



US 20250258741A1

(19) **United States**

(12) **Patent Application Publication**  
**ANWAR et al.**

(10) **Pub. No.: US 2025/0258741 A1**

(43) **Pub. Date: Aug. 14, 2025**

(54) **REPLICATING FILES GENERATED DURING PROCESSING**

(71) Applicant: **Splunk Inc.**, San Francisco, CA (US)

(72) Inventors: **Tameem ANWAR**, Dublin, CA (US);  
**John Nguyen**, San Francisco, CA (US);  
**Sai Krishna Sajja**, Dublin, CA (US);  
**Sanket Murlidhar Varkhede**, Santa Clara, CA (US); **Ruochoen Zhang**, Milpitas, CA (US)

(21) Appl. No.: **19/192,051**

(22) Filed: **Apr. 28, 2025**

**Related U.S. Application Data**

(63) Continuation of application No. 18/427,653, filed on Jan. 30, 2024, now Pat. No. 12,287,710.

**Publication Classification**

(51) **Int. Cl.**  
**G06F 11/14** (2006.01)  
**G06F 9/4401** (2018.01)

**G06F 16/11** (2019.01)

**G06F 16/178** (2019.01)

(52) **U.S. Cl.**

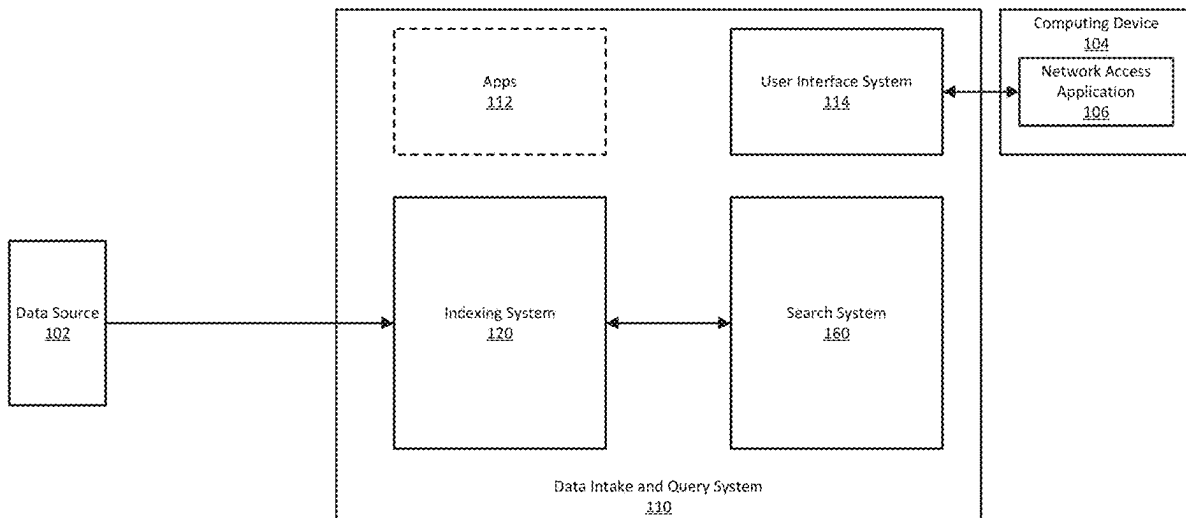
**CPC** ..... **G06F 11/1469** (2013.01); **G06F 9/4401** (2013.01); **G06F 16/116** (2019.01); **G06F 16/178** (2019.01); **G06F 2201/84** (2013.01)

(57)

**ABSTRACT**

A computer-implemented method for performing cross regional disaster recovery includes receiving event data at an active deployment of a data intake and query system. The method further includes processing the event data in a plurality of stages of the data intake and query system to convert the event data into searchable buckets of indexed data, wherein files generated at each stage of the processing are uploaded to a first scalable storage module in the active deployment. Further, the method includes replicating the files generated at each stage of the processing to a second scalable storage module in a standby deployment of the data intake and query system. Responsive to an outage at the active deployment, the method includes transitioning control to the standby deployment and recovering the event data at the standby deployment using the files replicated to the second scalable storage module.

