CloudFuice: A flexible Cloud-based Data Integration System

Andreas Thor[1], Erhard Rahm[2]

[1] University of Maryland Institute for Advanced Computer Studies, USA
[2] University of Leipzig, Department of Computer Science, Germany
thor@umiacs.umd.edu, rahm@informatik.uni-leipzig.de

Technical Report (April 21, 2011)

Abstract. The advent of cloud computing technologies shows great promise for web engineering and facilitates the development of flexible, distributed, and scalable web applications. Data integration can notably benefit from cloud computing because integrating web data is usually an expensive task. This paper introduces CloudFuice, a data integration system that follows a mashup-like specification of advanced dataflows for data integration. CloudFuice's task-based execution approach allows for an efficient, asynchronous, and parallel execution of dataflows in the cloud and utilizes recent cloud-based web engineering instruments. We demonstrate and evaluate CloudFuice's applicability for mashup-based data integration in the cloud with the help of a first prototype implementation.

Keywords: Cloud Data Management, Data Integration, Mashups

1 Introduction

Cloud computing technologies shows great promise for web engineering. Distributed data stores (e.g., Google's Bigtable), web-based queue services (e.g., Amazon SQS), and the ability to employ computing capacity on demand (e.g., Amazon EC2) facilitate the development of flexible, distributed, and scalable web applications. Furthermore, recent efforts in entity search engines [5] and the cloud of linked data [2] have lowered the barriers to easily access huge amounts of data.

Data integration [16] can notably benefit from cloud computing because accessing multiple data sources and integration of instance data are usually expensive tasks. For example, entity matching [14], i.e., identification of entities referring to the same real-world object, is key for linking multiple data sources. Since web data is usually dirty, sophisticated matching techniques must employ multiple similarity measures to make effective match decisions. The pair-wise similarity computation usually has quadratic complexity and is thus very expensive for large-scale data integration. On the other hand, data of interest is usually obtained by sending queries to external data sources. However, the use of existing search engines may require several queries for more complex integration tasks to obtain a sufficient number of relevant result entities [9]. Execution of many queries needs to be reliable, i.e., it requires handling of failed queries (e.g., due to network congestion) as well as dealing with source restrictions, such as access quota.

Mashup-based data integration [17] demonstrates a programmatic and dataflow-like integration approach which is complementary to common query- and search-based data integration approaches, e.g., data warehouses or query mediators. In fact, mashups are built on the idea of combining existing services so that they can also use existing search engines and query services. However, current mashup dataflows are mostly comparatively simple and do not yet exploit the full potential of programmatic data integration, e.g., as needed for enterprise applications or to analyze larger sets of web data. Although cloud services make it easy to host a mashup application on multiple servers in the cloud, the development of mashup dataflows that can transparently run on multiple nodes still requires substantial development effort.

In this paper we present CloudFuice, a data integration system that allows for the specification and execution of advanced dataflows for data integration that can be employed within mashups. It utilizes a simple access model for external sources to easily incorporate common web sources such as entity search engines, HTML pages, or RDF data sources. A script language provides operators for common data integration tasks such as query generation and entity matching. Beside fast script development the CloudFuice approach also strives for an efficient execution of dataflows in the cloud. Many integration scenarios may easily involve thousands of entities that need to be processed and thus demand scalability. CloudFuice supports an asynchronous and parallel execution of scripts on multiple machines in a cloud as well as elastic computing, i.e., available cloud capacities can immediately be used when they become available.

In this paper, we make the following contributions:

- We introduce a powerful data integration script language that is tailored for web data integration dataflows. (Section 2)
- We detail how a dataflow can be transformed into independent tasks that in turn can be executed asynchronously and in parallel in the cloud. (Section 3)
- We present the CloudFuice architecture facilitating dataflow execution in the cloud. We describe our prototype that also serves as a mashup development tool. (Section 4)
- We evaluate CloudFuice's parallel and asynchronous execution approach and demonstrate that it can significantly reduce the execution time of dataflows. (Section 5)

We discuss related work in Section 6 before we conclude.

2 Dataflow Definition

In this section we describe how a developer can specify data integration dataflows. To this end we present the structure of data sources and entities, data structures and operators, and their combined use within script programs.

2.1 Entities and data sources

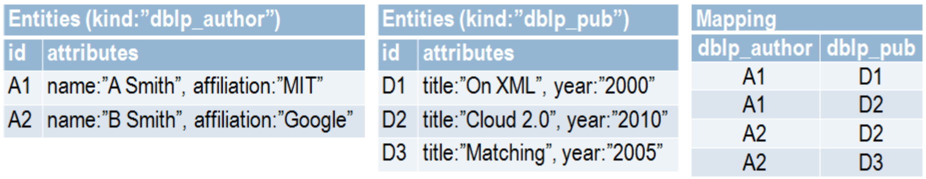
CloudFuice employs the simple and flexible entity-attribute-value model. An entity is of a certain kind and has a kind-specific id, i.e., the pair (kind, id) identifies an entity unambiguously. Each entity has a (possibly empty) list of attributes that can be single-valued (numbers, strings) or multi-valued (i.e., set of single values). This key-value format can be well supported by cloud-based key value stores.

CloudFuice accesses external data sources to obtain entities via three defined methods: search, get, and link. These methods reflect the characteristics of typical web data sources such as (static) web pages, (entity) search engines, and RDF data sources. We therefore assume that it is comparatively easy to implement the following methods (as web services) for a particular data source. Note that not all methods need to be available for all sources.

The **search method** returns entities of a given kind using a specified query. CloudFuice doesn't impose any restrictions on the type of query. The query could be a simple keyword if the source supports keyword search, e.g., a product search engine. If the source supports structured query languages such as SQL or SPARQL, the query could be more complex, e.g., a SQL WHERE condition. The **get method** retrieves all available information for a given entity. This is especially important for frequently changing information, e.g., the number of citations of a publication or the current offered price for a certain product. A possible realization of the get method is a request to an entity specific HTML page (e.g., a product detail page) with subsequent screen scraping. The **link method** retrieves all entities that are connected to a specified entity. This method reflects the nature of web data where the entities are inter-connected by HTML hyperlinks or RDF links in the cloud of linked data. Examples are the list of citing publications for a given publication or the list of products for a specified manufacturer.

