remote sensing

MDPI

Article

A Serverless-Based, On-the-Fly Computing Framework for Remote Sensing Image Collection

Jin Wu [1,2,*], Mingbo Wu [1,2], Haiyan Li [2], Lijuan Li [1,2] and Leilei Li [2,3]

1 State Key Laboratory of Resources and Environmental Information System, Institute of Geographic Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing 100101, China; wumingbo14@mails.ucas.ac.cn (M.W.); lilj.17b@igsnrr.ac.cn (L.L.)
2 University of Chinese Academy of Sciences, Beijing 100049, China; lihaiyan@ucas.ac.cn (H.L.); lileilei17@mails.ucas.ac.cn (L.L.)
3 State Key Laboratory of Desert and Oasis Ecology, Xinjiang Institute of Ecology and Geography, Chinese Academy of Sciences, Urumqi 830011, China
* Correspondence: wuj.17b@igsnrr.ac.cn

Abstract: The rapid growth of remote sensing data calls for the construction of new computational models for algorithmic exploration, which requires on-demand execution, instant response, and multitenancy. We call this model on-the-fly computing, which could reduce the complexity of cloud programming for remote sensing data analysis and benefit from efficient multiplexing. As an advancement of cloud computing, serverless computing makes it possible to realize the on-the-fly computational model. In the study, the concise definition of an on-the-fly computing model for remote sensing data analysis and the corresponding software architecture based on the serverless computing commodities are presented. The proof-of-concept experiments have suggested that the on-the-fly computing model for remote sensing data analysis can be efficiently implemented as a serverless software. The response time is mainly related to the tile reading operation and data structure conversion. In the case of high concurrency, the system can scale to hundreds of instances in seconds.

Keywords: serverless; cloud computing; on-the-fly; remote sensing data; DAG

Citation: Wu, J.; Wu, M.; Li, H.; Li, L.; Li, L. A Serverless-Based, On-the-Fly Computing Framework for Remote Sensing Image Collection. *Remote Sens.* **2022**, *14*, 1728. https://doi.org/10.3390/rs14071728

Academic Editors: Peng Yue, Ingo Simonis and Maged N. Kamel Boulos

Received: 20 February 2022
Accepted: 31 March 2022
Published: 3 April 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the advance in earth observation and surveying technology, remote sensing images are increasingly accumulated and piled to be processed, heading the community of geographical information science into an era of big data. Massive remote sensing images, multisource archives of petabytes, pose great challenges for the traditional geospatial information analysis infrastructure. Cloud computing is one of the most promising technologies to tackle these challenges, and the geographical information science community has developed various cloud computing platforms for massive remote sensing images analysis [1,2].

Among the existing massive remote sensing images analysis frameworks, the most influential one is Google Earth Engine (GEE) [3], which provides two types of computing services, namely on-the-fly computing and batch computing. Based on the traditional big data analysis techniques, batch computing processes all the input data as a whole [4]. As remote sensing data grows larger and larger, the execution time of batch computing is becoming longer and longer. However, exploratory analysis in scientific research usually requires an environment enabling instant read–eval–print loop (REPL), where a program is executed piecewise for a rapid result evaluation. To solve this problem, the on-the-fly computing technologies, featuring on-demand execution, instant response, and multitenancy, are developed to improve the data science productivity. Although GEE provides an excellent instance of the on-the-fly cloud computing paradigm and has been widely used by

the remote sensing community, its theoretical basis, design principles, and implementation details are not publicly available.

General big data analysis frameworks, such as Hadoop [5], Spark [4], and Dask [6], are based on individual servers with tightly integrated resources, which is called server-centric computing. As advancement in the computer architecture community enables the data-center disaggregation [7], serverless computing comes to light [8]. Serverless computing brings about cloud functions with greater elasticity and more lightweight virtualization while changing the pricing of cloud computing from paying resources allocated to paying in proportion to resources used. Unfortunately, to the best of our knowledge, none of the existing on-the-fly remote sensing image analysis frameworks, like Geonotebook [9], have yet adopted serverless computing technologies and could not switch to batch processing seamlessly at the same time.

In summary, this paper makes the following contributions:

(1) Proposing a definition for the on-the-fly cloud computing paradigm for remote sensing image collections, including some empirical or descriptive characteristics and a formal definition.
(2) Designing an entirely serverless architecture based on the serverless commodities of a public cloud, which consists of a data model, a programming model, and a series of key implementing technologies for remote sensing image collection analysis.
(3) Providing some concrete, proof-of-concept experiments suggesting that on-the-fly cloud computing for remote sensing images can effectively run on the serverless cloud platform.

The remainder of this paper is organized as follows: Section 2 is an overview of the definition for the on-the-fly computing paradigm and introduces its serverless software architecture. More details about the implementation are presented in Sections 3–5, and Section 6 shows some concrete proof-of-concept experiments of on-the-fly computing for remote sensing images. Section 7 discusses the results and concludes the paper.

2. On-the-Fly Cloud Computing

2.1. Cloud Computing vs. HPC

Currently, cloud computing has become the main paradigm of server programming, which can ship code to the big data. The key technologies include virtualization, distributed storage, and distributed computation. A large number of frameworks have been developed by the industry and scholars, which can be classified into the server-centric pattern or serverless pattern. There are some serverless computing commodities in the public cloud, such as AWS Lambda.

There are four requirements for any computing system, including ease of use, high performance, portability, and flexibility. The cloud computing system's first object is the ease of use while that of high-performance computing (HPC) is performance. Therefore, HPC provides programming abstractions with low-level details about computer architecture, such as MPI, and cloud computing systems have more automatic optimization mechanisms.

From the perspective of workload, the big data processing frameworks can be classified into batch processing and streaming processing. However, on-the-fly computing has a significant difference from the other two paradigms. The data source for on-the-fly computing is the same as batch processing, but it requires a quick response. The streaming processing can respond instantly, but its data source is real time. Therefore, existing, general cloud computing frameworks cannot be directly applied to the algorithm exploratory analysis. A new paradigm of geocomputation is needed.

2.2. Characteristics

The target computing model is called on-the-fly cloud geocomputation, which is implemented based on general purpose cloud computing technologies, oriented to exploratory analysis, and dedicated to remote sensing processing. We summarize the characteristics of on-the-fly cloud geocomputation from the perspective of human–computer interaction as shown in Figure 1.

(1) Shipping code to the remote sensing images persisted in the cloud storage instead of downloading the data locally for analysis.
(2) Seamlessly switching to batch processing without code modification, which requires the data abstraction and operators to be the same.
(3) Implicitly triggering the execution implied in specific operators, such as visualization and data export.
(4) Dynamically determining the spatial scope of remote sensing images to be processed based on the tiles visualized on the map.
(5) Responding as rapidly as possible when the user needs to evaluate without queueing of workloads.
(6) Executing user-defined codes based on the overviews of remote sensing images without explicitly provisioning and managing data allocation.
(7) Paying in proportion to remote sensing data used instead of paying for the computing resources allocated.

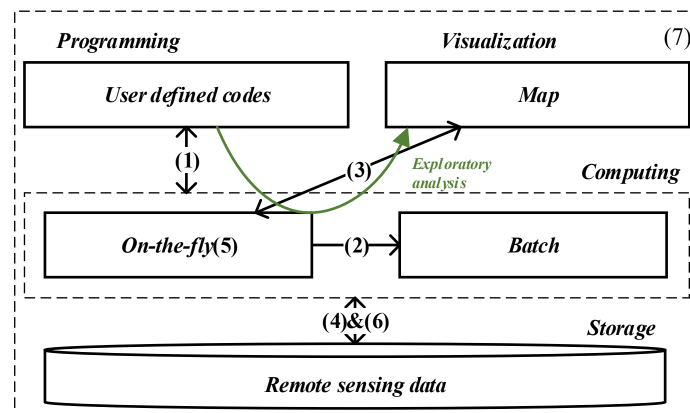

Figure 1. Characteristics of on-the-fly cloud geocomputation.

2.3. Formal Definition

The empirical and descriptive characteristics of on-the-fly cloud geocomputation are not sufficient to determine the structure of the target model. A directed acyclic graph (DAG)-based model of on-the-fly cloud geocomputation is presented in Definition 1. A DAG is a kind of intermediate representation for user-defined codes, and it is common in relational databases, where it is used to represent the query plans. The nodes of a DAG are function invocations with some edges for representing the inputs and outputs. Operators and user-defined codes are equivalent logically and can both be transformed into DAGs in the target framework.