100



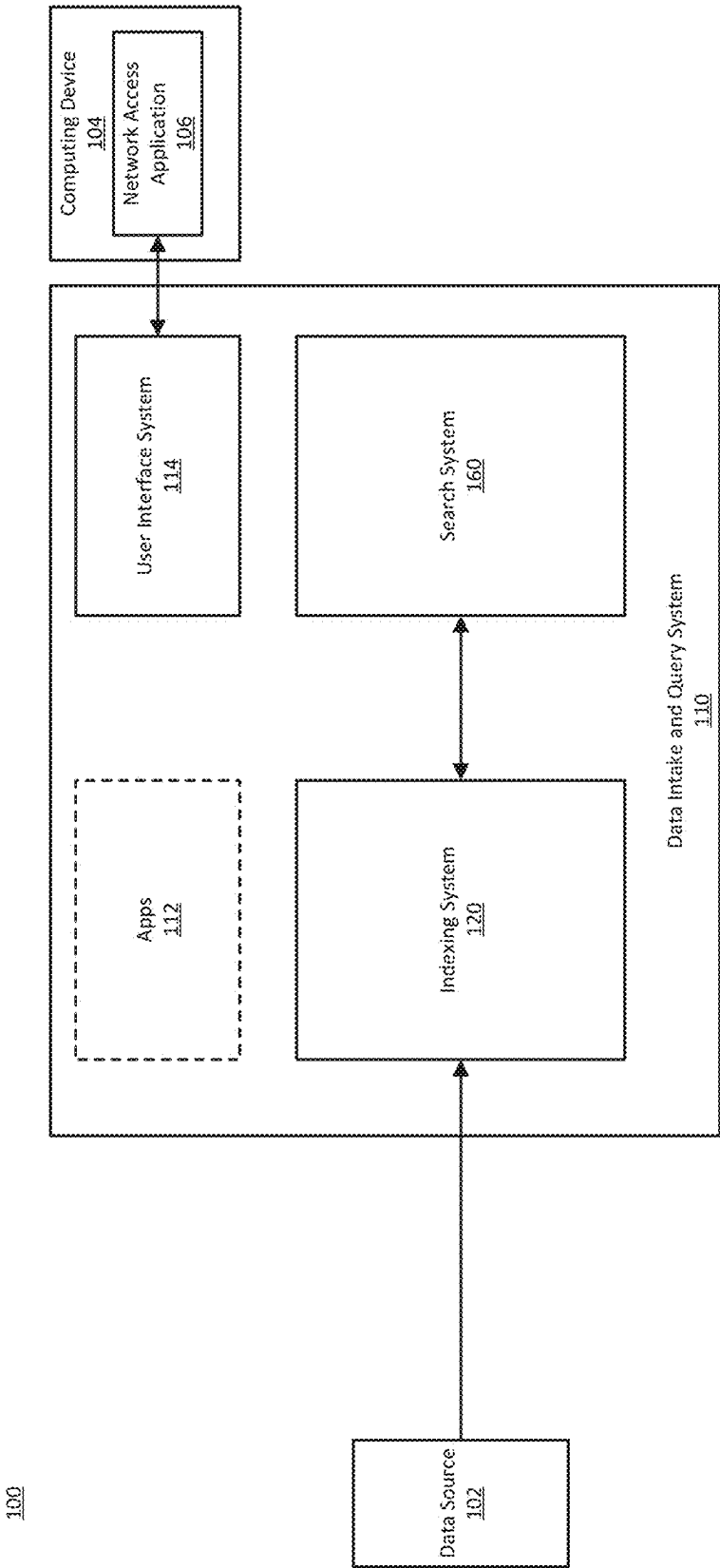


FIG. 1

220

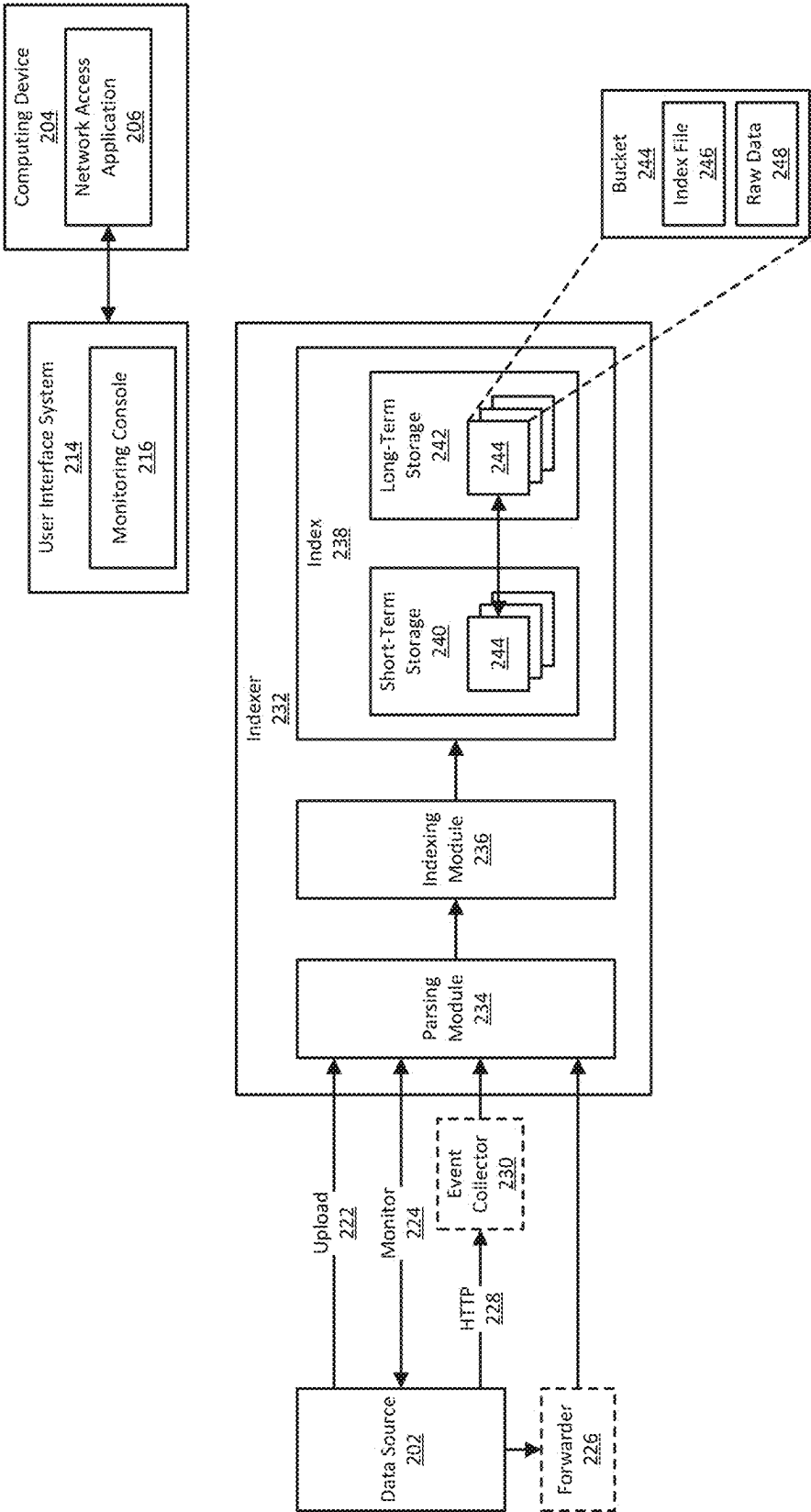


FIG. 2

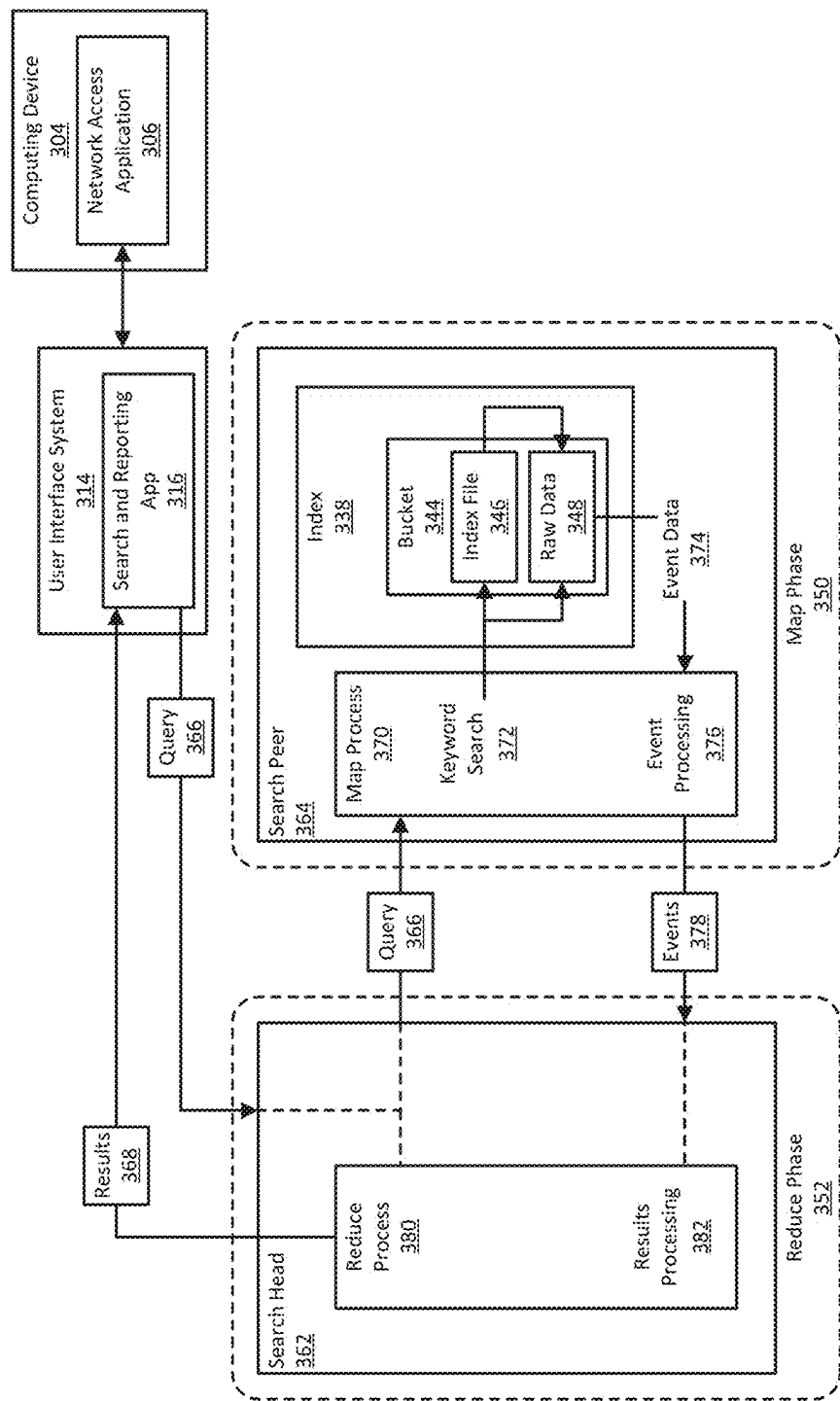


FIG. 3

360

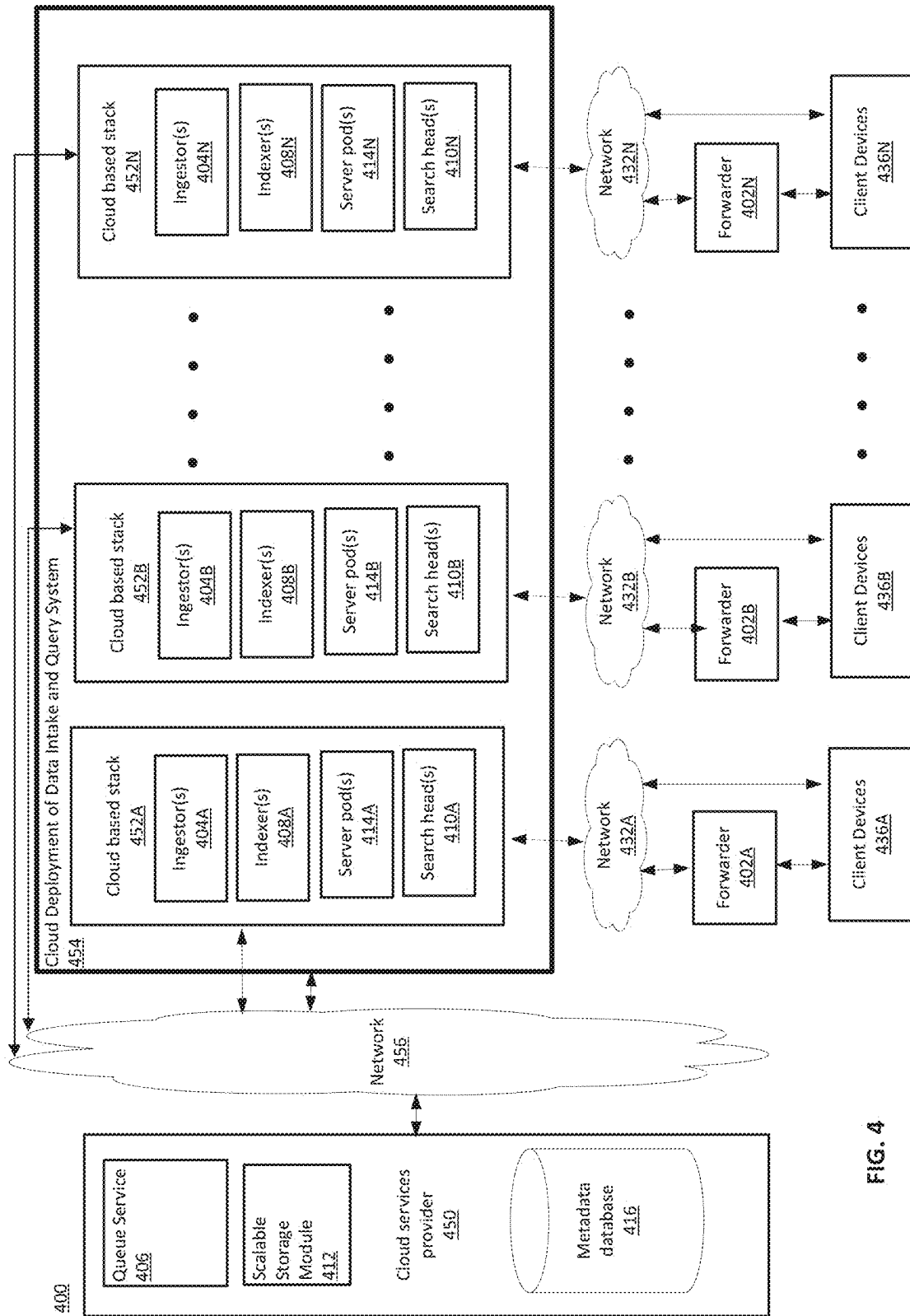


FIG. 4

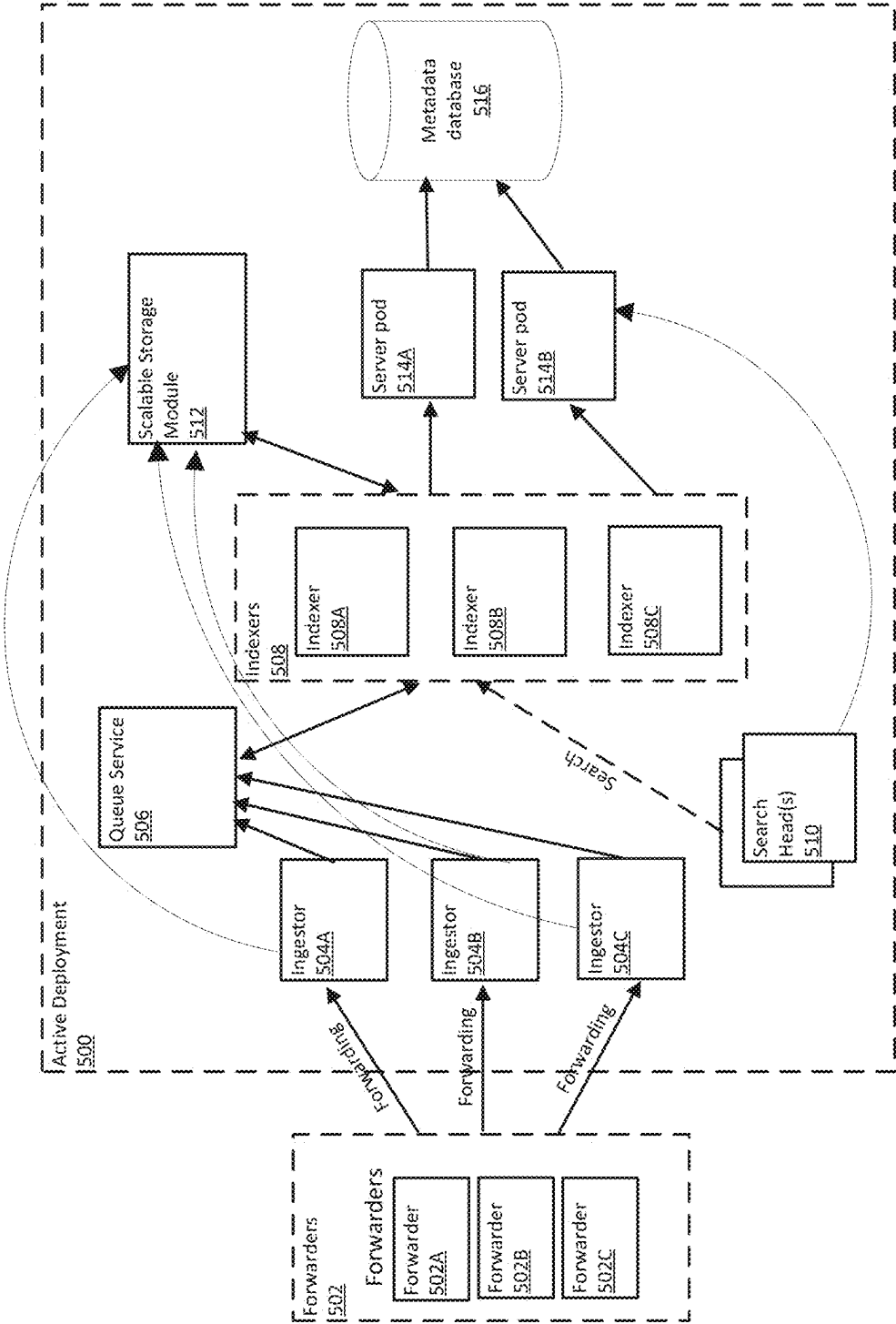
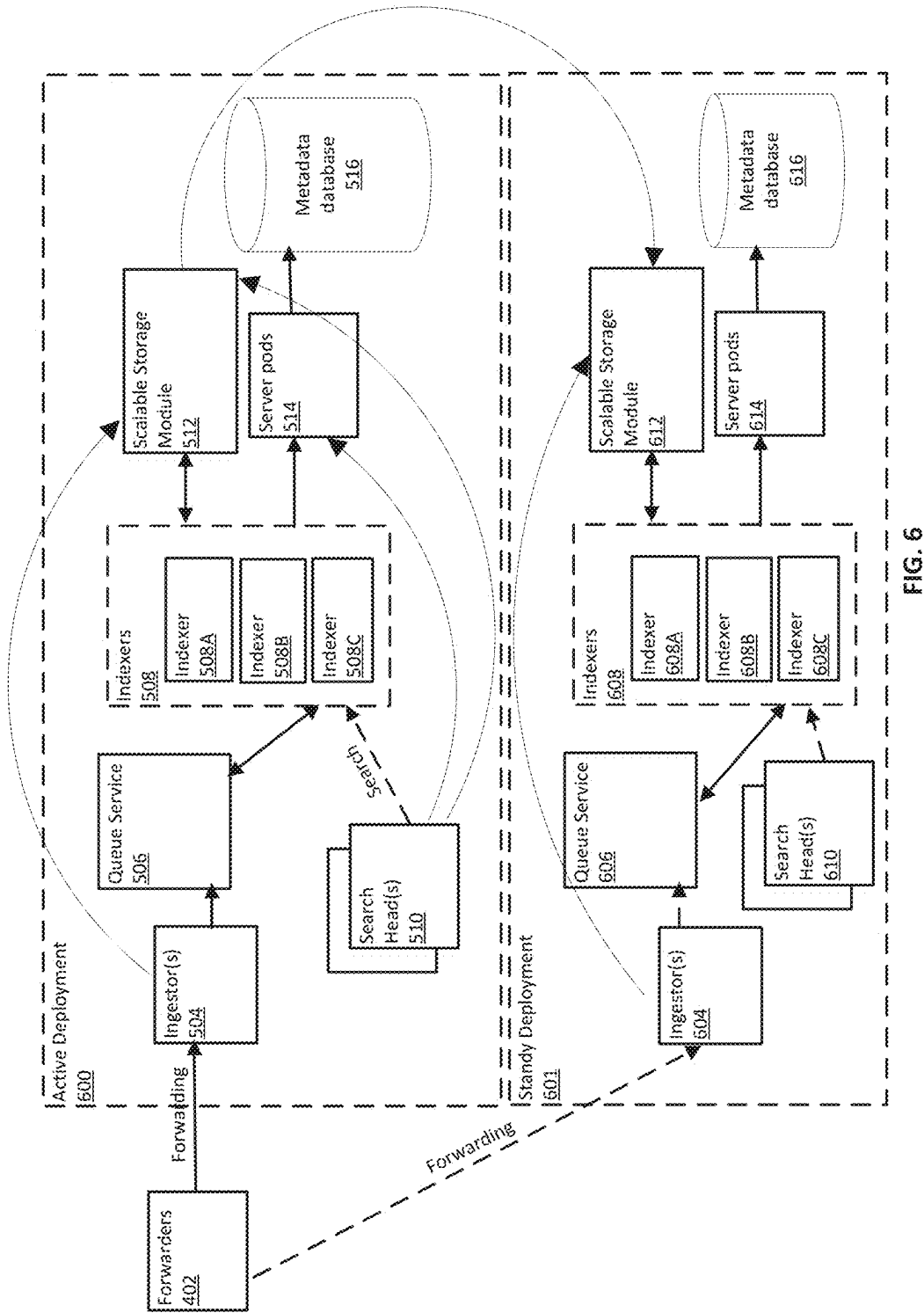


FIG. 5



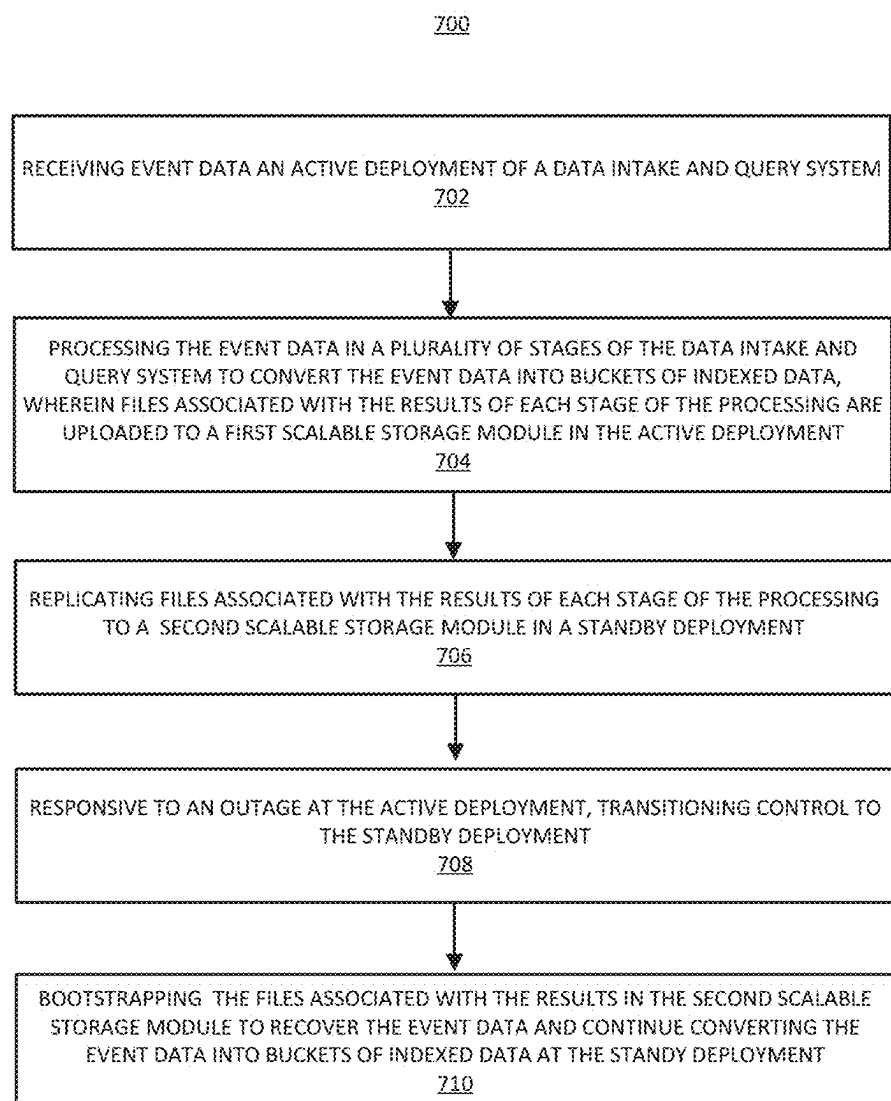


FIG. 7



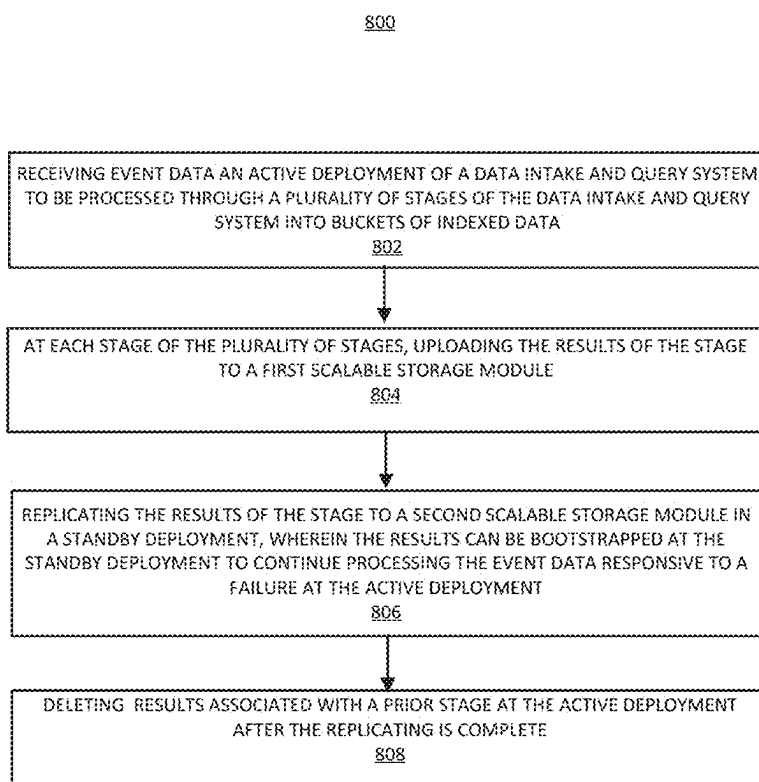


FIG. 8

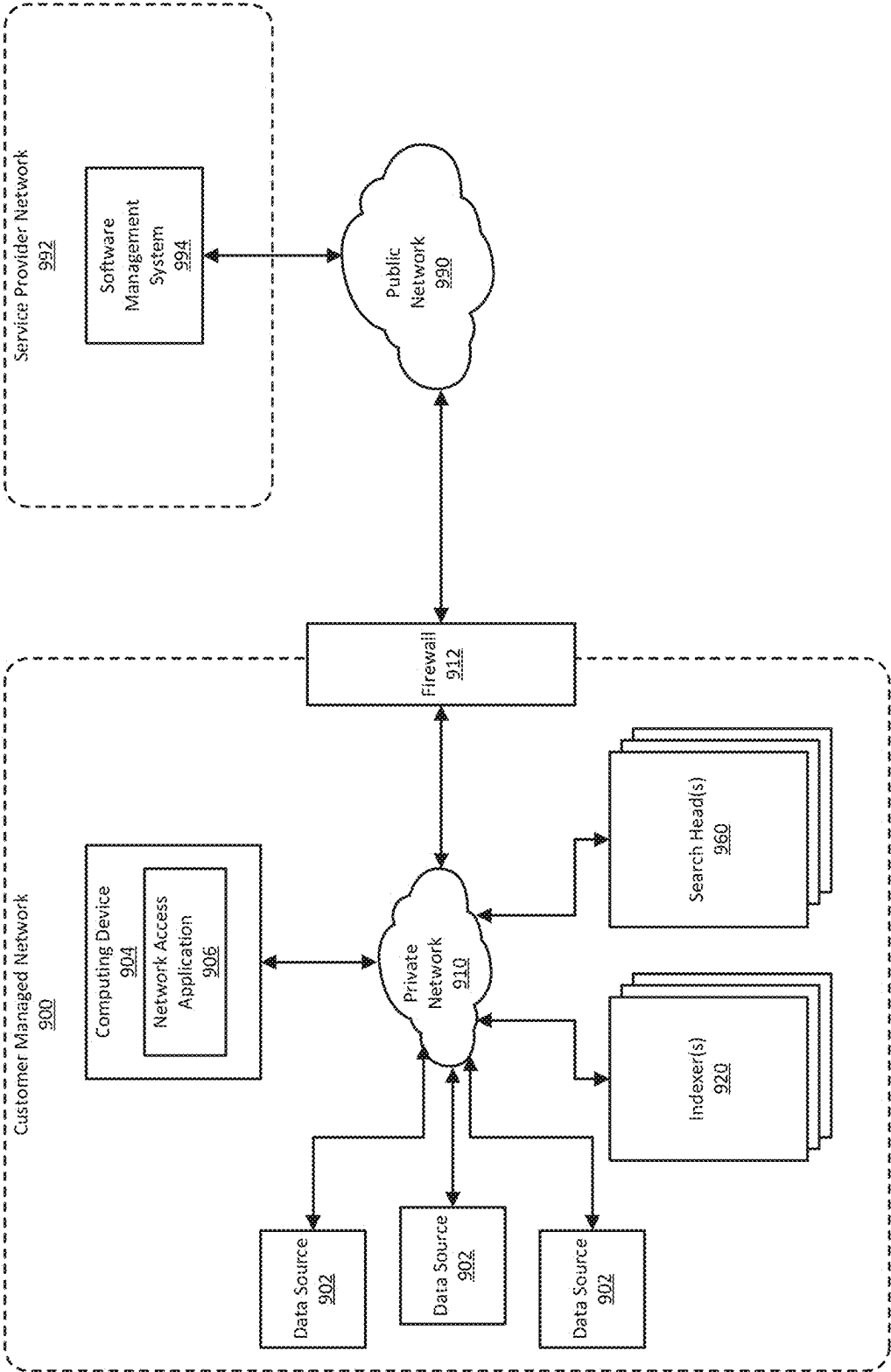


FIG. 9

## REPLICATING FILES GENERATED DURING PROCESSING

### CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application is a continuation of U.S. patent application Ser. No. 18/427,653, filed on Jan. 30, 2024. The entire contents of which are incorporated herein.

### BACKGROUND

[0002] Information technology (IT) environments can include diverse types of data systems that store large amounts of diverse data types generated by numerous devices. For example, a big data ecosystem may include databases such as MySQL and Oracle databases, cloud computing services such as Amazon web services (AWS), and other data systems that store passively or actively generated data, including machine-generated data (“machine data”). The machine data can include log data, performance data, diagnostic data, metrics, tracing data, or any other data that can be analyzed to diagnose equipment performance problems, monitor user interactions, and to derive other insights.

[0003] The large amount and diversity of data systems containing large amounts of structured, semi-structured, and unstructured data relevant to any search query can be massive and continues to grow rapidly. This technological evolution can give rise to various challenges in relation to managing, understanding and effectively utilizing the data. To reduce the potentially vast amount of data that may be generated, some data systems pre-process data based on anticipated data analysis needs. In particular, specified data items may be extracted from the generated data and stored in a data system to facilitate efficient retrieval and analysis of those data items at a later time. At least some of the remainder of the generated data is typically discarded during pre-processing.

[0004] However, storing massive quantities of minimally processed or unprocessed data (collectively and individually referred to as “raw data”) for later retrieval and analysis is becoming increasingly more feasible as storage capacity becomes more inexpensive and plentiful. In general, storing raw data and performing analysis on that data later can provide greater flexibility because it enables an analyst to analyze all of the generated data instead of only a fraction of it. Although the availability of vastly greater amounts of diverse data on diverse data systems provides opportunities to derive new insights, it also gives rise to technical challenges to store, search and analyze the data in a performant way.

[0005] One notable technical challenge in handling the ingestion and processing of extensive data, including indexing, is the necessity to seamlessly transition to a secondary region in the event of a major disaster affecting the primary servers within a cloud configuration’s primary region. Implementing cross-region disaster recovery introduces two primary challenges that conventional systems struggle to address effectively: minimizing data loss while ensuring data availability in the secondary region and achieving prompt service availability in the secondary region.