2.2 Data structures and operators

CloudFuice follows a script programming approach for defining information integration dataflows. It builds up on our iFuice idea [20] and is tailored to web data integration by providing simple and powerful operators. CloudFuice does not require prior generation of metadata models (global schemas) but lets developers take responsibility to define semantically meaningful integration within their scripts. For example, CloudFuice introduces queries as building blocks and thus enables developers to carefully construct queries in a programmatic way for specific data sources.



Entities (kind:"dblp_author")	
id	attributes
A1	name:"A Smith", affiliation:"MIT"
A2	name:"B Smith", affiliation:"Google"

Entities (kind:"dblp_pub")	
id	attributes
D1	title:"On XML", year:"2000"
D2	title:"Cloud 2.0", year:"2010"
D3	title:"Matching", year:"2005"

Mapping	
dblp_author	dblp_pub
A1	D1
A1	D2
A2	D2
A2	D3

Fig. 1. Examples for two entities sets (left) of kind dblp_author and dblp_pub, respectively, and a mapping (right).

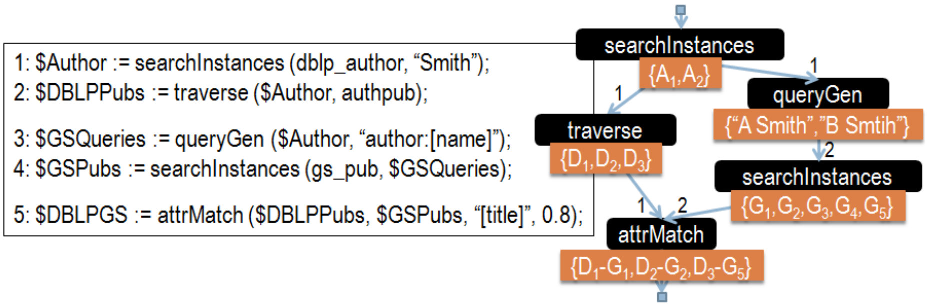
A CloudFuice script contains several operator calls like in programs of imperative programming languages. Figure 2 illustrates an example script that will be used throughout the paper and will be discussed in more detail in Section 2.3. The output of any operator can be bound to a variable for later use, e.g., as input to other operators. Script programmers should not be limited to the execution of one source access method at a time but are provided with powerful set-oriented operators. To this end, CloudFuice supports three set-based data structures.

Entities of the same kind A can be subsumed in a set E_A. Figure 1 (left) illustrates two sets of entities. One is a set of DBLP authors, the other a set of DBLP publications. **Mappings** represent directed binary relationships between entities. A mapping $M_{A \times B}$ is a set of correspondences, i.e., pairs of entities of kind A and B, respectively. Figure 1 (right) depicts an example mapping between the DBLP authors and DBLP publications. The author named "A Smith" is linked to the XML and the Cloud paper whereas "B Smith" relates to the Cloud and the Matching paper. Correspondences can be annotated with similarity values or other meta data, e.g., to reflect a confidence level for entity matching [22]. In this paper we restrict the mapping definition to entity pairs without annotations but we present an extension in Appendix A.

Queries are the third type of data structures because they are key to efficient data retrieval from web sources. Providing an explicit data structure for queries Q enables the generation and processing of queries and therefore allows for sophisticated query generation mechanisms [9] that are especially important for entity search engines.

All data structures can be both input and output for operators. CloudFuice provides operators for fetching data from sources and for matching entities. In addition, auxiliary operators are provided for basic data processing. In the following we give a brief overview of selected operators. A complete operator list (incl. signature and definition) can be found in Appendix B.

Source operators fetch data from (web) data sources by employing set-based source access methods. For example, the operatores searchInstances and getInstances take a set of queries or entities, respectively, and call the corresponding source method, search or get, for every single query/entity. The operator result is then the union of the source methods result. The operators traverse and map employ the link method in a similar way. The difference between these two operators is that traverse returns the union of all link results whereas map



```
1: $Author := searchInstances (dblp_author, "Smith");
2: $DBLPPubs := traverse ($Author, authpub);

3: $GSQueries := queryGen ($Author, "author:[name]");
4: $GSPubs := searchInstances (gs_pub, $GSQueries);

5: $DBLPGS := attrMatch ($DBLPPubs, $GSPubs, "[title]", 0.8);
```

Fig. 2. Left: Script notation of an example dataflow for the integration of publications from DBLP and Google Scholar (GS). **Right:** Graph representation of the example dataflow. The graph also contains example output for each operator, e.g., entities $\{A_1, A_2\}$ for the first searchInstances operator.

incorporates the input entities by returning correspondences between input entities and the resulting link entities. **Matching operators** provide techniques for entity matching. The input of a matching operator are two sets of entities and the output is a mapping containing the corresponding, i.e., matching entities. To further restrict the matching, the input can already be a mapping instead of two input sets. The matching operator then only compares entity pairs of the input mapping. CloudFuice provides a generic attribute matcher attrMatch for the common use case of entity matching using attribute similarity and a threshold. In addition, external match algorithms can be plugged in via the match operator. **Auxiliary operators** act as the "glue" between operators. Since all data structures are set-based, common set operations union, intersect, and diff can be applied. The compose operator allows for composition of mappings, e.g., for input correspondences (a, b) and (b, c) compose derives the output correspondence (a, c). The filter operator reduces a set of entities or correspondences using a filter criterion. Finally, queryGen allows for a flexible query generation based on entities' attribute values [9].

2.3 Scripts and dataflows

Figure 2 (left) shows an example CloudFuice script that integrates publication data from DBLP and Google Scholar. The first script line searches for authors in the DBLP data source and binds the resulting entities to the variable $Author. The retrieved authors are input to a traverse operator (line #2) that retrieves the corresponding DBLP publications D_1, D_2, and D_3. The author entities are also provided to the queryGen operator (line #3) that generates queries using the authors's names. In the example, the two authors A_1 and A_2 generate one query each and, thus, two queries ("A Smith" for A_1 and "B Smith" for A_2) are bound to the $GSQueries variable and sent to the subsequent searchInstances operator (line #4). The searchInstances operator executes both queries and returns a merged set of Google Scholar publications G_1 to G_5. Finally, the attrMatch operator (line #5) takes both the DBLP and the GS publications as input and determines matching publications. In the example script of Figure 2, two publications are considered to match if they have a title similarity greater than or equal to 0.8. The result is then stored in the variable named $DBLPGS.