Definition 1. *In order to simplify cloud programming, the target framework of the on-the-fly cloud computing model for remote sensing images should provide rich datatypes, analysis-ready data, and dedicated operators for remote sensing image analysis, which would significantly reduce the amount of user-defined codes. Its programming model could only acquire and generate the datatypes and operators accessible, according to user authentication in the frontend, and process data visible to users on maps, usually in the form of tiles, finally achieving instance response, on-demand execution, and mutitenancy.*

The on-the-fly computing model for the remote sensing data analysis M can be defined as a five-tuple:

$$M :: (E, S, U, V, A)$$

Here notation :: means "defined as", and E refers to the main elements to be processed, including operators, datatypes, and DAGs, which can be defined as a set:

$$E :: D \cup OP \cup G$$

D refers to the set of datatypes, also known as data models, commonly used in remote sensing image processing, such as $Image$ and $ImageCollection$. Each datatype D_i is a set of elements d_i, which means that d_i is a specific dataset of type D_i.

$$D :: \{D_i | i \in N\}$$

$$D_i :: \left\{ d_i^k \middle| k \in N \right\}, i \in N$$

There are some specific relations R between datatypes in D, which can be defined as a three-tuple. A predicate p is virtually a functional mapping from one datatype D_i to another datatype or operator. Superscript $*$ means receiving the power set.

$$r :: (\alpha, p,\ \beta) \Leftrightarrow p(\alpha) \rightarrow \beta; \alpha \in D \cup OP, p \in P, \beta \in D$$

Here notation ; means the end of an expression. There are only two predicates. The predicate cp means that α is a component of β, and more complex datatypes can be established through it. The predicate ih means that β is more specific and customed on the basis of α. We can build a classification based on predicate ih. It should be pointed out that all the datatypes and operators can be organized as a network through these two predicates.

$$cp(\alpha) \rightarrow \beta \Rightarrow \alpha \in \beta; \beta \in D, \alpha \in D \cup OP$$

$$ih(\alpha) \rightarrow \beta \Rightarrow \beta \in \alpha^*; \alpha, \beta \in D$$

OP refers to a set of operators dedicated to remote sensing image analysis, which is also known as the programming model. Each operator op is bounded with a specific datatype D_i through the predicate cp. Every operator is actually a function mapping from certain datatypes with or without a base operator to the output datatypes.

$$OP :: \{op_i | i \in N\}$$

$$op :: D^* \times OP \rightarrow D^*$$

G refers to a DAG, which represents the computational process of remote sensing images. The DAG is constructed by a series of computational nodes n_j, which represent functional calls with the output results α, operator name, and input arguments. Similar to an operator, G is virtually a function mapping from the input remote sensing data to the results. In the implementation of the framework, G can be modeled as a series of nested objects. Each of them records the input arguments, operator name, and returned datatype.

$$G :: \{n_j | j \in N\}$$

$$n :: (\alpha, op, \beta), \alpha \in D^*, \beta \in G^* | D^*$$

S refers to a set of states of the main elements, including datatypes, operators, and the DAG. Any datatype, operator, or DAG can be located only in the client or on the server, and it can be a static string, a callable proxy, a piece of code, or an executable cloud function. Notations st, cb, cd, and ex represent that the element is static, callable, code style,

or executable, respectively. Notations ct and sv mean that the element is located in the client or server, respectively.

$$S :: \{st, cb, cd, ex\} \times \{ct, sv\}$$

The notation U refers to end users. Each user has permitted access to certain operators. V refers to the viewpoint on a map, which can be defined as a set of tile numbers. The viewpoint determines the spatial scope of input remote sensing data to be processed.

$$U :: \{u_i | i \in N\}, u_i \in OP^*$$
$$V :: \{(x, y, z) | x, y, z \in N\}$$

Notation A refers to a set of actions to change the state of target elements to complete the whole computational process.

$$A :: \{get, init, gnrt, sbmt, schdl\}$$
$$a :: U \times S \rightarrow U \times S, a \in A$$

The action get can change the location of some elements that can be accessed by a certain user u_i. Notation: means value of the state S.

$$get :: u_i \times S : (st, sv) \rightarrow u_i \times S : (st, ct), u_i \in U$$

The action $init$ represents an action for the initialization of datatypes and operators, which translates the state of them from static to callable. A datatype or an operator that is callable means that it can be programmed but will not be actually executed.

$$init :: u_i \times S : (st, ct) \rightarrow u_i \times S : (cb, ct), u_i \in U$$

Notation $gnrt$ represents an action for DAG generation, which translates the user-defined script to a DAG object. The state of the DAG changes from code style to callable.

$$gnrt :: G \times S : (cd, ct) \rightarrow G \times S : (cb, ct)$$

In contrast to action get, $sbmt$ represents the action of the DAG submission, which can be modeled as translating the DAG to a static string and changing the location of the DAG from client to server.

$$sbmt :: G \times S : (cb, ct) \rightarrow G \times S : (st, sv)$$

The action $schdl$ represents the action of DAG scheduling, which changes the state of the DAG from static to executable and obtains the result tiles determined by viewpoint. The execution or scheduling of DAG depends on a run-time environment, which can be modeled by a process calculation [10]. Serverless has no formal foundation yet, and to simplify the definition, we do not model the execution details of the DAG in the backend.

$$schdl :: G \times S : (st, sv) \times V \rightarrow \{tile_i | i \in V\}$$

It should be noted that the essence of the element state change is a process of translation rather than a process of encapsulation and invocation.

2.4. Serverless Architecture

In this study, a pure serverless software architecture means that all the components are built on serverless commodities from the public cloud providers, mainly including the function computing (FC) [11], serverless workflow [12], Tablestore [13], message service (MNS) [14], relation database system (RDS) [15], and object storage service (OSS) [16] of

Alibaba Cloud. This architecture is shown in Figure 2, which introduces the high-level components in the target system and traces the execution flow of the UDF creation and pipeline execution. Due to the adoption of serverless technologies in software design, the cost of the system can be paid after the construction is completed. This will enable flexibility in the pricing of system services.

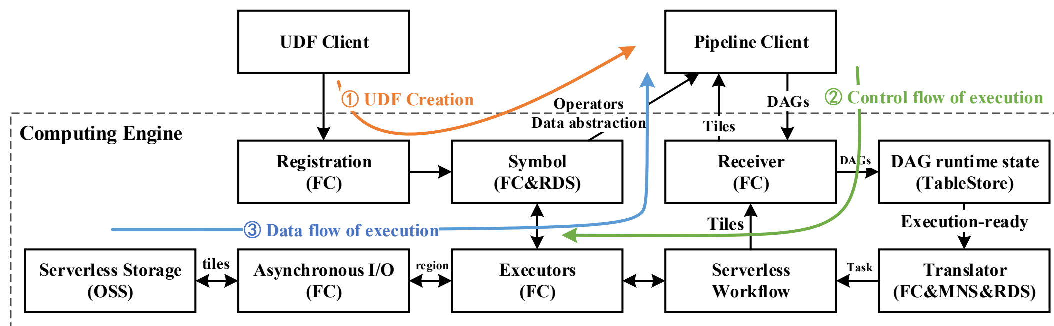

Figure 2. Serverless-based architecture.

① User-defined function (UDF) creation. Some basic operators are defined and submitted to a cloud function through the UDF client. This cloud function registers the UDF in the FC engine and a symbol database. As operators are dynamically generated, users can access the created UDFs through the pipeline client.

② Control flow of execution. User-defined code based on the pipeline client are expressions that can be translated to DAGs. These DAGs are submitted to the receiver based on a cloud function and then persisted in the DAG run-time storage. Another cloud function is triggered by the run-time storage update events to translate the execution-ready nodes to a parallel workflow.

③ Data flow of execution. Remote sensing images in the serverless storage OSS can be asynchronously accessed and processed by executors. Tiles of the results are returned to the receiver and visualized on the map in the frontend.

3. Data Model

3.1. Tiling

Because on-the-fly geocomputation emphasizes analyzing during visualization, the target system needs to constantly load only part of the target remote sensing images to memory, which means that it is necessary to divide the images into tiles at different levels and organize them as pyramids.

For the remote sensing images, Cloud Optimized GeoTIFF (COG) [17] is the most popular file format to build a pyramid for on-the-fly cloud geocomputation. Because the I/O time is supposed to be much less than the time of connecting to OSS, all the tiles of different levels that belong to the same band are organized together in a single GeoTIFF file. The tiles of the COG are usually organized in a sequence of a row major.

There are two kinds of tiles in the target system: one is for visualization and the other is introduced by the COG format for cloud storage. Notation t_v and t_s represent a tile for visualization or storage, respectively. The tile number (x_v, y_v, z_v) is usually determined according to Web Mercator projection.

$$t_v :: (x_v, y_v, z_v, value_v)$$

$$t_s :: (x_s, y_s, z_s, value_s)$$

In this paper, we highly recommend that these two tiling strategies are consistent and aligned to some extent. Images from different satellites may have different spatial projections and need conversions from (x_s, y_s, l_s) to (x_v, y_v, l_v); (x_s, y_s, l_s) is usually tiling locally along the rows and the columns of image and not aligned to (x_v, y_v, l_v).