### BRIEF DESCRIPTION OF THE DRAWINGS

[0006] Illustrative examples are described in detail below with reference to the following figures:

[0007] FIG. 1 is a block diagram illustrating an example computing environment that includes a data intake and query system.

[0008] FIG. 2 is a block diagram illustrating in greater detail an example of an indexing system of a data intake and query system, such as the data intake and query system of FIG. 1.

[0009] FIG. 3 is a block diagram illustrating in greater detail an example of the search system of a data intake and query system, such as the data intake and query system of FIG. 1.

[0010] FIG. 4 illustrates a network architecture that enables data replication in the context of disaster recovery for cloud-based data intake and query system, in implementations according to the present disclosure.

[0011] FIG. 5 illustrates a cloud-based network architecture that enables data ingestion and indexing of event data in an active deployment, in implementations according to the present disclosure.

[0012] FIG. 6 illustrates control being transferred from an active deployment of the data intake and query system to a standby deployment, in implementations according to the present disclosure.

[0013] FIG. 7 is a flowchart illustrating an example of a computer-implemented method for transferring control from an active deployment of a data intake and query system to a standby deployment, in implementations according to the present disclosure.

[0014] FIG. 8 is a flowchart illustrating another example of a computer-implemented method for transferring control from an active deployment of a data intake and query system to a standby deployment, in implementations according to the present disclosure.

[0015] FIG. 9 illustrates an example of a self-managed network 900 that includes a data intake and query system.

### DETAILED DESCRIPTION

[0016] Entities of various types, such as companies, educational institutions, medical facilities, governmental departments, and private individuals, among other examples, operate computing environments for various purposes. Computing environments, which can also be referred to as information technology environments, can include inter-networked, physical hardware devices, the software executing on the hardware devices, and the users of the hardware and software. As an example, an entity such as a school can operate a Local Area Network (LAN) that includes desktop computers, laptop computers, smart phones, and tablets connected to a physical and wireless network, where users correspond to teachers and students. In this example, the physical devices may be in buildings or a campus that is controlled by the school. As another example, an entity such as a business can operate a Wide Area Network (WAN) that includes physical devices in multiple geographic locations where the offices of the business are located. In this example, the different offices can be inter-networked using a combination of public networks such as the Internet and private networks. As another example, an entity can operate a data center at a centralized location, where computing resources (such as compute, memory,

and/or networking resources) are kept and maintained, and whose resources are accessible over a network to users who may be in different geographical locations. In this example, users associated with the entity that operates the data center can access the computing resources in the data center over public and/or private networks that may not be operated and controlled by the same entity. Alternatively or additionally, the operator of the data center may provide the computing resources to users associated with other entities, for example on a subscription basis. Such a data center operator may be referred to as a cloud services provider, and the services provided by such an entity may be described by one or more service models, such as to Software-as-a Service (SaaS) model, Infrastructure-as-a-Service (IaaS) model, or Platform-as-a-Service (PaaS), among others. In these examples, users may expect resources and/or services to be available on demand and without direct active management by the user, a resource delivery model often referred to as cloud computing.

**[0017]** Entities that operate computing environments need information about their computing environments. For example, an entity may need to know the operating status of the various computing resources in the entity's computing environment, so that the entity can administer the environment, including performing configuration and maintenance, performing repairs or replacements, provisioning additional resources, removing unused resources, or addressing issues that may arise during operation of the computing environment, among other examples. As another example, an entity can use information about a computing environment to identify and remediate security issues that may endanger the data, users, and/or equipment in the computing environment. As another example, an entity may be operating a computing environment for some purpose (e.g., to run an online store, to operate a bank, to manage a municipal railway, etc.) and may want information about the computing environment that can aid the entity in understanding whether the computing environment is operating efficiently and for its intended purpose.

**[0018]** Collection and analysis of the data from a computing environment can be performed by a data intake and query system such as is described herein. A data intake and query system can ingest and store data obtained from the components in a computing environment, and can enable an entity to search, analyze, and visualize the data. Through these and other capabilities, the data intake and query system can enable an entity to use the data for administration of the computing environment, to detect security issues, to understand how the computing environment is performing or being used, and/or to perform other analytics.

**[0019]** FIG. 1 is a block diagram illustrating an example computing environment 100 that includes a data intake and query system 110. The data intake and query system 110 obtains data from a data source 102 in the computing environment 100, and ingests the data using an indexing system 120. A search system 160 of the data intake and query system 110 enables users to navigate the indexed data. Though drawn with separate boxes in FIG. 1, in some implementations the indexing system 120 and the search system 160 can have overlapping components. A computing device 104, running a network access application 106, can communicate with the data intake and query system 110 through a user interface system 114 of the data intake and query system 110. Using the computing device 104, a user

can perform various operations with respect to the data intake and query system 110, such as administration of the data intake and query system 110, management and generation of "knowledge objects," (user-defined entities for enriching data, such as saved searches, event types, tags, field extractions, lookups, reports, alerts, data models, workflow actions, and fields), initiating of searches, and generation of reports, among other operations. The data intake and query system 110 can further optionally include apps 112 that extend the search, analytics, and/or visualization capabilities of the data intake and query system 110.

**[0020]** The data intake and query system 110 can be implemented using program code that can be executed using a computing device. A computing device is an electronic device that has a memory for storing program code instructions and a hardware processor for executing the instructions. The computing device can further include other physical components, such as a network interface or components for input and output. The program code for the data intake and query system 110 can be stored on a non-transitory computer-readable medium, such as a magnetic or optical storage disk or a flash or solid-state memory, from which the program code can be loaded into the memory of the computing device for execution. "Non-transitory" means that the computer-readable medium can retain the program code while not under power, as opposed to volatile or "transitory" memory or media that requires power in order to retain data.

**[0021]** In various examples, the program code for the data intake and query system 110 can be executed on a single computing device, or execution of the program code can be distributed over multiple computing devices. For example, the program code can include instructions for both indexing and search components (which may be part of the indexing system 120 and/or the search system 160, respectively), which can be executed on a computing device that also provides the data source 102. As another example, the program code can be executed on one computing device, where execution of the program code provides both indexing and search components, while another copy of the program code executes on a second computing device that provides the data source 102. As another example, the program code can be configured such that, when executed, the program code implements only an indexing component or only a search component. In this example, a first instance of the program code that is executing the indexing component and a second instance of the program code that is executing the search component can be executing on the same computing device or on different computing devices.

**[0022]** The data source 102 of the computing environment 100 is a component of a computing device that produces machine data. The component can be a hardware component (e.g., a microprocessor or a network adapter, among other examples) or a software component (e.g., a part of the operating system or an application, among other examples). The component can be a virtual component, such as a virtual machine, a virtual machine monitor (also referred as a hypervisor), a container, or a container orchestrator, among other examples. Examples of computing devices that can provide the data source 102 include personal computers (e.g., laptops, desktop computers, etc.), handheld devices (e.g., smart phones, tablet computers, etc.), servers (e.g., network servers, compute servers, storage servers, domain name servers, web servers, etc.), network infrastructure devices (e.g., routers, switches, firewalls, etc.), and "Internet

of Things” devices (e.g., vehicles, home appliances, factory equipment, etc.), among other examples. Machine data is electronically generated data that is output by the component of the computing device and reflects activity of the component. Such activity can include, for example, operation status, actions performed, performance metrics, communications with other components, or communications with users, among other examples. The component can produce machine data in an automated fashion (e.g., through the ordinary course of being powered on and/or executing) and/or as a result of user interaction with the computing device (e.g., through the user’s use of input/output devices or applications). The machine data can be structured, semi-structured, and/or unstructured. The machine data may be referred to as raw machine data when the data is unaltered from the format in which the data was output by the component of the computing device. Examples of machine data include operating system logs, web server logs, live application logs, network feeds, metrics, change monitoring, message queues, and archive files, among other examples.

**[0023]** As discussed in greater detail below, the indexing system **120** obtains machine data from the data source **102** and processes and stores the data. Processing and storing of data may be referred to as “ingestion” of the data. Processing of the data can include parsing the data to identify individual events, where an event is a discrete portion of machine data that can be associated with a timestamp. Processing of the data can further include generating an index of the events, where the index is a data storage structure in which the events are stored. The indexing system **120** does not require prior knowledge of the structure of incoming data (e.g., the indexing system **120** does not need to be provided with a schema describing the data). Additionally, the indexing system **120** retains a copy of the data as it was received by the indexing system **120** such that the original data is always available for searching (e.g., no data is discarded, though, in some examples, the indexing system **120** can be configured to do so).

**[0024]** The search system **160** searches the data stored by the indexing **120** system. As discussed in greater detail below, the search system **160** enables users associated with the computing environment **100** (and possibly also other users) to navigate the data, generate reports, and visualize search results in “dashboards” output using a graphical interface. Using the facilities of the search system **160**, users can obtain insights about the data, such as retrieving events from an index, calculating metrics, searching for specific conditions within a rolling time window, identifying patterns in the data, and predicting future trends, among other examples. To achieve greater efficiency, the search system **160** can apply map-reduce methods to parallelize searching of large volumes of data. Additionally, because the original data is available, the search system **160** can apply a schema to the data at search time. This allows different structures to be applied to the same data, or for the structure to be modified if or when the content of the data changes. Application of a schema at search time may be referred to herein as a late-binding schema technique.

**[0025]** The user interface system **114** provides mechanisms through which users associated with the computing environment **100** (and possibly others) can interact with the data intake and query system **110**. These interactions can include configuration, administration, and management of the indexing system **120**, initiation and/or scheduling of

queries that are to be processed by the search system **160**, receipt or reporting of search results, and/or visualization of search results. The user interface system **114** can include, for example, facilities to provide a command line interface or a web-based interface.

**[0026]** Users can access the user interface system **114** using a computing device **104** that communicates with data intake and query system **110**, possibly over a network. A “user,” in the context of the implementations and examples described herein, is a digital entity that is described by a set of information in a computing environment. The set of information can include, for example, a user identifier, a username, a password, a user account, a set of authentication credentials, a token, other data, and/or a combination of the preceding. Using the digital entity that is represented by a user, a person can interact with the computing environment **100**. For example, a person can log in as a particular user and, using the user’s digital information, can access the data intake and query system **110**. A user can be associated with one or more people, meaning that one or more people may be able to use the same user’s digital information. For example, an administrative user account may be used by multiple people who have been given access to the administrative user account. Alternatively or additionally, a user can be associated with another digital entity, such as a bot (e.g., a software program that can perform autonomous tasks). A user can also be associated with one or more entities. For example, a company can have associated with it a number of users. In this example, the company may control the users’ digital information, including assignment of user identifiers, management of security credentials, control of which persons are associated with which users, and so on.

**[0027]** The computing device **104** can provide a human-machine interface through which a person can have a digital presence in the computing environment **100** in the form of a user. The computing device **104** is an electronic device having one or more processors and a memory capable of storing instructions for execution by the one or more processors. The computing device **104** can further include input/output (I/O) hardware and a network interface. Applications executed by the computing device **104** can include a network access application **106**, such as a web browser, which can use a network interface of the client computing device **104** to communicate, over a network, with the user interface system **114** of the data intake and query system **110**. The user interface system **114** can use the network access application **106** to generate user interfaces that enable a user to interact with the data intake and query system **110**. A web browser is one example of a network access application. A shell tool can also be used as a network access application. In some examples, the data intake and query system **110** is an application executing on the computing device **106**. In such examples, the network access application **106** can access the user interface system **114** without going over a network.

**[0028]** The data intake and query system **110** can optionally include apps **112**. An app of the data intake and query system **110** is a collection of configurations, knowledge objects (a user-defined entity that enriches the data in the data intake and query system **110**), views, and dashboards that may provide additional functionality, different techniques for searching the data, and/or additional insights into the data. The data intake and query system **110** can execute

multiple applications simultaneously. Example applications include an information technology service intelligence application, which can monitor and analyze the performance and behavior of the computing environment 100, and an enterprise security application, which can include content and searches to assist security analysts in diagnosing and acting on anomalous or malicious behavior in the computing environment 100.

**[0029]** Though FIG. 1 illustrates only one data source, in practical implementations, the computing environment 100 contains many data sources spread across numerous computing devices. The computing devices may be controlled and operated by a single entity. For example, in an “on the premises” or “on-prem” implementation, the computing devices may physically and digitally be controlled by one entity, meaning that the computing devices are in physical locations that are owned and/or operated by the entity and are within a network domain that is controlled by the entity. In an entirely on-prem implementation of the computing environment 100, the data intake and query system 110 executes on an on-prem computing device and obtains machine data from on-prem data sources. An on-prem implementation can also be referred to as an “enterprise” network, though the term “on-prem” refers primarily to physical locality of a network and who controls that location while the term “enterprise” may be used to refer to the network of a single entity. As such, an enterprise network could include cloud components.

**[0030]** “Cloud” or “in the cloud” refers to a network model in which an entity operates network resources (e.g., processor capacity, network capacity, storage capacity, etc.), located for example in a data center, and makes those resources available to users and/or other entities over a network. A “private cloud” is a cloud implementation where the entity provides the network resources only to its own users. A “public cloud” is a cloud implementation where an entity operates network resources in order to provide them to users that are not associated with the entity and/or to other entities. In this implementation, the provider entity can, for example, allow a subscriber entity to pay for a subscription that enables users associated with subscriber entity to access a certain amount of the provider entity’s cloud resources, possibly for a limited time. A subscriber entity of cloud resources can also be referred to as a tenant of the provider entity. Users associated with the subscriber entity access the cloud resources over a network, which may include the public Internet. In contrast to an on-prem implementation, a subscriber entity does not have physical control of the computing devices that are in the cloud, and has digital access to resources provided by the computing devices only to the extent that such access is enabled by the provider entity.

**[0031]** In some implementations, the computing environment 100 can include on-prem and cloud-based computing resources, or only cloud-based resources. For example, an entity may have on-prem computing devices and a private cloud. In this example, the entity operates the data intake and query system 110 and can choose to execute the data intake and query system 110 on an on-prem computing device or in the cloud. In another example, a provider entity operates the data intake and query system 110 in a public cloud and provides the functionality of the data intake and query system 110 as a service, for example under a Software-as-a-Service (SaaS) model, to entities that pay for the user of

the service on a subscription basis. In this example, the provider entity can provision a separate tenant (or possibly multiple tenants) in the public cloud network for each subscriber entity, where each tenant executes a separate and distinct instance of the data intake and query system 110. In some implementations, the entity providing the data intake and query system 110 is itself subscribing to the cloud services of a cloud service provider. As an example, a first entity provides computing resources under a public cloud service model, a second entity subscribes to the cloud services of the first provider entity and uses the cloud computing resources to operate the data intake and query system 110, and a third entity can subscribe to the services of the second provider entity in order to use the functionality of the data intake and query system 110. In this example, the data sources are associated with the third entity, users accessing the data intake and query system 110 are associated with the third entity, and the analytics and insights provided by the data intake and query system 110 are for purposes of the third entity’s operations.

**[0032]** FIG. 2 is a block diagram illustrating in greater detail an example of an indexing system 220 of a data intake and query system, such as the data intake and query system 110 of FIG. 1. The indexing system 220 of FIG. 2 uses various methods to obtain machine data from a data source 202 and stores the data in an index 238 of an indexer 232. As discussed previously, a data source is a hardware, software, physical, and/or virtual component of a computing device that produces machine data in an automated fashion and/or as a result of user interaction. Examples of data sources include files and directories; network event logs; operating system logs, operational data, and performance monitoring data; metrics; first-in, first-out queues; scripted inputs; and modular inputs, among others. The indexing system 220 enables the data intake and query system to obtain the machine data produced by the data source 202 and to store the data for searching and retrieval.

**[0033]** Users can administer the operations of the indexing system 220 using a computing device 204 that can access the indexing system 220 through a user interface system 214 of the data intake and query system. For example, the computing device 204 can be executing a network access application 206, such as a web browser or a terminal, through which a user can access a monitoring console 216 provided by the user interface system 214. The monitoring console 216 can enable operations such as: identifying the data source 202 for data ingestion; configuring the indexer 232 to index the data from the data source 232; configuring a data ingestion method; configuring, deploying, and managing clusters of indexers; and viewing the topology and performance of a deployment of the data intake and query system, among other operations. The operations performed by the indexing system 220 may be referred to as “index time” operations, which are distinct from “search time” operations that are discussed further below.

**[0034]** The indexer 232, which may be referred to herein as a data indexing component, coordinates and performs most of the index time operations. The indexer 232 can be implemented using program code that can be executed on a computing device. The program code for the indexer 232 can be stored on a non-transitory computer-readable medium (e.g. a magnetic, optical, or solid state storage disk, a flash memory, or another type of non-transitory storage media), and from this medium can be loaded or copied to the

memory of the computing device. One or more hardware processors of the computing device can read the program code from the memory and execute the program code in order to implement the operations of the indexer 232. In some implementations, the indexer 232 executes on the computing device 204 through which a user can access the indexing system 220. In some implementations, the indexer 232 executes on a different computing device than the illustrated computing device 204.

[0035] The indexer 232 may be executing on the computing device that also provides the data source 202 or may be executing on a different computing device. In implementations wherein the indexer 232 is on the same computing device as the data source 202, the data produced by the data source 202 may be referred to as “local data.” In other implementations the data source 202 is a component of a first computing device and the indexer 232 executes on a second computing device that is different from the first computing device. In these implementations, the data produced by the data source 202 may be referred to as “remote data.” In some implementations, the first computing device is “on-prem” and in some implementations the first computing device is “in the cloud.” In some implementations, the indexer 232 executes on a computing device in the cloud and the operations of the indexer 232 are provided as a service to entities that subscribe to the services provided by the data intake and query system.

[0036] For a given data produced by the data source 202, the indexing system 220 can be configured to use one of several methods to ingest the data into the indexer 232. These methods include upload 222, monitor 224, using a forwarder 226, or using HyperText Transfer Protocol (HTTP 228) and an event collector 230. These and other methods for data ingestion may be referred to as “getting data in” (GDI) methods.

[0037] Using the upload 222 method, a user can specify a file for uploading into the indexer 232. For example, the monitoring console 216 can include commands or an interface through which the user can specify where the file is located (e.g., on which computing device and/or in which directory of a file system) and the name of the file. The file may be located at the data source 202 or maybe on the computing device where the indexer 232 is executing. Once uploading is initiated, the indexer 232 processes the file, as discussed further below. Uploading is a manual process and occurs when instigated by a user. For automated data ingestion, the other ingestion methods are used.

[0038] The monitor 224 method enables the indexing system 202 to monitor the data source 202 and continuously or periodically obtain data produced by the data source 202 for ingestion by the indexer 232. For example, using the monitoring console 216, a user can specify a file or directory for monitoring. In this example, the indexing system 202 can execute a monitoring process that detects whenever the file or directory is modified and causes the file or directory contents to be sent to the indexer 232. As another example, a user can specify a network port for monitoring. In this example, a monitoring process can capture data received at or transmitting from the network port and cause the data to be sent to the indexer 232. In various examples, monitoring can also be configured for data sources such as operating system event logs, performance data generated by an operating system, operating system registries, operating system directory services, and other data sources.