CloudFuice scripts define a **dataflow**, i.e., an acyclic directed graph with operators as nodes and edges that connect the output of an operator to an input parameter of another operator. Figure 2 (right) shows the dataflow graph for the example script. Each incoming edge is annotated with a parameter index to map the operator output to the specified input parameter. For example, the parameter indexes of two incoming edges for attrMatch in Figure 2 are 1 and 2, respectively, to indicate that the traverse output is the first parameter whereas the searchInstances output is the second parameter of attrMatch. Operator parameters that are not output of other operators, e.g., the similarity threshold of attrMatch, are considered to be available anytime and we therefore do not include them in the dataflow graph. The dataflow graph is the foundation for CloudFuice's execution approach that we will explain in the next section.

3 Dataflow Execution

We introduce CloudFuice's approach to dataflow execution and the application of inter- and intra-operator parallelism. Furthermore, we present the full execution plan for our running example.

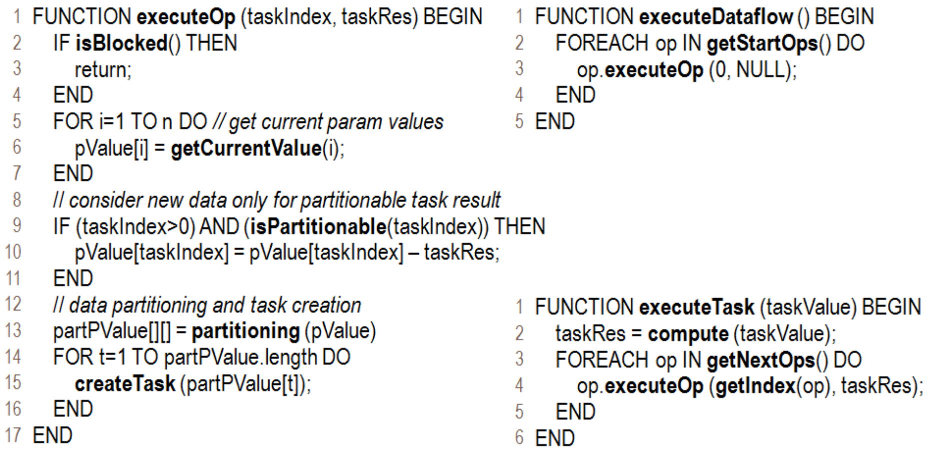
3.1 Dataflow execution approach

Operators of CloudFuice dataflows are executed within one or multiple operator-specific tasks that can be run in parallel on distributed cloud servers. Tasks may concurrently store their results in a distributed datastore. Furthermore, tasks can be executed as soon as computing resources are available in the cloud to obtain a minimal overall execution time. Finally, our execution model is based on a dynamic invocation of tasks such that each finished task invokes the generation of all operators following in the dataflow as we will detail below.

To support efficient, parallel execution of dataflows, we support both intra- and inter-operator parallelism similar as in parallel database systems [7]. In addition to pipeline parallelism between adjacent operators, data partitioning is utilized to run independent operators on different data in parallel and to parallelize operators on disjoint data partitions.

However, intra-operator parallelism is subject to **partitionable** input data (i.e., operator parameters) only. A parameter p_k of an n-ary operator op is called partitionable if and only if the following two properties hold: (1) The parameter p_k is a set of entities E, or a mapping M (i.e., set of correspondences), or a set of queries Q. (2) For any complete and disjoint partitioning $p_k = p_{k_1} \cup p_{k_2}$ of p_k holds: $op(p_1, \ldots, p_k, \ldots, p_n) = op(p_1, \ldots, p_{k_1}, \ldots, p_n) \cup op(p_1, \ldots, p_{k_2}, \ldots, p_n)$.

We call an operator blocking if none of its parameters is partitionable. A blocking operator is therefore unable to produce any (partial) result until all



```
FUNCTION executeOp (taskIndex, taskRes) BEGIN
  IF isBlocked() THEN
    return;
  END
  FOR i=1 TO n DO // get current param values
    pValue[i] = getCurrentValue(i);
  END
  // consider new data only for partitionable task result
  IF (taskIndex>0) AND (isPartitionable(taskIndex)) THEN
    pValue[taskIndex] = pValue[taskIndex] – taskRes;
  END
  // data partitioning and task creation
  partPValue[][] = partitioning (pValue)
  FOR t=1 TO partPValue.length DO
    createTask (partPValue[t]);
  END
END
```

```
FUNCTION executeDataflow () BEGIN
  FOREACH op IN getStartOps() DO
    op.executeOp (0, NULL);
  END
END
```

```
FUNCTION executeTask (taskValue) BEGIN
  taskRes = compute (taskValue);
  FOREACH op IN getNextOps() DO
    op.executeOp (getIndex(op), taskRes);
  END
END
```

Fig. 3. Pseudocode for executing dataflows, operators, and tasks.

input data is available. On the other hand, all parameters (of type E, M, or Q) of non-blocking operators are partitionable. For example, all source operators (e.g., searchInstances) are non-blocking. Operators that are neither blocking nor non-blocking are called partially blocking. For example, the diff operator computes the difference of two entity sets E_1 and E_2 of the same kind. The first parameter E_1 is partitionable whereas the second parameter E_2 is not. The operator needs to know all entities E_2 that must not appear in the operator result before returning any (partial) results. The Appendix of [23] specifies the type for each operator.

The advantages of partitionable input data for an operator are twofold. First, an operator can partition each partitionable parameter and thereby split the operator execution into multiple tasks that can be executed in parallel (intra-operator parallelism). The complete operator result can later be reconstructed from all task results. On the other hand, data partitioning can also be used for asynchronous (pipelined) operator execution. Individual task results, i.e., partial operator results, can already be handed off (or "pushed") to its succeeding operator (which in turn may already produce a partial result) if the corresponding input parameter of the succeeding operator is partitionable, too. This method therefore allows for an overlapping execution of neighboring operators in the dataflow graph. Finally, independent operators, i.e., operators that neither directly nor indirectly rely on each others' outputs, are run in parallel (inter-operator parallelism).

Dataflow execution thus entails the correct transformation of a dataflow into a series of independently executable tasks. Figure 3 shows the pseudo code for task generation (methods for dataflow and operator execution) and task execution. Initially the tasks for all start operators, i.e., operators that do not rely on input of other operators, are generated and executed (see method executeDataflow). All other operators will be dynamically invoked by the tasks creating their input parameters (see method executeTask). The input-generating task then hands over its complete result (taskRes) as well as the associated parameter index (taskIndex) as parameters to the executeOp call. For the invocation of a start operator that does not depend on input data, we use a taskIndex value of 0 (see executeDataflow).