3.2. Logical Region

Tile is the basic unit for storage and visualization. The region is a logical strategy of tiles grouping, which is aimed at the dynamic requirements of data-parallel execution. It can be modeled as a set of tile numbers, representing a spatially continuous coverage, and is the minimal unit for the algorithm design, task allocation, and geodata access.

The calculation from a region number (x_r, y_r, z_r) to visualization tile numbers $\{(x_v, y_v, z_v)\}$ can be modeled as an affine function f_t. a_x and a_y represent the width and height of the region, respectively. b_{rx} is an offset in region r along x while b_{ry} is an offset in region r along y. z_v and z_r are levels in the pyramid, and they usually have the same value. Obviously, the conversion f_t^{-1} from (x_v, y_v, z_v) to (x_r, y_r, z_r) is a kind of integer modular operation.

$$\begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix} \overset{f_t}{\leftrightarrow} \begin{pmatrix} x_r \\ y_r \\ z_r \end{pmatrix} \times (a_x, a_y, 1) \mp \begin{pmatrix} b_{rx} \\ b_{ry} \\ 0 \end{pmatrix}, a_x, a_y, b_{rx}, b_{ry} \in N$$

The region is composed of two independent types of tiles, which can be transformed to each other by f_c. Due to the transformation of the projection, a tile in the COG may belong to two different regions. In order to prevent the tiles belonging to different regions from being processed multiple times, a tile-masking strategy was designed. The mask is a set of tiles, which can indicate whether the target tiles have been processed. In computation, the region is reformed as a unified larger tile that can ensure the correctness of focal operators by the overlapped zones, such as sliding windows. The logical region LR can be defined formally as follows:

$$LR :: (T_v, T_s, f_t, f_c, M)$$

T_v and T_s are the set of t_v and t_s, respectively. M is also a cached set of t_s to indicate the tiles that have been processed. The logical region is as shown in Figure 3. In the OSS storage, the bytes to be read are determined by some information, including bucket, image collection, image, band, region number, tile order, and overlap. The *region* and *overlap* define the minimal basic unit for distributed geocomputation. The *overlap* is a dynamic value determined by the window operators. The spatial *order*, similar to the Hilbert curve or Z curve, determines the offset and length of bytes where the system begins to read.

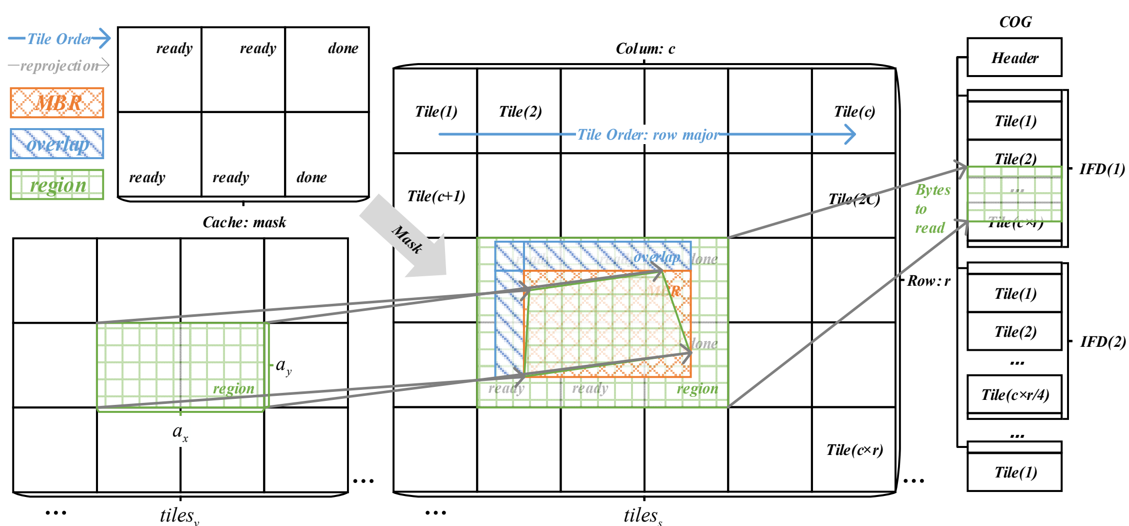

Figure 3. Logical region.

3.3. Datatypes

As a kind of raster data, the remote sensing image is the main type of geodata and can be analyzed by a map algebra or an array algebra system. In addition, vector data is another

type of geodata, often used to manage the image processing results. The logical region is a general purpose geodata abstraction for implicit parallel computing, which is not suitable for end-user programming and should be invisible to users. We propose a composite datatype of image collection based on the SpatioTemporal Asset Catalog (STAC) [18] and GeoJSON. The definition is as follows:

Image collection is the top-level data abstraction dedicated to remote sensing image processing and constructed based on predicates P and some basic datatypes. All datatypes and their relations are shown in Figure 4.

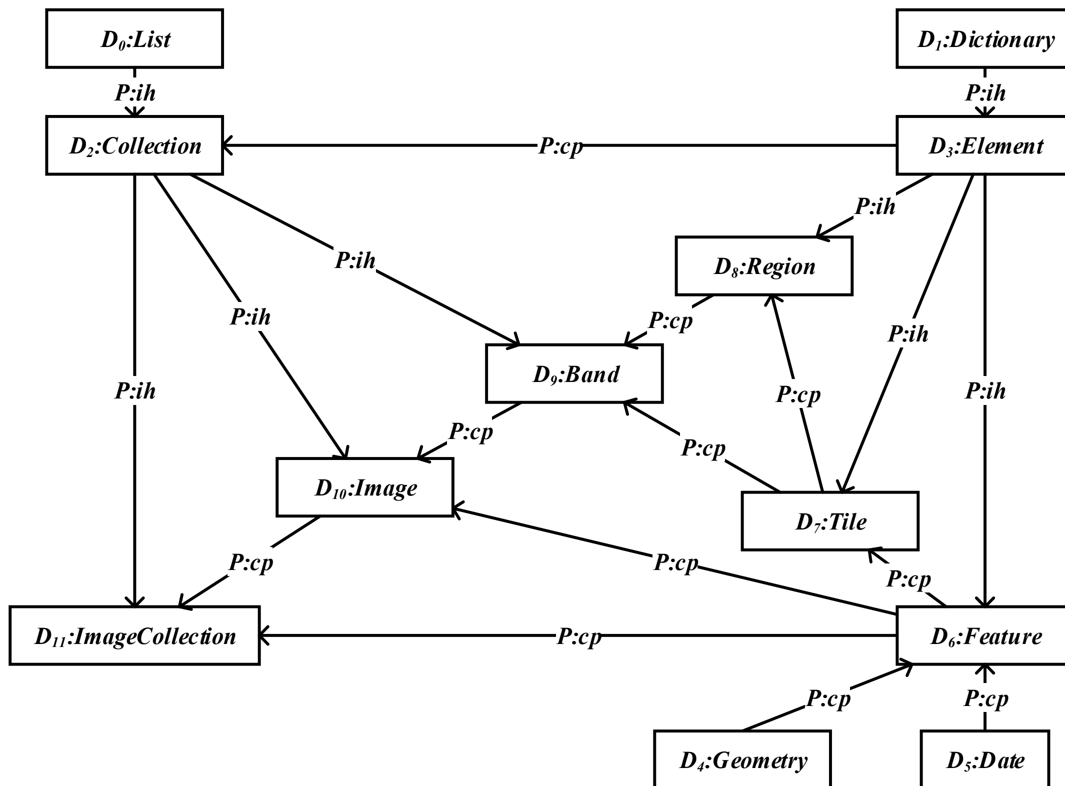

Figure 4. Definition of image collection.

4. Programming Model

4.1. Workflow

The DAG is virtually a workflow for modeling the computational process, where the nodes and edges represent operators and image collections. DAG generation, presented in Section 4.4, is usually separated from the end-user programming interface and invisible to users, which makes programmers avoid the burden of constructing a global DAG data structure for workflow. This paper proposes a design or definition for the interface of remote sensing processing workflow. The user programming interface consists of composite datatypes and workflow skeletons, which can be expressed as a two-tuple, $(composite\ types, skeletons)$.

(1) $composite\ type$. The composite datatypes are the user-visible components of data abstraction, defined in Section 3.3, except *region* and *tile*. In the construction of workflow, *ImageCollection* is the most common datatype.