[0039] Monitoring is available when the data source 202 is local to the indexer 232 (e.g., the data source 202 is on the computing device where the indexer 232 is executing). Other data ingestion methods, including forwarding and the event collector 230, can be used for either local or remote data sources.

[0040] A forwarder 226, which may be referred to herein as a data forwarding component, is a software process that sends data from the data source 202 to the indexer 232. The forwarder 226 can be implemented using program code that can be executed on the computer device that provides the data source 202. A user launches the program code for the forwarder 226 on the computing device that provides the data source 202. The user can further configure the forwarder 226, for example to specify a receiver for the data being forwarded (e.g., one or more indexers, another forwarder, and/or another recipient system), to enable or disable data forwarding, and to specify a file, directory, network events, operating system data, or other data to forward, among other operations.

[0041] The forwarder 226 can provide various capabilities. For example, the forwarder 226 can send the data unprocessed or can perform minimal processing on the data before sending the data to the indexer 232. Minimal processing can include, for example, adding metadata tags to the data to identify a source, source type, and/or host, among other information, dividing the data into blocks, and/or applying a timestamp to the data. In some implementations, the forwarder 226 can break the data into individual events (event generation is discussed further below) and send the events to a receiver. Other operations that the forwarder 226 may be configured to perform include buffering data, compressing data, and using secure protocols for sending the data, for example.

[0042] Forwarders can be configured in various topologies. For example, multiple forwarders can send data to the same indexer. As another example, a forwarder can be configured to filter and/or route events to specific receivers (e.g., different indexers), and/or discard events. As another example, a forwarder can be configured to send data to another forwarder, or to a receiver that is not an indexer or a forwarder (such as, for example, a log aggregator).

[0043] The event collector 230 provides an alternate method for obtaining data from the data source 202. The event collector 230 enables data and application events to be sent to the indexer 232 using HTTP 228. The event collector 230 can be implemented using program code that can be executed on a computing device. The program code may be a component of the data intake and query system or can be a standalone component that can be executed independently of the data intake and query system and operates in cooperation with the data intake and query system.

[0044] To use the event collector 230, a user can, for example using the monitoring console 216 or a similar interface provided by the user interface system 214, enable the event collector 230 and configure an authentication token. In this context, an authentication token is a piece of digital data generated by a computing device, such as a server, that contains information to identify a particular entity, such as a user or a computing device, to the server. The token will contain identification information for the entity (e.g., an alphanumeric string that is unique to each token) and a code that authenticates the entity with the

server. The token can be used, for example, by the data source **202** as an alternative method to using a username and password for authentication.

**[0045]** To send data to the event collector **230**, the data source **202** is supplied with a token and can then send HTTP **228** requests to the event collector **230**. To send HTTP **228** requests, the data source **202** can be configured to use an HTTP client and/or to use logging libraries such as those supplied by Java, JavaScript, and .NET libraries. An HTTP client enables the data source **202** to send data to the event collector **230** by supplying the data, and a Uniform Resource Identifier (URI) for the event collector **230** to the HTTP client. The HTTP client then handles establishing a connection with the event collector **230**, transmitting a request containing the data, closing the connection, and receiving an acknowledgment if the event collector **230** sends one. Logging libraries enable HTTP **228** requests to the event collector **230** to be generated directly by the data source. For example, an application can include or link a logging library, and through functionality provided by the logging library manage establishing a connection with the event collector **230**, transmitting a request, and receiving an acknowledgment.

**[0046]** An HTTP **228** request to the event collector **230** can contain a token, a channel identifier, event metadata, and/or event data. The token authenticates the request with the event collector **230**. The channel identifier, if available in the indexing system **220**, enables the event collector **230** to segregate and maintain separate data from different data sources. The event metadata can include one or more key-value pairs that describe the data source **202** or the event data included in the request. For example, the event metadata can include key-value pairs specifying a timestamp, a hostname, a source, a source type, or an index where the event data should be indexed. The event data can be a structured data object, such as a JavaScript Object Notation (JSON) object, or raw text. The structured data object can include both event data and event metadata. Additionally, one request can include event data for one or more events.

**[0047]** In some implementations, the event collector **230** extracts events from HTTP **228** requests and sends the events to the indexer **232**. The event collector **230** can further be configured to send events to one or more indexers. Extracting the events can include associating any metadata in a request with the event or events included in the request. In these implementations, event generation by the indexer **232** (discussed further below) is bypassed, and the indexer **232** moves the events directly to indexing. In some implementations, the event collector **230** extracts event data from a request and outputs the event data to the indexer **232**, and the indexer generates events from the event data. In some implementations, the event collector **230** sends an acknowledgement message to the data source **202** to indicate that the event collector **230** has received a particular request from the data source **202**, and/or to indicate to the data source **202** that events in the request have been added to an index.

**[0048]** The indexer **232** ingests incoming data and transforms the data into searchable knowledge in the form of events. In the data intake and query system, an event is a single piece of data that represents activity of the component represented in FIG. 2 by the data source **202**. An event can be, for example, a single record in a log file that records a single action performed by the component (e.g., a user login, a disk read, transmission of a network packet, etc.). An event

includes one or more fields that together describe the action captured by the event, where a field is a key-value pair (also referred to as a name-value pair). In some cases, an event includes both the key and the value, and in some cases the event includes only the value and the key can be inferred or assumed.

**[0049]** Transformation of data into events can include event generation and event indexing. Event generation includes identifying each discrete piece of data that represents one event and associating each event with a timestamp and possibly other information (which may be referred to herein as metadata). Event indexing includes storing of each event in the data structure of an index. As an example, the indexer **232** can include a parsing module **234** and an indexing module **236** for generating and storing the events. The parsing module **234** and indexing module **236** can be modular and pipelined, such that one component can be operating on a first set of data while the second component is simultaneously operating on a second set of data. Additionally, the indexer **232** may at any time have multiple instances of the parsing module **234** and indexing module **236**, with each set of instances configured to simultaneously operate on data from the same data source or from different data sources. The parsing module **234** and indexing module **236** are illustrated in FIG. 2 to facilitate discussion, with the understanding that implementations with other components are possible to achieve the same functionality.

**[0050]** The parsing module **234** determines information about incoming event data, where the information can be used to identify events within the event data. For example, the parsing module **234** can associate a source type with the event data. A source type identifies the data source **202** and describes a possible data structure of event data produced by the data source **202**. For example, the source type can indicate which fields to expect in events generated at the data source **202** and the keys for the values in the fields, and possibly other information such as sizes of fields, an order of the fields, a field separator, and so on. The source type of the data source **202** can be specified when the data source **202** is configured as a source of event data. Alternatively, the parsing module **234** can determine the source type from the event data, for example from an event field in the event data or using machine learning techniques applied to the event data.

**[0051]** Other information that the parsing module **234** can determine include timestamps. In some cases, an event includes a timestamp as a field, and the timestamp indicates a point in time when the action represented by the event occurred or was recorded by the data source **202** as event data. In these cases, the parsing module **234** may be able to determine from the source type associated with the event data that the timestamps can be extracted from the events themselves. In some cases, an event does not include a timestamp and the parsing module **234** determines a timestamp for the event, for example from a name associated with the event data from the data source **202** (e.g., a file name when the event data is in the form of a file) or a time associated with the event data (e.g., a file modification time). As another example, when the parsing module **234** is not able to determine a timestamp from the event data, the parsing module **234** may use the time at which it is indexing the event data. As another example, the parsing module **234** can use a user-configured rule to determine the timestamps to associate with events.



[0052] The parsing module 234 can further determine event boundaries. In some cases, a single line (e.g., a sequence of characters ending with a line termination) in event data represents one event while in other cases, a single line represents multiple events. In yet other cases, one event may span multiple lines within the event data. The parsing module 234 may be able to determine event boundaries from the source type associated with the event data, for example from a data structure indicated by the source type. In some implementations, a user can configure rules the parsing module 234 can use to identify event boundaries.

[0053] The parsing module 234 can further extract data from events and possibly also perform transformations on the events. For example, the parsing module 234 can extract a set of fields (key-value pairs) for each event, such as a host or hostname, source or source name, and/or source type. The parsing module 234 may extract certain fields by default or based on a user configuration. Alternatively or additionally, the parsing module 234 may add fields to events, such as a source type or a user-configured field. As another example of a transformation, the parsing module 234 can anonymize fields in events to mask sensitive information, such as social security numbers or account numbers. Anonymizing fields can include changing or replacing values of specific fields. The parsing component 234 can further perform user-configured transformations.

[0054] The parsing module 234 outputs the results of processing incoming event data to the indexing module 236, which performs event segmentation and builds index data structures.

[0055] Event segmentation identifies searchable segments, which may alternatively be referred to as searchable terms or keywords, which can be used by the search system of the data intake and query system to search the event data. A searchable segment may be a part of a field in an event or an entire field. The indexer 232 can be configured to identify searchable segments that are parts of fields, searchable segments that are entire fields, or both. The parsing module 234 organizes the searchable segments into a lexicon or dictionary for the event data, with the lexicon including each searchable segment (e.g., the field “src=10.10.1.1”) and a reference to the location of each occurrence of the searchable segment within the event data (e.g., the location within the event data of each occurrence of “src=10.10.1.1”). As discussed further below, the search system can use the lexicon, which is stored in an index file 246, to find event data that matches a search query. In some implementations, segmentation can alternatively be performed by the forwarder 226. Segmentation can also be disabled, in which case the indexer 232 will not build a lexicon for the event data. When segmentation is disabled, the search system searches the event data directly.

[0056] Building index data structures generates the index 238. The index 238 is a storage data structure on a storage device (e.g., a disk drive or other physical device for storing digital data). The storage device may be a component of the computing device on which the indexer 232 is operating (referred to herein as local storage) or may be a component of a different computing device (referred to herein as remote storage) that the indexer 238 has access to over a network. The indexer 232 can manage more than one index and can manage indexes of different types. For example, the indexer 232 can manage event indexes, which impose minimal structure on stored data and can accommodate any type of

data. As another example, the indexer 232 can manage metrics indexes, which use a highly structured format to handle the higher volume and lower latency demands associated with metrics data.

[0057] The indexing module 236 organizes files in the index 238 in directories referred to as buckets. The files in a bucket 244 can include raw data files, index files, and possibly also other metadata files. As used herein, “raw data” means data as when the data was produced by the data source 202, without alteration to the format or content. As noted previously, the parsing component 234 may add fields to event data and/or perform transformations on fields in the event data. Event data that has been altered in this way is referred to herein as enriched data. A raw data file 248 can include enriched data, in addition to or instead of raw data. The raw data file 248 may be compressed to reduce disk usage. An index file 246, which may also be referred to herein as a “time-series index” or tsidx file, contains metadata that the indexer 232 can use to search a corresponding raw data file 248. As noted above, the metadata in the index file 246 includes a lexicon of the event data, which associates each unique keyword in the event data with a reference to the location of event data within the raw data file 248. The keyword data in the index file 246 may also be referred to as an inverted index. In various implementations, the data intake and query system can use index files for other purposes, such as to store data summarizations that can be used to accelerate searches.

[0058] A bucket 244 includes event data for a particular range of time. The indexing module 236 arranges buckets in the index 238 according to the age of the buckets, such that buckets for more recent ranges of time are stored in short-term storage 240 and buckets for less recent ranges of time are stored in long-term storage 242. Short-term storage 240 may be faster to access while long-term storage 242 may be slower to access. Buckets may be moved from short-term storage 240 to long-term storage 242 according to a configurable data retention policy, which can indicate at what point in time a bucket is old enough to be moved.

[0059] A bucket’s location in short-term storage 240 or long-term storage 242 can also be indicated by the bucket’s status. As an example, a bucket’s status can be “hot,” “warm,” “cold,” “frozen,” or “thawed.” In this example, hot bucket is one to which the indexer 232 is writing data and the bucket becomes a warm bucket when the index 232 stops writing data to it. In this example, both hot and warm buckets reside in short-term storage 240. Continuing this example, when a warm bucket is moved to long-term storage 242, the bucket becomes a cold bucket. A cold bucket can become a frozen bucket after a period of time, at which point the bucket may be deleted or archived. An archived bucket cannot be searched. When an archived bucket is retrieved for searching, the bucket becomes thawed and can then be searched.

[0060] The indexing system 220 can include more than one indexer, where a group of indexers is referred to as an index cluster. The indexers in an index cluster may also be referred to as peer nodes. In an index cluster, the indexers are configured to replicate each other’s data by copying buckets from one indexer to another. The number of copies of a bucket can be configured (e.g., three copies of each buckets must exist within the cluster), and indexers to which buckets are copied may be selected to optimize distribution of data across the cluster.

[0061] A user can view the performance of the indexing system 220 through the monitoring console 216 provided by the user interface system 214. Using the monitoring console 216, the user can configure and monitor an index cluster, and see information such as disk usage by an index, volume usage by an indexer, index and volume size over time, data age, statistics for bucket types, and bucket settings, among other information.

[0062] FIG. 3 is a block diagram illustrating in greater detail an example of the search system 360 of a data intake and query system, such as the data intake and query system 110 of FIG. 1. The search system 360 of FIG. 3 issues a query 366 to a search head 362, which sends the query 366 to a search peer 364. Using a map process 370, the search peer 364 searches the appropriate index 338 for events identified by the query 366 and sends events 378 so identified back to the search head 362. Using a reduce process 382, the search head 362 processes the events 378 and produces results 368 to respond to the query 366. The results 368 can provide useful insights about the data stored in the index 338. These insights can aid in the administration of information technology systems, in security analysis of information technology systems, and/or in analysis of the development environment provided by information technology systems.

[0063] The query 366 that initiates a search is produced by a search and reporting app 316 that is available through the user interface system 314 of the data intake and query system. Using a network access application 306 executing on a computing device 304, a user can input the query 366 into a search field provided by the search and reporting app 316. Alternatively or additionally, the search and reporting app 316 can include pre-configured queries or stored queries that can be activated by the user. In some cases, the search and reporting app 316 initiates the query 366 when the user enters the query 366. In these cases, the query 366 maybe referred to as an “ad-hoc” query. In some cases, the search and reporting app 316 initiates the query 366 based on a schedule. For example, the search and reporting app 316 can be configured to execute the query 366 once per hour, once per day, at a specific time, on a specific date, or at some other time that can be specified by a date, time, and/or frequency. These types of queries maybe referred to as scheduled queries.

[0064] The query 366 is specified using a search processing language. The search processing language includes commands or search terms that the search peer 364 will use to identify events to return in the search results 368. The search processing language can further include commands for filtering events, extracting more information from events, evaluating fields in events, aggregating events, calculating statistics over events, organizing the results, and/or generating charts, graphs, or other visualizations, among other examples. Some search commands may have functions and arguments associated with them, which can, for example, specify how the commands operate on results and which fields to act upon. The search processing language may further include constructs that enable the query 366 to include sequential commands, where a subsequent command may operate on the results of a prior command. As an example, sequential commands may be separated in the query 366 by a vertical line (“|” or “pipe”) symbol.

[0065] In addition to one or more search commands, the query 366 includes a time indicator. The time indicator

limits searching to events that have timestamps described by the time indicator. For example, the time indicator can indicate a specific point in time (e.g., 10:00:00 am today), in which case only events that have the point in time for their timestamp will be searched. As another example, the time indicator can indicate a range of time (e.g., the last 24 hours), in which case only events whose timestamps fall within the range of time will be searched. The time indicator can alternatively indicate all of time, in which case all events will be searched.

[0066] Processing of the search query 366 occurs in two broad phases: a map phase 350 and a reduce phase 352. The map phase 350 takes place across one or more search peers. In the map phase 350, the search peers locate event data that matches the search terms in the search query 366 and sorts the event data into field-value pairs. When the map phase 350 is complete, the search peers send events that they have found to one or more search heads for the reduce phase 352. During the reduce phase 352, the search heads process the events through commands in the search query 366 and aggregate the events to produce the final search results 368.

[0067] A search head, such as the search head 362 illustrated in FIG. 3, is a component of the search system 360 that manages searches. The search head 362, which may also be referred to herein as a search management component, can be implemented using program code that can be executed on a computing device. The program code for the search head 362 can be stored on a non-transitory computer-readable medium and from this medium can be loaded or copied to the memory of a computing device. One or more hardware processors of the computing device can read the program code from the memory and execute the program code in order to implement the operations of the search head 362.

[0068] Upon receiving the search query 366, the search head 362 directs the query 366 to one or more search peers, such as the search peer 364 illustrated in FIG. 3. “Search peer” is an alternate name for “indexer” and a search peer may be largely similar to the indexer described previously. The search peer 364 may be referred to as a “peer node” when the search peer 364 is part of an indexer cluster. The search peer 364, which may also be referred to as a search execution component, can be implemented using program code that can be executed on a computing device. In some implementations, one set of program code implements both the search head 362 and the search peer 364 such that the search head 362 and the search peer 364 form one component. In some implementations, the search head 362 is an independent piece of code that performs searching and no indexing functionality. In these implementations, the search head 362 may be referred to as a dedicated search head.

[0069] The search head 362 may consider multiple criteria when determining whether to send the query 366 to the particular search peer 364. For example, the search system 360 may be configured to include multiple search peers that each have duplicative copies of at least some of the event data and are implanted using different hardware resources. In this example, the sending the search query 366 to more than one search peer allows the search system 360 to distribute the search workload across different hardware resources. As another example, search system 360 may include different search peers for different purposes (e.g., one has an index storing a first type of data or from a first data source while a second has an index storing a second type of data or from a second data source). In this example,

the search query 366 may specify which indexes to search, and the search head 362 will send the query 366 to the search peers that have those indexes.

[0070] To identify events 378 to send back to the search head 362, the search peer 364 performs a map process 370 to obtain event data 374 from the index 338 that is maintained by the search peer 364. During a first phase of the map process 370, the search peer 364 identifies buckets that have events that are described by the time indicator in the search query 366. As noted above, a bucket contains events whose timestamps fall within a particular range of time. For each bucket 344 whose events can be described by the time indicator, during a second phase of the map process 370, the search peer 364 performs a keyword search 374 using search terms specified in the search query 366. The search terms can be one or more of keywords, phrases, fields, Boolean expressions, and/or comparison expressions that in combination describe events being searched for. When segmentation is enabled at index time, the search peer 364 performs the keyword search 372 on the bucket's index file 346. As noted previously, the index file 346 includes a lexicon of the searchable terms in the events stored in the bucket's raw data 348 file. The keyword search 372 searches the lexicon for searchable terms that correspond to one or more of the search terms in the query 366. As also noted above, the lexicon includes, for each searchable term, a reference to each location in the raw data 348 file where the searchable term can be found. Thus, when the keyword search identifies a searchable term in the index file 346 that matches a search term in the query 366, the search peer 364 can use the location references to extract from the raw data 348 file the event data 374 for each event that include the searchable term.

