BabelMR: A Polyglot Framework for Serverless MapReduce

Fabian Mahling*, Paul Rößler*, Thomas Bodner and Tilmann Rabl

Hasso Plattner Institute, University of Potsdam, Germany

Abstract

The MapReduce programming model and its open-source implementation Hadoop have democratized large-scale data processing by providing ease-of-use and scalability. Subsequently, systems such as Spark have dramatically improved efficiency. However, for a large number of users and applications, using these frameworks remains challenging, because they typically restrict them to specific programming languages or require cluster management expertise.

In this paper, we present BabelMR, a data processing framework that provides the MapReduce programming model to arbitrary containerized applications to be executed on serverless cloud infrastructure. Users provide application logic in *Map* and *Reduce* functions that read and write their inputs and outputs to the ephemeral filesystem of a serverless function container. BabelMR orchestrates the data-parallel programs across stages of concurrent cloud function executions and efficiently integrates with serverless storage systems and columnar storage formats. Our evaluation shows that BabelMR reduces the entry hurdle to analyzing data in a distributed serverless environment in terms of development effort. BabelMR's I/O and data shuffle building blocks outperform handwritten Python and C# code, and BabelMR is competitive with state-of-the-art serverless MapReduce systems.

1. Introduction

Data processing workflows are becoming increasingly complex. Interdisciplinary teams of data engineers, analysts, and scientists build data pipelines that combine different programming languages, runtime environments, and computing frameworks [1]. The resulting multi-language programs are commonly integrated via three different approaches. First, relational database systems are extended to support *user-defined functions* (UDFs). UDFs are constrained to be written in specific language dialects because of security or performance concerns of the database vendors and do not allow to express complex analytical algorithms. Second, *MapReduce systems* are more flexible and support general analytical dataflows [2, 3]. They prioritize a few popular languages (for example, Python [4] and SQL [5]) and provide hooks for other languages in their individual OS environment [6]. Third, *OS-level containers* are employed to encapsulate arbitrary applications along with their dependencies [7]. Redshift, BigQuery, and Snowflake have integrated UDFs as containers [8, 9, 10]. SAP HANA, Databricks, and ClickHouse have containerized their entire architectures [11, 12, 13]. The user code and system components are coupled loosely via REST APIs, which greatly simplifies integration but impedes the performance optimizations available to the prior approaches.

Another entry barrier for users of distributed data processing systems is the management of the underlying compute infrastructure. Conventional compute services based on virtual servers require decisions on instance types, cluster sizes, and pricing models [14, 15]. Virtual containers need orchestration and observability [16, 17].

In our view, these hurdles are not inherent issues of scalable data processing and recent developments in serverless computing can help to alleviate them.

Multiple cloud providers have introduced serverless compute services that allocate resources based on consumption, removing the need for manual provisioning and scaling [18, 19, 20]. The services have supported lightweight functions initially and now also run full-fledged containers [21]. Recent research has indicated the potential of data processing systems built on serverless functions. PyWren [22], Starling [23], and Lambada [24] have demonstrated performance and scalability that are competitive to systems on regular virtual machines.

Building on this work, we present the BabelMR framework for multi-language, data-intensive applications. Users of BabelMR package applications in containers and declare them to be mappers or reducers. Collocated with the user's application artifacts in a container is the BabelMR execution engine, which efficiently integrates with cloud storage and standard file formats. Between the BabelMR engine and the user application, data is passed locally through the container's ephemeral filesystem. The distributed computation of the containers within and across map and reduce stages is orchestrated by the BabelMR coordinator and carried out on the serverless compute service AWS Lambda. This enables users to write their applications in any language and run them in a data-parallel fashion on serverless cloud infrastructure with no operational overhead.

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Workshop on Serverless Data Analytics (SDA'23), August 28 - September 1, 2023, Vancouver, Canada

*These authors contributed equally.

fabian.mahling@student.hpi.uni-potsdam.de (F. Mahling); gerd.roessler@student.hpi.uni-potsdam.de (P. Rößler); thomas.bodner@hpi.de (T. Bodner); tilmann.rabl@hpi.de (T. Rabl)

CEUR Workshop Proceedings (CEUR-WS.org)