The executeOp method partitions the input data based on an operator-specific partitioning function partitioning and creates a corresponding task for each partition. CloudFuice thereby allows tailored partitioning strategies for operators (see Section 5 for a discussion and evaluation). The task execution first employs the operator-specific computation (compute) to achieve the task result which is then stored in a distributed data store. Afterwards all following operators are called (with the task result and the parameter index) which in turn may eventually create new tasks.

The methods for operator and task execution make use of a few auxiliary functions. The function isBlocked returns true if there is at least one non-partitionable input parameter p_i with incomplete data, i.e., the corresponding input operator has not finished yet, and thus the operator cannot be executed yet. It makes use of isPartitionable(i) which is true if the parameter p_i is partitionable. getCurrentValue(i) reads the current value of parameter p_i from the datastore. The function getStartOps returns all start operators of the dataflow and getNextOps returns the following operators for a task. With the help of getIndex(nextOp) the parameter index for which the current task provides input to operator nextOp is determined. Finally, the function createTask generates a new task that will eventually be executed by the runtime infrastructure (see Figure 5).

The approach for asynchronous executions of operators/tasks also allows for notifications when an operator is finished and, thus, when the value of the bounded script variable is available. An operator is finished if all of its input operators are finished and there are no unfinished operator tasks. To this end each operator keeps track of the number of generated tasks as well as the number of finished tasks. Finally, a script is finished if all operators are finished. This additional functionality is not shown in Figure 3 in favor of readability.

3.2 Execution example

Finally, we demonstrate how the example script of Figure 2 will be executed. Figure 4 illustrates the dataflow along with the generated tasks and their execution time frame. The only start operator, searchInstances, is invoked for one query at the beginning and as a result it generates one search task. Once finished, it triggers the two subsequent operators traverse and queryGen. They can run in parallel because they are independent from each other (inter-operator parallelism). The traverse operator makes use of intra-operator parallelism and generates two tasks because it is given two input entities (DBLP authors A_1 and A_2). The example assumes that there is no access restriction to the data source so that both tasks can be executed immediately. The queryGen operator is also invoked for A_1 and A_2 but generates only one task that emits two queries.

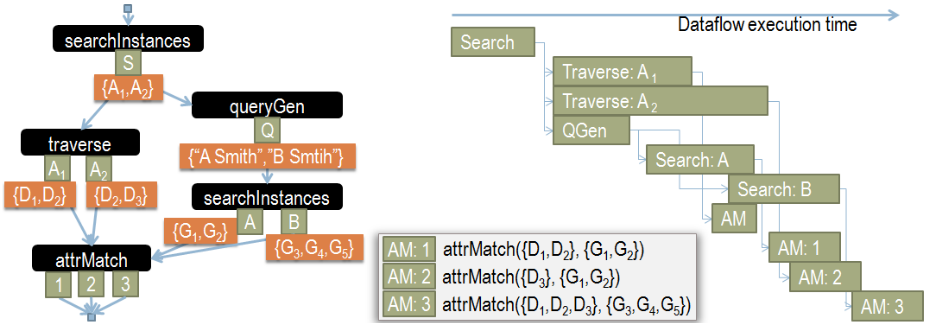


Fig. 4. Example execution of the dataflow depicted in Figure 2. In addition to Figure 2 the dataflow graph also shows the generated tasks (green quadrats below the operator box). The right part shows a timeline of the executed tasks.

This task – once finished – invokes the second searchInstances operator. The two input queries lead to two search tasks. In this example we assume that there is a source access quota so that the first task can be executed immediately whereas the second task is scheduled with consideration of a reasonable waiting time (handled by a task scheduler, see Section 4).

The last operator, attrMatch, takes as input the output of the two operators traverse and searchInstances. The operator input data comes in four data chunks because both input operators employ two tasks each for their execution. The order of the incoming data chunks is arbitrary and in the example of Figure 4 we assume the following order: traverse:A1 $(= \{D_1, D_2\})$, search:A $(= \{G_1, G_2\})$, traverse:A2 $(= \{D_2, D_3\})$, and search:B $(= \{G_3, G_4, G_5\})$. Once finished, each task invokes the attrMatch operator.

In our example we consider a non-blocking implementation of attrMatch, i.e., attrMatch process all pairs of the Cartesian product independently, e.g., with the help of common string similarity measures such as Edit Distance. It is therefore amenable to asynchronous execution, i.e., it can produce partial results on partial input data. However, the first operator call does not yet create any tasks because at this point there are only DBLP publications available but no GS publications. The second operator call generates the first task AM:1 that matches all available DBLP entities with the new available GS entities (see the box in the middle of Figure 4). The third operator call completes the DBLP input entities and creates task AM:2. This task process the new DBLP entities (only D_3 is new because D_1 and D_2 have been already retrieved by the previous traverse task) along with all available GS entities. Finally, the second query of searchInstances is finished and the corresponding task calls attrMatch for the fourth time. To this end, attrMatch creates task AM:3 that takes as input all available DBLP entities and the newly added GS entities $\{G_3, G_4, G_5\}$. The box in the center of Figure 4 summarizes the three employed attrMatch tasks along with their input data. It thereby illustrates that the attrMatch operator indeed processes the entire cross product of all DBLP and GS entities. In the example of Figure 4 the partitioning

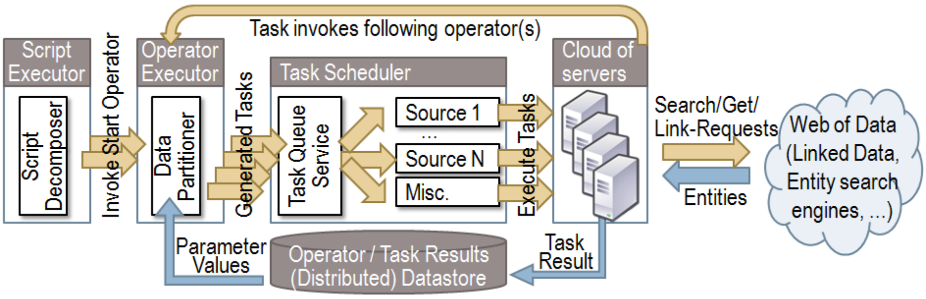


Fig. 5. Architecture of the CloudFuice approach. It employs a distributed data store, a task queue service, and a cloud of servers for task execution.