[0071] In cases where segmentation was disabled at index time, the search peer 364 performs the keyword search 372 directly on the raw data 348 file. To search the raw data 348, the search peer 364 may identify searchable segments in events in a similar manner as when the data was indexed. Thus, depending on how the search peer 364 is configured, the search peer 364 may look at event fields and/or parts of event fields to determine whether an event matches the query 366. Any matching events can be added to the event data 374 read from the raw data 348 file. The search peer 364 can further be configured to enable segmentation at search time, so that searching of the index 338 causes the search peer 364 to build a lexicon in the index file 346.

[0072] The event data 374 obtained from the raw data 348 file includes the full text of each event found by the keyword search 372. During a third phase of the map process 370, the search peer 364 performs event processing 376 on the event data 374, with the steps performed being determined by the configuration of the search peer 364 and/or commands in the search query 366. For example, the search peer 364 can be configured to perform field discovery and field extraction. Field discovery is a process by which the search peer 364 identifies and extracts key-value pairs from the events in the event data 374. The search peer 364 can, for example, be configured to automatically extract the first 100 fields (or another number of fields) in the event data 374 that can be identified as key-value pairs. As another example, the search peer 364 can extract any fields explicitly mentioned in the search query 366. The search peer 364 can, alternatively or additionally, be configured with particular field extractions to perform.

[0073] Other examples of steps that can be performed during event processing 376 include: field aliasing (assigning an alternate name to a field); addition of fields from lookups (adding fields from an external source to events based on existing field values in the events); associating event types with events; source type renaming (changing the name of the source type associated with particular events); and tagging (adding one or more strings of text, or a "tags" to particular events), among other examples.

[0074] The search peer 364 sends processed events 378 to the search head 362, which performs a reduce process 380. The reduce process 380 potentially receives events from multiple search peers and performs various results processing 382 steps on the received events. The results processing 382 steps can include, for example, aggregating the events received from different search peers into a single set of events, deduplicating and aggregating fields discovered by different search peers, counting the number of events found, and sorting the events by timestamp (e.g., newest first or oldest first), among other examples. Results processing 382 can further include applying commands from the search query 366 to the events. The query 366 can include, for example, commands for evaluating and/or manipulating fields (e.g., to generate new fields from existing fields or parse fields that have more than one value). As another example, the query 366 can include commands for calculating statistics over the events, such as counts of the occurrences of fields, or sums, averages, ranges, and so on, of field values. As another example, the query 366 can include commands for generating statistical values for purposes of generating charts or graphs of the events.

[0075] The reduce process 380 outputs the events found by the search query 366, as well as information about the events. The search head 362 transmits the events and the information about the events as search results 368, which are received by the search and reporting app 316. The search and reporting app 316 can generate visual interfaces for viewing the search results 368. The search and reporting app 316 can, for example, output visual interfaces for the network access application 306 running on a computing device 304 to generate.

[0076] The visual interfaces can include various visualizations of the search results 368, such as tables, line or area charts, Choropleth maps, or single values. The search and reporting app 316 can organize the visualizations into a dashboard, where the dashboard includes a panel for each visualization. A dashboard can thus include, for example, a panel listing the raw event data for the events in the search results 368, a panel listing fields extracted at index time and/or found through field discovery along with statistics for those fields, and/or a timeline chart indicating how many events occurred at specific points in time (as indicated by the timestamps associated with each event). In various implementations, the search and reporting app 316 can provide one or more default dashboards. Alternatively or additionally, the search and reporting app 316 can include functionality that enables a user to configure custom dashboards.

[0077] The search and reporting app 316 can also enable further investigation into the events in the search results 316. The process of further investigation may be referred to as drilldown. For example, a visualization in a dashboard can include interactive elements, which, when selected, provide options for finding out more about the data being displayed by the interactive elements. To find out more, an interactive

element can, for example, generate a new search that includes some of the data being displayed by the interactive element, and thus may be more focused than the initial search query 366. As another example, an interactive element can launch a different dashboard whose panels include more detailed information about the data that is displayed by the interactive element. Other examples of actions that can be performed by interactive elements in a dashboard include opening a link, playing an audio or video file, or launching another application, among other examples.

#### Bootstrapping Techniques for Performing Cross Region Disaster Recovery

**[0078]** Ingesting data by a data intake and query system includes processing and storing of data produced by data sources (e.g., data source 102 of FIG. 1) associated with a subscriber entity (or tenant) of the data index and query system where the data is transmitted to the data intake and query system using one or more forwarders (e.g., forwarder 226 of FIG. 2). Processing of the data can include parsing the data to identify individual events, where an event, as mentioned previously, is a discrete portion of machine data that can be associated with a timestamp. Processing of the data can further include generating an index of the events, where the index is a data storage structure in which the events are stored and the computing device performing the indexing is referred to as an indexer (e.g., indexer 232 of FIG. 2).

**[0079]** Managing the ingestion and processing of large data volumes becomes particularly complex when the data intake and query system 110 is offered as a service in a public cloud, following a Software as a Service (SaaS) model. This scenario introduces a substantial technical challenge, as any failure—whether triggered by a disaster or malfunction in the cloud’s provisioned resources—potentially exposes the system to the risk of substantial data loss. Effectively addressing this challenge entails the establishment of a seamless transition plan to a secondary region. This contingency plan becomes critical in the face of a significant disaster or malfunction that impacts the primary servers within the primary region of the cloud configuration. The goal is to ensure uninterrupted data operations and maintain the integrity of the service under varying circumstances.

**[0080]** Traditional disaster recovery methods for data intake and query systems often guarantee data availability within a specific region but lack assurances regarding the secure replication of data across multiple regions in the event of a complete regional outage. Additionally, even in cases of intra-region replication, conventional approaches to backing up data from an active deployment to a standby deployment lack speed and efficiency in replicating data and ensuring consistent data synchronization across distinct deployments.

**[0081]** Implementations of a cloud deployment of a data intake and query system are designed to effectively address challenges associated with data loss in the event of a disaster. Deployments of the cloud-based data intake and query system are engineered to effectively address challenges associated with data loss through a robust replication mechanism between an active and a standby deployment, situated either within the same region or in different regions. In addition to replication, the cloud deployment implementations include bootstrapping techniques to ensure consistency

between the replicated data in the standby deployment and the active deployment, minimizing data loss. Bootstrapping is often associated with the process of initiating or starting up a system. In the context of data replication and disaster recovery, bootstrapping involves initializing or establishing a system or data store using existing information. Within the specific context of the cloud deployment of the data intake and query system, bootstrapping comprises using replicated data to bring the standby deployment to a functional state where it can seamlessly take over in case of an active deployment failure.

**[0082]** Implementations of the cloud deployment of the data intake and query system also prioritize comprehensive data replication, aiming to preserve substantially all data during the process. For example, the active deployment maintains redundant data in the pipeline until it receives confirmation that the standby deployment has successfully received and recovered the necessary data. Maintaining redundant data in the pipeline can be necessary sometimes in the event of a disaster or malfunction occurring during the process of replicating data between the active and standby deployment. Waiting for a confirmation from the standby deployment prior to deleting any redundant data ensures accurate bootstrapping of all data from the active deployment to the standby deployment and further ensures consistency while safeguarding against any data loss. In this way, implementations of the cloud deployment of the data intake and query system maintain data integrity and continuity in disaster recovery scenarios.

**[0083]** Implementing a system in accordance with the disclosed techniques that ensures consistency between active and standby deployments offers several key advantages in the realm of data management and disaster recovery. By maintaining synchronized and replicated data between these deployments, the system minimizes the risk of data loss during a disaster scenario. This replication mechanism, coupled with efficient bootstrapping techniques, enables a seamless transition from the active to the standby deployment, ensuring continuity and minimal downtime in the face of unforeseen events. The redundancy provided by this approach enhances overall data integrity and system reliability. Moreover, the ability to initiate the standby deployment with accurate and up-to-date information contributes to a faster and more reliable recovery process. This strategic alignment of active and standby deployments not only safeguards against potential data inconsistencies but also fortifies the system’s resilience, offering a robust foundation for organizations reliant on continuous and secure data availability.

**[0084]** FIG. 4 illustrates a network architecture that enables data replication in the context of disaster recovery for cloud-based data intake and query system, in implementations according to the present disclosure. The architecture 400 of FIG. 4 illustrates a cloud deployment of a data intake and query system 454 (“cloud deployment”) that is operated by a provider entity operating cloud-based network resources. The cloud deployment 454 may be a distributed implementation of the data intake and query system (e.g., a distributed implementation of the data intake and query system 110).

**[0085]** As discussed previously, “cloud” or “in the cloud” refers to a network model in which an entity operates network resources (e.g., processor capacity, network capacity, storage capacity, etc.), located for example in a data

center, and makes those resources available to users and/or other entities over a network. A “public cloud” is a cloud implementation where an entity operates network resources in order to provide them to users that are not associated with the entity and/or to other entities. In this implementation, the provider entity can, for example, allow a subscriber entity to pay for a subscription that enables users associated with the subscriber entity to access a certain amount of the provider entity’s cloud resources, possibly for a limited time. A subscriber entity of cloud resources can also be referred to as a tenant of the provider entity. Users associated with the subscriber entity access the cloud resources over a network, which may include the public Internet. In contrast to an on-prem implementation (which includes computing resources physically located on a client or subscriber entity’s premises), a subscriber entity does not have physical control of the computing devices that are in the cloud, and has digital access to resources provided by the computing devices only to the extent that such access is enabled by the provider entity.

**[0086]** In the illustration of FIG. 4, the provider entity operates the cloud deployment of the data intake and query system **454** and provides the functionality of the data intake and query system (e.g., the data intake and query system **110**) as a service using the cloud, for example under a Software-as-a-Service (SaaS) model, to entities that pay for the user of the service on a subscription basis. In this example, the provider entity can provision a separate tenant (or possibly multiple tenants) in the public cloud network for each subscriber entity, where each tenant executes a separate and distinct instance of the data intake and query system **110**. These entities can, for example, be subscribers or customers who subscribe to the cloud deployment **454**.

**[0087]** As shown in FIG. 4, each tenant can be associated with a respective cloud deployment of a customer stack **452** (referred to herein as a “cloud-based stack”) (e.g., **452A**, **452B** . . . **452N**). For explanatory purposes, multiple instances of like objects are denoted with reference numbers identifying the object and letters identifying the instance, where needed. In some implementations, each cloud-based stack **452** in FIG. 4 executes a separate instance of data intake and query system **110** discussed in connection with FIG. 1. In other words, each cloud-based stack **452** is provisioned in the cloud deployment **454** and is a separate instantiation of a data intake and query system **110** (from FIG. 1) that is associated with a subscriber entity or customer. In some implementations, each subscriber entity may be associated with one or more tenants.

**[0088]** In some implementations, the entity providing the cloud deployment of the data intake and query system **454** is itself subscribing to the cloud services of a cloud service provider **450**. As an example, a first entity provides computing resources under a public cloud service model (e.g., cloud service provider **450**), a second entity subscribes to the cloud services of the first entity and uses the cloud computing resources to operate the cloud deployment of the data intake and query system **454** (e.g., the provider entity of the cloud-based data intake and query system **454** also referred to herein as the “provider entity”), and a third entity can subscribe to the services of the second provider entity in order to use the functionality of the cloud deployment of the data intake and query system **454** (e.g., a subscriber entity or tenant associated with each cloud-based stack **452A**, **452B**, . . . **452N**). In the illustration of FIG. 4, the data sources from

which data is ingested and processed are associated with a tenant (or subscriber entity), users accessing the cloud deployment of the data intake and query system **454** are associated with the tenant, and the analytics and insights provided by the cloud deployment of the data intake and query system **454** are for purposes of the tenant’s operations.

**[0089]** A stack comprises a group of computing devices associated with a subscriber entity that implement a separate instance of the data intake and query system **110** and run independently from computing devices associated with other subscriber entities in the platform operated by the provider entity. A stack can either be a cloud-based stack (e.g., cloud-based stack **452**), or an on-prem stack (not shown in FIG. 4). The disclosed techniques, however, are particularly effective in relation to cloud-based implementations of the data intake and query system where replicating data across regions is critical for avoiding data loss in case of disasters.

**[0090]** Each cloud-based stack **452** (e.g., cloud-based stacks **452A**, . . . **452N**) can include, without limitation, a respective ingestor **404** (e.g., ingestors **404A**, **404B**, . . . **404N**), a respective indexer **408** (e.g., indexers **408A**, **408B**, . . . **408N**), a respective server pod **414** (e.g., server pod **414A**, **414B**, . . . **414N**), and a respective search head **410** (e.g., search head **410A**, **410B**, . . . **410N**). The cloud services provider **450** can include, without limitation, a queue service **406**, a scalable storage module **412** and a metadata database **416**.

**[0091]** As noted above, in some implementations, the provider entity making the cloud deployment of the data intake and query system **454** available is itself subscribing to the cloud services of a cloud service provider **450**. Accordingly, the provider entity that controls the cloud deployment **454** operates network resources (e.g., processor capacity, network capacity, storage capacity, etc.) that are leased from the cloud services provider **450** and located, for example, in a data center controlled by the cloud services provider **450**. The provider entity makes these network resources available to subscriber entities. For example, the provider entity controls the computing devices included within each cloud-based stack (e.g., ingestors **404**, indexers **408**, server pods **414**, search heads **410** and other components associated with the data intake and query system **110** discussed in connection with FIGS. 1-3) and can, for example, allow a subscriber entity to pay for a subscription that enables users associated with subscriber entity to access a certain amount of the provider entity’s cloud resources, including the computing devices within the cloud-based stack for a limited time. The subscriber entity does not have physical control of the computing devices that are in the cloud, and has digital access to resources provided by the computing devices only to the extent that such access is enabled by the provider entity.

**[0092]** In some implementations, where the provider entity of the cloud deployment of the data intake and query system **454** subscribes to the cloud services of a cloud service provider **450**, the provider entity can communicate with the cloud services provider **450** through a network **456**. As noted above, the entire cloud deployment **454** may be provisioned utilizing cloud computing resources offered by or leased from the cloud services provider **450**. Furthermore, the provider entity has the capability to request the cloud services provider **450** to increase (or decrease) the allocation of cloud computing resources for the overall cloud deploy-

ment 454 or for individual stacks 452. For example, if the provider entity determines that a particular cloud-based stack 452 has requested an upgraded subscription or that the workloads associated with the particular cloud stack 452 has increased, the provider entity may communicate with the cloud service provider 450 and request an increase in allocation of computing resources for the particular stack. Alternatively, in some implementations, each stack 452 may be capable of negotiating or requesting an increase in compute resources directly from the cloud services provider 450. For example, if data ingesting and pre-processing requirements associated with a stack 452 increase, the stack 452 can directly communicate with the cloud services provider 450 through the network 456 and request an increase in computational resources. In response, the cloud services provider 450 may increase an allocation in the number of nodes (e.g., indexers, search heads, etc.) made available to the stack 452.

[0093] In some implementations, the queue service 406, the scalable storage module 412 and the metadata database 416 are associated with services provided by the cloud services provider 450. In some implementations, the ingestors 404, indexers 408, server pods 414 and search heads 410 are controlled and operated by the provider entity that subscribes to the cloud services of the cloud services provider 450 and uses the cloud computing resources to operate the cloud deployment of the data intake and query system 454. In some implementations, the indexers 408, search heads 410, the server pods 414, the ingestors 404 and other computing devices needed for each stack can be implemented using program code that can be executed on the computing devices within the stack, as explained previously in connection with FIGS. 1, 2 and 3. In some implementations, the ingestors 404, indexers 408, server pods 414 and search heads 410 can comprise computing instances that allow users to run applications and host various workloads on scalable and resizable virtual servers in the cloud. Additionally, the ingestors 404, indexers 408, server pods 414 and search heads 410 can comprise computing instances executing on physical servers in data centers controlled and operated by the cloud services provider 450. The provider entity can choose from a variety of instance types optimized for different use cases, such as compute-optimized, memory-optimized, storage-optimized, and more.

[0094] In some implementations, the indexers 408 perform substantially the same functions as the indexer 232 discussed in connection with FIG. 2. In some implementations, the search heads 410 perform substantially the same functions as the indexer 362 discussed in connection with FIG. 3. In some implementations, the server pods 414 can coordinate the activities of the indexers 408. As will be further explained in connection with FIG. 5, these server pods 414 are capable of monitoring the activation and inclusion of one or more indexers 408 into a stack. Similarly, they can detect instances when one or more indexers 408 become inactive or are deactivated. In some implementations, the ingestors 404 can perform processing on the event data received from client devices 436 (e.g., client devices 436A, 436B, . . . 436C). For example, the ingestors 404 can parse the events, merge a set of events and transform the set of events into file blocks that can be uploaded to the scalable storage module 412. When a file block is uploaded to the scalable storage module 412, the ingestors 404 will also transmit the location of the file block (e.g., using a pointer)

to be stored on the queue service 406 as will be discussed further in connection with FIG. 5.

[0095] In some implementations, the queue service 406 is provided by the cloud services provider to allow different components of a distributed system to communicate by sending and receiving messages. For example, among other things, the queue service 406 can be used to store a pointer to an ingested file block in the scalable storage module 412. Any other components in the system needing access to the file block can retrieve the pointer from the queue service 406. In some implementations, the scalable storage module 412 is a storage service provided by the cloud services provider 450 designed to store and retrieve large amounts of data in a highly scalable and durable manner. In some implementations, objects (e.g., ingested file blocks) are stored in the scalable storage module 412 in containers and each object has a unique key within a container. In some implementations, the metadata database 416 comprises a relational database engine designed to deliver high performance, scalability, and availability. The metadata database 416 is used by the server pods 414 to store metadata corresponding to the indexers 408, as will be further explained in connection with FIG. 5.

[0096] In some implementations, each client device 436 (e.g., client devices 436A, 436B, . . . 436N) can communicate with a respective stack 452 (e.g., 452A, 452B, . . . 452N) over a respective network 432 (e.g., 432A, 432B, . . . 432N) through a user interface system. Secure communications between a client device 436 (and other data sources associated with a subscriber entity) and a cloud-based stack 452 are enabled via the network 432 in accordance with example implementations. In various implementations, the network 432A . . . 432N may be portions of the same network or may include one or more shared network components that communicate with each of the networks 432A . . . 432N. In various implementations, the cloud deployment 454 can communicate with a network interface 432 of a given client device 436 through use of a mobile gateway that facilitates communication between devices behind a firewall, or set of firewalls.