(2) $skeleton$. Workflow skeletons are high-level operators representing the basic workflow semantics. There are six operators related to workflow construction, including $create$, $filt$, $integrate$, $transform$, $aggregrate$, and $show$. These skeletons are functions mapping from one image collection to another.

$$Skeleton :: D_{11} \times OP \rightarrow D_{11}$$

Model, *Condition*, *Relation*, *Base operator*, *Aggregrator*, *Reducer*, and *Scheme* are kinds of user-defined functions, defined in Section 4.2. The definition is shown in Figure 5.

Every *skeleton* has two parts, the lefthand expression $\langle lhs \rangle$ and the righthand expression $\langle rhs \rangle$. All the input datatypes of *skeleton* are *ImageCollection*. The *create* operator is used to load an image collection with a file path of cloud storage or a data model *Model* describing the content of the certain image collection. The *Filt* operator is used to construct spatiotemporal or regular conditions for filtering the input image collection. Because each image collection has a corresponding data model *Model*, the *Integrate* operator provides a way to integrate different image collections with a *Relation* between different data models. The *skeleton Transform* maps a base intraimage operator to every item of the input image collection while the *Aggregate* operator maps a base interimage operator according to the *Aggregater*, which also groups the items of the image collection by the selected dimensions in *Model*, such as time or space. The *Show* operator triggers the execution of the workflow and obtains the tiles of the input image collection visible to users.

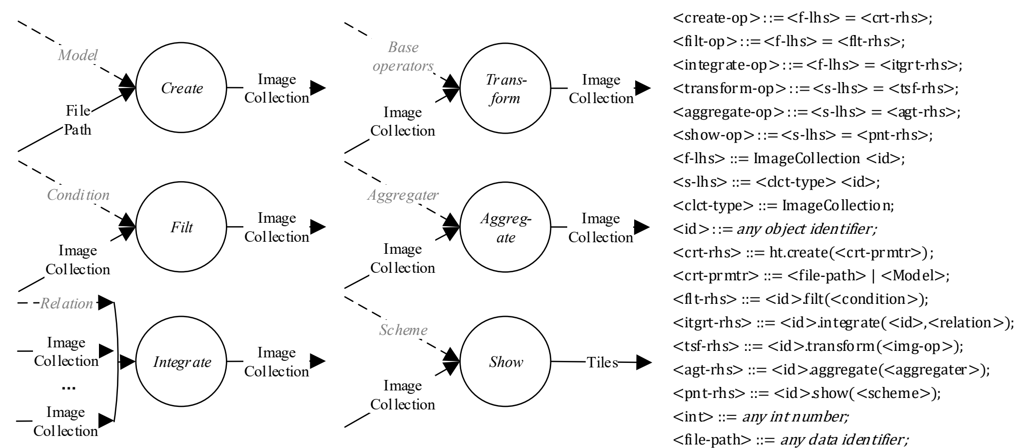

Figure 5. Workflow skeletons.

4.2. User-Defined Function

The *skeletons* are collection-level operators and lack the flexibility of image-level or pixel-level operations. According to the definition of *workflow*, there are six operators at the image level and pixel level, which can be defined and published by the user, including *Model*, *Condition*, *Relation*, *Base operator*, *Aggregrater*, and *Scheme*.

Similar to the constructor of programming language, *Model* is a configuration for certain satellite images, including band number, coverage, resolution, spatial projection, and other attributes in a key-value form. The *Condition* is a logical expression, especially the spatial-temporal topological relationship, for selecting the target images. The *Realtion* is a rename operator, which specifies a unified model for the input image collections.

Base operator refers to the image-level algorithm defined and implemented on the band-level and pixel-level interface. The band-level operators, band math, or map algebra can be regarded as some window operators on linear algebra. The *reducer* defines some algorithms to integrate all the images in a collection into a single image along a certain axis, such as *space*, *time*, *bands*, or other metadata.

Besides all the above operators, there are also some operators related to publishing UDFs, such as *register*, which provide an interface for registering the basic information of UDFs, including results datatype, function name, input arguments, and description, etc. More details about operator publication are shown in Section 4.3.

The syntax description is based on the augmented Backus–Naur form (ABNF), shown in Figure 6.

```
<band> ::= {<<band-name>, <band-value>>, <band-meta>};
<band-name> ::= an identifier for the index of band;
<band-value> ::= a 2 dimensional numeric array [[..]...[..]];
<band-meta> ::= {<'projection', <projection>>, <meta-data>*};
<meta-data> ::= <key, value>;//all metadata is organized as pairs;
<projection> ::= an identifier of projection defined according EPSG;

<band-op> ::= <band-lhs> = <band-rhs>;
<band-lhs> ::= Band <id>;
<band-rhs> ::= ht.band() | ht.load(<band-name>)
                  <id>.addMeta(<band-meta> | {<meta-data>*}) |
                  <id>.setValue(<band-name>,<band-value>) |
                  <id>.deleteMetaByName(<key>) |
                  <id>.deleteArray(<band-name>);

<img> ::= {<<band-name>, <band>>+; <img-meta>};
<img-op> ::= <img-lhs> = <img-rhs>;
<img-lhs> ::= Image <id>;
<img-rhs> ::= ht.image() | ht.load(<img-name>) |
                 <id>.addBands(<band>*) |
                 <id>.deleteBands(<band-name>*) |
                 <id>.updateBands(<band>*) |
                 <id>.selectBand(<band-name>*);

<imgclct> ::= {<<img-name>, <img>>+; <metadata>};
<imgclct-op> ::= <imgclct-lhs> = <imgclct-rhs>;
<imgclct-lhs> ::= ImageCollection <id>;
<imgclct-rhs> ::= ht.imageCollection() | ht.load(<imgclct-name>) |
                    <id>.imgClctAdd(<img>*) |
                    <id>.imgClctDelByName(<img-name>*) |
                    <id>.imgClctUpdate(<img>*);
                    <id>.selectImg(<img-name>*);
<Model-udf> ::= ImageCollection <id>(<img-lhs>*){
                    <img-op>+ | <band-op>*};

//defined according the GeoJSON specification
<geometry>::=<Point>|<LineString>|<LinearRing>|<Polygon>|
                <MultiPoint>|<MultiLineString>|<MultiPolygon>|
                <GeometryCollection>|<Feature>|<FeatureCollection>;
<spatial-rel> := Adjacent | Connection | Conjunction | Inclusion;

//Defined according the time ontology in OWL;
<time> ::= <time-point> | <time-interval>;
<time-rel> := Before | After | Meets | MetBy | Overlaps | Starts|
                OverlappedBy | StartedBy | During | Contains |
                Finishes | FinishedBy | Equals | In | Disjoint;
```

```
<str-format> is an regular expression;
<c-op> := EQ | NE | LT | GT | LE | GE; //=, ≠, <, >, ≤, ≥
<l-op> := and | or;

<condition-op> := <cond-lhs> = <cond-rhs>;
<cond-lhs> := Condition <id>;
<cond-rhs> := ht.condition() |
                <id>.addCond(<id>,<l-op>) |
                <id>.addGeometry(<gemetry>,<spatial-rel>,<l-op>) |
                <id>.addDatetime(<time>,<time-rel>,<l-op>) |
                <id>.addMetaStr(<key>,<str-format>,<l-op>) |
                <id>.addMetaNum(<key>,<c-op>,<value>,<l-op>);

<condition-udf> ::= Condition <id>(<cond-lhs>*){
                      <condition-op>+};

//add a common scheme to the candidate image collections
<relation-udf> ::= ImageCollection <id>(<destination>, <source>+){
                     <Model-op>+}

<destination>, <source> ::=<imgclct-lhs>;//both are image collections

<scheme-op> ::= <imgclct-lhs> = <schm-rhs>;
<schm-rhs> ::= <source>.<metadata>.as(<destination>.<metadata>);

<img-op-udf>::=Image <id>(<img-lhs>,<win-lhs>){
                 <img-op>+};

<window-op> ::= <win-lhs> = <win-rhs>;
<win-lhs> ::= Window <id>;
<win-rhs> ::= <id>.create(<<left>, <right>>+|<geometry>+|
                            <time-interval>+);
<left>, <right> ::= an positive integer indicating relative coordinate;
<pixel-op> ::= <u-op> | <b-op> | <c-op>;

<u-op> ::= Neg | Not | Log | Ceilling | Floor | Log2;
<b-op> ::= Add | Sub | Mul | Div | <l-op>;
<win-op-udf> ::= Window <id>(<win-lhs>){
                   <pixel-op>+};

//for single band or multi-bands
<img-op> ::= <img-lhs> = <id>.focal(<win-op-udf>, <win-lhs>);
                          <id>.apply(<img-lhs>, <b-op>);

<aggregater-udf> ::= ImageCollection <id>(<imgclct-lhs>, <win-lhs>){
                       <imgclct-op>+ | <img-op>+ | <pixel-op>+};
```

Figure 6. Syntax description.