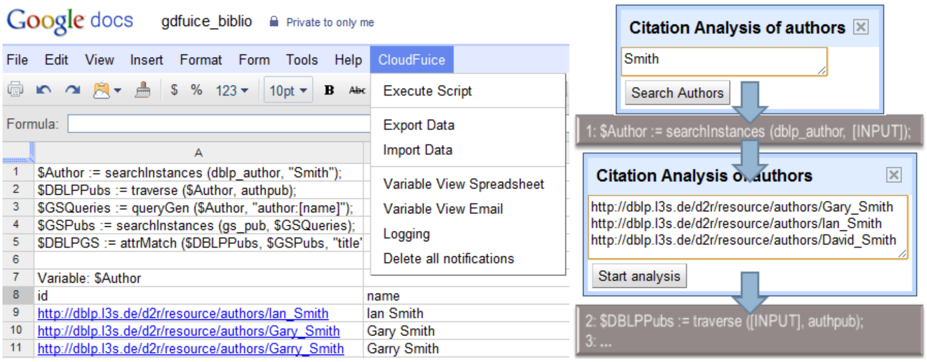
is solely driven by the outcome and the execution order of the input operator tasks. However, large entity sets may require an additional partitioning to reduce the workload per task (see Section 5).

4 Web-based Architecture and Prototype

Figure 5 depicts the overall CloudFuice architecture. It comprises a script and operator executor, a task scheduler, and an elastic cloud of servers. The script and operator executor realize CloudFuice's execution approach. A given script is first converted into a dataflow and then decomposed into a set of operator calls which are in turn transformed into multiple tasks. Tasks are subject to a scheduling mechanism that takes into account access restrictions of external sources. Tasks are executed on a cloud of servers that are capable of (parallel) executing tasks as web services. All tasks store their results in a distributed data store and invoke other operators if necessary. If the last task of an operator has finished, it may notify external applications about the availability of an operator result (not shown in Figure 5).

The script executor analyzes a given CloudFuice script and decomposes it into a data flow. The script executor then invokes all start operators, i.e., all operators that do not rely on input of other operators. The operator executor retrieves the relevant parameter values from the datastore. If the operator can be executed, the executor applies an operator-specific partitioning strategy on the input data and generates one or multiple tasks. Each task is assigned the relevant partition of the operator's input data as well as the list of following operators. Tasks are sent to the task scheduler.

The task scheduler provides a task queue for each source and a task is appended to a task queue if it is supposed to send a request to the corresponding source. Each task queue employs a simple bucket algorithm for scheduling task execution because source access is typically limited by quotas and exceeded quotas may cause requests failures. The bucket size determines how many tasks can be executed in parallel, i.e., how many requests can be sent to the source simultaneously. The execution rate controls the average number of task executions for



	A	
1	$Author := searchInstances (dblp_author, "Smith");	
2	$DBLPPubs := traverse ($Author, authpub);	
3	$GSQueries := queryGen ($Author, "author:[name]");	
4	$GSPubs := searchInstances (gs_pub, $GSQueries);	
5	$DBLPGS := attrMatch ($DBLPPubs, $GSPubs, "title'	
6		
7	Variable: $Author	
8	id	name
9	http://dblp.l3s.de/d2r/resource/authors/Ian_Smith	Ian Smith
10	http://dblp.l3s.de/d2r/resource/authors/Gary_Smith	Gary Smith
11	http://dblp.l3s.de/d2r/resource/authors/Garry_Smith	Garry Smith

Smith

```
1: $Author := searchInstances (dblp_author, [INPUT]);
```

http://dblp.l3s.de/d2r/resource/authors/Gary_Smith
http://dblp.l3s.de/d2r/resource/authors/Ian_Smith
http://dblp.l3s.de/d2r/resource/authors/David_Smith

```
2: $DBLPPubs := traverse ([INPUT], authpub);
3: ...
```

Fig. 6. Left: Screenshot of a Google Spreadsheet document with extended menu bar. The spreadsheet may contain multiple scripts as well as variable content. **Right:** Example mashup workflow using user input dialogs and parametrized CloudFuice scripts.

a time window, e.g., 1 task (request) per second. The scheduler also provides an additional unlimited task queue ("miscellaneous") for all other tasks. Since tasks mail fail due to several reasons (e.g., network congestion or server unavailability) each task queue also provides a retry mechanism for failed tasks.

The CloudFuice architecture realizes task execution by web service requests. Requests can be handled by different servers (nodes) that all have access to a distributed data store for storing task results. The actual task implementations are encapsulated as web services and can, thus, be realized in virtually any programming language. Each task may request data from the Web using one of the source access methods (search, get, or link). Note that the task implementation does not need to deal with any source constraints because this is already handled by the task scheduler. Moreover, no global coordinator is needed for the task execution because each task knows what operator is supposed to be invoked after its completion.

The current prototype implementation employs Google App Engine, a platform for running web applications on multiple nodes. New server instances become automatically (un)available based on the number of tasks. All operators and tasks are implemented as REST-based web services and use JSON as data exchange format. The prototype employs Google App Engine's datastore that implements the Bigtable data model [4]. Task results can be concurrently stored using a unique task id which is assigned during task generation. Each task result contains a reference to the corresponding operator and, thus, operator results can be retrieved by merging all corresponding task results. The prototype employs Google Spreadsheet as script development environment (see Figure 6 (left) for a screenshot). The spreadsheet contains both CloudFuice scripts and data and, thus, will serve as an integrated development environment. Google Spreadsheet executes CloudFuice scripts and retrieves data by HTTP requests. On the other hand, Google Spreadsheet's API enables the CloudFuice server to directly update results in a spreadsheet (data pushing) even if the spreadsheet

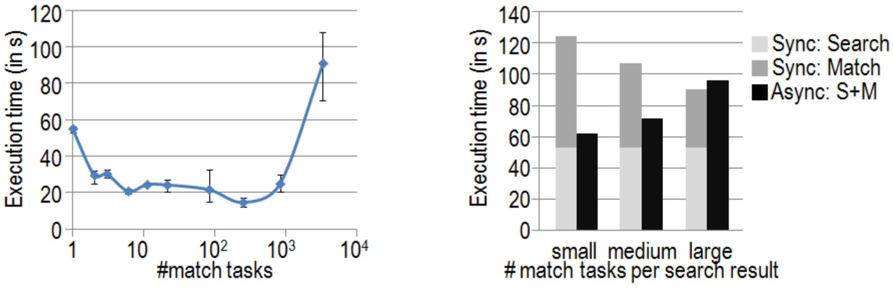


Fig. 7. Evaluation of partitioning strategies for entity matching (left) and the influence of asynchronous dataflow execution (right).

is closed. Google Apps Script is used to implement simple user interfaces and execute parametrized CloudFuice scripts. This mechanism acts as a simple way for mashup development. Figure 6 (right) illustrates an example mashup.