[0097] In some implementations, the user interface system in each client device 436 may perform substantially the same functions as the user interface system 114 of the data intake and query system 110 discussed in connection with FIG. 1. In some implementations, in addition to the client devices 436 interfacing with a respective stack 452 directly, data produced by data sources associated with each tenant (not shown in FIG. 4) can also be processed by one or more forwarders 402 (e.g., forwarders 402A, 402B, . . . 402N) and communicated through the respective network 432 to a respective stack 452. As indicated in the discussion associated with FIG. 2, a forwarder is a software process that ingests and forwards data from a data source to one or more of the ingestors 404. The forwarder 402 can be implemented using program code that can be executed on the computer device that provides the data source. A forwarder 402 can be configured, among other things, to specify a receiver for the data being forwarded (e.g., one or more ingestors, one or more indexers, another forwarder, and/or another recipient system), to enable or disable data forwarding, and to specify a file, directory, network events, operating system data, or other data to forward, among other operations. The forwarder 402 can provide various capabilities. For example, the forwarder 402 can send the data unprocessed or can

perform minimal processing on the data before sending the data to an ingestor **404**. Minimal processing can include, for example, adding metadata tags to the data to identify a source, source type, and/or host, among other information, dividing the data into blocks, and/or applying a timestamp to the data. In some implementations, the forwarder **402** can break the data into individual events and send the events to a receiver, such as the ingestor **404**. Other operations that the forwarder **402** may be configured to perform include buffering data, compressing data, and using secure protocols for sending the data, for example.

[0098] As noted previously, a data source (e.g., data source **102**) is a component of a computing device that produces machine data. Machine data is electronically generated data that is output by the component of the computing device and reflects activity of the component. The component can be a hardware component (e.g., a microprocessor or a network adapter, among other examples) or a software component (e.g., a part of the operating system or an application, among other examples). The component can be a virtual component, such as a virtual machine, a virtual machine monitor (also referred to as a hypervisor), a container, or a container orchestrator, among other examples. Examples of computing devices that can provide the data source include personal computers (e.g., laptops, desktop computers, etc.), handheld devices (e.g., smart phones, tablet computers, etc.), servers (e.g., network servers, compute servers, storage servers, domain name servers, web servers, etc.), network infrastructure devices (e.g., routers, switches, firewalls, etc.), and “Internet of Things” devices (e.g., vehicles, home appliances, factory equipment, etc.), among other examples. Accordingly, in some implementations, client device **416** may in itself be a data source that produces data ingested by a respective stack **406**.

[0099] Using the client device **436**, a user can perform various operations with respect to the corresponding cloud-based stack **452**, such as administration of the data intake and query system instance comprised within the cloud-based stack **452**, management and generation of “knowledge objects,” (user-defined entities for enriching data, such as saved searches, event types, tags, field extractions, lookups, reports, alerts, data models, workflow actions, and fields), initiating of searches, and generation of reports, among other operations. In some implementations, the client device **436** can be executing a network access application, such as a web browser or a terminal, through which a user can access a monitoring console provided by the user interface system. The monitoring console performs substantially the same functions as monitoring console **216** discussed in connection with FIG. 2.

[0100] In some implementations, using the network access application (e.g., network access application **206** of FIG. 2) installed on the client device **436**, a subscriber entity or client can input a query into a search field in a GUI of the user interface system (e.g., user interface system **314** in FIG. 3). This GUI can be made available through an application installed on the client device **436** (e.g., search and reporting app **316** also discussed in connection with FIG. 3). Alternatively, or additionally, the search and reporting app can include pre-configured queries or stored queries that can be activated by the user. In some cases, the search and reporting app initiates the query when the user enters the query. In these cases, the query maybe referred to as an “ad-hoc” query. In some cases, the search and reporting app initiates

the query based on a schedule. For example, the search and reporting app can be configured to execute the query once per hour, once per day, at a specific time, on a specific date, or at some other time that can be specified by a date, time, and/or frequency. These types of queries maybe referred to as “scheduled queries.”

#### Data Ingestion and Indexing in an Active Deployment With Failover Mechanisms

[0101] FIG. 5 illustrates a cloud-based network architecture that enables data ingestion and indexing of event data in an active deployment, in implementations according to the present disclosure. In some implementations, forwarders **502** (e.g., including forwarders **502A**, **502B**, . . . **502N**), typically located on a subscribing entity’s premises and included in the subscriber entity’s “on the premises” implementation (or “on-prem” implementation), forwards data from a data source (e.g., data source **102** in FIG. 1) to the active deployment **500**.

[0102] In some implementations, the active deployment **500** comprises a cloud-based deployment of a data intake and query system and includes, without limitation, one or more ingestors **504** (e.g., ingestors **504A**, **504B** . . . **504N**), a queue service **506**, a scalable storage module **512**, one or more indexers **508** (e.g., indexers **508A**, **508B** . . . **508N**), one or more server pods **514** (e.g., server pods **514A**, **514B**, etc.), one or more search heads **510** and a metadata database **514**. As discussed in connection with FIG. 4, not all the components of the active deployment **500** are controlled by the same entity. For example, the ingestors **504**, the indexers **508**, the server pods **514** and the search heads **510** can be controlled by the provider entity and executed on physical servers leased from a cloud services provider (e.g., cloud services provider **450** in FIG. 4). Meanwhile, the queue service **506**, the scalable storage module **512** and the metadata database **514** can be controlled by the cloud services provider. Note that the one or more ingestors **504** (e.g., ingestors **504A**, **504B** . . . **504N**), the queue service **506**, the scalable storage module **512**, the one or more indexers **508** (e.g., indexers **508A**, **508B** . . . **508N**), the one or more server pods **514** (e.g., server pods **514A**, **514B**, etc.), the one or more search heads **510** and the metadata database **514** perform substantially the same functions as corresponding components illustrated and discussed above in connection with FIG. 4. In some implementations, the active deployment **500** corresponds to a single cloud-based stack **452** illustrated in FIG. 4. However, the active deployment **500** is not limited to a single stack **452** and can also represent cloud deployments of data intake and query systems corresponding to multiple stacks **452**.

[0103] In some implementations, data from the forwarders **502** is transmitted to the ingestors **504** within the active deployment **500**. The ingestors **504** process the incoming data by parsing, merging, and transforming events into distinct file blocks, subsequently uploading these file blocks into the scalable storage module **512**. Concurrently, the ingestors **504** initiate the insertion of corresponding metadata messages into the queue service **506** for each file block. These metadata messages include essential information specifying the location of the respective file block within the scalable storage module **512**. For example, if the forwarders **502** forward two hundred events to the ingestors **504**, the ingestors **504** can ingest the two hundred events and create two exemplary file blocks, file block A and file block B, each



with a hundred events. Each file block is uploaded to the scalable storage module 512 (e.g., to an ingested files directory) and a metadata message containing a location reference or pointer is added to the queue service 506 for each of the two file blocks, file block A and file block B.

[0104] In some implementations, each instance of creating and uploading a file block to the scalable storage module 512, along with the insertion of a corresponding message into the queue service 506, prompts the responsible ingestor 504 to transmit an acknowledgment message to one or more of the forwarders 502. Upon receipt of this acknowledgment from the ingestor 504, the forwarder 502 concludes the process by removing the events associated with the acknowledged file block from the forwarder's memory. It is worth noting that, for failover purposes, the forwarder 502 must retain copies of the events in the forwarder's memory until the acknowledgment is received. Post-acknowledgment, the forwarder 502 can continue to process other events from client devices.

[0105] In some implementations, the indexers 508 periodically check the queue service 506 to determine if any messages pertaining to new file blocks have been added to the queue service 506. In some implementations, the indexers 508 can poll the queue service 506 continuously to check for new messages. When an indexer 508 finds a new message, the indexer 508 determines if there is a location pointer in the new message for a file block residing on the scalable storage module 512. If a location pointer to the scalable storage module 512 is found inside a message, the indexer 508 downloads the corresponding file block from the scalable storage module 512 and indexes the events in the file block. An indexed file block is stored as a "hot bucket slice" at the indexer and also uploaded to the scalable storage module 512.

[0106] As discussed previously, the indexers 508 (which perform substantially the same function as indexing module 236 discussed in connection with FIG. 2) organize files in an index (e.g., the index 238 referred to in FIG. 2) in directories referred to as buckets. The files in a bucket (e.g., bucket 244 of FIG. 2) can include raw data files, index files, and possibly also other metadata files. A bucket's location in short-term storage (e.g., short-term storage 240 in FIG. 2) or long-term storage (e.g., long-term storage 242 in FIG. 2) can also be indicated by the bucket's status. As an example, a bucket's status can be "hot," "warm," "cold," "frozen," or "thawed." For example, a bucket can be considered "hot" if an indexer 508 is writing data to the bucket and the bucket becomes a warm bucket when the index 508 stops writing data to it. In some implementations, both hot and warm buckets can reside in short-term storage (e.g., short-term storage 240 in FIG. 2). When a warm bucket is moved to long-term storage (e.g., long-term storage 242 in FIG. 2), the bucket becomes a cold bucket. A cold bucket can become a frozen bucket after a period of time, at which point the bucket may be deleted or archived. An archived bucket cannot be searched. When an archived bucket is retrieved for searching, the bucket becomes thawed and can then be searched.

[0107] In some implementations, one or more search heads 510 are communicatively coupled with one or more indexers 508 and send queries the search heads 510 receive (e.g., from a client device or otherwise) to the indexers (or search peers) 508. As discussed in connection with FIG. 3, using a map process, the search peer 508 searches the

appropriate index for events identified by the query and sends events so identified back to the search head 508. Using a reduce process, the search head 508 processes the events and produces results to respond to the query. The results can provide useful insights about the data stored in the indexers 508. These insights can aid in the administration of information technology systems, in security analysis of information technology systems, and/or in analysis of the development environment provided by information technology systems.

[0108] Continuing the discussion from above, each file block can be indexed by an indexer 508 to create a hot bucket slice that is stored at the indexer 508 and also uploaded to the scalable storage module 512 (e.g., to an indexer file directory). Uploading the hot bucket slice to the scalable storage module 512 prevents a loss of the hot bucket slice in the event that the indexer fails. Subsequent to uploading the hot bucket slices to the scalable storage module 512, the messages corresponding to the file blocks associated with the hot bucket slices can be deleted from the queue service 506 and, further, the ingested file blocks can also be deleted from the scalable storage module 512. Deleting the ingested blocks from the scalable storage module 512 and the corresponding messages from the queue service is copacetic from a failover perspective because the indexed hot bucket slices corresponding to the file blocks have been uploaded to the scalable storage module 512.

[0109] Referring to the example with file blocks A and B from above, an indexer 508 can poll the queue service 506 to discover messages associated with file blocks A and B. Using the location pointers for the two file blocks, the indexer 508 can download the file blocks from the scalable storage module 512 and index the events in the file blocks. For each file block, the indexer 508 can create a separate hot bucket slice and store the hot bucket slices in a hot bucket directory at the indexer 508 and also upload the hot bucket slices to a dedicated hot bucket directory or indexer file directory stored at the scalable storage module 512. Once, the hot bucket slices are uploaded to the scalable storage module 512, the messages corresponding to the file blocks A and B can be deleted from the queue service 506 and the scalable storage module 512.

[0110] Note that because the hot bucket slices corresponding to file blocks A and B have already been uploaded to the scalable storage module 512, the prior stage's data (corresponding to data in the queue service 506 and the ingested file blocks stored by the scalable storage module 512) can be safely deleted without losing the data. Further note that at each stage, once the current stage has backed up the data to the scalable storage module 512, the prior stage's data can be deleted. For example, as discussed earlier, each instance of creating and uploading a file block to the scalable storage module 512, along with the insertion of a corresponding message into the queue service 506, prompts the responsible ingestor 504 to transmit an acknowledgment message to one or more of the forwarders 502, which then deletes the events associated with the acknowledged file block from the forwarder's memory. Similarly, once hot bucket slices have been uploaded to the scalable storage module 512, the prior stage's data (corresponding to data in the queue service 506 and the ingested file blocks backed up to the scalable storage module 512) can be safely deleted without losing the data.

[0111] After a threshold number of hot bucket slices associated with a given bucket directory has been created by



one or more indexers 508, a hot bucket is created by the one or more indexers 508. The hot bucket, among other things, comprises a concatenation of the hot bucket slices, commonly known as a “raw data journal.” Further, because no new data can be added to the created hot bucket, the hot bucket is rolled over to a warm bucket status. Note that newly indexed data lands in a hot bucket, where it is actively read and written, but when a predetermined number of hot bucket slices is reached, or when the size of the data in the hot buckets exceeds the specified threshold, the hot bucket is rolled to a warm bucket. Warm buckets reside on the same tier of storage as hot buckets. Note that the warm bucket comprises the raw data journal (i.e., the concatenated hot bucket slices), and also includes index files (e.g., time-series index files or tsidx files, etc.), search related files, and possibly also other metadata files. A time-series index file associates each unique keyword in the data with location references to events, which are stored in a companion raw data file (e.g., raw data files associated with the hot bucket slices). The search related files can be used to support or expedite searching of the indexed data in the raw data journal. The warm bucket is thereafter uploaded by the one or more indexers 508 to the scalable storage module 512. Note that in some implementations, the warm bucket is uploaded to a dedicated directory for buckets on the scalable storage module 512 that is different from the directory in which the hot bucket slices are uploaded.

[0112] As previously highlighted, in the FIG. 5 implementation, data linked to a preceding stage undergoes deletion only after the current stage has completed processing and successfully uploaded the data to the scalable storage module 512. This design ensures the prevention of data loss through the incorporation of failover mechanisms. Referring back to the example above, consider a scenario where an ingestor 504 encounters a failure during or immediately after processing the two hundred events linked to file block A and file block B, but before transmitting an acknowledgment to the forwarders 502. As emphasized earlier, the forwarder 502 adheres to a protocol wherein the forwarder 502 retains these events in memory until receiving an acknowledgment from the ingestor 504. In the event of an ingestor 504 failure before acknowledgment transmission, the forwarder initiates another attempt to convey the two hundred events to an alternate ingestor 504. In essence, the forwarder 502 is programmed to persistently re-transmit the events until the file blocks associated with the events are successfully uploaded to the scalable storage module 512, metadata messages containing reference pointers to these file blocks are deposited into the queue service 506, and an acknowledgment message is received by the forwarder 502. This failover mechanism, characterized by the forwarder’s persistent attempts to transmit events to ingestors 504 until acknowledgment, serves to avert data loss and uphold data integrity.

[0113] Referencing the example above again, consider a situation where an acknowledgment message is successfully sent to the forwarder 502 for the events linked to file blocks A and B. Following this, an indexer 508 (e.g., indexer 508A) examines the queue service 506 for new messages, retrieves messages associated with file blocks A and B, and downloads either file block A, file block B, or both from the scalable storage module 512. However, a failure occurs during the indexing process. As mentioned earlier, messages in the queue service 506 are only deleted after hot bucket

slices corresponding to the file blocks associated with the messages are uploaded to the scalable storage module 512 by one or more indexers 508. Consequently, if an indexer 508 (e.g., indexer 508A) encounters a failure while indexing data related to file blocks A or B, the messages associated with these file blocks persist in the queue service 506. Subsequently, after a designated amount of time has passed, these messages become visible to a different indexer 508 (e.g., indexer 508B) actively polling the queue service 506 or periodically checking for messages. The new indexer 508 (e.g., indexer 508B) can then proceed to index the events, generate hot bucket slices, and upload them to the scalable storage module 512.

[0114] Building upon the earlier example, consider a situation where the newly introduced indexer 508 (e.g., indexer 508B) takes steps to index the events linked to file blocks A and B. The indexer 508 successfully generates hot bucket slices associated with these file blocks and proceeds to upload them to the scalable storage module 512. Further, the indexer 508 (e.g., indexer 508B) takes steps to delete the file blocks A and B from the scalable storage module 512 and also delete messages associated with the file blocks A and B from the queue service 506. However, shortly thereafter, the indexer experiences downtime or failure. In this scenario, the hot bucket slices associated with file blocks A and B remain stored on the scalable storage module 512. However, the indexer 508 (e.g., indexer 508B) responsible for uploading these hot bucket slices to the designated directory on the scalable storage module 512 is currently inactive. As elaborated further below, the server pods 514 identify the inactive status and downtime of the indexer 508 (e.g., indexer 508B) and prompt another indexer (e.g., indexer 508C) to initiate the recovery of the hot bucket slices.

[0115] In some implementations, the server pods 514 orchestrate the actions of the various indexers 508 within the architectural framework. The server pods 514, among their various functionalities, have the ability to monitor instances of indexer downtime, trace the reactivation of one or more indexers 508, monitor the hot buckets or hot bucket directories generated by an indexer 508, and provide instructions to indexers for addressing broken or incomplete buckets arising from indexer downtime. For instance, prior to an indexer 508 initiating the indexing of events in a designated file block, the indexer 508 establishes a directory for the hot bucket slices and informs one or more server pods 514 about the directory creation, along with the corresponding bucket identifier. Thus, the server pods 514 contain information about the presence of directories (or buckets) established by any of the indexers 508, where the directories store hot bucket slices, for example. In some implementations, the server pods 514 store metadata pertaining to the indexers 508 and the associated buckets created by the indexers 508 in the metadata database 516.

[0116] Building further upon the earlier example then, when an indexer 508 (e.g., indexer 508B) fails after uploading the hot bucket slices associated with file blocks A and B to the scalable storage module 512, the server pods 514 react to the indexer 508 (e.g., indexer 508B) failing by notifying a different indexer 508 (e.g., indexer 508C) to repair the hot bucket associated with the hot bucket slices that were created by the inactive indexer (e.g., indexer 508B) prior to failure. Put differently, when an indexer (e.g., indexer 508B) experiences a failure, the server pods 514 detects that the indexer has failed. Due to the information stored at the

server pods **514** about the buckets created by the failed indexer (e.g., indexer **508B**), the server pods **514** can instruct another indexer (e.g., indexer **508C**) to rectify the orphaned bucket resulting from the inactivity of the failed indexer (e.g., indexer **508B**). Because the server pods **514** track the directory or location of the hot bucket slices generated by each of the indexers **508**, the server pods **514** can inform the active indexer **508** (e.g., indexer **508C**) of the location of the incomplete bucket (or directory) that needs to be repaired. As noted above, in some implementations, the directories into which the hot bucket slices are uploaded by the indexers **508** are tracked by the server pods **514** based on bucket identifiers.