For the reason of embedding into other programming languages as an internal domain-specific language (DSL) [19], the production rules adopt some symbols different from the regular ABNF [20]. All the symbols of the production rules in Figures 4 and 5 are defined in Table 1.

Table 1. Description of the symbols.

Symbols	Description	Example
<>	Denotes an operator or variable in programming;	<img-clct>
[]	Indicates creating a data structure of numeric array;	[[. . .] . . . [. . .]]
{}	Body of the UDF or a data structure of dictionary;	{<metadata>*}
()	Indicates the inputs of the UDF;	(<img-lhs>)
|	Choice operator for two candidate expressions;	EQ | NE
*	Zero or more occurrences of the preceding element;	<img-op>*
+	One or more occurrences of the preceding element;	<img-op>+
.	Denotes the attribute or method of an object;	<id>.apply(. . .)
;	Indicates end of a BNF statement;	Image <id>;
//	Annotation of the production rules;	//annotation
::=	Means being defined as the right-hand expressions;	<l-op> ::= or;

4.3. Operator Publication

Similar to the symbol table of programming language, operators can be regarded as a kind of computational symbol with input and output information in the remote sensing processing workflows. An operator brings two kinds of information, one for user programming and the other for explicitly cloud functions calling, which can be called high-level attribute and low-level attribute of operators.

The high-level attribute is some information exposed to users, which would be sent to the pipeline client through a dynamic operator generation. It contains four kinds of

information expressed as datatype, operator name, input arguments, return types, and description. The low-level attribute is a pointer to a certain cloud function, which contains two kinds of information, including operator name and function call location.

When users create operators, they need to publish in the backend before these operators can be used to create workflows. The process of operator publication is shown in Figure 7, presented in Python style. When UDFs have been published by users, they can access the published UDFs through agent datatypes through the pipeline client, and they can be used in the construction of workflows.

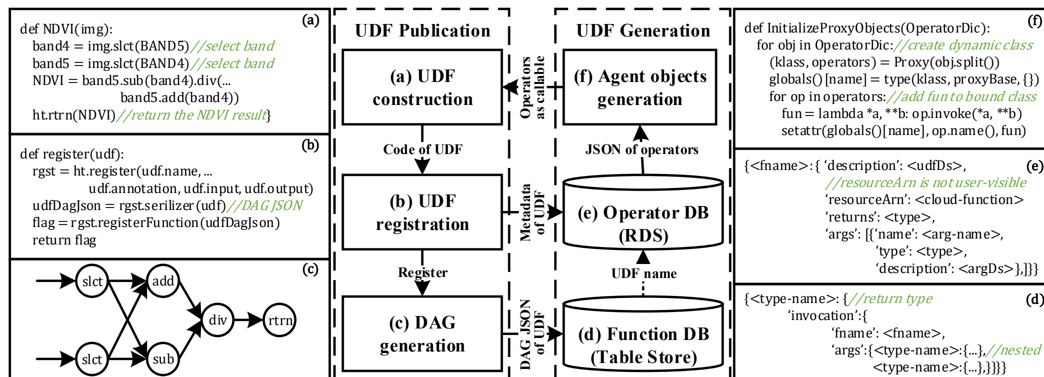

Figure 7. Operator publication and generation.

4.4. DAG Generation

The pipeline and UDF clients have the ability of dynamically generating operators in the frontend based on the metaprogramming technology and the high-level attributes of cloud functions. All the operators in the frontend belong to specific proxy objects, and they actually refer to one same function just recording the invocation between cloud functions, which make up a sense of code execution. This relation can be modeled as a DAG, which is a kind of intermediate representation between the remote sensing image processing pipeline and the underlying cloud functions, and generated through a series of callable proxy objects in the frontend, similar to the way of the GEE API Client.

4.5. Trigger of Execution

Although the DAGs represent the pipeline execution, they are generated in the frontend. When users invoke specific functions, the framework is triggered to submit DAGs from client to cloud. In order to receive results as soon as possible, the DAGs should have an initial viewpoint or would receive a default one if not. The framework can compute results on the tiles around the viewpoint. The trigger of execution can be defined as a tuple of $(function, viewpoint)$. The $function$ can submit the DAGs through a request and obtain a map identifier to fetch tiles. The $viewpoint$ defines the initial scope of input tiles. Therefore, the users can receive the result very quickly, complying with the design principle of on-the-fly geocomputation.

The most important aspect of DAG execution is to guarantee the correctness of translating the focal operators from band level to region level. For the reason that logical region is invisible to end users, the translator of the run-time environment should be able to determine the shape of logical region automatically, i.e., automatic data partition.

5. DAG Execution

5.1. Data Partition

Different from the DAGs generated automatically by the machine learning engine, the DAGs manually constructed by users are usually not too big to be partitioned for task-parallel execution. This paper only focuses on the algorithms of data-parallel execution. We need an algorithm for automatic data partition.

The focal operator has a characteristic of a structural locality, which means that the result at the location of (i, j) is not only determined by the value of (i, j) but also its neighbors. The structural locality of the remote sensing processing can be defined as a window so that the computational model of the data partition for a remote sensing image can be expressed as a function with a window as the input and a kind of logical region as the output.

$$region \leftarrow f_{image}(window)$$

Since all the remote sensing images are organized as a COG and persisted in the OSS, the data partition algorithm is ideally determined by the characteristics of OSS, the shape of the window, and the physical layout of the COG. As the COG is a highly customized format in the OSS for streaming, progressive rendering, and supporting on-the-fly random reading, the impacts of the OSS and the physical layout of the COG can be neglected in this paper.

A single remote sensing image can be regarded as a three-dimensional array, so the window defines the scope and neighboring pixels at a specified location of (i, j, k). Different from the common window definition, such as $7 \times 7 \times 3$, this paper adopts a more flexible window form, i.e., a set of neighboring pixels similar to ArrayUDF [21]. Expression $\{r, c, b\}$ represents the three dimensions of an image. The shape of the overlapping zone $\{O_k\}$ guarantees the correctness of operators across logic regions and can be derived from the window parameters, $[L_i, R_i]$.

$$W([L_i, R_i]) \leftarrow f\left(\{P_{\delta_r,\delta_c,\delta_b} | \delta_i \in [L_i, R_i]\}\right), \forall i \in \{r, c, b\}$$
$$\{O_k\} \leftarrow Sum(L_k, R_k), \forall k \in \{r, c, b\}$$

5.2. Execution

To implement computing while visualizing, we propose an architecture based on the producer–consumer pattern. The producer receives tile numbers indicating data to be processed, applies the DAG to be executed on tiles determined by regions, and puts the results in a workspace to be consumed. The data-parallel DAG execution algorithm is shown in Algorithm 1.

Algorithm 1 Data-Parallel DAG Execution

```
Input: Tile numbers: {t_v}; Task ID: DAG_id; Serverless Engine: Executor
Output: Result of DAG execution on specific images: {t_s}
 1: function CONSUMER({t_v}, DAG_id, Executor)
 2:     {R_v} ← TILETOREGION({t_v})                     ▷ Converts tile numbers to region numbers
 3:     {Img} ← OPENOSSIMGS(DAG_id)                     ▷ Receives the handler of {Img} to be processed
 4:     for each R_v ∈ CACHE.GETREGIONS(DAG_id) do
 5:         {t_s} ← CACHE.GETTILES({R_v}, DAG_id)       ▷ Receives the processed tiles of {Img}
 6:         {R_v} ← {R_v} − R_v                         ▷ Deletes the region that has been processed
 7:         PRODUCER.PUTS({t_s})                        ▷ Puts the tiles to workspace to be consumed
 8:     end for
 9:     Overlap ← GETWINDOW(DAG_id).CALOVERLAP()        ▷ Receives the overlap zone
10:     {MBR} ← {Img}.REGIONTOMBR({R_v})  ▷ Receives MBRs of {R_v} according {Img} projects
11:     for each MBR ∈ {MBR} do
12:         {t_s} ← {Img}.RELATTILES(MBR)               ▷ Receives tiles related to MBR of all {Img}
13:         {t_s} ← CACHE.GETMASK({t_s})                ▷ Sets mask for tiles have been processed
14:         {t_s} ← Executor.Apply(DAG_id, {t_s}, Overlap) ▷ Generates Cloud Functions for execution
15:         PRODUCER.PUTS({t_s})                        ▷ Puts the tiles to workspace to be consumed
16:     end for
17: end function
```

5.3. Cache

Considering that some tiles may be requested for more than one time in the DAG execution, the framework must maintain the execution states, including DAGs and tiles, in caches. The consumer firstly obtains tiles in the target region from the cache in the frontend and generates tiles dynamically from the upper or lower tiles maintained in the cache if they have been requested and then sends a request to the backend to receive the target tiles generated by the producers from the workspace.