5 Evaluation

We evaluate the practicability of our approach with the help of our prototype and thereby demonstrate that the use of cloud technologies can improve the runtime performance of CloudFuice scripts. In particular we will examine partitioning strategies for entity matching and the effect of asynchronous script execution.

The first experiment deals with effective and efficient data partitioning which is key to a correct and efficient execution of dataflows. Recall that the execution approach ensures that only partitionable data is subject to intra-operator parallelism and that different partitioning strategies can be applied. Fine-grained partitioning strategies generate small tasks that can be executed in parallel but suffer from the additional overhead of task creation, scheduling, and execution. Furthermore, danger of skew effects also grows with more tasks since the slowest task determines the overall execution time. On the other hand coarse-grained strategies result in few large tasks that may not fully exploit the power of the available computing resources. For intra-operator parallelism, we evaluate size-based partitioning functions for entity matching similar to [13]. A maximal block size b ensures that each match task only process a limited part of the Cartesian product, i.e., at most $b \times b$ entities per task. For our experiment we employ attrMatch with two entity sets of size $|R| = 471$ and $|S| = 16,269$, respectively. Each set is evenly divided into blocks of maximal size b and one task is generated for each pair of blocks. For example, a block size $b = 100$ leads to 5 blocks for R and 163 blocks for S and thus $5 \cdot 163 = 815$ match tasks are created.

Figure 7 (left) shows the measured average script execution time (average over three runs, minimal and maximal values are shown as error bars) for different block sizes. The x-axis denotes the resulting number of tasks. The execution time can be significantly reduced from 55 seconds (computation is realized by one task only) to 14 seconds if approx. 250 tasks are employed. However, if the number of tasks is increased even further the execution time is overwhelmed by the additional task management overhead. Note that the prototype runs on Google App Engine that does not allow configuration of computing capacities, i.e., creation and utilization of node instances cannot be controlled. Instances become instead available and unavailable, respectively, by an internal heuristics based on the current number of tasks.

In a second experiment we demonstrate the influence of asynchronous script execution (inter-operator parallelism). The evaluation scripts contains two steps. First, 1,180 entities are requested by 49 queries (searchInstances). We assume a reasonable source quota of 1 query per second, i.e., query execution process takes about 50 seconds. In a second step the obtained entities are matched against a given set of 16,269 entities like in our previous experiment. As shown in our running example (see Figure 4) both input sets of attrMatch are partitionable, i.e., entity matching can already process partial input data. For a synchronous execution we assume that none of the attrMatch parameters are partitionable and thus the operator cannot start until all input data becomes fully available. Synchronous execution behavior of attrMatch is achieved by modifying the isBlocked function accordingly (see pseudo code in Figure 3).

Figure 7 (right) compares the overall times for synchronous vs. asynchronous execution using different partitioning strategies for attrMatch. The synchronous execution is composed of the search time and matching time. The search time remains the same for all partitioning strategies because it is dominated by the source quota. On the other hand, an increasing number of tasks decreases the time for entity matching similar to what we have observed in Figure 7 (left). Figure 7 (right) proves that CloudFuice's asynchronous dataflow execution can significantly reduce the execution time by an early hand-off of partial results. The source access is, of course, a lower bound but the asynchronous model already performs entity matching while still querying the data source. It thereby reduces the remaining workload after the last query result becomes available. However, the asynchronous model is characterized by an increasing execution time for an increasing task number. This is due to the fact that each search result retrieves a comparatively small number of entities (≈ 24) which then have to be matched against the large set of 16,269 entities. Obviously a fine-grained partitioning is not beneficial for such imbalanced match tasks. For example, if we assume that each search result retrieves at most 100 entities a block size of $b = 100$ would result in 163 tasks per search result. The overall number of match tasks is therefore 49 (search queries) $\times$ 163 = 7,987 that deteriorates execution time.

In general, CloudFuice's execution model has to combine a partitioning strategy with an incremental availability of input data due to asynchronous execution. This is especially important against the background of an additional overhead for task generation, scheduling, and transferring. We will therefore investigate in adaptive partitioning strategies in future work. For example, a partitioning strategy may not create any tasks until a minimal number of entities is available when dealing with partial results. Furthermore, operators that do not have following operators might apply a different partitioning strategy because there is no benefit in forwarding partial results.

6 Related Work

Mashup-based data integration has become very popular in recent years and many tools and frameworks have been developed [17]. Applications need components for data, process, and presentation level [18] and CloudFuice mainly focuses on the data level. A popular approach are pipes, e.g., as used in Yahoo! Pipes, Damia [21], or [15], that process entity sets via relatively simple user-specified dataflows. These tools have demonstrated to be applicable in different settings since they offer a powerful and easy-to-use interface for inexperienced users. However, they are not designed for more advanced data integration problems that have to deal with dirty data and, thus, require advanced operators. For example, entity matching and query generation are key to achieve accurate and complete integrated results. This, on the other hand, requires more advanced skills in mashup development. CloudFuice therefore strives for a good balance between powerful data integration operators and a simple scripting language.

A few recent mashup platforms also deal with mashup efficiency. CoMaP [10] targets a distributed mashup execution that minimizes the overall mashup execution time of multiple hosted mashups. It is based on a general mashup dataflow model with operators that can be executed on different nodes. A dynamic scheduling takes into account several parameters such as network and users. The AMMORE [11] system even modifies original mashup dataflows to avoid duplicate computations and unnecessary data retrievals. To this end, AMMORE identifies common operator sequences in different mashups and executes them together. CoMap, AMMORE, and CloudFuice share the same goal of efficient dataflow execution but have slightly different focuses. CloudFuice does not deal with multiple mashups and users but task scheduling is driven by data source constraints and available computing capacity instead of usage or network traffic.

