User-Defined Aggregate Functions for Python in AsterixDB

Luke Ren

June 2022

Michael J. Carey
Donald Bren School of Information and Computer Science
Honors Thesis

Abstract

With the rise of Python as a popular data science language, it has become all the more important for modern Big Data Management Systems to support user-defined functions (UDFs) in Python. AsterixDB, a parallel, distributed BDMS for semi-structured data, supports scalar UDFs in both Python and Java, but it lacks support for user-defined aggregate functions (UDAs) which provide stateful or cumulative behavior.

In this work, Python user-defined aggregate functions were incorporated into the AsterixDB system by re-using much of the architecture for the existing scalar user-defined aggregate function system. The interface for UDAs in AsterixDB was drawn from other Big Data systems that have already implemented a similar feature, namely Apache Cassandra, Apache Spark, and Oracle. Minimal modification was made to the original UDF architecture in implementing UDAs in order to minimize the complexity of the system without sacrificing performance.

With this implementation of the Python interface for user-defined aggregate functions, users are now able to develop aggregate functions in Python that can run in a parallel, distributed fashion within the AsterixDB ecosystem. However, the true performance and use cases for such a system are still not fully known.

Table of Contents

1 Introduction

AsterixDB[1] is a Big Data Management System that employs a semi-structured data model and a SQL-like language as well as a scalable runtime that runs on multiple partitions of large datasets. Because AsterixDB supports a flexible data model and distributed computing, it marks itself as a strong candidate for machine learning and Big Data applications.

Python has become a popular data science language because it hosts a wide collection of machine learning and data mining packages and is also an easy-to-use, dynamically-typed language. Thus, proper integration with Python became a goal for AsterixDB, since data science applications are a primary target of the system.

Though the most recent snapshot of AsterixDB supports scalar user-defined functions (UDF) in both Python and Java, the system lacks support for aggregate behavior. In order to support the stateful and cumulative nature of machine learning and other Big Data applications, a system beyond the current user-defined function system is required.

In this project, an initial implementation for user-defined aggregate functions (UDA) was designed and integrated into AsterixDB. The goal for this implementation was to integrate user-defined aggregate functions into AsterixDB while minimizing changes to the current scalar UDF system. Re-using the existing architecture helped simplify the system overall.

2 Background

User-defined functions are a feature of Big Data Management Systems that allows for users of the BDMS to define their own functions with an external language that the BDMS supports, such as Java or Python. This enables users to use their own functions within the system as if they were one of the builtin functions.

User-defined functions can be scalar, which means that they receive a single tuple as an input and return a single tuple as an output. Scalar user-defined functions are currently the only type of user-defined function that AsterixDB supports for Python and Java. Scalar user-defined functions lack state, which allows for various optimizations but may not capture the full range of behaviors that the user wishes to perform.

User-defined aggregate (UDA) functions on the other hand receive a range of tuples and apply themselves to each of their inputs before aggregating all of those sub-results into a single output tuple. Because user-defined aggregate functions aggregate a result over multiple inputs, they can be used to implement stateful and cumulative behavior.

User-defined aggregate functions have already been implemented in other Big Data Management Systems for various other languages. The UDA interfaces for several of these other Big Data Management Systems were analyzed and taken into account in implementing UDAs for AsterixDB.

The three other systems that were considered were Apache Spark, Apache Cassandra, and Oracle. By analyzing the interface for each of these UDA systems, an appropriate interface was developed that matches the syntax of AsterixDB.

2.1 Apache Cassandra

The interface for defining user-defined aggregate functions in Apache Cassandra involves defining two intermediate scalar user-defined functions and an aggregate function that combines the two. These intermediate functions are created inside Apache Cassandra using its user-defined functions interface in CQL (Cassandra Query Language). The first intermediate UDF represents the state of the aggregate function as it accumulates values throughout a given range. The example below was taken from the Apache Cassandra UDA page [2] and represents the state function for the average aggregate function for integer data:

```
CREATE OR REPLACE FUNCTION avgState
( state tuple<int,bigint>, val int )
CALLED ON NULL INPUT
RETURNS tuple<int,bigint> LANGUAGE java AS
'if(val!=null){
state.setInt(0,state.getInt(0)+1);
state.setLong(1,state.getLong(1)+val.intValue());
}
return␣state;';
```

As the example above demonstrates, the average function's state can be represented using an integer and long, representing the total count and sum respectively over all of the inputs. The state function is implemented in Java and is scalar, meaning that it receives a single tuple as an input and returns the state tuple as an output.