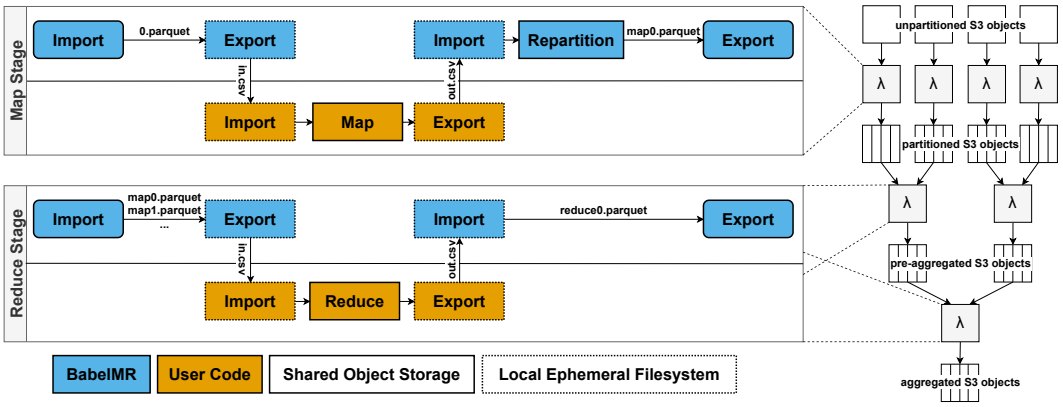

Figure 1: BabelMR dataflow: Stages are executed in parallel on serverless functions. Within one serverless function, BabelMR imports data from shared object storage (converts it into common formats) and outputs it to the local filesystem. The user code interacts with the filesystem only. BabelMR loads its output (performs repartitioning) and exports the data to shared storage.

In summary, we make the following contributions:

- We implement a MapReduce-style system on serverless cloud infrastructure. Particularly, we propose a solution for the data-parallel execution of any user code with serverless containers.
- We conduct microbenchmarks to determine the startup time of containers in AWS Lambda and the performance of the ephemeral filesystem for low-touch inter-process communication.
- We use queries from the TPC-H and TPCx-BB benchmarks in multiple languages to evaluate the development efficiency and the performance of BabelMR, as compared to state-of-the-art serverless data processors.

2. Overview of BabelMR

In this section, we present the most relevant serverless compute and storage services of AWS and introduce the architecture of BabelMR. The presented concepts apply to the services of other cloud providers [25, 26] as well.

2.1. Serverless Cloud Infrastructure

AWS Lambda [18] is a serverless, event-driven compute service. Users deploy applications in Lambda as so-called functions. The user's application code is embedded into the Lambda Runtime, which is executed on lightweight, stateless virtual machines [27, 28]. For each function, the memory size can be configured from 128 MB to 10 GB. Compute power scales linearly with the memory size up to 6 vCPUs. Each function has ephemeral (local) storage that can be configured between 512 MB and 10 GB.

AWS Lambda handles concurrent invocations automatically, by spinning up multiple instances of VMs that run the user code. The inherent elasticity this provides is ideal for the sporadic nature of the workloads that serverless data processing targets.

Amazon S3 [29] is an object storage service providing elastic scalability. Objects are stored in buckets and are identified by a unique key. Compared to other serverless storage systems (e.g., key-value stores or filesystems), storage costs are low and network bandwidth is high. This renders S3 most suitable for big data processing.

2.2. System Architecture

In BabelMR, data pipelines are run in stages. Per stage, Lambda-based workers execute the map or reduce user functions concurrently, as shown in Figure 1. We co-deploy the BabelMR execution engine and the user code on the Lambda functions. During a function invocation, the BabelMR engine imports a partition of the dataset into the local filesystem. Then, user code is executed, which reads the locally stored files containing key-value batches. The user code stores its output back to disk. Finally, the results are read by the BabelMR engine, partitioned, and written back to shared storage.

2.3. Programming Model and Interface

BabelMR exposes a MapReduce-style interface. A job in BabelMR is configured with map and reduce function binaries and object storage locations for the input and output data. We allow the specification of multiple inputs (e.g., different tables) for map jobs. This facilitates the implementation of map-side joins (see Code Listing 2). Additionally, the key and value attributes are specified, which are used for data partitioning and distribution.

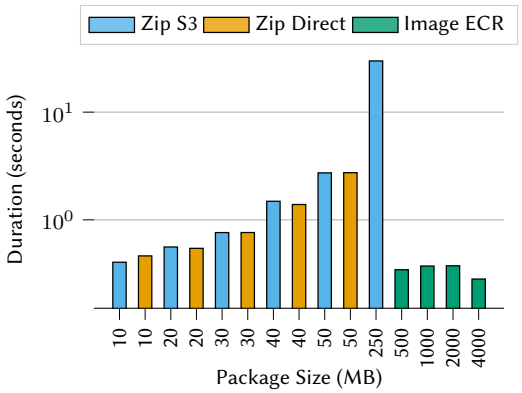

Figure 2: Initialization times for different package sizes, uploaded directly, through S3, or as an image through ECR. Measured on Lambda workers with 2,048 MB RAM.