The recent shift in web infrastructures from high-end server systems to clusters of commodity hardware (also known as cloud) has triggered research in parallel data processing in a wide variety of applications. Driving forces are simple and powerful parallel programming models such as MapReduce [6] and Dryad [12]. Although the MapReduce program model is limited to two dataflow primitives (map and reduce), it has proven to be very powerful for a wide range of applications. On the other hand Dryad supports general dataflow graphs. Freely available frameworks such as Hadoop further stimulate the popularity of distributed data processing. Higher-level languages can be layered on top of these infrastructures. Examples include the high-level dataflow language Pig Latin [19], Nephele/PACTs [1] (based on MapReduce), DryadLINQ [24] as well as SCOPE [3] that offers a SQL-like scripting language on top of Microsoft's distributed computing platform Cosmos. Similar to Cloudfuice, a high-level program is decomposed into small buildings blocks (tasks) that are transparently executed in a distributed environment. On the other hand, CloudFuice targets the fast development of dataflows for web applications and therefore relies on a common web engineering tools. Furthermore CloudFuice deals with asynchronous dataflow execution, e.g., due to continuous data input from query results, in contrast to synchronized offline data analysing/processing. For example, MapReduce enforces synchronization between the map and reduce phase (a disadvantage that has been recently addressed in [8]).

7 Conclusions and Future Work

We presented CloudFuice, a flexible system for specification and execution of dataflows for data integration. The task-based execution approach allows for an efficient, asynchronous, and parallel execution within the cloud and is tailored to recent cloud-based web engineering instruments. We have demonstrated CloudFuice's applicability for mashup-based data integration in the cloud with the help of a first prototype implementation.

Future work includes the development of adaptive partitioning strategies as well as the further development toward a comprehensive mashup platform. We will also investigate how dataflows can be automatically generated from declarative queries.

References

1. Battré, Ewen, Hueske, Kao, Markl, and Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, 2010.
2. Bizer, Heath, and Berners-Lee. Linked data - the story so far. *IJSWIS*, 5(3), 2009.
3. Chaiken, Jenkins, Larson, Ramsey, Shakib, Weaver, and Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.
4. Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes, and Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26, 2008.
5. Cheng, Yan, and Chang. Entityrank: Searching entities directly and holistically. *VLDB*, 2007.
6. Dean and Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.
7. DeWitt and Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6), 1992.
8. Elteir, Lin, and Feng. Enhancing mapreduce via asynchronous data processing. In *ICPADS*, 2010.
9. Endrullis, Thor, and Rahm. Evaluation of Query Generators for Entity Search Engines. In *USETIM*, 2009.
10. Hassan, Ramaswamy, and Miller. Comap: A cooperative overlay-based mashup platform. *CoopIS*, 2010.
11. Hassan, Ramaswamy, and Miller. Enhancing Scalability and Performance of Mashups Through Merging and Operator Reordering. In *ICWS*, 2010.
12. Isard, Budiu, Yu, Birrell, and Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *EuroSys Conference*, 2007.

13. Kirsten, Kolb, Hartung, Gross, Köpcke, and Rahm. Data Partitioning for Parallel Entity Matching. In *QDB*, 2010.
14. Köpcke and Rahm. Frameworks for entity matching: A comparison. *Data Knowl. Eng.*, 69(2), 2010.
15. Le-Phuoc, Polleres, Hauswirth, Tummarello, and Morbidoni. Rapid Prototyping of semantic Mash-ups through semantic Web Pipes. In *WWW*, 2009.
16. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.
17. Lorenzo, Hacid, Paik, and Benatallah. Data Integration in Mashups. *SIGMOD Rec.*, 38, 2009.
18. Maximilien, Wilkinson, Desai, and Tai. A domain-specific Language for Web APIs and Services Mashups. In *ICSOC*, 2007.
19. Olston, Reed, Srivastava, Kumar, and Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.
20. Rahm, Thor, Aumueller, Do, Golovin, and Kirsten. iFuice - Information Fusion utilizing Instance Correspondences and Peer Mappings. In *WebDB*, 2005.
21. Simmen, Altinel, Markl, Padmanabhan, and Singh. Damia: Data Mashups for Intranet Applications. In *SIGMOD*, 2008.
22. Thor and Rahm. MOMA - A Mapping-based Object Matching System. *CIDR*, 2007.
23. Thor and Rahm. CloudFuice: A flexible Cloud-based Data Integration Approach. Technical report, University of Leipzig, 2011. `http://dbs.uni-leipzig.de/publication/year/2011`.
24. Yu, Isard, Fetterly, Budiu, Erlingsson, Gunda, and Currey. Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

A Extended Mapping Semantics

Mappings represent instance level correspondences between entities. A simple representation is a set of entity pairs, i.e., $M_{A\times B} = \{(a,b)|a \in A \wedge B \in B\}$. We extend this semantics and introduce two meta-data values, *confidence* and *weight*, that are used to annotate each entity pairs: $M_{A\times B} = \{(a,b,c,w)|a \in A \wedge B \in B \wedge c \in [0,1] \wedge w \in \mathbb{N}^{+}\}$.[1] The value c describes the confidence in the correctness of the entity correspondences and a value of $c = 1$ indicates the highest confidence. Confidence values are usually employed by entity matching algorithms to reflect the similarity of the corresponding entities. The weight w is a relative (positive) number to characterize the relative importance of a correspondence within a mapping. It is used to for a meaningful computation of confidence values for the operators union, intersect, and compose. The default value for both c and w equals 1, i.e., the simple mapping semantics (without confidence and weight) can be represented using the extended semantics and $c = w = 1$.

The extended mapping semantics, of course, requires an extended operator definition. Operators combining two mappings need to determine confidence and

[1] A mapping M still represents a duplicate free set of entity pairs, i.e., $(a,b,c_1,w_1) \in M \wedge (a,b,c_1,w_2) \in M \rightarrow c_1 = c_2 \wedge w_1 = w_2$.