The second intermediate UDF represents a final function that computes the output value of the average aggregate function given the last result of the state function:

```
CREATE OR REPLACE FUNCTION avgFinal
( state tuple<int,bigint> )
CALLED ON NULL INPUT
RETURNS double LANGUAGE java AS
'double␣r=0;
if(state.getInt(0)==0)return␣null;
r=state.getLong(1);
r/=state.getInt(0);
return␣Double.valueOf(r);';
```

The final function is also scalar and created using Apache Cassandra's UDF system. Combining these two UDFs into a single aggregate function at the query level allows users to define aggregate functions based on scalar UDFs.

The Apache Cassandra query language, CQL, has a special statement for defining aggregate functions that merges these two UDFs and declares an initial condition for the state:

```
CREATE AGGREGATE IF NOT EXISTS average ( int )
SFUNC avgState STYPE tuple<int,bigint>
FINALFUNC avgFinal
INITCOND (0,0);
```

As the example above demonstrates, the average aggregate function can be represented using the two scalar UDF functions defined above as well as an initial state tuple.

Given that there is no merge function to combine intermediate state function results between multiple nodes, it would seem that the Apache Cassandra interface for user-defined aggregate functions does not support parallelism in its implementation. This is a problem since one of the major motives behind AsterixDB is distributed computing.

Though the interface for Python UDAs in AsterixDB will likely not mirror that of Apache Cassandra entirely, there are a few points of inspiration that were drawn from the Apache Cassandra interface. The first is that the definition for aggregate functions has its own unique statement within CQL, involving the "AGGREGATE" keyword. Since AsterixDB also defines its own query language, SQL++, it was within reach to define a new unique keyword and format for UDAs in AsterixDB as well. The second is that Apache Cassandra utilizes much of its existing user-defined function system to support the definition of user-defined aggregates. While the interface for explicitly merging the intermediate state and final UDFs may be different in AsterixDB, the reuse of an existing code architecture makes the implementation and usage of UDAs in AsterixDB much simpler.

2.2 Apache Spark

The Apache Spark interface for user-defined aggregates, unlike that of Apache Cassandra, has a high-level of integration with its supported external languages, Scala and Java. Since Apache Spark already supports a robust and maintained API library for Scala and Java, creation of a user-defined aggregate in either language simply involves implementing an interface rather than running statements in a query language. To take the Java example on the Apache Spark UDA page [3] for the average aggregate function for the long integer data type:

```
public static class Average implements Serializable {
  private long sum;
  private long count;

  // Constructors, getters, setters...
```

```
}

// an aggregator class that averages long integer data
public static class MyAverage
    // extend the aggregator class with format
    // Aggregator<Tuple, SubResult, Output>
  extends Aggregator<Long, Average, Double> {
  // A zero value for this aggregation.
  // Should satisfy the property that any b + zero = b
  public Average zero() {
    return new Average(0L, 0L);
  }
  // Combine two values to produce a new value.
  // For performance, the function may modify `buffer`
  // and return it instead of constructing a new object
  public Average reduce(Average buffer,
    Long value) {
    long newSum = buffer.getSum() + value;
    long newCount = buffer.getCount() + 1;
    buffer.setSum(newSum);
    buffer.setCount(newCount);
    return buffer;
  }
  // Merge two intermediate values
  public Average merge(Average b1, Average b2) {
    long mergedSum = b1.getSum() + b2.getSum();
    long mergedCount = b1.getCount() + b2.getCount();
    b1.setSum(mergedSum);
    b1.setCount(mergedCount);
    return b1;
  }
  // Transform the output of the reduction
  public Double finish(Average reduction) {
    return ((double) reduction.getSum())
      / reduction.getCount();
  }
  // Specifies the Encoder for the
  // intermediate value type
  public Encoder<Average> bufferEncoder() {
    return Encoders.bean(Average.class);
  }
  // Specifies the Encoder for the
  // final output value type
  public Encoder<Double> outputEncoder() {
    return Encoders.DOUBLE();
  }
```

```
}
```

As the example above demonstrates, the highly-integrated API library for Apache Spark allows Java developers to remain comfortable writing Java-like code. The Apache Spark system, thus, must do the work under-the-hood of connecting this interface directly to the data system.

