

SELECTING QUERIES FOR EXECUTION ON A STREAM OF REAL-TIME DATA**TECHNICAL FIELD**

The present application relates to computer systems, methods and machine-readable hardware storages to select queries for execution on a stream of data, in particular real-time data. Specifically, implementations relate to dataflow graph-based computations implementing a plurality of active queries to be executed on the stream of data that may originate from a distributed network of data sources.

BACKGROUND

Many networked data processing systems require timely access to critical information for decision making. A traditional approach to meet such requirements has been to build data warehouses, data marts and reporting applications. These approaches have not been able to meet strict service level agreement (SLA) requirements for truly real-time applications, such as fraud detection, service monitoring, gaming and real-time trending, because these approaches are not able to monitor a stream of continuously arriving data, while searching for patterns or conditions.

Additionally, databases and Structured Query Language (SQL) do not have constructs to wait for data arrival. SQL works on historical data that is present in a repository, when the query is fired. This processing of historical data stored in a data warehouse often fails to meet many latency requirements, as it takes time to collect, cleanse and integrate data (commonly known as ETL – Extract Transform Load) in a data warehouse and as it also takes time to start a query for processing of the warehoused data.

SUMMARY

In an implementation, a computer-implemented method for executing a query on data items located at different places in a stream of (e.g., near real-time) data to provide (e.g., near-real time) intermediate results for the query, as the query is being executed, includes: from time to time (e.g., periodically), executing, by one or more computer systems, the query on two or more of the data items located at different places in the stream, optionally with the two or more data items being accessed in near real-time with respect to each of the two or more data items; generating information indicative of results of executing the query; and as the query continues being executed, generating intermediate results of query execution by aggregating the results

with prior results of executing the query on data items that previously appeared in the stream of (e.g., near real-time) data; and transmitting to one or more client devices the intermediate results of query execution, prior to completion of execution of the query. A system of one or more computers can be configured to perform particular operations or actions by virtue of having
5 software, firmware, hardware, or a combination of them installed on the system that in operation causes or cause the system to perform the actions. One or more computer programs can be configured to perform particular operations or actions by virtue of including instructions that, when executed by data processing apparatus, cause the apparatus to perform the actions.

The actions include, at a subsequent point in time, aggregating the intermediate results
10 with results for executing the query at the subsequent point in time to generate final results. The actions also include storing the final results and discarding the intermediate results. Executing the query from time to time includes: executing the query on (i) one or more first data items in the stream of (e.g., near real-time) data, with the one or more first data items being located in a first portion of the stream, and (ii) one or more second data items in the stream of (e.g., near real-
15 time) data, with the one or more second data items being located in a second portion of the stream. Executing the query includes: periodically executing a dataflow graph that represents the query, with the dataflow graph including executable computer code to implement the query, and with the dataflow graph receiving as input query specifications for the query. The dataflow graph includes components that represent operations to be performed in execution of the first
20 query, and wherein the method further includes: for a component: performing a checkpoint operation that saves a local state of the component to enable recoverability of a state of the dataflow graph. The query is executed on data items that appear in the stream of near real-time data during a period of time the end of which is unknown at a start of executing the query. An amount of data items in the stream on which the query is executed is unknown at a start of
25 executing the query.

The actions include accessing information indicative of user-defined custom operations for data transformation on the aggregated results; executing the user-defined custom operations on the aggregated results; and transforming the aggregated results in accordance with the user-defined custom operations. The actions include generating, based on the aggregated results, a
30 near real-time alert to alert a user of detection of a pre-defined condition. The stream of near real-time data includes a data stream in which data items are (i) periodically received at different

times, or (ii) continuously received at different times. The actions include receiving the stream of near real-time data from a data queue, a data repository, or a data feed. The query is a first query and wherein the method further includes: selecting a second query for execution on two or more of the data items that appear at different locations in the stream; and executing the first and
5 second queries in (e.g., near real-time) with respect to the data items of the stream. The actions include generating information for a user interface that when rendered on a display device includes: input fields for input of information defining queries to be executed on the stream of near real-time data. The actions include accessing, in a control repository, pre-defined queries that are candidates for execution on the stream of near real-time data.

10 All or part of the foregoing may be implemented as a computer program product including instructions that are stored on one or more non-transitory machine-readable storage media and/or one or more computer-readable hardware storage devices that are a hard drive, a random access memory storage device, such as a dynamic random access memory, machine-readable hardware storage devices, and other types of non-transitory machine-readable storage
15 devices, and that are executable on one or more processing devices. All or part of the foregoing may be implemented as an apparatus, method, or electronic system that may include one or more processing devices and memory to store executable instructions to implement the stated functions.