[0117] In some implementations, the active indexer **508** (e.g., indexer **508C**) can, thereafter, repair the bucket left incomplete because of the failure of the inactive indexer (e.g., indexer **508B**). Repairing the bucket can comprise concatenating the hot bucket slices already uploaded into a raw data journal, and using the raw data journal to generate the other files needed to build the bucket including the time-series index file, metadata files, and other search related files. The repaired hot bucket is subsequently rolled over to a warm bucket status and uploaded to the scalable storage module **512** by the active indexer (e.g., indexer **508C**). After uploading the warm bucket to the scalable storage module **512**, the active indexer (e.g., indexer **508C**) can delete the hot bucket slices previously uploaded to the scalable storage module **512** by the inactive indexer (e.g., indexer **508B**) and can further notify the server pods **514** that the hot bucket left incomplete due to the failure of the inactive indexer (e.g., indexer **508B**) has been repaired and restored.

[0118] The above example highlights the failover mechanisms embedded at each stage within the architecture discussed in relation to FIG. 5. These mechanisms are designed to ensure the recovery and restoration of data in the event of a failure at any stage. Indeed, data within the architecture depicted in FIG. 5 can be recovered even in a scenario where all ingestors **504** and indexers **508** (e.g., indexers **508A**, **508B**, **508C**) are inadvertently wiped clean during an outage. Imagine a hypothetical situation where, following the creation and upload of ten warm buckets of event data to the scalable storage module **512**, both ingestors **504** and indexers **508** undergo a complete wipeout during an outage. In such a case, the server pods **514** and the metadata database **516** retain information concerning the inactive indexers **508** and the buckets they previously generated. Consequently, if new indexers **508** (e.g., indexers **508D**, **508E**, **508F**, not shown in FIG. 5) are activated, the server pods **514** can guide these new indexers to identify the buckets uploaded to the scalable storage module **512** by the now inactive indexers. The new indexers (e.g., indexers **508D**, **508E**, **508F**, not shown in FIG. 5) can, in some implementations, download the buckets and effectively replace the inactive indexers **508**. Accordingly, when a new search query is transmitted to the new indexers **508** from a search head **510**, the new indexers can use the data in the buckets to facilitate a response to the search query.

[0119] Expanding upon the previous example, envision a situation where, in addition to the inadvertent wipeout of ingestors **504** and indexers **508** (e.g., indexers **508A**, **508B**, **508C**), the metadata database **516** is also unintentionally cleared. In this scenario, aside from the ten warm buckets on the scalable storage module **512**, no other information

regarding the presence of indexed data exists within the architecture of FIG. 5. In certain implementations addressing this circumstance, new indexers (e.g., indexers **508D**, **508E**, **508F**, not shown in FIG. 5) and a new metadata database **516** are activated and brought online. The server pods **514** are programmed to inspect the scalable storage module in the dedicated directory for warm buckets, checking for warm buckets that require bootstrapping or rediscovery. These server pods **514** can identify the ten warm buckets on the scalable storage module **512** in the specified directory, extract associated metadata, and populate the extracted information into the newly activated metadata database **516**. Once the metadata for the warm buckets has been bootstrapped and populated in the new metadata database **516**, the aforementioned procedure can be employed to initiate the new indexers (e.g., indexers **508D**, **508E**, **508F**, not shown in FIG. 5) to make the warm buckets searchable. The example above illustrates that as long as event data is successfully uploaded to the scalable storage module **512**, the data can be recovered even if there is a failure or wipeout of one or most of the other network elements (besides the scalable storage module **512**) illustrated in FIG. 5.

#### Transferring Control From an Active Deployment to a Standby Deployment

[0120] In certain situations, however, there can be widespread outage during which even the scalable storage module **512** is wiped clean. As discussed previously, traditional disaster recovery methods for data intake and query systems often guarantee data availability within a specific region but lack assurances regarding the secure replication of data across multiple regions in the event of a complete regional outage. Additionally, even in cases of intra-region replication, conventional approaches to backing up data from an active deployment to a standby deployment lack speed and efficiency in replicating data and ensuring consistent data synchronization across distinct deployments.

[0121] FIG. 6 illustrates control being transferred from an active deployment of the data intake and query system to a standby deployment, in implementations according to the present disclosure. As shown in FIG. 6, the active deployment **600** includes, without limitation, ingestor(s) **504**, a queue service **506**, indexers **508** (e.g., indexers **508A**, **508B**, **508C**, etc.), scalable storage module **512**, search head(s) **510**, server pods **514** and the metadata database **516**. The active deployment **600** receives data from one or more forwarders **402**. As discussed in connection with the active deployment **500** of FIG. 5, the active deployment **600** can similarly correspond to a single tenant stack (e.g., cloud based stack **452**) or multiple tenant stacks. The active deployment **600** performs substantially the same function as the active deployment **500** and each of the network elements within the active deployment **600** perform substantially the same functions as corresponding components illustrated and discussed above in connection with FIG. 5. The standby deployment **601** includes, without limitation, ingestors **604**, a queue service **606**, indexers **608** (e.g., indexers **608A**, **608B**, **608C**, etc.), scalable storage module **612**, search heads **610**, server pods **614** and the metadata database **616**. Note that the network elements of the standby deployment perform substantially the same functions as corresponding components illustrated and discussed above in connection with FIG. 5.

[0122] The standby deployment 601 can either be in a different region from the active deployment or it can be within the same region. For example, the standby deployment 601 can be in the eastern region of the US while the active deployment 600 can be in the western region of the US. Alternatively, both deployments can be in the eastern or western region.

[0123] In some implementations, data in the scalable storage module 512 is backed up to the scalable storage module 612 in the standby deployment 601 asynchronously. In some implementations, data is backed up over predetermined periodic intervals. In some implementations, data can be backed up whenever new data is uploaded to the scalable storage module 512 in the active deployment 600. As discussed above, data stored in the scalable storage module 512 can include, without limitation, file blocks uploaded by one or more ingestors 504 (with corresponding messages uploaded to the queue service 506), hot bucket slices uploaded by the indexers 508, and warm buckets created from the hot bucket slices that are also uploaded by the indexers 508. Each type of data stored in the scalable storage module 512 can be backed up to the scalable storage module 612 asynchronously. As noted before, once data from each stage is uploaded to the scalable storage module 512, the prior stage's data can be deleted. Accordingly, the architecture minimizes replication both in the active deployment 600, and also during the process of backing up the active deployment to the standby deployment 601.

[0124] By way of example, consider a scenario similar to the example discussed above, where the forwarders 402 forward two hundred events to the ingestors 504. The ingestors ingest the two hundred events and create two exemplary file blocks, file block A and file block B, each with a hundred events. Each file block is uploaded to the scalable storage module 512 (e.g., to an ingested files directory) and a metadata message containing a location reference or pointer is added to the queue service 506 for each of the two file blocks, file block A and file block B. In some implementations, an acknowledgement message is transmitted to the forwarders 402 only after the file blocks are backed up to the scalable storage module 612 in the standby deployment 601 and the metadata messages are backed up to the queue service 606 in the standby deployment 606.

[0125] In alternative implementations, however, it may be desirable to not stall the progress on the part of the forwarders 402 to prevent congestion or data loss on the client side, in which case the acknowledgment can be sent to the forwarders 402 right after the file blocks A and B are uploaded to the scalable storage module 512 but before the file blocks are backed up to the standby deployment 601. In this situation, there is a chance that during the time between sending an acknowledgment message to the forwarders 402 and backing up the file blocks to the standby deployment 601, the active deployment 600 goes down. In such implementations, the architecture is subject to a recovery point objective (RPO), which refers to the maximum acceptable amount of data loss measured in time. It indicates the point in time to which data must be restored in case of a system failure or data loss event. For example, if an organization has an RPO of fifteen minutes, it means that in the event of a failure, the data should be restored to a state no more than fifteen minutes before the failure occurred. In this particular setup, with the assurance that the backup from the active

deployment 600 to the standby deployment 601 is completed within a fifteen-minute timeframe, the Recovery Point Objective (RPO) is set at fifteen minutes. This signifies that should the active deployment 600 experience a crash before data is securely backed up to the standby deployment 601, any data uploaded to the active deployment within that fifteen-minute window can be at risk of loss. However, the design of the architecture ensures that this is the only stage of the deployment that may be subject to a data loss. As will be explained below, implementations of the architecture illustrated in FIG. 6 are configured to avoid data loss at any other stage except for the ingestion stage.

[0126] In some implementations, once the ingested file blocks A and B are backed up to the standby deployment 601, any data loss is prevented even in situations where the entire active deployment 600 experiences an outage. If the active deployment 601 experiences an outage, the control switches to the standby deployment 601, and the standby deployment 601 becomes the new active deployment. For example, forwarders 402 can begin to forward data to the ingestors 604 in the standby deployment 601, which is now the new active deployment and the data ingestion and indexing can continue thereafter using the standby deployment 601.

[0127] In certain implementations, there is a periodic backup process for ingested file blocks on the scalable storage module 512 to be mirrored onto the scalable storage module 612. Subsequently, these mirrored ingested file blocks can undergo bootstrapping to extract metadata. This extracted metadata is then utilized to populate the queue service 606. Consequently, the indexers 608 can retrieve location points from the queue service 606, enabling them to locate the ingested file blocks on the scalable storage module 612 and initiate the indexing process for these file blocks. Continuing the example from above, once the ingested file blocks A and B are backed up to the scalable storage module 612 in the standby deployment 601 (i.e. the new active deployment), the ingestors 604 in the deployment 601 can access the ingested file blocks on the scalable storage module 612 and add corresponding metadata messages associated with reference pointers to the queue service 606. Because the event data is available in the file blocks, the ingestors 604 are able to work backwards and extract the requisite metadata and location information from the ingested file blocks to populate the messages in the queue service 606. In this way, the file blocks on the scalable storage module 612 can be bootstrapped and the information therein be made available for the indexers 608. Once the metadata messages with the corresponding reference locations are available on the queue service 612, the indexers 608 that periodically check or poll the queue service 606 are notified that new file blocks are available for indexing on the scalable storage module 612.

[0128] In certain implementations, a parallel bootstrapping process is applicable to hot bucket slices uploaded to the scalable storage module 512. Hot bucket slices, generated by the indexers 508 during the indexing of one or more file blocks, undergo periodic backup to the standby deployment 601. In the event of an outage at the active deployment 600, the server pods 614 in the standby deployment 601 are notified that the standby deployment 601 is the new active deployment. Thereafter, the server pods 614 initiate a bootstrapping sequence for the hot bucket slices backed up to the scalable storage module 612. This involves instructing one

of the indexers **608** to recover and repair orphan buckets residing on the scalable storage module **612**. Due to the predetermined path at which the hot bucket slices are stored on the scalable storage module **612**, the indexers **608** can efficiently locate and recover these slices. Subsequently, they convert the hot bucket slices into a hot bucket and transition it to a warm status. To illustrate further, imagine a scenario where one or more indexers **508** are actively indexing file blocks A and B, uploading hot bucket slices (yet to transition to warm bucket status) to the scalable storage module **512**. These hot bucket slices are routinely backed up to the standby deployment **601**. Now, envision a circumstance where the active deployment **600** encounters a failure subsequent to the upload of these hot bucket slices by one or more indexers **508**. In response, the server pods **614** instruct an indexer (e.g., indexer **608A**) to access the predetermined path for the hot bucket slices, recover them, and undertake the necessary repairs to the bucket. The indexer **508** completes the repair process, and the bucket is rolled over to a warm status, as previously discussed. As detailed above, repairing the bucket can comprise concatenating the hot bucket slices already uploaded into a raw data journal, and using the raw data journal to generate the other files needed to build the bucket including the time-series index file, metadata files, and other search related files.

[0129] In certain implementations, a similar bootstrapping process is also available to warm buckets uploaded to the scalable storage module **612**. Warm buckets uploaded to the scalable storage module **512** are periodically and asynchronously copied over to the scalable storage module **612** in the standby deployment **601**. If the active deployment **600** experiences an outage, the server pods **614** are notified that the standby deployment **601** is the new active deployment. Thereafter, the server pods **614** initiate a bootstrapping sequence for the warm buckets backed up to the scalable storage module **612**. This involves instructing one of the indexers **608** to bootstrap the warm buckets. Bootstrapping the warm buckets includes discovering the bucket identifiers for the warm bucket and reporting the identifiers and other metadata associated with the buckets back to the server pods **614**. As explained above in connection with the examples from FIG. 5, the server pods **614** can populate the extracted information into the newly activated metadata database **616**. Once the metadata for the warm buckets has been bootstrapped and populated in the new metadata database **616**, the indexers **608** are able to make the data searchable.

[0130] It is important to highlight that the diverse examples discussed in relation to FIG. 5, detailing the recovery of data at each stage during a failure of one or more network components, similarly extend to the implementations delineated in FIG. 6. Specifically, when a standby deployment **601** transitions into the new active deployment and encounters an outage in any of its network components (e.g., ingestors **604**, indexers **608**, metadata database **616**), the identical procedures expounded above remain applicable in this scenario.

[0131] As previously mentioned in the context of FIG. 5, following the processing of data at each stage within the data intake and query system deployment, and subsequent uploading of results into a scalable storage module, the data from the preceding stage, stored in the scalable storage module, can be safely removed. This practice serves to prevent data duplication within the pipeline and contributes to a reduction in the required storage space. When transfer-

ring control from an active deployment **600** to a standby deployment **601**, however, implementations of the cloud-based data intake and query system need to ensure that the data from a current processing stage of the data intake and query system pipeline in an active deployment **600** is replicated and backed up to the standby deployment **601** safely prior to deleting the corresponding data from a prior stage in the pipeline.

[0132] Referencing the example discussed previously, consider a scenario where the ingested file blocks A and B are uploaded to the scalable storage module **512** by the ingestors **504** and two corresponding metadata messages are added to the queue service **506**. Following this, the indexers **508** detect the messages on the queue service **506** and access file blocks A and B from the scalable storage module **512** for indexing purposes. In the subsequent stage, after indexing file block A, an indexer **508** generates and uploads a corresponding hot bucket slice (hot bucket slice corresponding to file block A) to the scalable storage module **512**. In contrast to single deployment implementations (as explored in connection with FIG. 5), where an indexer **508** can promptly delete file block A and the associated message from the queue service **506**, implementations with a standby deployment **601** necessitate waiting. Here, an indexer **508** must wait until the hot bucket slice linked to file block A completes replication to the scalable storage module **612** in the standby deployment **601**. Only after ensuring the replication of the data from the current stage, exemplified by the hot bucket slice corresponding to the ingested file block A, to the standby deployment **601**, can the data from the prior stage—specifically, the ingested file block A uploaded to the scalable storage module **512**—safely be removed from the active deployment **600**.

[0133] Failure to wait for the backup to the standby deployment **601** before deleting data from the prior stage can lead to unintended consequences and potential data loss. Continuing with the aforementioned example, if the ingested file block A is deleted from the scalable storage module **512** prior to the full replication of the corresponding hot bucket slice to the standby deployment **601**, the risk of data loss escalates. In situations where the active deployment **600** encounters an outage before the hot bucket slice fully replicates to the standby deployment **601**, the deletion of the ingested file block A from scalable storage module **512** gets replicated to the scalable storage module **612**. Note that typically, the ingested file block A would also have been replicated to the scalable storage module **612** in the standby deployment **601** at the prior stage shortly after the ingestors **504** uploaded the file block A to scalable storage module **512**. Nevertheless, deleting ingested file block A in the active deployment **600** before replicating the corresponding hot bucket slice introduces a discrepancy because the deletion is replicated to the standby deployment **601**. The deletion of the ingested file block A from scalable storage module **512** outpaces the replication of the corresponding hot bucket slice from scalable storage module **512** to scalable storage module **612**. Consequently, recovery of the data becomes impossible, as the ingested file block A is erased from both deployments, and the hot bucket slice corresponding to file block A is lost during the outage experienced at the active deployment **600**. Accordingly, it is necessary for data from the current stage (exemplified by the hot bucket slice in this example) of the active deployment **600** to be backed up safely to the standby deployment **601** prior to the deletion

of the data from the prior stage from the active deployment **600** (where the deletion is then promptly replicated at the standby deployment **601**).

[0134] By way of further example, consider a situation where that indexer **508** has finished indexing both file blocks A and B, uploaded the hot slices corresponding to both file blocks to the scalable storage module **512** in the active deployment **600** and subsequently rolled over the hot bucket slices to a warm bucket. Similar to the discussion above, prior to deleting the hot bucket slices (that continue to exist in a different directory on the scalable storage module **512** even after a warm bucket has been created and stored at the scalable storage module **512**), the entire warm bucket needs to first be replicated to the scalable storage module **612** of the standby deployment **601**. This ensures that no data is lost if an outage is experienced if the active deployment **600** experiences an outage before backing up the entire warm bucket to the standby deployment **601**.

[0135] FIG. 7 is a flowchart illustrating an example of a computer-implemented method for transferring control from an active deployment of a data intake and query system to a standby deployment, in implementations according to the present disclosure. The example process **700** can be implemented, for example, by a computing device that comprises a processor and a non-transitory computer-readable medium. The non-transitory computer readable medium can be storing instructions that, when executed by the processor, can cause the processor to perform the operations of the illustrated process **700**. Alternatively, or additionally, the process **700** can be implemented using a non-transitory computer-readable medium storing instructions that, when executed by one or more processors, cause the one or more processors to perform the operations of the process **700** of FIG. 7.

[0136] At block **702**, event data is received at an active deployment **600** of a data intake and query system. The data intake and query system can be associated with a single customer stack (e.g., stack **452**). The event data can be transmitted using forwarders **402** located on the premises of a client of the data intake and query system. The event data can comprise raw machine data segmented into events and timestamped.

[0137] At block **704**, the event data is processed in a plurality of stages of the data intake and query system to convert the event data into buckets of indexed data. In some implementations, the files associated with the results of each stage of the processing are uploaded to scalable storage module **512** in the active deployment **600**. To illustrate, file blocks generated by the ingestors **504** undergo uploading to the scalable storage module **512**, while pertinent information, detailing the location of these file blocks on the scalable storage module **512**, is concurrently uploaded to the queue service **506**. Additionally, as an illustrative instance, hot bucket slices resulting from the indexers **508** processing ingested file blocks are uploaded to a designated directory for hot bucket slices on the scalable storage module **512**. Moreover, warm buckets, formed upon reaching a pre-defined number of hot bucket slices that transition to warm bucket status under the indexers **508**, are also uploaded to the scalable storage module **512** and archived in a dedicated directory within the scalable storage module **512**.

[0138] At block **706**, the files associated with the results of each stage are replicated to the scalable storage module **612** in the standby deployment **601**. For example, at each stage

described in connection with block **704** above, whenever files associated with the event data (e.g., ingested file blocks, hot bucket slices, warm buckets, etc.) are uploaded to the scalable storage module **512**, the same files are also copied and backed up to the scalable storage module **612**. For example, after the ingestors **504** upload the ingested file blocks to the scalable storage module **512**, the files associated with the ingested blocks are also backed up to the scalable storage module **612**. Similarly, the files associated with the hot bucket slices and warm buckets are also replicated to the standby deployment **601**.