The Apache Spark interface supports three more functions than the Apache Cassandra interface. The merge function combines intermediate results of the state of the aggregate function from different threads, which implies that Apache Spark is able to run separate threads of computation for the aggregate in parallel before merging those intermediate results. This function will be necessary for the final implementation of UDAs in Python, since AsterixDB aims to perform these aggregations in a distributed fashion, as is done in the rest of the AsterixDB system. The other two new functions are encoder and decoder methods, which are not necessary in AsterixDB if the architecture for Python UDFs is reused since it already has strong support for serializing semi-structured data.

There were two major insights from analyzing the Apache Spark interface. The first is that supporting an interface for the native language in which the UDA is being written in, such as Python for the case of AsterixDB, provides a clean and easy-to-use solution for users of the UDA system. The other insight is that a merge function will be necessary for parallelism to exist in the interface, and it will involve merging intermediate states of the aggregate function. This is important, as parallelism is a main feature of AsterixDB that it aims to keep for its UDA system.

2.3 Oracle

The last UDA interface that was examined was that of Oracle. The Oracle interface is similar to the Apache Cassandra interface in that it executes query statements in order to define UDAs; however, the structure of the interface implies that it also supports parallel computation of the aggregate function, similarly to the Apache Spark implementation.

The Oracle implementation for UDAs, defined as the ODCIAggregate interface, involves defining four subroutines that, when combined, perform the work of an aggregate function: ODCIAggregateInitialize, ODCIAggregateIterate, ODCIAggregateMerge, and ODCIAggregateTerminate. These functions are synonymous to the intermediate scalar functions in the Apache Spark implementation, minus the encoder and decoder methods that Apache Spark uses to serialize its data. The example implementation for the spatial union function using Oracle's UDA interface found on the UDA page for Oracle [4] is as follows:

```
CREATE TYPE SpatialUnionRoutines(
   STATIC FUNCTION ODCIAggregateInitialize( ... ) ...,
MEMBER FUNCTION ODCIAggregateIterate(...) ... ,
MEMBER FUNCTION ODCIAggregateMerge(...) ...,
MEMBER FUNCTION ODCIAggregateTerminate(...)
);
```

```
CREATE TYPE BODY SpatialUnionRoutines IS
 ...
END;
```

Each of the member functions can be written in any of the Oracle-supported external languages, similarly to the Apache Cassandra interface. After defining this intermediate interface, the actual aggregate function can be defined using a unique Data Definition Language (DDL) statement:

```
CREATE FUNCTION SpatialUnion(x Geometry) RETURN Geometry
AGGREGATE USING SpatialUnionRoutines;
```

As the example above demonstrates, the interface for UDAs in Oracle is very similar to that of Apache Cassandra. Both interfaces employ a SQL-like query language to define intermediate scalar user-defined functions or routines that combine together to form a user-defined aggregate function. Both interfaces also use a unique "AGGREGATE" identifier to indicate the creation of an aggregate in the DDL statement of their respective query languages.

The structure of the creation of the aggregate function in Oracle differs slightly than that of Apache Cassandra in that the "AGGREGATE" identifier appears slightly later in the query, which potentially allows less modification of the grammar for that language than having it be its own creation statement.

2.4 Observations & Inferences

After performing an analysis on each of the interfaces of the UDA systems for Apache Cassandra, Apache Spark, and Oracle, the following observations and inferences were made:

User-defined aggregate functions are composed of intermediate scalar functions that manage and accumulate a state value across the input range of data. The first of these functions is an initialization function that defines the initial state of the aggregate function. Then comes a step function that takes a tuple from the input range of data and applies it to the current state. If the interface supports parallelism, then a merge function is necessary to combine two intermediate state values. The final function finishes the aggregate function by applying a transformation on the state before returning a single scalar value that represents the output of the aggregate function.