Except for the cache in the frontend, there is also a cache in the backend, which plays an import role in the DAG execution. If the whole region requested by the consumers has been processed, the cache will put the tiles directly in the workspace. As some tiles are shared by different regions and the target region is partially processed, the cache will generate a mask to declare which tiles have been processed to reduce the computational cost.

6. Case Study

6.1. Data and Result

This study conceptually validates the feasibility of serverless-based, on-the-fly computation framework with a simple NDVI use case on a remote sensing image. The NDVI code is shown in Figure 8. All the remote sensing images are from Landsat8 and organized in the COG file format. Every overview of different levels in the image pyramid is tiled into tiles of 256 × 256 pixels and encoded into an individual TIFF file. This study provides a Python client integrated as an algorithm library in Jupyter. The NDVI code, the input image, and the result tiles are shown in Figure 8. The NDVI function is performed on four tiles, numbered as $(0,7)$, $(0,8)$, $(1,7)$, and $(1,8)$.

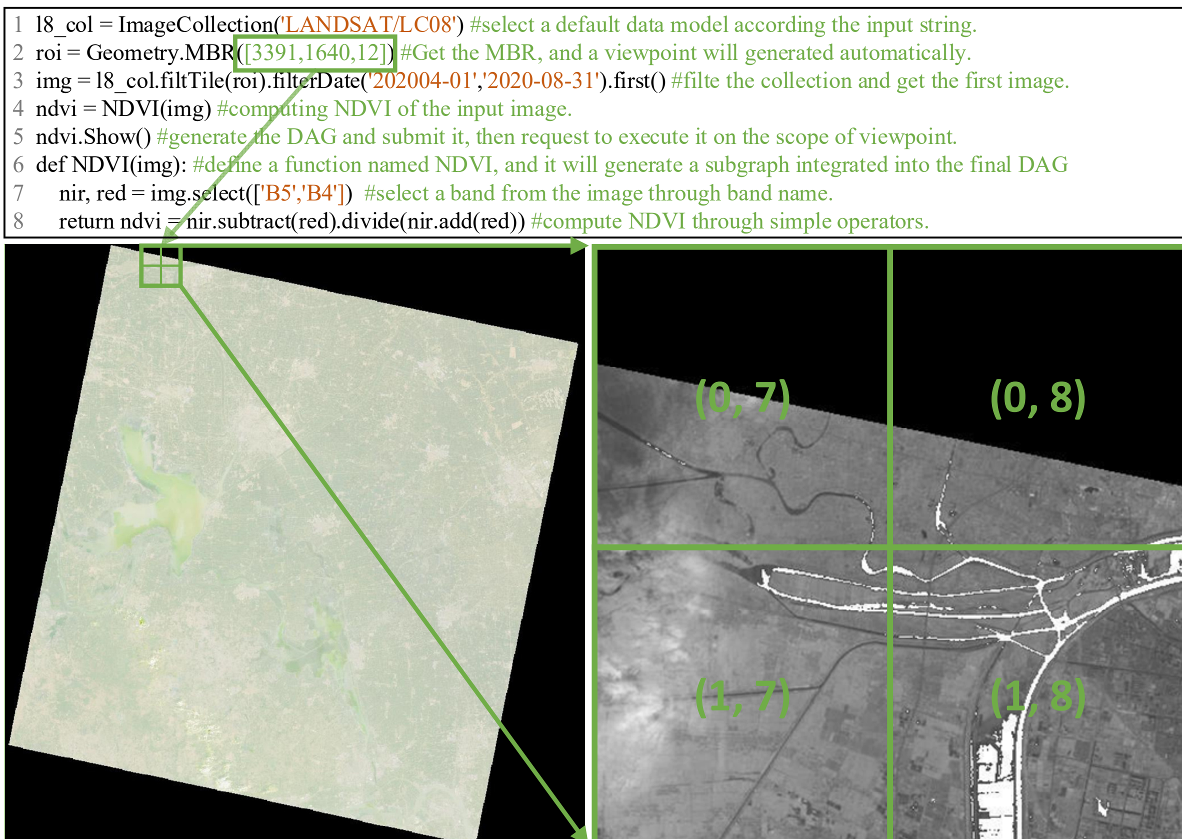

Figure 8. The Landsat8 data and the result tiles.

Before the system begins to process the target tiles, a DAG will be generated and submitted to the data center. Part of the DAG in Figure 9 is corresponding to the NDVI code from line 1 to line 3. It should be pointed out that the time point is modeled as a range from a start time to the end time. A DAG is actually a nested object returned by the final operator. The DAG generation is a process of creating this nested object.

Figure 9. Part of the DAG.

6.2. Response Time

Once the algorithm is determined, it will be applied to the whole remote sensing collection. The response time is a critical characteristic of on-the-fly cloud computing for remote sensing data analysis during the phase of DAG construction or algorithm exploration. When users request to run the user-defined code, the frontend firstly submits a DAG and receives a task identifier, and then the backend executes it on the specified tiles determined by VP.

The response time is related to the efficiency of the scheduler, the complexity of the related algorithms, the latency of the communication, the characteristics of the serverless platform, and the concurrency. In the study, the response time is tested through a series of continuous requests shown in Figure 10.

The initialization phase of a serverless platform is called a cold start. The cold start of the NDVI case is around 1200 ms, which is about three times longer than that consumed by a base cloud function, which only responds to the request without doing anything. Once the serverless platform has been started, the response time will reduce to around 700 ms in the elastic mode whereas in the performance mode, the computation time will reduce to less than 400 ms. It is shown that the NDVI case benefits a lot from better hardware, but anything larger than a 4 GB memory could not further reduce the response time.

The response time of the NDVI case is mainly determined by reading tiles from the OSS, putting them together, and transforming them into an image. The time of reading a COG header and four tiles from the OSS is about 200 ms, and the NDVI case needs to repeat this process. Then, the system will put four tiles together and transform them into an image, and it will take about another 200 ms. The scheduler only takes less than 100 ms, for the total response time is less than 700 ms. To reduce the response time, the system adopts a strategy similar to lazy evaluation, in which the operation of reading data from the OSS is only triggered by some specific operators that must manipulate the data. In the

NDVI case, the operation of reading tiles from the OSS is deferred to the eleventh round of task scheduling.

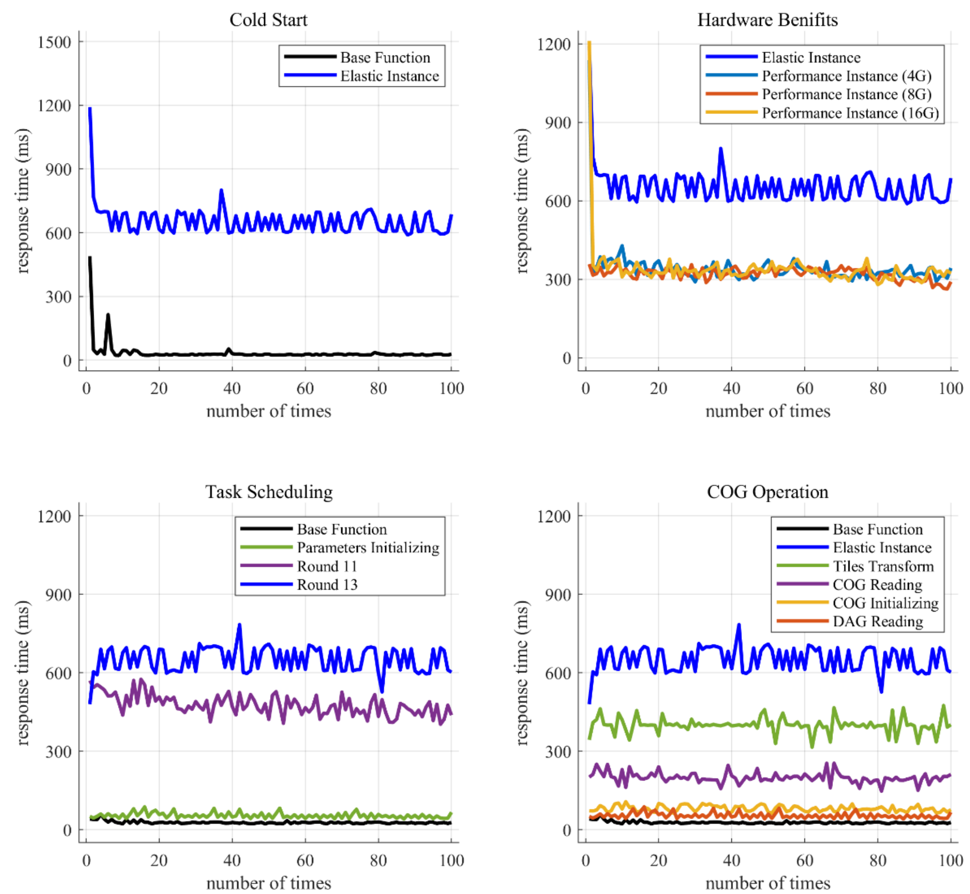

Figure 10. Response time of a series of the continuous request.