While large computer networks potentially can collect huge amounts of data from various
20 sources distributed within the networks (e.g., data sources distributed across the globe), they suffer from latency when it comes to the challenge of extracting, transforming and loading the collected data, especially when the data includes a stream of continuously arriving data. Such a distributed network can be a logistical network (e.g., airports, train stations, harbors, or other logistic centers), a security-related network (e.g., credit card information processing, banking, or
25 other authentication systems), or a network of machines performing industrial processes. Embodiments described herein allow to process huge amounts of data in a very fast manner (e.g., real-time/near real-time), while also maintaining reliability and/or safety of the computations performed on the data. The approach described in context of these embodiments is scalable in a flexible manner with respect to the amount of data, the size of the underlying network from
30 which the data originates and the amount/variety of active queries to be executed on the data stream.

The details of one or more embodiments are set forth in the accompanying drawings and the description below. Other features, objects, and advantages of the techniques described herein will be apparent from the description and drawings, and from the claims.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of a data processing system.

FIGS. 2, 5B and 8 are each a conceptual diagram of aggregating results of data processing.

FIGS. 3-5A are diagrams of dataflow graphs.

FIGS. 6-7 are flowcharts

FIG. 9 is a diagram of dynamic execution of multiple queries.

FIG. 10A is a block diagram of one embodiment of the invention showing the interrelationship of principal elements.

FIG. 10B is a block diagram of a data flow graph.

FIG. 11 is a block diagram of a typical graph having a rollup component and a sort component 2004 with designated runtime parameters.

FIG. 12 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid that would be associated with a graph.

FIG. 13 is a flowchart that summarizes the process of using a runtime parameter.

FIG. 14 is a diagram of one embodiment of a graphical dialog generated by the `key` prompt.

FIG. 15 is a diagram of one embodiment of a graphical dialog generated by the `filter` prompt.

FIG. 16 is a diagram of one embodiment of a graphical dialog generated by the `rollup` prompt.

FIG. 17 is a diagram of one embodiment of a graphical dialog generated by the `reformat` prompt.

FIG. 18A is a block diagram of a first graph in which a MergeJoin component joins data from files A and B and outputs the result to an output file.

FIG. 18B is a block diagram of a second graph in which a Rollup component aggregates data from file A and outputs the result to an output file.

FIG. 18C is a block diagram of a graph in which a MergeJoin component joins data from files A and B, and a Rollup component aggregates the resulting data and outputs a final result to an output file.

FIG. 19 is a diagram of one embodiment of a graphical dialog presenting a Condition having a Condition-interpretation control.

FIG. 20 is a diagram of a graph showing a situation in which poisoning arises.

FIG. 21 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Remove Completely* conditional component.

FIG. 22 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Replace With Flow* conditional component for a particular embodiment of the invention.

FIG. 23 is a diagram of a graph representing a rollup application without runtime parameters.

FIG. 24 is a diagram of a graph representing a runtime parameterized version of the rollup application of FIG. 23.

FIG. 25 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application of FIG. 24.

FIG. 26A is a diagram of one embodiment of a graphical dialog representing a form generated by the Web Interface from the information in the parameters grid of FIG. 25.

FIG. 26B is a diagram of the form of FIG. 26A filled in by a user with parameter values.

FIG. 27 is a diagram of a graph representing a runtime parameterized rollup and join application.

FIG. 28 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application of FIG. 27.

FIG. 29 is a diagram of one embodiment of a graphical dialog representing a form generated by the Web Interface from the information in the parameters grid of FIG. 28.

FIG. 30 is a diagram of a graph representing a runtime parameterized rollup-join-sort application.

FIG. 31 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid for the example application shown in FIG 30.

FIG. 32A is a diagram of a graph in which metadata is propagated.

FIG. 32B is a diagram of a sub-graph for a component in the graph of FIG. 32A.

5 FIG. 33 is a flowchart for a metadata propagation process.

FIG. 34A is a graph including parameters that have intra-component and inter-component dependencies.

FIGS. 34B and 34C are dependency graphs representing dependencies among the parameters of the graph in FIG. 34A.

10 FIG. 35 is a diagram of a modified topological sort process.

FIG. 36 is a block diagram of an example arrangement of a system in which a dataflow graph is executed.

FIG. 37 is an illustration of an exemplary dataflow graph including a micrograph.