An observation that was made that influenced the implementation of Python UDAs for AsterixDB is that the native language interface for Scala and Java in Apache Spark was especially simple and easy-to-use. This encouraged a similar interface for Python such that Python developers would be more comfortable using the system. Another observation that was made was that in both Apache Cassandra and Oracle, the scalar user-defined function feature of those systems was re-used heavily for integrating user-defined aggregate functions. Since Python scalar user-defined functions already exist in AsterixDB, much of that same architecture was reused to implement Python UDAs.

3 Methodology

3.1 Planning

Given the observations and inferences made on the interfaces for Apache Cassandra, Apache Spark, and Oracle, an initial implementation for Python UDAs in AsterixDB was designed. This initial implementation started as a non-parallel, single-threaded UDA, which meant that no merge function was defined. By building this initial implementation, the feasibility of the new system feature was tested before the full, multi-threaded implementation was built.

Given the simplicity of the native language interface that Apache Spark exemplifies, it was decided that the UDA system for AsterixDB should employ the same easy-to-use approach for Python. This is done by having users define a Python class that implements scalar methods that combine together to form an aggregate function. For the initial, non-parallel implementation of the system, the interface has three functions, init, step, and finish. The example below demonstrates the interface for the count aggregate function using Python:

```
class MyCount:
  def init(self):
    self.count = 0
  def step(self, tuple):
    self.count += 1
  def finish(self):
    return self.count
```

As the example above demonstrates, the init, step, and finish methods resemble the initialize, step, and finish functions from previous implementations of UDAs. These functions are initialized and then called as scalar UDFs using the existing AsterixDB UDF system infrastructure.

Something unique about this implementation that's different from the UDA implementation for Apache Cassandra, Apache Spark, and Oracle is that the state for the aggregate function lives inside of the instance of the external class rather than an AsterixDB tuple. Although this decision was mostly made to reduce the complexity of the new system, it also enables the user to control the state behavior of the function, which could make an impact on how this feature will be used in the future.

However, since AsterixDB doesn't support as highly-integrated an API library for Python as Apache Spark does for Scala and Java, the creation of this Python class is not enough to introduce completely a UDA into the system. The robust, SQL-like language for AsterixDB, SQL++, must be used to define the aggregate function within AsterixDB, similarly to the Apache Cassandra and Oracle interfaces.

Drawing the most inspiration from the Oracle implementation, the following SQL++ DDL statement was designed:

```
CREATE FUNCTION foo(f1, f2: int)
(RETURN baz)?
```

```
[NULL CALL]
AS "modulename", "AggregateClass"
AT "pylib" AGGREGATE;
```

The largest consideration when designing the SQL++ DDL statement for UDAs was that the statement should be structured similarly to the current DDL statement for scalar UDFs in SQL++. Structuring the DDL statement for UDAs similarly to UDFs allows for the least amount of modification to the SQL++ grammar. Given that the Oracle implementation for this DDL statement already has a similar syntax for the creation of UDFs, it was a strong inspiration for the UDA interface for AsterixDB's SQL++.

For the example of the MyCount Python UDF described above, the following DDL statement could be used to initialize it:

```
CREATE FUNCTION my_count
AS "lib", "MyCount"
AT "pylib" AGGREGATE;
```

After a UDA is defined using a DDL statement like the ones above, it can be called just as an AsterixDB array-aggregate function is called. The example below demonstrates the my_count aggregate function being called on a range of data:

```
SELECT my_count((SELECT * FROM Data));
```

3.2 Implementation

Implementing the interface described in the planning phase above required work on the SQL++ grammar, compiler, and run-time, as the feature requires modifying each of these components.

A non-parallel, single-threaded version of the UDA feature was developed first to first test the feasibility of the concept.