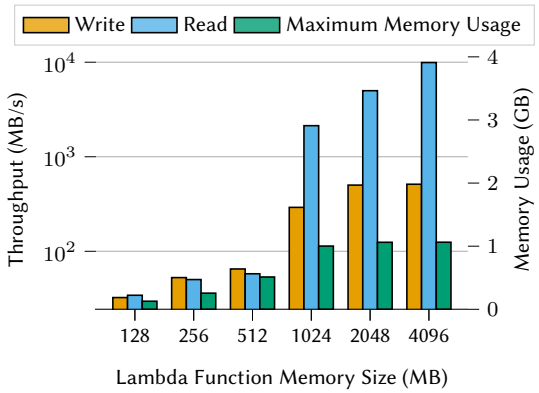

Figure 3: Read and write speeds for the local filesystem of Lambda functions with respect to function size. 1 GiB of data was written chunks of 16 MB.

Runtime Customization. We use a custom Lambda runtime to orchestrate user function binaries written in any language, and thus are not limited to the languages supported by the standard runtimes [30]. Our file-based, inter-process communication spares the implementation of language-specific adapters. Within a stage and worker, the BabelMR engine loads a partition of the input data and writes a file to the local filesystem, containing key-value pairs. The user code must load and process the file contents and output a file that again contains key-value pairs together with a schema description. The BabelMR engine exports the output data to remote shared storage.

Container Integration. To deploy their applications to AWS Lambda, users directly upload zipped binaries to their functions, upload the zipped binary to S3 or upload a container image [31] to Amazon Elastic Container Registry (ECR) [32]. The uncompressed sizes of the Zip files and container images are limited to 250 MB and 10 GB, respectively.

```
# Base image containing BabelMR engine
FROM hpides/babelmr:latest
# Installation of runtime environment
RUN yum install python3 -y
COPY requirements.txt ./
RUN python3 -m pip install -r /var/task/
    requirements.txt
COPY *.py ./
# Will be executed after BabelMR import
ENV MR_CMD="python3 /var/task/app.py"
```

Code Listing 1: Dockerfile to install the user's execution environment and code on top of our provided base image.

In BabelMR, the user's application is co-deployed with its runtime environment and the BabelMR engine. This may quickly exceed the size limitation of Zip-based Lambda functions. Container image-based functions do not entail such a strict limit and have worked for all of our use cases, ranging from simple scalar UDFs to full-fledged, in-memory database engines [33]. To evaluate the impact of these deployment methods on function startup times, we build different function packages, each containing a dynamically sized BLOB and a minimal, running function. Then, we measure the time it takes for a function to be initialized and operational.

Figure 2 shows that container images deployed via ECR impose no initialization overhead. Instead, initialization times decrease compared to Zip-based uploads and stay constant for different package sizes. This is due to Lambda employing lazy loading for images [21]. Thus, files within the images are only loaded on the running function, when they are accessed/read - in this case only the minimal function. When reading the blob, we get a throughput of roughly 150 MB/s for lazy loading files from ECR for 50 concurrently invoked Lambda functions. After the blob is read for the first time, subsequent read speeds did not differ based on the deployment variant.

Deploying the code packages as a Docker image overcomes potential size limitations. It also simplifies the user's integration with BabelMR, as we can provide a base image, which contains all our dependencies. Based on this image, the MapReduce executable and the execution environment are added (in our tests, the Dockerfiles were less than 10 lines long, see Listing 1). Using image-based Lambda functions is, therefore, the selected variant in our system.

```python
item=ParquetFile("/tmp/0.parquet").to_pandas()
sale=ParquetFile("/tmp/1.parquet").to_pandas()

filtered_item = item[
  item["i_category_id"].isin([1, 2, 3])
].filter(["i_item_sk"])

filtered_sale = store_sale[
  store_sale["ss_store_sk"].isin([10, 20, 33])
].filter(["ss_item_sk", "ss_ticket_number"])

filtered_sale["item_id"] = filtered_sale[
  "ss_item_sk"
]
joined = (
  filtered_sale.set_index("item_id")
  .join(
    filtered_item.set_index("i_item_sk"),
    how="inner")
)

joined.reset_index(drop=True, inplace=True)
fastparquet.write("/tmp/out.parquet", joined)
```

Code Listing 2: Python code for the first map function of TPCx-BB Q1 in BabelMR.

```python
# Read only specified row groups of file.
def read_partitioned(fd, columns, opener):
  dfs = []
  inputfile = fd["bucket"] + "/" + fd["key"]
  partitions = fd["partitions"]
  pf = ParquetFile(inputfile, open_with=opener)
  for i, rg in enumerate(pf):
    # Check if current partition is specified.
    if partitions == [] or i in partitions:
      dfs.append(rg.to_pandas(columns=columns))
  return pd.concat(dfs)

def read_partitioned_parallel(columns, opener,
    file_descriptions):
  # Spin up multiple threads for parallel read.
  with ThreadPoolExecutor() as executor:
    result_dfs = list(
      executor.map(
        lambda fd: read_partitioned(
          fd, columns, opener
          ),
        file_descriptions))
  return pd.concat(result_dfs)
```

Code Listing 3: Parallel import of Parquet files written in Python for the BabelMR variant All Custom.