FIG. 38A is an illustration of an exemplary specialized component.

15 FIG. 38B is an illustration of an exemplary micrograph.

FIG. 39 is an illustration of an exemplary specialized component with a plurality of micrographs in memory.

FIG. 40A is an illustration of an exemplary specialized component with multiple micrographs processing data flows.

20 FIG. 40B is an illustration of an exemplary dataflow graph with multiple specialized components.

FIG. 41 is block diagram of an exemplary specialized component including pre and post processing components.

25 FIG. 42 is an illustration of example of multiple data flow components, each interacting with a single database.

FIG. 43 is an illustration of an exemplary data flow component interacting with multiple databases.

FIG. 44A is an illustration of an exemplary data flow component interacting with multiple databases.

30 FIG. 44B is an illustration of an exemplary data flow component interacting with multiple databases using a common protocol layer.

FIG. 45 is a block diagram of an example arrangement of a system in which a dataflow graph is compiled prior to run-time.

Like reference symbols in the various drawings indicate like elements.

5

DESCRIPTION

A system consistent with this disclosure implements a Real-Time Transaction Tracker (RTTT). Generally, RTTT is a software program that implements operations (e.g., queries) for obtaining information from a stream of data over an extended (and unknown) period of time. RTTT allows for the ad hoc monitoring of queries in real-time applications. RTTT also supports the execution of multiple queries, added dynamically, and provides for visibility of partial results, as further described below. As used herein, “real-time” includes, but is not limited to, near real-time and substantially real-time, for each of which there may be a time lag between when data is received or accessed and when processing of that data actually occurs, but the data is still processed in live time as the data is received (e.g. continuously or periodically). Using RTTT, a user can specify operations (e.g., queries) to be performed (e.g., concurrently) on the data stream in real-time. RTTT generates intermediate results of the executed queries and (e.g., periodically) updates the intermediate results with new, subsequent results of the executed queries. Generally, an intermediate result includes a result of query execution that occurs prior to completion of query execution. RTTT also provides the user with interfaces to (e.g., periodically) generate (at any time) new queries for execution and/or to provide feedback to the user whether the specified operations (e.g. queries) can be performed (in the desired manner, e.g., in real-time) on the data stream by the system. As further described below, a user may also stop or terminate a query at will.

RTTT also stores only the results of the queries and discards individual items that occur in the data stream. With RTTT, a user is able to periodically and/or continuously monitor data and operate on it frequently (in real-time) to obtain information.

Referring to FIG. 1, data processing environment 100 (e.g. runtime environment) includes system 102 for implementing RTTT 103. RTTT 103 includes source data reader 106 for receiving source data, control repository 112 for storing queries 107a ... 107n and query objects 111a ... 111n, query driver or execution module 108 for determining which of query objects 111a ... 111n to execute, and query worker engines 110a ... 110n for executing queries on

the source data (or on portions thereof). RTTT 103 supports execution of multiple queries, using the techniques described herein. The below description may refer to “a query” or “the query.” These references to “a query” or “the query” are for purposes of convenience, without limiting the attendant description to a single query. The techniques described below are applicable to multiple queries.

Generally, source data includes data received from third party data sources (e.g., systems) that are external to system 102. Source data includes real-time data stream 104 (e.g., near real-time data stream), which source data reader 106 receives from external systems. Real-time data stream 104 includes a data stream in which data items are located in different places in the stream, e.g., as the stream is being periodically, intermittently and/or continuously received. For example, some data items are located in one portion of the stream and other data items are located in another, subsequent portion of the stream. Real-time data stream 104 also includes a data stream in which data items are (i) periodically received at different times, or (ii) continuously received at different times. Real-time data stream 104 includes various types of data, including, e.g., transaction log feed data that is indicative of transactions (e.g., logistic transactions, machine data transactions, debit card transactions, automated teller machine (ATM) transactions, credit card transactions, charge card transactions, stored-value card transactions, international transactions for a payment card, domestic transactions for a payment card, manual cash transactions, and so forth). In the example of FIG. 1, file 109 is received in real-time data stream 104. In this example, a file is one type of source data. Generally, a file includes a set of records (e.g., two-hundred thousand records), with each record representing items of data. Records in a file may be arranged into work units, e.g., by RTTT 103. Generally, a work unit is a collection of data to be processed. For example, a work unit may represent a set of records or a subset of records. Source data reader 106 scans multiple directories (not shown) in system 102 for arrival of source data from the third party data sources. These directories are pre-configured in system 102 to receive the source data. Source data reader 106 reads the data from real-time data stream 104. Source data reader 106 reads this data a single time and uses this read data to service many queries (e.g., queries 107a ... 107n). In this example, system 102 does not include a data store for storing (e.g., permanently) data items occurring in real-time data stream 104. Rather, real-time data stream 104 is read by source data reader 106 and then discarded, except for being temporarily stored in a buffer cache of system 102, while queries are being executed

against the data in real-time data stream 104. Following execution of the queries, the contents of the buffer cache are also discarded.