6.3. Concurrency

An important characteristic of serverless computing and the requirement for on-the-fly cloud geocomputation is concurrency. Once the computation request is sent by a map client automatically, the serverless framework needs to respond to a large number of computation requests in a short period. The tested concurrency performance of the serverless-based remote sensing image analysis framework is shown in Figure 11.

The framework in the elasticity mode is tested through a series of asynchronous requests at the scale of tens and hundreds. It is shown that the response time stays below 1.5 s until the concurrency approaches about 700, and then the response time starts to increase linearly. When the asynchronous request is under 200, the maximum execution time of the DAG is less than about 1.5 s. The average response time is still under about 1.5 s though the scale of asynchronous requests reaches 1000 and the maximum execution time is close to 3.5 s. In contrast with the traditional technologies of cloud computing, which may scale in minutes or hours, the serverless-based framework can increase the number of functions and instances in seconds to handle new requests.

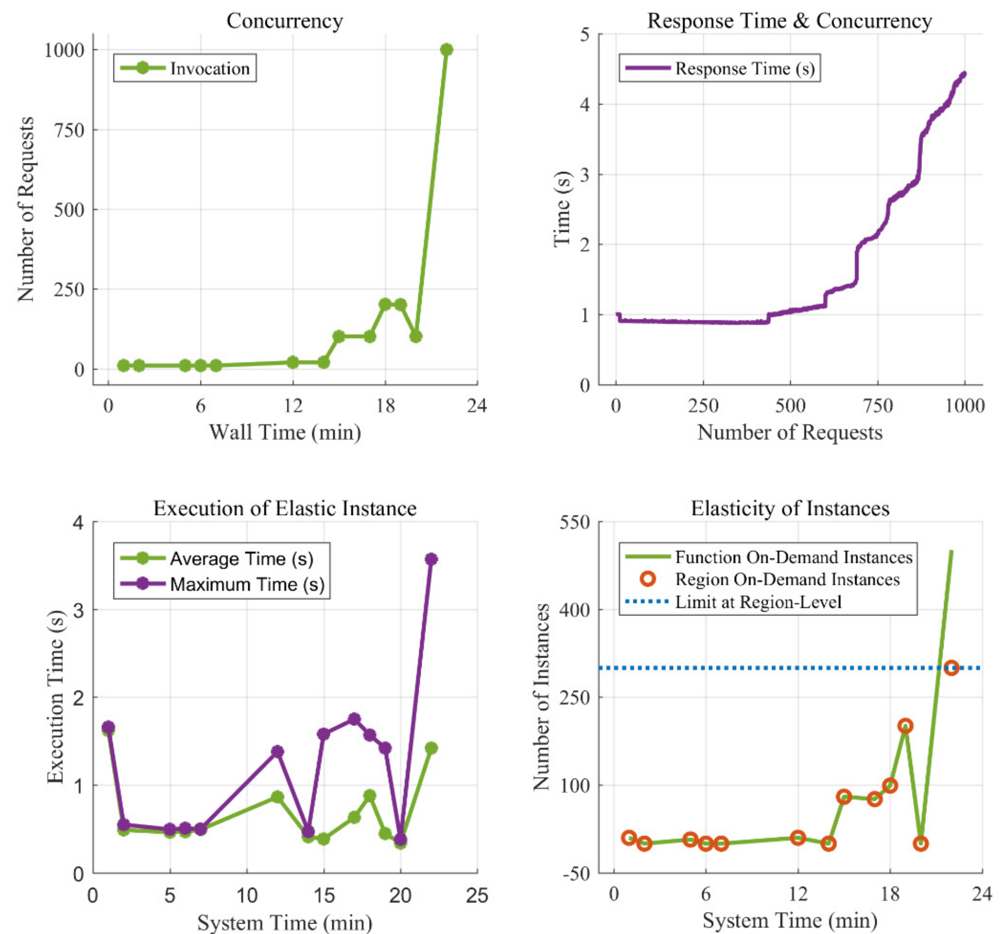

Figure 11. Elasticity and concurrency.

7. Discussion and Conclusions

There are some remote sensing frameworks developed from traditional parallel databases and cloud computing technologies. The paper [1] analyzed the related works of three types, including spatial databases, programming and software tools, and big spatial data infrastructures, and categorized them further into ten types from underlying general technologies. Though extensive in the scope of investigation, it only focuses on the underlying technologies without evaluating them in terms of computing service. Different from this review, here, we analyze the representative remote sensing image analysis frameworks from both computing service types and underlying cloud computing technologies.

As one of the most popular remote sensing analysis cloud platforms, GEE provides two types of computing services, namely on-the-fly computing and batch computing. On-the-fly computing is used for rapid prototyping for tiled remote sensing images while batch computing provides the capability of planetary-scale processing. Despite their different focuses, they both have the same programming interface and can switch to each other seamlessly based on GEE's ability to translate the user-defined codes or functions to DAGs. However, GEE cannot express focal operation with structure locality at a pixel-level. In the study, the ability of the focal operation is implemented through window operators based on relative coordinates and overlapped logical regions.

Different from GEE's local tiling strategy, the Open Data Cube (ODC) [22] reformats the raw, remote sensing images into analysis-ready data with a global tiling strategy. ODC's rapid prototyping and parallel computing capabilities are provided by the xarray [23] and Celery framework, respectively. Although Celery has high performance, from the perspective of ease of use, ODC only has a low-level programming model compared with highly customed remote sensing processing APIs or operators and lacks control over

operator access. As cloud computing systems are considered to prioritize the ease of use over high performance, we provide end users with a customed DAG-centric programming model that includes various datatypes and operators dedicated to remote sensing data analysis and operator-level access control capabilities.

GeoTrellis [24,25] and GeoSpark [26], built on the top of Spark, are generally used for batch processing of raster data and similar to the batch computing component of GEE, which is based on FlumeJava [27]. These frameworks usually have a distributed data abstraction customed to raster data based on the RDD data structure [28]. However, at the phase of algorithm exploratory, they cannot be used as REPL tools to provide a public computing service and rapid prototyping. Besides, these frameworks require the computation to be built on the distributed datatypes, which limits the expressiveness and flexibility of end-user programming for remote sensing collection processing. Although Spark can be refactored on serverless technologies, GeoTrellis and GeoSpark cannot be directly used as public cloud commodities to provide end-user programming services.

Iris [29] is a python library used for the analysis and visualization of meteorological data, which provides the ability of batch geocomputation based on the distributed numerical computing framework Dask. Contrary to the Spark-based frameworks, Iris has a higher communication efficiency and is capable of performing certain high-performance computing tasks, such as a dense linear algebra calculation. Nevertheless, Iris is highly customed to the analysis of meteorological data, which is usually organized as a NetCDF file or its variants, and, therefore, could not be applied directly to remote sensing images analysis. Besides, as Iris is deeply bound to Dask, it cannot control operator access and provide the service of paying in proportion to resources used, similar to a serverless public commodity.

With the rise of the disaggregation datacenter, serverless computing is believed to become the default computing paradigm of cloud computing and bring closure to the client–server era [8]. Although GEE has the feature of scaling automatically and billing on usage, it does not claim to be built on serverless technologies. All other remote sensing image processing frameworks need explicit resource provisioning, which can be regarded as based on a server-centric computing paradigm. Despite that serverless cloud functions are becoming more and more lightweight and have been successfully employed for several types of general workloads [30,31], there are still many limitations. The cloud functions are stateless without fine-grained coordination and do not provide high-level parallel operators, posing difficulty for remote sensing data processing workloads.

This paper presents the empirical characteristics and a formal definition of on-the-fly cloud geocomputation for the first time. Then, we give a serverless-based software architecture and some proof-of-concept experiments, which suggest that on-the-fly cloud geocomputation can be efficiently implemented with serverless technologies, such as the object storage system and function computing engine. At the frontend, we provide a DAG-based, end-user programming environment for remote sensing data analysis, which contains a series of customed datatypes and operators. The DAG is one of the core designs, which bridges the user-defined code and the cloud functions at the backend. The logical region is another core design, which guarantees the correctness of focal operators through overlapped zones and controls the amount of input image tiles to achieve a rapid response.