2.4. Execution

In a MapReduce system, the I/O-bound tasks of remote data transfer and data shuffling are often the bottleneck and focus of optimization. In BabelMR, the worker-local communication between the engine process and the user application's process also needs to be efficient.

Distributed Communication. BabelMR employs scan and shuffle operators that are optimized for processing on serverless infrastructure. These operators largely follow the design of the state-of-the-art Starling and Lambada systems. They encapsulate efficient reading, writing, and format conversions between CSV, Apache Parquet [34], and ORC [35]. Also, they are fine-tuned to the CPU and network characteristics of Lambda and S3.

Inter-process Communication. BabelMR transfers data between its engine process and the application's process via the local filesystem. Using shared memory or message passing may yield performance superior to filesystem I/O [36]. However, it requires the user code to implement complex read and write operations or use libraries, such as Arrow Flight [37], which are not widely supported across programming languages. Conversely, using the filesystem yields high usability as file access is supported by every language. To evaluate whether the filesystem achieves acceptable performance under this usability trade-off, we measure the throughput of Lambda's local ephemeral storage.

To measure I/O throughput, we write 1 GiB of data in chunks of 16 MB into the /tmp/ directory of the Lambda instance with a single thread. We flush the file streams and then read the files serially. We executed the benchmark with 10 repetitions on 32 concurrently invoked Lambda functions. Figure 3 shows the median read and write speeds and the memory used with respect to the function's RAM. The measured write speed increases with the function size from 32 MB/s to 512 MB/s. The read speed for a 128 MB function is 34 MB/s and goes up to 9.9 GB/s for the 4 GB function. This is likely due to file accesses being cached in memory - as also indicated by the memory used.

Thus, when RAM is sufficiently available, using the filesystem as intermediate storage is reasonable. Our experiments indicate that execution times of big data pipelines benefit from larger workers, in particular, because CPU resources scale accordingly. Additionally, we do not import more than 340 MB of data per function invocation as the network throughput diminishes after an initial network burst budget. Consequently, we do not process larger-than-memory data on a single function.

Optimizations can be applied, to further improve data exchange. Instead of storing data as CSV files in S3, choosing more efficient file formats like Parquet [34] or ORC [35] yields better performance. Less data needs to be read from S3 and partitioned data can be stored in one single file. As the BabelMR engine can convert between standard formats, user code can operate on the preferred format independent of the format used in S3.

Programming languages may entail long initialization times (e.g., for dependency loading). In a regular Lambda

runtime, the execution environment is cached when an invocation finishes. The initialization overhead occurs once per function execution lifecycle [38]. In BabelMR, the application is newly initialized per invocation, because it's process is not kept alive by the underlying custom runtime. For many languages, it is possible to employ wrappers that keep application state cached, eliminating repeated initialization. This, however, is not supported out-of-the-box and requires per-language effort.

3. Evaluation

To evaluate the usability and performance of BabelMR, we write BabelMR applications for the relational query TPC-H Q1 [39] and the MapReduce job TPCx-BB Q1 [40, 41]. We implement the applications in Python and C# in three variants. The source code can be found on GitHub under https://github.com/hpides/babelmr-applications.

All Custom implements all functionalities a MapReduce system requires. The complete workflow includes S3 access (partitioned import/export), the application logic (map/reduce), and repartitioning methods. The mappers export partitioned Parquet files and the reducers import individual row groups from multiple map outputs. We use established dataframe libraries for the application logic, the AWS SDK for S3 access, and implement repartitioning and parallelized reads to the best of our knowledge.

System-side Shuffle takes care of data shuffling. As partitioning strategies are independent of the application logic, a generic intermediate pipeline is supplied, which shuffles between map and reduce stages. To configure the intermediate pipeline, we specify the attributes that should be available on each reducer (i.e., the values), the attributes the data should be partitioned on (i.e., the key) and the number of reducers n. The intermediate pipeline reads the map outputs and yields output files with n partitions, which are consumed by subsequent reducers.

BabelMR entirely relieves the user of the responsibility to deal with cloud storage access as we collocate the user-defined application together with the BabelMR engine on one function (see Section 2). We build an executable that reads input from the filesystem, performs the batched job and writes the result to the filesystem.

	TPC-H Q1		TPCx-BB Q1	
	Python	C#	Python	C#
All Custom	152	150	210	149
System-side Shuffle	143	100	201	94
BabelMR	120	71	170	69

Table 1: Lines of code to implement the MapReduce programs. The values for BabelMR include a Dockerfile with 4 LoC for C# and 6 LoC for Python.

Additionally, we implement TPC-H Q1 in the serverless MapReduce frameworks Corral [42] and PyWren as well as with PySpark [4] and Ray [43]. We execute the latter with AWS Elastic MapReduce (EMR) [44] and AWS Glue [45]. We compare them against BabelMR (Go) and BabelMR (Python), where we use Go and Python for the application code. Here, we process CSV data, as Parquet is not supported by Corral.