3.2.1 Scalar Aggregate

The grammar was first modified to include the new "AGGREGATE" keyword and its corresponding DDL statement. This keyword was added as a modifier to the "CREATE FUNCTION" statement, which checks a flag for the UDF that designates it as an aggregate function when it is stored in the metadata. The implementation took strong influence from the Oracle implementation of the interface since it involved minimal modification to the grammar.

After modifying the SQL++ grammar to include the UDA DDL statement, the compiler was modified to recognize UDFs marked with the aggregate flag as UDAs during compile time. Given the state of the optimizer at the time, the initial implementation of this compiler work involved creating a scalar version of the aggregate that calls the aggregate on a listified version of the data. Later, the optimizer was configured to perform 2-step, parallel aggregates instead.

To support the run-time for UDAs, a new implementation of the aggregate run-time interface in AsterixDB was designed and implemented. The run-time interface for an aggregate function in AsterixDB required implementing an initialize, step, and finish function for the aggregate that each call their Python interface counterpart.

To implement this interface, the Python class that contains the methods to be used for the aggregate function is first initialized, and then the 3 scalar UDFs that compose the UDA are initialized.

The initialization of the UDA class was a new feature that did not exist in the original UDF system, but was necessary to unify the separate instance methods. In the original UDF system, each UDF was initialized as a method on a new instance of the parent class that caused each member UDF to function on a separate instance of the class. In order to unify these functions, a shared instance of the Python class must first be initialized in order to associate each member UDF with it.

After initializing the UDA class and then the component UDFs, the initialize, step, and finish functions were implemented by calling their equivalent UDF as a scalar UDF. This implementation required minimal change to the existing UDF system while still implementing the UDA feature efficiently. The caveat to this initial implementation was that it did not support parallel computation or a merging of sub-results.

3.2.2 Two-Step Aggregate

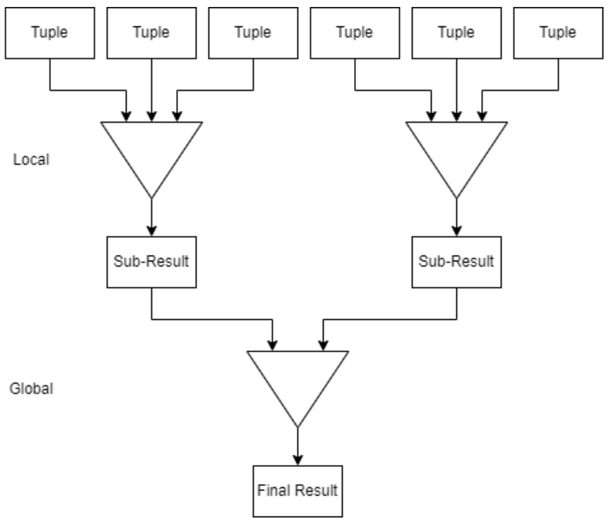

Figure 1: Two-Step Aggregate

AsterixDB represents parallel aggregates as two-step aggregates, which involves creating a local and global version of the aggregate. As demonstrated in

Figure 1, the local version computes a sub-result that represents an intermediate result for the aggregate on a partition of the inputs, and the global version merges these sub-results to yield the final aggregated result. To implement this version of the aggregate, the optimizer was first coerced into treating the aggregate as a two-step aggregate and then the run-time was modified to implement the two-step interface.

To enable two-step aggregates for UDAs, the optimizer was modified to store UDAs as four separate functions inside of the metadata: the scalar-wrapped version of the aggregate, the local version of the aggregate, the global version of the aggregate, and the aggregate function itself. These separate versions are stored in the metadata when the UDA is created using its DDL statement, and when the function is called, the compiler splits the aggregate into its local and global versions.

In order to implement the two-step aggregate run-time interface, a modification to the Python interface was necessary. A function to represent the intermediate sub-results of the local versions of the aggregate were necessary, as well as a function to merge those sub-results. Given these two new features, the following Python interface was designed and implemented:

```
class MyAverage:
    def init(self):
        self.count = 0
        self.total = 0
    def step(self, x):
        self.total += x
        self.count += 1
    def serialize(self):
        return [self.total, self.count]
    def merge(self, x):
        self.total += x[0]
        self.count += x[1]
    def finish(self):
        return self.total / self.count
```

The serialize function, as demonstrated above, allows AsterixDB to store the intermediate state as a tuple that can be passed into the global version of the aggregate. The global version of the aggregate can then call the merge function from this interface to aggregate the sub-results computed by other threads to its own state. The result is then computed and returned using the finish function similarly to the previous implementation.