Nevertheless, several aspects of the proposed serverless-based system could be further improved. First, the technology stack of current serverless commodities lacks in-memory storage, similar to Redis, which limits further performance improvements. Future work could refer to Anna [32], which can provide high-performance memory storage services to improve the efficiency of COG reading and cache. In addition, the scheduler in the current system adopts a staged scheduling method, which does not pay attention to the difference in execution time between different nodes. Therefore, it is necessary to develop a scheduling algorithm specially oriented to serverless computing to achieve the optimization of both job completion time and cost of execution [33].

Author Contributions: Conceptualization, J.W. and M.W.; methodology, J.W.; software, J.W.; validation, H.L., L.L. (Lijuan Li), and L.L. (Leilei Li); investigation, J.W.; resources, J.W.; data curation, J.W.; writing—original draft preparation, J.W.; writing—review and editing, L.L. (Lijuan Li); visualization, J.W.; project administration, J.W.; funding acquisition, J.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under grant number 2021YFB3900900. This work was jointly supported by the Fundamental Research Funds for the Central Universities, National Natural Science Foundation of China under grant number 41776197, the Strategic Priority Research Program of Chinese Academy of Sciences under grant number XDB42010403, and the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under grant number 2018YFE0204203.

Data Availability Statement: All satellite data used in the study are available for free download from their respective data portals (https://earthexplorer.usgs.gov/ accessed on 1 January 2022).

Acknowledgments: The authors would like to thank the editors and the anonymous reviewers for their crucial comments, which improved the quality of this paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alam, M.M.; Torgo, L.; Bifet, A. A Survey on Spatio-temporal Data Analytics Systems. *arXiv* **2021**, arXiv:2103.09883. [CrossRef]
2. Gomes, V.; Queiroz, G.; Ferreira, K. An Overview of Platforms for Big Earth Observation Data Management and Analysis. *Remote Sens.* **2020**, *12*, 1253. [CrossRef]
3. Gorelick, N.; Hancher, M.; Dixon, M.; Ilyushchenko, S.; Thau, D.; Moore, R. Google Earth Engine: Planetary-scale geospatial analysis for everyone. *Remote Sens. Environ.* **2017**, *202*, 18–27. [CrossRef]
4. Zaharia, M.; Xin, R.S.; Wendell, P.; Das, T.; Armbrust, M.; Dave, A.; Meng, X.; Rosen, J.; Venkataraman, S.; Franklin, M.J. Apache spark: A unified engine for big data processing. *Commun. ACM* **2016**, *59*, 56–65. [CrossRef]
5. White, T. *Hadoop: The Definitive Guide*; O'Reilly Media, Inc.: Sebastopol, CA, USA, 2012.
6. Rocklin, M. Dask: Parallel computation with blocked algorithms and task scheduling. In Proceedings of the 14th Python in Science Conference, Austin, TX, USA, 6–12 July 2015.
7. Han, S.; Egi, N.; Panda, A.; Ratnasamy, S.; Shi, G.; Shenker, S. Network support for resource disaggregation in next-generation datacenters. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, College Park, MD, USA, 21–22 November 2013; pp. 1–7.
8. Jonas, E.; Schleier-Smith, J.; Sreekanti, V.; Tsai, C.-C.; Khandelwal, A.; Pu, Q.; Shankar, V.; Carreira, J.; Krauth, K.; Yadwadkar, N. Cloud programming simplified: A berkeley view on serverless computing. *arXiv* **2019**, arXiv:1902.03383.
9. Ozturk, D.; Chaudhary, A.; Votava, P.; Kotfila, C. GeoNotebook: Browser based Interactive analysis and visualization workflow for very large climate and geospatial datasets. *AGU Fall Meet. Abstr.* **2016**, *2016*, IN53A-1876.
10. Jangda, A.; Pinckney, D.; Brun, Y.; Guha, A. Formal foundations of serverless computing. *Proc. ACM Program. Lang.* **2019**, *3*, 1–26. [CrossRef]
11. AlibabaCloud. Function Computing. Available online: https://help.aliyun.com/product/50980.html (accessed on 3 April 2022).
12. AlibabaCloud. Serverless Workflow. Available online: https://help.aliyun.com/product/113549.html (accessed on 3 April 2022).
13. AlibabaCloud. TableStore. Available online: https://help.aliyun.com/product/27278.html (accessed on 3 April 2022).
14. AlibabaCloud. Message Service. Available online: https://help.aliyun.com/product/27412.html (accessed on 3 April 2022).
15. AlibabaCloud. Relation Database System. Available online: https://help.aliyun.com/product/26090.html (accessed on 3 April 2022).
16. AlibabaCloud. Object Storage Service. Available online: https://help.aliyun.com/product/31815.html (accessed on 3 April 2022).
17. COG. Cloud Optimized GeoTIFF. Available online: https://www.cogeo.org/ (accessed on 3 April 2022).
18. STAC. SpatioTemporal Asset Catalogs. Available online: http://stacspec.org/ (accessed on 3 April 2022).
19. Hennessy, J.; Patterson, D. A New Golden Age for Computer Architecture: Domain-Specific Hardware/Software Co-Design, Enhanced Security, Open Instruction Sets, and Agile Chip Development. In Proceedings of the Turing Lecture Given at ISCA'18, Los Angeles, CA, USA, 2–6 June 2018; Volume 10.
20. Crocker, D.; Overell, P. *Augmented BNF for Syntax Specifications: ABNF*; RFC 2234; HKU Sandy Bay RFC Ltd.: Pok Fu Lam, China, 1997.
21. Dong, B.; Wu, K.; Byna, S.; Liu, J.; Zhao, W.; Rusu, F. ArrayUDF: User-defined scientific data analysis on arrays. In Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, Washington, DC, USA, 26–30 June 2017; pp. 53–64.

22. Lewis, A.; Oliver, S.; Lymburner, L.; Evans, B.; Wyborn, L.; Mueller, N.; Raevksi, G.; Hooke, J.; Woodcock, R.; Sixsmith, J.; et al. The Australian Geoscience Data Cube—Foundations and lessons learned. *Remote Sens. Environ.* **2017**, *202*, 276–292. [CrossRef]
23. Hoyer, S.; Hamman, J. xarray: ND labeled arrays and datasets in Python. *J. Open Res. Softw.* **2017**, *5*, 10. [CrossRef]
24. Eldawy, A.; Mokbel, M.F. The era of big spatial data: A survey. *Foundations and Trends in Databases* **2016**, *6*, 163–273. [CrossRef]
25. Geotrellis. GeoTrellis is a Geographic Data Processing Engine for High Performance Applications. Available online: https://geotrellis.io/ (accessed on 3 April 2022).
26. Yu, J.; Wu, J.; Sarwat, M. Geospark: A cluster computing framework for processing large-scale spatial data. In Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, 3–6 November 2015; pp. 1–4.
27. Chambers, C.; Raniwala, A.; Perry, F.; Adams, S.; Henry, R.R.; Bradshaw, R.; Weizenbaum, N. FlumeJava: Easy, efficient data-parallel pipelines. *ACM Sigplan Not.* **2010**, *45*, 363–375. [CrossRef]
28. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauly, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
29. Hamman, J.; Rocklin, M.; Abernathy, R. Pangeo: A big-data ecosystem for scalable earth system science. In Proceedings of the EGU General Assembly Conference Abstracts, Vienna, Austria, 8–13 April 2018; p. 12146.
30. Taibi, D.; El Ioini, N.; Pahl, C.; Niederkofler, J.R.S. Serverless cloud computing (function-as-a-service) patterns: A multivocal literature review. In Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020), Prague, Czech Republic, 7–9 May 2020.
31. Shankar, V.; Krauth, K.; Vodrahalli, K.; Pu, Q.; Recht, B.; Stoica, I.; Ragan-Kelley, J.; Jonas, E.; Venkataraman, S. Serverless linear algebra. In Proceedings of the 11th ACM Symposium on Cloud Computing, Seattle, WA, USA, 19–21 October 2020; pp. 281–295.
32. Wu, C.; Faleiro, J.; Lin, Y.; Hellerstein, J. Anna: A kvs for any scale. *IEEE Trans. Knowl. Data Eng.* **2019**, *33*, 344–358. [CrossRef]
33. Zhang, H.; Tang, Y.; Khandelwal, A.; Chen, J.; Stoica, I. Caerus:{NIMBLE} Task Scheduling for Serverless Analytics. In Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), Boston, MA, USA, 12–14 April 2021; pp. 653–669.