RTTT 103 also includes query creator 114 that provides a user (e.g., an analyst) with user interfaces for creating queries and obtaining feedback whether the created queries can be performed by the system. Query creator 114 includes a module that provides a user interface for an analyst to generate queries. In an example, query creator 114 is a user interface for data entry. In another example, query creator 114 includes a spreadsheet template. Query creator 114 validates the query syntax and generates query worker engines (from a template), which is the executable code for a query. The executable code is stored as files on disk in system 102 and control repository 112 tracks when the queries start and stop. Query worker engines include generic executable code that be used to execute any query object, as described below. Query creator 114 also registers the query metadata (e.g., a query state) with control repository 112, which is used for control, metadata, security and audit. The query state specifies whether a query is currently executing or when as the last time the query executed.

Through the user interfaces of query creator 114, client device 116 transmits query 107a to system 102. A user may define multiple queries 107a ... 107n that are transmitted to system 102 and stored. Client device 116 transmitting query 107a is operated by user 117 creating query 107a. Query 107a includes query syntax information that specifies a syntax for the query. Responsive to receiving query 107a, query creator 114 validates the query syntax and generates query object 111a. Query 107a includes a query specification, i.e., a detailed description of the query (e.g., query terms).

For example, query 107a is the user entered information that specifies how to process an incoming data stream (e.g., real-time data stream 104) and how to process the results, e.g. for display (e.g., what results the user wants to see). Query 107a also specifies criteria for the query to execute. Query object 111a is the machine representation (i.e., machine language representation) of query 107a, so that a query worker engine can read and process the query. A query worker engine executes the query object. For purposes of convenience, and without limitation, the term query may be used to refer to a user-specified query (e.g., query 107a) or a query object (e.g., query 111a).

As previously described, the query worker engine is the executable code for a query. This executable code includes an uninstantiated dataflow graph, e.g., a dataflow graph that is not

represented for a concrete instance. This uninstantiated dataflow graph includes various components. Upon reading in the query object, the query worker engine or pre-execution module turns off certain components and turns on others, e.g., based on the components needed to execute the query as specified by the query object. That is, the query object specifies which components are to be locally executed by the query worker engine to satisfy the specifications of the query. In this example, query creator 114 generates a query object for each query. For example, query creator 114 generates query objects 111a ... 111n for queries 107a ... 107n, respectively.

In an example, system 102 generates data for a graphical user interface (GUI) through which a user inputs data that is used in producing the query and the query specification. Through the GUI, the user inputs a query identifier (ID), which is associated with the query and used to identify the query. The query ID is also used to retrieve the query specification and to name per-query results files, e.g., files for storing intermediate and final results of query execution, as described in further details below. The user also inputs, into the query specification, data specifying a start time for execution of the query, a stop time for execution of the query, and a reporting interval (e.g., how often a user may view results of executing the query). By specifying the start/stop time for the query, a user can schedule the query to run at a specific time and to run for specific times. In a variation, system 102 provides a user with a control (e.g., via a display in a graphical user interface) that provides for automatic starting and stopping of queries.

Through query creator 114, a user may dynamically add one or more queries for real-time execution. Once a query is created, the user may dynamically add the query (e.g., to a set of multiple queries being executed) by updating the query specification to specify a new start time (e.g., to be a current time) and/or a new stop time. That is, via the query creator 114, a user may dynamically add additional queries to a set of multiple queries that are already executing. The user may dynamically and in real-time add multiple queries for real-time, present execution.

Query 107a also includes a name of a compiled dataflow graph to execute to implement the query. Generally, a dataflow graph is a directed graph. This name of the compiled dataflow graph is saved as part of the query specification. As described in further detail below, the query is implemented via a dataflow graph that applies the query to portions of real-time data stream 104. In this example, data processing environment 100 is an environment (e.g., development environment) for developing a query as a dataflow graph that includes vertices (representing data

processing graph components or datasets) connected by directed links (representing flows of work elements between the components, i.e., data) between the vertices. For example, such an environment is described in more detail below. A system for executing such graph-based queries is described in U.S. Patent 5,966,072, titled "Executing Computations Expressed as Graphs."