To implement this interface, the instance methods were divided among the local and global versions of the aggregate. The step and serialize functions are called as the step and finish functions of the local aggregate interface, respectively, and the merge and finish functions are called as the step and finish functions of the global aggregate interface.

The init function is utilized by both the local and global versions of the aggregate, which is based on an assumption that the global version of the aggregate should be initialized in the same way that the local version should. Given the currently limited understanding of all the future potential uses for this feature, this feature was kept.

4 Performance

After the interface for two-step aggregates was fully defined, tests were run in order to compare the performance of this new method with the original scalar UDA implementation. These tests were performed by running some simple time trials on a local machine for both versions on a medium-size data set. The data set contains 60,000 tuples that represent shipping orders and totals around 195 MB of data. Each tuple in the data set contains a field named o_ol_cnt, which stores an integer value. An example document of this Order dataset is below:

```
{
    "o_all_local": 1,
    "o_c_id": 1893,
    "o_carrier_id": 1,
    "o_d_id": 1,
    "o_entry_d": "2014-09-18 10:11:30",
    "o_id": 1003,
    "o_ol_cnt": 12,
    "o_orderline": [
      {
        "ol_amount": 0,
        "ol_delivery_d": "2015-01-22 01:15:21",
        "ol_dist_info": "psbafsmkkzshqdzdhchdocbv",
        "ol_i_id": 24302,
        "ol_number": 0,
        "ol_quantity": 35,
        "ol_supply_w_id": 1
      },
      ...
    ],
    "o_w_id": 1
}
```

The function computed on the o_ol_cnt field from each tuple was the average function, which was defined in Python for both versions as below:

```
# scalar version
class MyAverageInitial:
    def init(self):
        self.count = 0
        self.total = 0
    def step(self, x):
        self.total += x
        self.count += 1
    def finish(self):
        return self.total / self.count

# 2-step version
class MyAverage2Step:
    def init(self):
        self.count = 0
        self.total = 0
    def step(self, x):
        self.total += x
        self.count += 1
    def serialize(self):
        return [self.total, self.count]
    def merge(self, x):
        self.total += x[0]
        self.count += x[1]
    def finish(self):
        return self.total / self.count
```

The function was defined in AsterixDB with the following DDL statement and then run with the following SQL++ statements below:

```
# create the aggregate function
CREATE FUNCTION averageudf(x)
AS "lib", "Average" AT pylib
AGGREGATE;

# run the aggregate function
SELECT averageudf((SELECT VALUE o.o_ol_cnt from Orders o));
```

A total of ten trials were performed for both versions of the aggregate function. While this may not be a large sample size, it proved to be large enough to identify a distinct difference in performance between the two versions.

After running the trials on a local machine, the two-step aggregate ran significantly faster than the scalar version. For both versions, the first, last, and average trial time were computed from the trials. The results of the trials are shown in Table 1.

	First Trial (sec)	Last Trial (sec)	Average Trial (sec)
Scalar	50.63	48.47	47.96
Two-Step	7.76	4.81	5.27

Table 1: Scalar vs. Two-Step Aggregate Results on a Data Set of 60,000 Tuples (10 trials each)

As the results of the trials demonstrate, the two-step aggregate version performed significantly faster than the scalar version on average. This is due to the system not having to first materialize and then pass in the entire array argument to the scalar version of the UDA in the two-step UDA case and the two-step aggregate making use of multiple nodes to compute several sub-results. The tests were run sequentially, with the trials for the scalar version being run before the trials of the two-step version.

These tests were performed on one machine with 2 simulated nodes and 4 partitions using a Windows 10 Pro laptop running Windows Subsystem Linux. The machine contains an Intel i7-8565U CPU @ 1.80GHz with 4 cores and 8 logical processors, and the storage used was a 512 GB M.2 SSD. Multiple simulated nodes were used in order to exercise the parallel aspects of the feature given just one laptop.

