notice that the join on multiple fields does not always speed up the query execution. Queries 2, 5, 7, 9 and 20 all include this new feature. While for query 5 we note a significant performance boost, the execution time of the other queries is either stagnant or became worse (Q9, Q20).

Regarding the `semijoin` operator, the new version of Q4 differs from the old one mainly by the addition of this operator. The execution times show that the `semijoin` improves performance. The execution time of Q4 decreased by approximately 25%. We could even increase the performance boost by implementing a second semijoin operator. In that case, we would have a right respective left semijoin operator, where one builds its hash table on the outer collection and probes on the inner collection. The additional performance boost derives from the fact that the hash map has to fit into the cache to achieve a good performance. Whenever the size difference between inner and outer collection is important and the hash map of one collection fits in the cache and the hash map from other collection does not, building on the smaller collection leads to a better performance. And currently for query 4, we are building on the larger collection, whose hash map does not fit into the cache. The second semijoin operator would allow us to change this and build on the smaller collection. The same issue occurs in query 20 with the only difference that in this query building on the wrong collection even leads to a slowdown compared to the previous workaround.

The third query which includes the `semijoin` operator is query 21. Again, we notice a slowdown of the execution time. This time, the reason for the slow down by factor 3 of the execution time is, in addition to the `semijoin` operator, the `antijoin` operator. The issue with the `antijoin` operator is the same as for the `semijoin` operator. As previously mentioned the collection on which the hash map is built plays an important role. For query 21, the `antijoin` operator builds the hash table on the large collection instead of the small collection. Consequently it performs worse than the previous workaround, which used a join and builds on the small collection. Same as with the `semijoin` operator, a second version for the `antijoin`, building on the opposite collection, would solve this problem.

Furthermore, the current `antijoin` implementation is too sophisticated for simple antijoin operations such as

Query	Time[ms]					
	CVM		MonetDB	HyPer		
	old	new	[6]	HyPer [6]	HyPer webinterface [7]	Hyper SF10/10 [8]
1	59	59	63	67	123	**49,8**
2	19	19	210	163	6,26	**4,7**
3	108	104	75	**66**	93,80	96,9
4	141	110	6003	194	**53,34**	72,5
5	230	89	5930	1276	**51,38**	82,1
6	16	16	123	**9**	34,21	20,7
7	98	98	1713	1151	**77,74**	80,4
8	73	73	172	399	53,02	**46,7**
9*	425	463	208	206	**136,38**	178,2
10^{c}	89	87	6209	1871	**72,30**	84,6
11	17	17	35	33	11,17	**11,1**
12	51	51	192	156	74,84	**46,0**
13	X	X	284	**185**	263,06	330,7
14	34	34	722	122	37,23	**28,3**
15^{c}	43	43	533	528	46,06	**22,7**
16	X	X	3562	1353	**61,79**	111,2
17	53	53	342	159	54,53	**48,5**
18	143	143	2505	**108**	237,32	306,1
19	**45**	**45**	1698	103	188,85	166,6
20*^{c}	37	49	750	114	38,74	**34,5**
21	280	870	329	**46**	123,20	134,9
22	X	80	141	7	25,35	**17,7**

Table 6.1: Execution Time per Query for SF=1
Q*: Query is not entirely executed in CVM
Q^{c}: Query is missing an output column

the one in query 22. When the user-defined function is redundant and we do not have to check a condition on additional fields, the processing of the function is redundant work which downgrades performance. Consequently, an implementation of the `antijoin` operator without the condition function improves performance. The frontend could switch automatically between the versions, if it detects a UDF or not.

6.2 System comparison

We compare CVM to other database systems to see how it compares in relation to those. For scale factor 1, we compare against MonetDB and HyPer, whereas for scale factor 10 we compare against Flare and HyPer. HyPer is one of the leading database systems and its running times are a good indication of whether CVM is competitive. We cannot run the TPC-H benchmark on Hyper ourselves, as the system is not open-source. Still we want to have robust comparison times and therefore we procure the execution times from different external sources. For some queries comparing the execution times with other systems must be done with caution as we do not execute the whole query in CVM (5.3) or do not process all the columns. We have marked these queries with adequate symbols in the tables.

First, we analyse the results for scale factor 1. Comparing the execution times of CVM with MonetDB, we notice that for the large majority, CVM outperforms MonetDB. Two exceptions are query 3 and 9. Taking a closer look at these queries may provide an idea on where there is still room for improvement in CVM. At first sight, we notice that in both queries, we compress multiple fields into a single field to perform a `reduce_by_key` operation on this compressed field. The workload to compress the fields may constitute a performance issue.

When we compare CVM to Hyper, it does not perform so well comparatively. CVM is slower for almost all queries to the best HyPer execution time. Only in query 19, CVM is able to outperform HyPer in scale factor 1. However, when running the queries on scale factor 10, CVM outperforms HyPer in 4 queries (Q6, Q17, Q18, Q19) and Flare in 5 queries (Q6, Q10, Q12, Q17, Q19).