5 Dataflow graphs made in accordance with this system may provide methods for getting information into and out of individual processes represented by graph components, for moving information between the processes, and for defining a running order for the processes. This system may include algorithms that choose inter-process communication methods from any available methods (for example, communication paths according to the links of the graph can use
10 TCP/IP or UNIX domain sockets, or use shared memory to pass data between the processes).

A dataflow graph (e.g. a micrograph) as described herein may be implemented by a system, wherein the system may include: a data storage; a computing environment coupled to the data storage and configured to generate a data flow graph that implements a graph-based computation associated with a plurality of active queries to be executed against a file, wherein
15 the graph-based computation includes executable computer code to implement the plurality of queries, the graph-based computation performed on data flowing from one or more input data sets through a graph of processing graph components to one or more output data sets, wherein the data flow graph is specified by data structures in the data storage, the dataflow graph having a plurality of vertices being specified by the data structures and representing the graph
20 components connected by one or more links, the links being specified by the data structures and representing data flows between the graph components; a runtime environment, being hosted on one or more computers, to execute the graph-based computation and being coupled to the data storage, the runtime environment including: a pre-execution module configured to read the stored data structures specifying the data flow graph and to allocate and configure computing
25 resources for performing the computations of the graph components that are assigned to the data flow graph by the pre-execution module, and an execution module to schedule and control execution of the assigned computations such that the operations of one or more of the methods described herein are executed.

The dataflow graphs developed using system 102 can be stored in control repository 112
30 (or another data storage) and accessed by RTTT 103. RTTT 103 may execute the dataflow

graphs to carry out queries associated with the components of the dataflow graphs to process the data received in real-time data stream 104.

Using query creator 114, a user may specify various query criteria and query parameters for a query. Generally, query criteria include conditions (e.g., types of data to be queried) to be
5 satisfied prior to execution of a query. In an example, queries are selected for execution based on query start times, e.g., rather than based on query criteria. That is, a user specifies a start time for a query and the query “wakes-up” or starts executing at that time. There are various types of query parameters, including, e.g., attributes to be displayed in results (e.g., which attributes a
10 user wants to view in query results), aggregation keys that specify how to aggregate the query results, filter conditions, sort conditions (e.g., information specifying how to sort the query results, such as based on date, output styles (e.g., information specifying a format for output of the query results), and alert thresholds (information specifying one or more pre-defined conditions that trigger sending an alerts).

For example, a user may specify filter conditions – information specifying how to filter
15 items of data included in real-time data stream 104. Filter conditions indicate which types of data to filter out of (exclude from) the query results and which types of data to filter into (include in) the query results. Based on the user provided filter conditions, system 102 updates query object 111a with the filter conditions and provides the query object 111a to a query worker engine to enable the query worker engine to filter the query results in accordance with the user’s
20 filter conditions.

Using query creator 114, a user may also define custom operations for data transformation on the query results. A user may specify an operation to read and to convert Extended Binary Coded Decimal Interchange Code (EBCDIC) date times. A user may specify another operation to convert a format of data items representing currency amounts into a United
25 States dollar (USD) currency. A user may specify still another type of operation to determine a payment card type (e.g., a debit card, an ATM card, a stored-value card, a credit card, a charge card and so forth) from types of data in real-time data stream 104. Through the query parameters and user-defined custom operations, a query worker engine (e.g., one of query worker engines 110a ... 110n) is able to customize the query results in accordance with the user’s specification.
30 In particular, the query worker engine accesses (e.g., via query object 111a in control repository 112) information indicative of the user-defined custom operations for data transformation on

aggregated (e.g., final) results of executing a query. The query worker engine executes the user-defined custom operations on the aggregated results and transforms (e.g., modifies) the aggregated results in accordance with the user-defined custom operations.

5 Query creator 114 stores query 107a and query object 111a (in association with each other) in control repository 112 (e.g., a data repository for storage of queries and associated query objects). In this example, control repository 112 stores queries 107a ... 107n and query objects 111a ... 111n to be executed on real-time data stream 104. In this example, queries 107a ... 107n include query 107a and query objects 111a ... 111n include query object 111a. Query creator 114 provides for audit and security of query 107a, as previously described. Query creator
10 114 also registers query objects 111a ... 111n with query driver 108.