3.1. Development Efficiency

By using BabelMR, the required lines of code are reduced by 19% and 53% compared to All Custom (see Table 1). We do not have to implement any S3 access, partitioning, or shuffling logic. In fact, in BabelMR, the application code does not use the AWS SDK at all.

Listing 2 shows the Python code for the first batched map job of TPCx-BB Q1. Here, we can utilize well-known APIs, as data is stored in the filesystem. Import and export code (lines 1-2 and 23) become simple and short. The majority of the file (lines 4-20) contains application logic. Likewise, we implement the reduce jobs of the query. With the Dockerfile in Listing 1, we install dependencies and code on top of the BabelMR base image.

For All Custom and System-side Shuffle, we implement S3 access ourselves. Listing 3 shows the implementation of a parallelized import from S3.

All of the MapReduce frameworks abstract from the distribution logic. BabelMR applications are similar in length to programs in Corral and PyWren. PySpark and Ray programs are shorter due to the mature integration into both dataframe and cloud service libraries.

3.2. Performance

The benchmarks were executed from May to July 2023. We configured Lambda workers with 5,120 MB RAM and 512 MB local storage. We invoked workers of the same stage concurrently from a c5.2xlarge EC2 instance within the same region (us-east-1). For Ray on Glue, we use *Ray 2.4* and *Glue 4.0*. We conduct benchmarks for EMR on elastic and static clusters using *emr-6.11.0*. Elastic clusters are initialized with minimum resources and grow while executing a job. Static clusters start the job execution with initialized resources. We measure runtimes of the three variants and related systems on four scale factors and take the average of 10 warm runs.

For scale factor n, we use n workers in the map stage of the TPC-H Q1, and one worker for the reduce stage. Intermediate shuffle stages are executed on one worker for TPC-H Q1 and $\lceil \frac{n}{50} \rceil$ workers for TPCx-BB Q1. For TPCx-BB Q1, we use $\lceil \frac{n}{2} \rceil$ workers in the first map stage and $\lceil \frac{n}{10} \rceil$ workers in each reduce stage. The data is stored in Parquet format across n files for TPC-H Q1 and $\lceil \frac{n}{2} \rceil$ files for TPCx-BB Q1. We evaluate pipelines that process

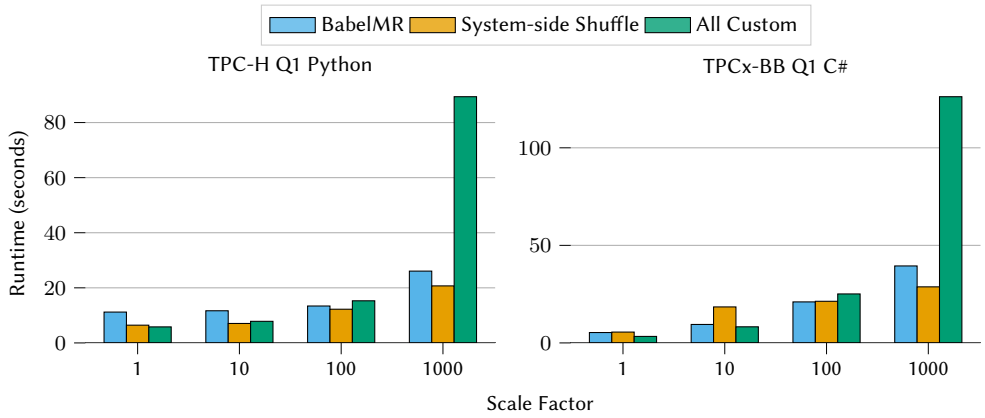

Figure 4: End-to-End runtimes for TPC-H Q1 and TPCx-BB Q1 executed on different scale factors with the different proposed architectures in Python and C#. The data was stored in S3 and partitioned into multiple Parquet files.

CSV data for TPC-H Q1 only, where we have $5n$ files and use $\lceil 2.5n \rceil$ mappers. For AWS EMR, elastic and static clusters may utilize one driver (8 vCPUs/16 GB RAM) and 1, 4, 40 or 400 worker (16 vCPUs, 32 GB RAM each) for scale factors 1 to 1,000. For Ray clusters, we use Z.2x machines for the worker. That way, the clusters have the same amount of resources at disposal as the serverless systems using Lambda workers.

TPC-H Q1. For our experiments with TPC-H Q1 (see Figure 4), System-side Shuffle has the fastest runtimes. This is due to the map stage generating very small results ($\approx n \cdot 8$ KB). These small files are read fast in parallel by the intermediate shuffle stage. The shuffle outputs one file that can be easily consumed by the reducer.