[0139] At block **708**, responsive to an outage at the active deployment **600**, control is transitioned to the standby deployment **601**. If the active deployment **600** goes down, the standby deployment **601** becomes the new active deployment and the forwarders **402** are alerted so that the forwarders **402** can start forwarding data to the standby deployment **601**.

[0140] At block **710**, the files associated with the results of each stage backed up to the standby deployment **601** are bootstrapped to recover the event data (in whichever format it was in when it was backed up). Further, the data intake and query system in the standby deployment, which is the new active deployment, continues to process the event data into buckets of indexed data that will eventually be uploaded to the scalable storage module **612**. Several examples illustrating the bootstrapping process using the backed-up files have been detailed earlier. In different ways, ingested file blocks, hot bucket slices, and warm buckets can undergo bootstrapping to ensure the operational status of the standby deployment **601**. This enables the continued processing of files, generating warm buckets that are stored on the scalable storage module **612** and can be accessed when responding to search queries from the search heads **610** received at the indexers **608**.

[0141] FIG. 8 is a flowchart illustrating another example of a computer-implemented method for transferring control from an active deployment of a data intake and query system to a standby deployment, in implementations according to the present disclosure. The example process **800** can be implemented, for example, by a computing device that comprises a processor and a non-transitory computer-readable medium. The non-transitory computer readable medium can be storing instructions that, when executed by the processor, can cause the processor to perform the operations of the illustrated process **800**. Alternatively, or additionally, the process **800** can be implemented using a non-transitory computer-readable medium storing instructions that, when executed by one or more processors, cause the one or more processors to perform the operations of the process **800** of FIG. 8.

[0142] At block **802**, event data is received at an active deployment **600** of a data intake and query system. The data intake and query system can be associated with a single customer stack (e.g., stack **452**). The event data can be transmitted using forwarders **402** located on the premises of a client of the data intake and query system. The event data can comprise raw machine data segmented into events and timestamped. As explained above, the event data is processed in a plurality of stages of the data intake and query system to convert the event data into buckets of indexed data.

[0143] At block **804**, the files associated with the results of each stage of the processing are uploaded to scalable storage

module 512 in the active deployment 600. To illustrate, file blocks generated by the ingestors 504 undergo uploading to the scalable storage module 512, while pertinent information, detailing the location of these file blocks on the scalable storage module 512, is concurrently uploaded to the queue service 506. Additionally, as an illustrative instance, hot bucket slices resulting from the indexers 508 processing ingested file blocks are uploaded to a designated directory for hot bucket slices on the scalable storage module 512. Moreover, warm buckets, formed upon reaching a pre-defined number of hot bucket slices that transition to warm bucket status under the indexers 508, are also uploaded to the scalable storage module 512 and archived in a dedicated directory within the scalable storage module 512.

[0144] At block 806, the files associated with the results of each stage are replicated to the scalable storage module 612 in the standby deployment 601. For example, at each stage described in connection with block 704 above, whenever files associated with the event data (e.g., ingested file blocks, hot bucket slices, warm buckets, etc.) are uploaded to the scalable storage module 512, the same files are also copied and backed up to the scalable storage module 612. For example, after the ingestors 504 upload the ingested file blocks to the scalable storage module 512, the files associated with the ingested blocks are also backed up to the scalable storage module 612. Similarly, the files associated with the hot bucket slices and warm buckets are also replicated to the standby deployment 601. In some implementations, responsive to an outage at the active deployment 600, control is transitioned to the standby deployment 601. If the active deployment 600 goes down, the standby deployment 601 becomes the new active deployment and the forwarders 402 are alerted so that the forwarders 402 can start forwarding data to the standby deployment 601. Thereafter, the files associated with the results of each stage backed up to the standby deployment 601 are bootstrapped to recover the event data (in whichever format it was in when it was backed up). Several examples illustrating the bootstrapping process using the backed-up files have been detailed earlier. In different ways, ingested file blocks, hot bucket slices, and warm buckets can undergo bootstrapping to ensure the operational status of the standby deployment 601. This enables the continued processing of files, generating warm buckets that are stored on the scalable storage module 612 and can be accessed when responding to search queries from the search heads 610 received at the indexers 608.

[0145] At step 808, following the backup of files associated with the results at each stage to the standby deployment 601, the corresponding counterparts from a prior stage in the active deployment 600 are purged from the scalable storage module 512. For instance, if hot bucket slices generated by the indexers 508 are replicated to the scalable storage module 612 (within the dedicated directory for hot bucket slices), the ingested file blocks corresponding to those hot bucket slices remain in the active deployment 600 until the files are fully replicated to the scalable storage module 612. This precautionary measure aims to avert potential data losses resulting from deletions executed at the scalable storage module 512 that might propagate to the standby deployment 601 before an outage occurs at the active deployment 600 and before the files associated with the current stage are successfully copied over.

[0146] FIG. 9 illustrates an example of a self-managed network 900 that includes a data intake and query system. “Self-managed” in this instance means that the entity that is operating the self-managed network 900 configures, administers, maintains, and/or operates the data intake and query system using its own compute resources and people. Further, the self-managed network 900 of this example is part of the entity’s on-premise network and comprises a set of compute, memory, and networking resources that are located, for example, within the confines of an entity’s data center. In some implementations, the self-managed network 900 may be substantially similar to the optional on-prem stack 408 of FIG. 4. These resources can include software and hardware resources. The entity can, for example, be a company or enterprise, a school, government entity, or other entity. Since the self-managed network 900 is located within the customer’s on-prem environment, such as in the entity’s data center, the operation and management of the self-managed network 900, including of the resources in the self-managed network 900, is under the control of the entity. For example, administrative personnel of the entity have complete access to and control over the configuration, management, and security of the self-managed network 900 and its resources.

[0147] The self-managed network 900 can execute one or more instances of the data intake and query system. An instance of the data intake and query system may be executed by one or more computing devices that are part of the self-managed network 900. A data intake and query system instance can comprise an indexing system and a search system, where the indexing system includes one or more indexers 920 and the search system includes one or more search heads 960.

[0148] As depicted in FIG. 9, the self-managed network 900 can include one or more data sources 902. Data received from these data sources may be processed by an instance of the data intake and query system within self-managed network 900. The data sources 902 and the data intake and query system instance can be communicatively coupled to each other via a private network 910.

[0149] Users associated with the entity can interact with and avail themselves of the functions performed by a data intake and query system instance using computing devices. As depicted in FIG. 9, a computing device 904 can execute a network access application 906 (e.g., a web browser), that can communicate with the data intake and query system instance and with data sources 902 via the private network 910. Using the computing device 904, a user can perform various operations with respect to the data intake and query system, such as management and administration of the data intake and query system, generation of knowledge objects, and other functions. Results generated from processing performed by the data intake and query system instance may be communicated to the computing device 904 and output to the user via an output system (e.g., a screen) of the computing device 904.

[0150] The self-managed network 900 can also be connected to other networks that are outside the entity’s on-premise environment/network, such as networks outside the entity’s data center. Connectivity to these other external networks is controlled and regulated through one or more layers of security provided by the self-managed network 900. One or more of these security layers can be implemented using firewalls 912. The firewalls 912 form a layer of security around the self-managed network 900 and regu-

late the transmission of traffic from the self-managed network 900 to the other networks and from these other networks to the self-managed network 900.

[0151] Networks external to the self-managed network can include various types of networks including public networks 990, other private networks, and/or cloud networks provided by one or more cloud service providers. An example of a public network 990 is the Internet. In the example depicted in FIG. 9, the self-managed network 900 is connected to a service provider network 992 provided by a cloud service provider via the public network 990.

[0152] In some implementations, resources provided by a cloud service provider may be used to facilitate the configuration and management of resources within the self-managed network 900. For example, configuration and management of a data intake and query system instance in the self-managed network 900 may be facilitated by a software management system 994 operating in the service provider network 992. There are various ways in which the software management system 994 can facilitate the configuration and management of a data intake and query system instance within the self-managed network 900. As one example, the software management system 994 may facilitate the download of software including software updates for the data intake and query system. In this example, the software management system 994 may store information indicative of the versions of the various data intake and query system instances present in the self-managed network 900. When a software patch or upgrade is available for an instance, the software management system 994 may inform the self-managed network 900 of the patch or upgrade. This can be done via messages communicated from the software management system 994 to the self-managed network 900.

[0153] The software management system 994 may also provide simplified ways for the patches and/or upgrades to be downloaded and applied to the self-managed network 900. For example, a message communicated from the software management system 994 to the self-managed network 900 regarding a software upgrade may include a Uniform Resource Identifier (URI) that can be used by a system administrator of the self-managed network 900 to download the upgrade to the self-managed network 900. In this manner, management resources provided by a cloud service provider using the service provider network 992 and which are located outside the self-managed network 900 can be used to facilitate the configuration and management of one or more resources within the entity's on-prem environment. In some implementations, the download of the upgrades and patches may be automated, whereby the software management system 994 is authorized to, upon determining that a patch is applicable to a data intake and query system instance inside the self-managed network 900, automatically communicate the upgrade or patch to self-managed network 900 and cause it to be installed within self-managed network 900.

[0154] Various examples and possible implementations have been described above, which recite certain features and/or functions. Although these examples and implementations have been described in language specific to structural features and/or functions, it is understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or functions described above. Rather, the specific features and functions described above are disclosed as examples of implementing the claims, and

other equivalent features and acts are intended to be within the scope of the claims. Further, any or all of the features and functions described above can be combined with each other, except to the extent it may be otherwise stated above or to the extent that any such embodiments may be incompatible by virtue of their function or structure, as will be apparent to persons of ordinary skill in the art. Unless contrary to physical possibility, it is envisioned that (i) the methods/steps described herein may be performed in any sequence and/or in any combination, and (ii) the components of respective embodiments may be combined in any manner.

[0155] Processing of the various components of systems illustrated herein can be distributed across multiple machines, networks, and other computing resources. Two or more components of a system can be combined into fewer components. Various components of the illustrated systems can be implemented in one or more virtual machines or an isolated execution environment, rather than in dedicated computer hardware systems and/or computing devices. Likewise, the data repositories shown can represent physical and/or logical data storage, including, e.g., storage area networks or other distributed storage systems. Moreover, in some embodiments the connections between the components shown represent possible paths of data flow, rather than actual connections between hardware. While some examples of possible connections are shown, any of the subset of the components shown can communicate with any other subset of components in various implementations.

[0156] Examples have been described with reference to flow chart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products. Each block of the flow chart illustrations and/or block diagrams, and combinations of blocks in the flow chart illustrations and/or block diagrams, may be implemented by computer program instructions. Such instructions may be provided to a processor of a general purpose computer, special purpose computer, specially-equipped computer (e.g., comprising a high-performance database server, a graphics subsystem, etc.) or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor(s) of the computer or other programmable data processing apparatus, create means for implementing the acts specified in the flow chart and/or block diagram block or blocks. These computer program instructions may also be stored in a non-transitory computer-readable memory that can direct a computer or other programmable data processing apparatus to operate in a particular manner, such that the instructions stored in the computer-readable memory produce an article of manufacture including instruction means which implement the acts specified in the flow chart and/or block diagram block or blocks. The computer program instructions may also be loaded to a computing device or other programmable data processing apparatus to cause operations to be performed on the computing device or other programmable apparatus to produce a computer implemented process such that the instructions which execute on the computing device or other programmable apparatus provide steps for implementing the acts specified in the flow chart and/or block diagram block or blocks.

[0157] In some embodiments, certain operations, acts, events, or functions of any of the algorithms described herein can be performed in a different sequence, can be added, merged, or left out altogether (e.g., not all are

necessary for the practice of the algorithms). In certain embodiments, operations, acts, functions, or events can be performed concurrently, e.g., through multi-threaded processing, interrupt processing, or multiple processors or processor cores or on other parallel architectures, rather than sequentially.

1. A computer-implemented method, comprising: receiving event data at an active deployment of a data intake and query system; processing the event data in a plurality of stages of the data intake and query system to convert the event data into searchable buckets of indexed data, wherein files generated at each stage of the processing are uploaded to a first scalable storage module in the active deployment; and replicating the files generated at each stage of the processing to a second scalable storage module in a standby deployment of the data intake and query system.
2. The computer-implemented method of claim 1, further comprising responsive to an outage at the active deployment, transitioning control to the standby deployment.
3. The computer-implemented method of claim 1, further comprising recovering the event data at the standby deployment using the files replicated to the second scalable storage module.
4. The computer-implemented method of claim 1, further comprising: recovering the event data at the standby deployment; and processing the recovered event data at the standby deployment to generate buckets of indexed data searchable at the standby deployment.
5. The computer-implemented method of claim 1, wherein the data intake and query system comprises a cloud deployment of the data intake and query system, and wherein the data intake and query system is associated with one or more stacks, wherein each stack is dedicated to a subscriber of the data intake and query system.
6. The computer-implemented method of claim 1, wherein the data intake and query system is associated with a stack, wherein the stack comprises a group of computing devices dedicated to a subscriber of the data intake and query system, and wherein the group of computing devices comprises at least one ingestor to ingest event data received at the active deployment, a search head communicatively coupled with one or more indexers, wherein the one or more indexers index and search data associated with the subscriber, wherein the search head processes events returned by the one or more indexers in response to a query and produces results to respond to the query.
7. The computer-implemented method of claim 1, wherein the data intake and query system is associated with a group of computing devices, and wherein the group of computing devices comprises at least one queue service, a scalable storage server, at least one server pod and at least one metadata database, wherein the at least one queue service stores messages referencing locations associated with the event data on the scalable storage server, wherein the scalable storage server is included in the first scalable storage module and the second scalable storage module and stores the files generated at each stage of the processing, wherein the at least one server pod coordinates one or more indexers deployed in the active deployment that index the event data, and wherein the metadata database stores meta-

data associated with the indexed event data generated by the one or more indexers and stored on the scalable storage server.

8. The computer-implemented method of claim 1, wherein at least one stage of the plurality of stages comprises: ingesting the event data and transforming the event data into ingested file blocks; generating messages corresponding to the ingested file blocks to be uploaded to a queue, wherein each message contains a reference pointer to a location on the first scalable storage module where a corresponding ingested file block is stored; and uploading the ingested file blocks to the first scalable storage module.
9. The computer-implemented method of claim 1, wherein at least one stage of the plurality of stages comprises: accessing a queue to retrieve a location pointer for an ingested file block associated with the event data; retrieving the ingested file block using the location pointer from the first scalable storage module; indexing event data in the ingested file block; and uploading hot bucket slices comprising indexed event data to the first scalable storage module.
10. The computer-implemented method of claim 1, wherein at least one stage of the plurality of stages comprises: responsive to a determination that a threshold number of hot bucket slices have been uploaded by one of more indexers in the active deployment to the first scalable storage module, concatenating the hot bucket slices; generating a metadata file and a time-series index file associated with the hot bucket slices; consolidating the concatenated hot bucket slices, the metadata file and the time-series index file into a hot bucket; and changing the status of the hot bucket to a warm bucket, wherein the hot bucket remains operable for writing by an indexer, and wherein the warm bucket is no longer open for writing.
11. The computer-implemented method of claim 1, further comprising recovering the event data at the standby deployment, wherein recovering the event data at the standby deployment comprises: accessing an ingested file block from the second scalable storage module; extracting metadata from the ingested file block comprising a location pointer to the ingested file block on the second scalable storage module; and uploading a message to a queue in the standby deployment comprising the metadata.
12. The computer-implemented method of claim 1, further comprising recovering the event data at the standby deployment, wherein recovering the event data at the standby deployment comprises: accessing one or more hot bucket slices from the second scalable storage module; repairing the one or more hot bucket slices; concatenating the hot bucket slices into a journal; generating one or more other files used in a creation of a bucket; and creating a warm bucket using the journal and the one or more other files.



13. The computer-implemented method of claim 1, further comprising recovering the event data at the standby deployment, wherein recovering the event data at the standby deployment comprises:

- accessing one or more warm buckets from the second scalable storage module;
- extracting metadata from the one or more warm buckets;
- populating the metadata in a metadata database; and
- making the warm buckets available for searching by one or more indexers.

14. A computing device, comprising:

- a processor; and

- a non-transitory computer-readable medium having stored thereon instructions that, when executed by the processor, cause the processor to perform operations including:

- receiving event data at an active deployment of a data intake and query system;

- processing the event data in a plurality of stages of the data intake and query system to convert the event data into searchable buckets of indexed data, wherein files generated at each stage of the processing are uploaded to a first scalable storage module in the active deployment; and

- replicating the files generated at each stage of the processing to a second scalable storage module in a standby deployment of the data intake and query system.

15. The computing device of claim 14, wherein the data intake and query system comprises a cloud deployment of the data intake and query system, and wherein the data intake and query system is associated with one or more stacks, wherein each stack is dedicated to a subscriber of the data intake and query system.

16. The computing device of claim 14, wherein the data intake and query system is associated with a stack, wherein the stack comprises a group of computing devices dedicated to a subscriber of the data intake and query system, and wherein the group of computing devices comprises at least one ingestor to ingest event data received at the active deployment, a search head communicatively coupled with one or more indexers, wherein the one or more indexers index and search data associated with the subscriber, wherein the search head processes events returned by the one or more indexers in response to a query and produces results to respond to the query.

17. The computing device of claim 14, wherein the data intake and query system is associated with a group of computing devices, and wherein the group of computing

devices comprises at least one queue service, a scalable storage server, at least one server pod and at least one metadata database, wherein the at least one queue service stores messages referencing locations associated with the event data on the scalable storage server, wherein the scalable storage server is associated with the first scalable storage module and the second scalable storage module and stores the files generated at each stage of the processing, wherein the at least one server pod coordinates one or more indexers deployed in the active deployment that index the event data, and wherein the metadata database stores meta-data associated with the indexed event data generated by the one or more indexers and stored on the scalable storage server.

18. The computing device of claim 14, wherein at least one stage of the plurality of stages comprises:

- ingesting the event data and transforming the event data into ingested file blocks;

- generating messages corresponding to the ingested file blocks to be uploaded to a queue, wherein each message contains a reference pointer to a corresponding ingested file block; and

- uploading the ingested file blocks to a scalable storage module deployed in the active deployment.

19. A non-transitory computer-readable medium having stored thereon instructions that, when executed by one or more processors, cause the one or more processor to perform operations including:

- receiving event data at an active deployment of a data intake and query system;

- processing the event data in a plurality of stages of the data intake and query system to convert the event data into searchable buckets of indexed data, wherein files generated at each stage of the processing are uploaded to a first scalable storage module in the active deployment; and

- replicating the files generated at each stage of the processing to a second scalable storage module in a standby deployment of the data intake and query system.

20. The non-transitory computer-readable medium of claim 19, wherein the data intake and query system comprises a cloud deployment of the data intake and query system, and wherein the data intake and query system is associated with one or more stacks, wherein each stack is dedicated to a subscriber of the data intake and query system.

\* \* \* \* \*