 Query driver 108 is a portion of RTTT 103 that selects one or more of query objects 111a ... 111n for execution and determines when to execute those selected query objects, as described in further detail below. Query driver 108 includes a continuous application that reads a real-time stream of source data on an incoming message bus. In an example, query driver 108
15 continuously executes a dataflow graph. Via the dataflow graph, query driver 108 is configured such that when it detects or receives more data (e.g., from a data stream), query driver 108 checks control repository 112 to determine which query object to run. Query driver 108 instantiates one instance of query worker engine for every query object that needs to be run.

 Query driver 108 polls (e.g., intermittently) control repository 112 for queries that are
20 active and scheduled to be run (e.g., at a current time and/or at an upcoming time). By intermittently polling control repository 112 for active queries, query driver 108 enables execution of multiple queries that are added dynamically, e.g., by being scheduled to start at a particular time. As described herein, queries can be scheduled to start and to stop at a specified time. For a given (original) set of queries that are already executing, query driver 108 may
25 identify additional queries to be added to the original set of executing queries, e.g., based on results of polling control repository 112. In this example, these additional queries are scheduled for execution at a time that is subsequent to the execution time for the queries in the original set. By intermittently polling control repository 112, query driver 108 dynamically adds to the already executing queries, by adding more additional queries to be executed. In a variation,
30 control repository 112 pushes to query driver 108 information specifying which queries to execute (and when), when the queries are registered with control repository 112. This pushed

information includes information specifying an active query worker engine for the registered query. There are various ways in which a query is registered with control repository 112, including, e.g., by generating a query object for the query and storing the query object in control repository 112.

5 Query driver 108 batches the incoming data into small work units, e.g., all data that arrived in the last 5 seconds. For each work unit, query driver 108 determines which query objects to execute by polling control repository 112 to determine the active query worker engines at that time, as previously described. These active query worker engines represent the currently active queries. As previously described, a query has a start time. The query worker engine is
10 programmed to turn on (e.g., be listed as active) at that start time. Query driver 108 calls one instance of the query worker engine for every query to be executed for the current unit of work. When the query worker engines finish execution, the control passes back to query driver 108 and it updates a query state in the control database (e.g., control repository 112).

 In a variation, when query creator 114 registers one of query objects 111a ... 111n with
15 query driver 108, and query driver 108 stores (e.g., in its memory cache) data specifying query criteria for execution each of query objects 111a ... 111n and query IDs for each of query objects 111a ... 111n. Using the stored information, query driver 108 determines which of query objects 111a ... 111n should be executed against real-time data stream 104 in real-time and as real-time data stream 104 is received. Query driver 108 makes this determination by determining the user-
20 specified start and stop times of queries, e.g., based on the query specification for these queries. In some examples, a query specification specifies a start and a stop time for a query. In another example, query driver 108 makes this determination by comparing the query criteria to qualities (e.g., attributes) of data items in real-time data stream 104. When query driver 108 determines
25 that query criteria of one or more of query objects 111a ... 111n is satisfied (and/or that certain queries should start in accordance with start times), query driver 108 uses the appropriate query IDs to retrieve, from control repository 112, the queries for which the query criteria are satisfied and to retrieve query objects for the appropriate queries. In another example, the query specification indicates that the query should be executed for as long as real-time data stream 104 includes data satisfying the query criteria of the queries being executed.

30 Query driver 108 implements a dataflow graph as described above (e.g., a continuous group, a periodic graph, and so forth) that reads real-time data stream 104 from source data

reader 106. Using the graph, query driver 108 determines which data items of real-time data stream 104 satisfy query criteria of queries query objects 111a ... 111n stored in control repository 112. Upon identification of a query for which query criteria are satisfied, query driver 108 calls an instance of a query worker engine (e.g., one of query worker engines 110a ... 110n).

5 In a variation, query driver 108 polls control repository 112 for a list of active query workers engines and applies these active query worker engines to the real-time data stream 104 or portions thereof. Query driver 108 also records, as part of the query state for a query in control repository 112, query execution statuses (e.g., an audit trail) to track which queries have been executed, when the queries are executed, and for how long.

10 Generally, a query worker engine is a generic program that can support a user query. One instance of the query worker engine is spawned by query driver 108 for every user query that needs to be executed for a given unit of work. Only those query worker engines are instantiated where the current system date time is within the monitoring period for a corresponding query. The actual user query drives the features of the query worker (for example,
15 if the specific user query has no aggregation or sort, these functions are turned off for the corresponding instance of the query worker). The query worker engine also computes alert metrics and delivers alerts at threshold, if requested to do so.