While BabelMR also benefits from the optimized read during the reduce stage, System-side Shuffle executes roughly 4 s faster. This is because of a shorter map stage, where every worker works on exactly one input file. Here, invoking BabelMR to materialize the Parquet file into the ephemeral storage leads to an overhead, compared to Python directly loading from S3.

While All Custom's parallelized import (see Listing 3) is roughly 10 times faster than importing sequentially, it still scales badly. At scale factor 1,000, the reduce function needs to resolve 1,000 imports, resulting in around 60 seconds being spent on imports. This leads to ~3.6 times slower end-to-end runtimes compared to BabelMR and System-side Shuffle at scale factor 1,000.

TPCx-BB Q1. Figure 5 breaks down the execution times of individual functions for TPCx-BB Q1. The query consists of three stages (and two additional shuffle stages for System-side Shuffle). Note the difference for workers of the System-side Shuffle and All Custom for Stage 2 and Stage 3. The consumed data is the same, but workers in All Custom need to execute 50 or 10 imports respectively. Workers in System-side Shuffle only have to import two files, due to the upstream repartitioning and combining of the intermediate shuffle stage (two blue lines).

Stages 2 and 3 in BabelMR profit from the same principle. Additionally, the import of BabelMR in the first stage outperforms the other variants. Only 3 of the 23 columns of the store_sales table are required. The Parquet import of BabelMR is better optimized to read only the required columns than the custom implementation in C#.

At large scale factors, BabelMR beats All Custom with regards to usability and performance on both TPC-H Q1 and TPCx-BB Q1. Compared to the System-side Shuffle, performance is similar and usability is better. In contrast to System-side Shuffle, BabelMR materializes data once more and has a larger deployable, leading to overheads.

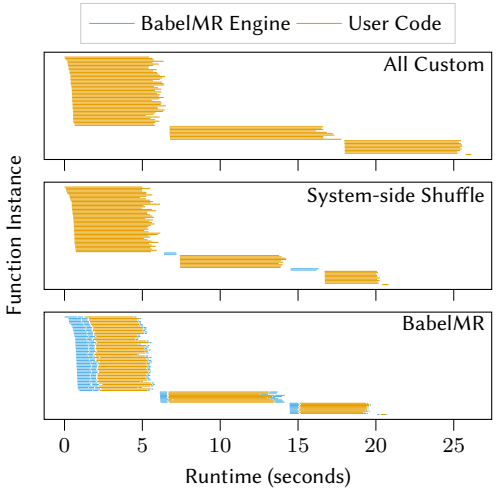

Figure 5: Break down of execution times per function for TPCx-BB Q1 (sf=100) for the three variants written in C#. Each line represents the lifetime of one Lambda function.

PyWren. As PyWren is unmaintained, we had to rebuild the underlying Python runtime and do minor bug fixes on PyWren. Still, compared to 2017 [22] setup times decreased from 14.2 s to 7 s and Lambda start latencies from 9.7 s to at most 1 s now (see Figure 2).

We execute the same Python map and reduce functions for TPC-H Q1 in PyWren and BabelMR. As we measure end-to-end runtimes, serialization of the Python functions and their upload to S3 is also measured. BabelMR outperforms PyWren by approximately 14 s for scale factors 1 and 10. This originates from the fact, that setup costs in PyWren are doubled, once for the map and once for the reduce phase [22]. On scale factor 100, BabelMR is on average 17.5 s faster than PyWren. At start-up, PyWren creates S3 objects for each worker's input, which leads to this additional overhead at high scale factors.

Corral. Corral exposes the traditional tuple-based MapReduce interface without the support of combiners. As a result, the outputs of the map stage are significantly larger ($\approx n \cdot 60$ MB) than those of BabelMR Go. Additionally, the map stage produces only four distinct keys. Therefore, execution times do not scale properly as four reducers have to process the entire map output (see Figure 6). For scale factor 100 the input for one reducer is larger than its memory, which is not supported by Corral and results in a failing pipeline. Corral performs slightly better compared to BabelMR Go for scale factor 1. This is partly due to the materialization overhead discussed earlier. Additionally, Corral distributes the rows of the five input files evenly among workers, whereas BabelMR assigns CSV input on a per-file granularity,

PySpark on EMR. BabelMR outperforms PySpark on all scale factors on TPC-H Q1. PySpark exposes the high level Spark [5] API to the user, so underlying execution details are handled by Spark. Thus, three instead of two stages are executed, wherein the first stage, Spark reads metadata from input files. This takes 2 s at scale factor 1 and 20 s at scale factor 1,000. Execution time is also spent on managing the cluster and allocating the required executors. At scale factor 1,000, PySpark runs 28 s until all 240 executors work on jobs in the static cluster and even longer on the elastic cluster.

Ray on Glue. Ray handles distribution and scheduling of jobs, utilizing it's high level dataset API. While Ray yields good usability and is well integrated in AWS Glue, execution times are higher than those of the introduced systems working on AWS Lambda. Start-up latencies are one reason for this. For scale factor 10, it takes 35 s to start all workers.