Query plan comparison

HyPer's query plans are available online at `https://hyper-db.de/interface.html`. Consequently, we can compare them with ours and analyse for each query, if the time difference originates from different query plans

or from an implementation difference of the individual operators. In the rest of the section, we analyse the potential difference of the query plans between HyPer and CVM.

Query 1/5/6/7/14/17/19 Same query plan.

Query 2 HyPer's query plan differs from our query plan. The order of the join operations on the tables is different in both query plans.

Query 3 Different query plan: HyPer uses a `groupjoin` operator which combines the join and grouping operation in a single operation. CVM does not have a similar operator and we have to use a simple `join` operator followed by a `reduce_by_key` operator.

Query 4/20 Same query plan: However, HyPer has a `semijoin` operator that builds on the outer collection and one that builds on the inner collection. In this query, it uses the operator that builds on the outer collection as this collection is smaller. CVM offers only a operator that builds on the inner collection and consequently suffers a performance loss.

Query 8 Same query plan: However, at three occasions HyPer performs an indexed nested loops join, while we perform a hash join.

Query 9/10/12/15 Same query plan: However, at one occasion HyPer performs an indexed nested loops join, while we perform a hash join.

Query 11/22 Same query plan: However, at one occasion HyPer performs a block nested loops join, while we perform a hash join.

Query 18 HyPer's query plan differs from our query plan. Whereas Hyper uses a semijoin operator and two join operators in their query plan, we do without the semijoin operator and perform only two join operations in the query.

Query 21 Different query plan: HyPer has a `semijoin` operator which can semijoin on multiple arguments in the same operation. This is not possible with the `semijoin` operator that is included in CVM and we have to perform an additional join operation during the query.

Overall, we notice that most query plans are similar. However, the implementation of the operators differs and leads to performance differences. The `join` operator is the best example. Next to the hash join, HyPer has two different join methods: the indexed nested loop join and the block nested loops join. These join methods are used in seven queries and likely offer a performance boost compared to a hash join, which we use in CVM for all join operations. Furthermore, HyPer has multiple `semijoin` and `antijoin` operators and consequently can build the hash table always on the smaller collection. Whereas in CVM we build always on the inner collection even if the inner collection is larger than the outer collection. Additionally, HyPer has a `groupjoin` operator, which does not exist in CVM, and a more powerful `semijoin` operator.

We can conclude that CVM is able to compete with HyPer, but is not yet at the same level. Additional extensions in the future could change this as currently the reason for the worse execution times in most queries are the more efficient operators in HyPer.

Query	**Time (ms)**			
	CVM		**HyPer**	**Flare**
	Old	New	[8]	[8]
1	634	634	**498**	550
2	177	175	**47**	98
3	1266	1271	969	**481**
4	1973	1581	725	**544**
5	15813	1379	821	**527**
6	**175**	**175**	207	212
7	1208	1182	804	**596**
8	1256	1256	**467**	1016
9*	3121	6036	**1782**	2688
10[c]	1007	980	**846**	1429
11	117	117	111	**53**
12	612	612	**460**	643
13	X	X	**3307**	3499
14	473	473	283	**256**
15	574	575	227	**175**
16	X	X	1112	**679**
17	**329**	**329**	485	1924
18	2607	2556	3061	**1150**
19	**462**	**462**	1666	2236
20*[c]	363	649	**345**	836
21	3007	9428	**1349**	1533
22	X	1597	**177**	241

Table 6.2: Execution Time per Query for SF=10

Chapter 7

Conclusion

In this thesis, we implemented the TPC-H benchmark using the “Collection Virtual Machine". Along this process, we extended CVM to increase its operation radius and improve performance. Mainly, four new operators and an extension to an existing operator originated from the thesis and overcame limitations CVM had before. In the end, we were able to run 20 of 22 queries in the CVM frontend. We run the queries at two different scale factors, 1 and 10, which equal respectively a database size of 1 GB and 10 GB. A comparison of the executing times with HyPer showed that CVM can compete with a leading database system and is not far off state-of-the-art performance. Still, there exist some extensions and optimizations to further improve CVM's performance in the future.

References

[1] I. Müller, R. Marroquín, D. Koutsoukos, M. Wawrzoniak, S. Akhadov, and G. Alonso, "The collection Virtual Machine," 2020.

[2] "TPC-H." [Online]. Available: http://www.tpc.org/tpch/

[3] Transaction Processing Performance Council, "TPC Benchmark TM H Standard Specification Revision 2.17.1," 2014.

[4] "Unix Timestamp." [Online]. Available: http://www.https://www.unixtimestamp.com/

[5] "C++ stl::sort." [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/sort

[6] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *Proceedings - International Conference on Data Engineering*, 2011.

[7] "Hyper Webinterface." [Online]. Available: https://hyper-db.de/interface.html

[8] G. Essertel, R. Tahboub, J. Decker, G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, "Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data Flare: Optimizing Apa," *Osdi'18*, 2018.