Though this single initial performance test may not be a strong indicator of the true benefits of parallel computation, it provides ample evidence to show that two-step aggregation has the potential to significantly outperform scalar aggregation.

5 Conclusion

After making modifications to the compiler and run-time implementation of Python UDFs, AsterixDB can now support Python UDAs. This feature supports parallel, two-step aggregates which run significantly faster than an initial scalar version that was initially implemented.

The UDA interface was implemented by composing several user-defined functions that together act as an user-defined aggregate function. This implementation was chosen because it minimized modification to the original UDF system without sacrificing performance or simplicity.

While the feature has been implemented into AsterixDB and supports computation of basic aggregate functions, much work still has to be done in order to finalize the system. In particular, thorough testing must still be done in order to evaluate the correctness and true performance of the aggregate function system using various kinds of aggregate methods. User-defined aggregates with more complex state, such as machine learning methods, still must be tested.

Even though the scalar version of UDAs still exists in the system, it is unclear how to conveniently switch between the scalar and two-step versions or even in which situations it might be worth doing so.

Further work must also be done to fully embed UDAs into the SQL++ language, such as perhaps also supporting the traditional SQL-92 aggregate calling convention. The system should also support the deletion of UDAs, since several aggregate versions are added to the metadata in order to support aggregate function lookup but only single UDFs are removed with the delete method.

While the potential use cases for this system are still not well-understood, there is now an opportunity to explore new horizons for UDAs and UDFs in AsterixDB and how they may potentially be used in the future.

References

[1] Sattam Alsubaiee et al. "AsterixDB: A scalable, open source BDMS". In: *arXiv preprint arXiv:1407.0454* (2014).

[2] *Creating user-defined aggregate function (UDA).* URL: https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useCreateUDA.html.

[3] *User Defined Aggregate Functions (UDAFs).* URL: https://spark.apache.org/docs/latest/sql-ref-functions-udf-aggregate.html.

[4] *User-Defined Aggregate Functions.* URL: https://docs.oracle.com/cd/B10501_01/appdev.920/a96595/dci11agg.htm.

[5] *User-defined Functions.* URL: https://asterixdb.apache.org/docs/0.9.8/udf.html.

6 Appendix

6.1 Example Usage

The source code for the implemented UDA system can be found on a fork of the AsterixDB repository: https://github.com/lukeren314/asterixdb on the "uda" branch.

In order to implement your own UDA, first follow the instructions on the most recent documentation page for setting up the user-defined function environment [5]. This includes cloning the AsterixDB respository, setting up authentication, and initializing a dataverse for testing your UDFs.

After setting up the UDF environment, write a Python file that contains a class that implements the UDA interface for AsterixDB. The example below is for the aggregate function, and is stored in a file named lib.py:

```python
class Average:
    def init(self):
        self.count = 0
        self.total = 0
    def step(self, x):
        self.total += x
        self.count += 1
    def serialize(self):
        return [self.total, self.count]
    def merge(self, x):
        self.total += x[0]
        self.count += x[1]
    def finish(self):
        return self.total / self.count
```

After writing the class in the Python file, follow the AsterixDB UDF instructions for uploading the Python file into AsterixDB. This includes packaging the file and its dependencies using the shiv package and uploading its result file to the cluster.

After uploading the packaged file onto your AsterixDB cluster, define your new UDA using the SQL++ DDL statement for UDAs. The example below defines the aggregate class as defined above for use in SQL++:

```
CREATE FUNCTION averageUdf(x)
AS "lib", "Average"
AT "pylib" AGGREGATE;
```

After defining the function, call it on the data of your choosing, making sure to use the array-aggregate style for calling AsterixDB functions. The example below demonstrates how to call the averageUdf function defined above on the id field of the TestData dataset:

```
SELECT averageUdf((SELECT VALUE t.id FROM TestData t));
```

After running the query above, the result of the UDA should be returned as a tuple containing the result of the aggregate.