In summary, BabelMR is performance competitive with other MapReduce systems running applications written in the languages that they support, despite the more generic multi-language approach.

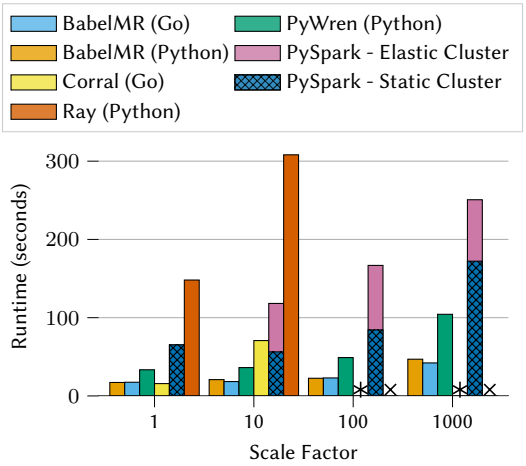

Figure 6: Execution times for TPC-H Q1 on PyWren, Corral, PySpark, Ray, and BabelMR. The data was stored in S3 and partitioned into multiple CSV files. $*$ denotes a failed run; $\times$ denotes a run omitted due to AWS quota constraints.

4. Related Work

Most closely related to our work are data processing systems that build on serverless infrastructure to alleviate their users from provisioning, management, and scaling of resources. There are the MapReduce-style systems PyWren [22], Corral [42], Flint [46], and Qubole [47] that each offer an interface for a popular programming language, e.g., Python, Go, or Java. In addition, there has been work on serverless SQL query processors with Starling [23] and Lambada [24], and our own prior work on Skyrise [48]. They all integrate efficiently with cloud storage options for storage access and data shuffling. They, however, all target only a single programming language.

Saur et al. acknowledge the demand for custom code executing arbitrary computation written in any programming language [36]. They suggest containerized UDFs enabling dependency management, portability, and encapsulation. They exchange input and output tuples between the running containers and the database. While Saur et al. assume a database server communicating to the containers, in our serverless environment data is stored in S3. We propose to collocate BabelMR into the container that handles the S3 access, communicating data to the user code via the file system.

While not a full-fledged data processing system, the distributed map abstraction from AWS Step Functions [49] allows to orchestrate large-scale parallel workloads of containerized applications. In contrast to BabelMR, they cannot execute reduce jobs, and thus are limited to embarrassingly parallel applications.

5. Conclusion

Distributed data processing frameworks present high programming and operations barriers for many users. In this paper, we propose the container-based, language-agnostic BabelMR system built on serverless cloud infrastructure. BabelMR inherits the flexibility of OS containers and the simplicity of serverless cloud services.

Our evaluation shows that BabelMR is performance-competitive with language-specific MapReduce frameworks while at the same time, it is more flexible and easier to use.

Acknowledgments

We would like to thank Theo Radig and Fabian Engel for valuable discussions and code contributions. This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), the European Union's Horizon 2020 research and innovation programme (ref. 957407), and the AWS Cloud Credit for Research Program.

References

[1] B. Haynes, A. Cheung, M. Balazinska, PipeGen: Data Pipe Generator for Hybrid Analytics, in: M. K. Aguilera, B. Cooper, Y. Diao (Eds.), ACM SoCC, 2016, pp. 470–483.

[2] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: USENIX OSDI, 2004, pp. 137–150.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, Spark: Cluster Computing with Working Sets, in: USENIX HotCloud, 2010.

[4] Apache Software Foundation, PySpark, https://spark.apache.org/docs/latest/api/python/, 2023.

[5] Apache Software Foundation, Spark SQL, DataFrames and Datasets Guide, https://spark.apache.org/docs/latest/sql-programming-guide.html, 2023.

[6] Apache Hadoop contributors, Hadoop Streaming, https://hadoop.apache.org/docs/stable/hadoop-streaming/HadoopStreaming.html, 2023.

[7] Docker, Use Containers to Build, Share and Run your Applications, https://www.docker.com/resources/what-container/, 2023.

[8] Google Cloud Platform, Working with Remote Functions, https://cloud.google.com/bigquery/docs/remote-functions/, 2023.

[9] Amazon, Creating a Scalar Lambda UDF, https://docs.aws.amazon.com/redshift/latest/dg/udf-creating-a-lambda-sql-udf.html, 2023.

[10] Snowflake, Writing External Functions, https://docs.snowflake.com/en/sql-reference/external-functions, 2023.

[11] SAP, SAP HANA in Containers, https://blogs.sap.com/2020/03/13/at-your-service-sap-hana-2.0-an-introduction-2/, 2023.

[12] Databricks, Announcing Serverless Compute for Databricks SQL, https://www.databricks.com/blog/2021/08/30/announcing-databricks-serverless-sql.html, 2023.