A query worker engine includes a dataflow graph as described above (e.g., a single generic graph) for a query, e.g., one of query objects 111a ... 111n. The query object is executed
20 through execution of the graph. Query driver 108 generates or selects one instance of a query worker engine as a graph, per executed query. In the example of FIG. 1, query driver 108 executes (e.g., concurrently) query objects 111a ... 111n against data items in real-time data stream 104. To execute these query objects, query driver 108 generates or selects query worker engines 110a ... 110n. In this example, query worker engines 110a ... 110n execute query objects
25 111a ... 111n, respectively.

In an example, an amount of data items in real-time data stream 104 is unknown at a start of executing one of query objects 111a ... 111n. Additionally, data items appear in real-time data stream 104 for a period of time. The end of this period of time is unknown at a start of executing one of query objects 111a ... 111n. In this example, source data reader 106 receives file 109 in
30 real-time data stream 104. In this example, source data reader 106 receives a new file at specified time intervals (e.g., every five minutes). Query driver 108 determines which of query

objects 111a ... 111n should be executed against file 109. In this example, query driver 108 determines that each of query objects 111a ... 111n should be executed against file 109. As such, query driver 108 generates or retrieves query worker engines 110a ... 110n to execute (concurrently) query objects 111a ... 111n against file 109. This process is then repeated when
 5 the next file arrives, e.g., for queries with start and end times that specify that the query is still active and executing. In this example, source data reader 106 reads in file 109 and transmits file 109 to query driver 108. Query driver 108 transmits file 109 to each of query worker engines 110a ... 110n to use when executing query objects 111a ... 111n, respectively.

A query worker engine also generates alert metrics and delivers alerts, for a query (e.g. if
 10 the user input does not comply with pre-defined requirements for the processing of the queries). For example, when a user is defining a query specification, the user will specify various pre-defined events that trigger an alert (e.g., a notification message specifying the occurrence of an event). These events may be stored in a query object for an associated query. In another example, the user may specify a threshold for an event, with the user requesting to be notified
 15 when a number of the events exceeds the threshold. Generally, an event includes an occurrence of one or more pre-specified data values or types. Query driver 108 transmits to each of query worker engines 110a ... 110n data indicative of the predefined events for the various queries being executed via query worker engines 110a ... 110n, to enable query worker engines 110a ... 110n to generate appropriate alerts.

Based on execution of query objects 111a ... 111n against data items in real-time data
 20 stream 104, query worker engines 110a ... 110n generate intermediate results 118a ... 118n. Intermediate results 118a ... 118n are available (for user review) throughout the duration of query execution. As described in further detail below, query worker engines 110a ... 110n combine intermediate results 118a ... 118n, respectively, with subsequent results of subsequent executions
 25 of query objects 111a ... 111n. These combined results produce either new intermediate results or final results (e.g., when execution of a query has ceased).

Using intermediate results 118a ... 118n (or final results), query worker engines 110a ... 110n generate alerts 120 and query results 122 (e.g., in comma separated values (CSV) form or in Structured Query Language (SQL) format) for the executed queries. In an example, the query
 30 results include partial results, e.g., the results of query execution up to a current point in time, while the query is still executing and prior to completion. In this example, the partial results are

continuously and/or periodically updated with new results, e.g., as the queries continue to be executed. The partial results are also displayed in real-time, e.g., as RTTT 103 detects that one or more items of real-time data satisfy one or more of the conditions of the query. As RTTT 103 continuously detects that new items of the real-time data satisfy the query condition (s), RTTT 103 updates and displays the partial results, e.g., based on the newly detected items. In this example, system 102 transmits alerts 120 and query results 122 to client device 125. In an example, query results 122 are stored in control repository 112 for subsequent retrieval.

Generally, an alert includes a notification message that informs the user of a detected event.

Generally, query results include a visualization of results of executing a query and include

information specifying how often and when a particular event is detected. Alerts 120 may be email or simple messaging service (SMS) alerts. System 102 may be configured to deliver the alerts at defined thresholds, e.g., information specifying a magnitude (e.g., a magnitude of events) that must be exceeded for an alert to be sent. Alerts 120 and query results 122 are each transmitted to client device 124, e.g., for viewing by user 125 of client device 124. System 102 outputs various types of query results, including, e.g., a consolidated output and an appended output. Generally, a consolidated output includes data that quantifies the results into charts and other visualizations. Generally, an appended output includes a data file of the results. Because the queries are run in real-time (and as real-time data stream 104 is received), system 102 is able to deliver results in real-time and within minutes of data arrival.