Operator	Parameters	Definition
union	$M1_{A \times B}$, $M2_{A \times B}$	$= \{(a, b, c, w) \mid (a, b, c_1, w_1) \in M1 \wedge (a, b, c_2, w_2) \in M2 \wedge c = \frac{c_1 \cdot w_1 + c_2 \cdot w_2}{w_1 + w_2} \wedge w = \mathsf{max}(w_1, w_2)\}$ $\cup \mathsf{diff}(M1, M2) \cup \mathsf{diff}(M2, M1)$
intersect	$M1_{A \times B}$, $M2_{A \times B}$	$= \{(a, b, c, w) \mid (a, b, c_1, w_1) \in M1 \wedge (a, b, c_2, w_2) \in M2 \wedge c = \frac{c_1 \cdot w_1 + c_2 \cdot w_2}{w_1 + w_2} \wedge w = \mathsf{min}(w_1, w_2)\}$
diff	$M1_{A \times B}$, $M2_{A \times B}$	$= \{(a, b, c_1, w_1) \mid (a, b, c, w_1) \in M1 \wedge \not\exists (c_2, w_2) : (a, b, c_2, w_2) \in M2\}$
compose	$M1_{A \times X}$, $M2_{X \times B}$	$= \{(a, b, c, w) \mid T = \{x_k \mid (a, x_k, c_{1,k}, w_{1,k}) \in M1 \wedge (x_k, b, c_{1,k}, w_{1,k}) \in M2\} \wedge \lvert T \rvert > 0$ $\wedge c = \frac{\Sigma_{x_k \in T} \left(s_{1,k} \cdot w_{1,k} \cdot s_{2,k} \cdot w_{2,k} \right)}{\Sigma_{x_k \in T} \left(w_{1,k} \cdot w_{2,k} \right)} \wedge w = \Sigma_{x_k \in T} \left(w_{1,k} \cdot w_{2,k} \right)$

Table 1. Definition of operators that combine two mappings using the extended mapping semantics. The operator definition comprises the corresponding entity pairs (a, b) as well as the annotated confidence c and weight w. As with the simple semantics (see Table 2) the intersect operator is non-blocking and diff is partially blocking. The extended semantics transforms union and compose into blocking operators.

weight values for each resulting correspondence. Table 1 lists the according definitions.

Note that the union operator is now blocking. If union "sees" a correspondence (a, b, c_1, w_1) in either one of the input mapping it still guarantees that the entity pair (a, b) appears in the operator result which is why union is non-blocking under the simple semantics. However, the values for confidence c and weight w depend on the existence of a correspondence (a, b, c_2, w_2) in the other input mapping and, thus, union is blocking under the extended semantics. Similarly, compose is also blocking because it relies on the complete set of correspondences involving intermediate entities x_k to determine the annotations for (a, b). The diff operator remains partial blocking and intersect remains non-blocking.

B List of operators

Table 2 lists all CloudFuice operators. It shows their signature and provides a brief semi-formal definition. The last column indicates if the operator is non-blocking (nb) or partially blocking (pb). In the latter case the non-distributive parameter is additionally given. The operators union and compose are blocking (b) if the extended mapping semantics is used. We introduce this semantics in Appendix A and argue why the mentioned operators become blocking operators.

The operator attrMatch is described with a reduced parameter list. If the names of the comparison attributes differ for the two entity sets, the attr parameter can be replaced by two parameters attr1 and attr2. The operator then compares attribute values a.attr1 with b.attr1. The similarity computation employs a default similarity metrics. The usage of a different metrics can be indicated by an additional parameter sim.

Operator	Parameters	Definition	Blocking
Source Operators			
searchInstances	kind, Q	$E_{kind} = \bigcup_{q\in Q}$ search(kind, q)	nb
getInstances	E_A	$E_A = \bigcup_{e\in E}$ get(e.kind, e.id)	nb
traverse	E_A, type	$E_B = \bigcup_{e\in E}$ link(e.kind, e.id, type)	nb
map	E_A, type	$M_{A\times B} = \bigcup_{e\in E} \{(e,l) \| l \in$ link(e.kind, e.id, type)$\}$	nb
Match Operators			
attrMatch	E_A, E_B, attr, thr $MI_{A\times B}$, attr, thr	$M_{A\times B} = \{(a,b) \| a \in E_A \wedge b \in E_B \wedge$ sim(a.attr, b.attr) $\geq$ thr$\}$ $M_{A\times B} = \{(a,b) \| (a,b) \in MI_{A\times B} \wedge$ sim(a.attr, b.attr) $\geq$ thr$\}$	nb nb
match	E_A, E_B, matcher, thr $MI_{A\times B}$, attr, thr	$M_{A\times B} = \{(a,b) \| a \in E_A \wedge b \in E_B \wedge$ matcher$(a,b) \geq$ thr$\}$ $M_{A\times B} = \{(a,b) \| (a,b) \in MI_{A\times B} \wedge$ matcher$(a,b) \geq$ thr$\}$	nb nb
Auxiliary Operators			
union	$E1_A$, $E2_A$ $M1_{A\times B}$, $M2_{A\times B}$ $Q1$, $Q2$	$E_A = E1_A \cup E2_B$ $M_{A\times B} = M1_{A\times B} \cup M2_{A\times B}$ $Q = Q1 \cup Q2$	nb nb / b$^{\text{ext}}$ nb
intersect	$E1_A$, $E2_A$ $M1_{A\times B}$, $M2_{A\times B}$ $Q1$, $Q2$	$E_A = E1_A \cap E2_B$ $M_{A\times B} = M1_{A\times B} \cap M2_{A\times B}$ $Q = Q1 \cap Q2$	nb nb nb
diff	$E1_A$, $E2_A$ $M1_{A\times B}$, $M2_{A\times B}$ $Q1$, $Q2$	$E_A = E1_A - E2_B$ $M_{A\times B} = M1_{A\times B} - M2_{A\times B}$ $Q = Q1 - Q2$	pb ($E2_A$) pb ($M2_{A\times B}$) pb ($Q2$)
compose	$M_{A\times X}$, $M_{X\times B}$	$M_{A\times B} = \{(a,b) \| (a,x) \in M_{A\times X} \wedge (x,b) \in M_{X\times B}\}$	nb / b$^{\text{ext}}$
filter	E_A, $cond$ $M_{A\times B}$, $cond$ Q, $cond$	$E_A = \{a \in E_A \| cond(a)\}$ $M_{A\times B} = \{(a,b) \in M_{A\times B} \| cond(a,b)\}$ $Q = \{q \in Q \| cond(q)\}$	nb nb nb

Table 2. List of all CloudFuice operators incl. signature and definition. Some operators are polymorphic, i.e., they support multiple signatures. The last column indicates operators' the blocking-type, i.e., non-blocking (nb), partially blocking (pb), or blocking (b).