[13] ClickHouse, Building ClickHouse Cloud From Scratch in a Year, https://clickhouse.com/blog/building-clickhouse-cloud-from-scratch-in-a-year/, 2023.

[14] Amazon, Amazon EC2, https://aws.amazon.com/ec2/, 2023.

[15] Amazon, EC2 Instance Types, https://aws.amazon.com/ec2/instance-types/, 2023.

[16] Apache Software Foundation, kOps - Kubernetes Operations, https://kops.sigs.k8s.io/, 2023.

[17] Amazon, Amazon Elastic Kubernetes Service, https://aws.amazon.com/eks/, 2023.

[18] Amazon, AWS Lambda, https://aws.amazon.com/lambda/, 2023.

[19] Microsoft, Azure Functions, https://azure.microsoft.com/services/functions/, 2023.

[20] Google, Google Cloud Functions, https://cloud.google.com/functions/, 2023.

[21] M. Brooker, M. Danilov, C. Greenwood, P. Piwonka, On-demand Container Loading in AWS Lambda, in: USENIX ATC, 2023, pp. 315–328.

[22] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the Cloud: Distributed Computing for the 99%, in: ACM SoCC, 2017, pp. 445–451.

[23] M. Perron, R. C. Fernandez, D. J. DeWitt, S. Madden, Starling: A Scalable Query Engine on Cloud Functions, in: ACM SIGMOD, 2020, pp. 131–141.

[24] I. Müller, R. Marroquín, G. Alonso, Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure, in: ACM SIGMOD, 2020, pp. 115–130.

[25] Microsoft, Azure Serverless, https://azure.microsoft.com/solutions/serverless/, 2023.

[26] Google, Serverless, https://cloud.google.com/serverless, 2023.

[27] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D. Popa, Firecracker: Lightweight Virtualization for Serverless Applications, in: USENIX NSDI, 2020, pp. 419–434.

[28] Firecracker contributors, Firecracker: Secure and Fast microVMs for Serverless Computing, https://github.com/firecracker-microvm/firecracker/, 2023.

[29] Amazon, AWS S3, https://aws.amazon.com/s3/, 2023.

[30] Amazon, Lambda Runtimes, https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html, 2023.
[31] D. Merkel, Docker: Lightweight Linux Containers for Consistent Development and Deployment, Linux journal 2014 (2014) 2.
[32] Amazon, Amazon Elastic Container Registry, https://aws.amazon.com/ecr/, 2023.
[33] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, H. Plattner, Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management, in: EDBT, 2019, pp. 313–324.
[34] Apache Software Foundation, Apache Parquet, https://parquet.apache.org/, 2023.
[35] Apache Software Foundation, Apache ORC - High-Performance Columnar Storage for Hadoop, https://orc.apache.org/, 2023.
[36] K. Saur, T. Mirmira, K. Karanasos, J. Camacho-Rodríguez, Containerized Execution of UDFs: An Experimental Evaluation, PVLDB 15 (2022) 3158–3171.
[37] Apache Arrow contributors, Introducing Apache Arrow Flight: A Framework for Fast Data Transport, https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/, 2023.
[38] Amazon, Understanding the Lambda Execution Environment, https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environment.html, 2023.
[39] Transaction Processing Performance Council, TPC-H Benchmark, https://www.tpc.org/tpch/, 2023.
[40] P. Cao, B. Gowda, S. Lakshmi, C. Narasimhadevara, P. Nguyen, J. Poelman, M. Poess, T. Rabl, From BigBench to TPCx-BB: Standardization of a Big Data Benchmark, in: TPCTC, volume 10080 of *Lecture Notes in Computer Science*, 2016, pp. 24–44.
[41] Transaction Processing Performance Council, TPCx-BB Benchmark, https://www.tpc.org/tpcx-bb/, 2023.
[42] B. Congdon, Corral: A Serverless MapReduce Framework Written for AWS Lambda, https://github.com/bcongdon/corral/, 2023.
[43] Anyscale, Inc., Ray, https://www.ray.io/, 2023.
[44] Amazon, Amazon EMR, https://aws.amazon.com/emr/, 2023.
[45] Amazon, Amazon Glue, https://aws.amazon.com/glue/, 2023.
[46] Y. Kim, J. Lin, Serverless Data Analytics with Flint, in: IEEE CLOUD, 2018, pp. 451–455.
[47] Qubole, Qubole: Apache Spark on AWS Lambda, https://github.com/qubole/spark-on-lambda/, 2023.
[48] T. Bodner, Elastic Query Processing on Function as a Service Platforms, in: VLDB PhD Workshop, 2020.
[49] Amazon, Step Functions: Using Map State in Distributed Mode, https://docs.aws.amazon.com/step-functions/latest/dg/concepts-asl-use-map-state-distributed.html, 2023.