In an example, data processing environment 100 is used to determine when a payment card (e.g., a debit card, an ATM card, a stored-value card, a credit card, a charge card and so forth) is at risk for fraudulent activity. In this example, system 102 receives from an entity issuing the card (e.g., the issuing entity) information specifying one or more cards that are at risk. In this example, source data reader 106 receives, from systems of the issuing entity, a real-time data stream indicative of transactions. Based on the real-time data stream, system 102 monitors transactions from cards at risk and determines a count of the number and types of transactions initiated by these cards. There are various types of transactions, including, e.g., a manual cash transaction, an ATM transaction, a domestic use transaction, an international use transaction, a debit card transaction, a charge card transaction, a stored-value card transaction, a credit card transaction, and so forth. In this example, an employee (e.g., an analyst in the fraud protection division) uses query creator 114 to specify alert conditions. For example, an alert condition

specifies that when system 102 determines a threshold amount of transactions of a various type (e.g., international use transaction) for a card at risk to alert the systems of the issuing entity (e.g., so that the issuing entity may decline the transaction).

RTTT 103 may be used to monitor a total number of logistic events or goods processed
5 by region, country, and so forth. RTTT 103 is also used for sales analysis to determine total sales and a count of transactions for a particular product in a particular country for a particular period of time (e.g., for a period of N hrs. starting 15/NOV/2013 08:00:00). RTTT 103 may be used to monitor a total number of marketing offer declines by region, country, and so forth. A user may configure RTTT 103 to specify that he/she would like to be notified if the total number of
10 declines exceeds a threshold. RTTT 103 may be used to generate an alert if any of the mandatory columns in incoming data is NULL or blank.

In an example, RTTT 103 also performs real-time rolling window operations (e.g., aggregations) for alerting. For a defined time window, there are various types of operations, including, e.g., a sum operation, a count operation, an average operation, a minimum operation, a
15 maximum operation, and so forth. The size of the time window itself is a user-defined parameter (e.g., 10 minutes, 60 minutes, and so forth). For example, a user could configure RTTT 103 to count a number of declines of an offer for a rolling ten minute time window. A user may also define various alert thresholds that specify when and how (e.g., via e-mail, via SMS, and so forth) the user wants to be notified. For example, the user may specify that when the count of
20 declines is greater than a threshold amount to notify the user by sending an email alert.

In another example, a financial institution is planning to launch a new product portfolio in a region and the business users would like to monitor the uptake of the product for the first 'N' hours (monitoring period) after the launch. They want to monitor several metrics such as the count of customers who have signed up, the total amount committed, broken by product and
25 geography. They also want an alert during the monitoring period, when a new customer invests more than a certain threshold amount in a product. They need to monitor the uptake of the portfolio regularly during the monitoring interval and need the information on demand to plot a trending chart. In this example, the business users do not want to wait to the end of the monitoring period, when the entire dataset will be available in the data warehouse for SQL
30 queries and analysis. In this example, the incoming data feed is not stored in a new dedicated reporting repository for the short term monitoring requirement, which would cause duplication of

the data with the data warehouse. Using the techniques described herein, the data is queried “on the fly” (e.g. in real-time) to provide requested monitoring information to the users in real-time. Additionally, users can pre-configure hundreds of queries, each with their own monitoring period. These queries run concurrently. Each query does real-time computation on a continuous stream of data and continually generates results that are available to the business. At the end of the monitoring interval for each individual query, the system pushes the final result to the recipients and automatically purge the query to free up system resources.

In an example, there is one stream of data for each query driver. In this example, a stream of data includes records of one format. In this example, for multiple streams of data, with each stream representing a records of a particular format, a data processing environment includes multiple query drivers, with each query driver being assigned to one of the data streams.

Referring to FIG. 2, diagram 150 conceptually illustrates aggregating intermediate results of query execution with subsequent results of query execution. In this example, a particular query is executed at time T1. Based on execution of the query, RTTT 103 generates results 152 (e.g., results at time T1). At time T2 that is subsequent to time T1, the query is again executed (e.g., this time against different data). For example, the query may be re-executed against a new file that is received. Based on execution of the query at time T2, RTTT 103 generates results 156, which are results of executing the query at time T2. RTTT 103 combines results 152 with results 156 to generate intermediate results 158 at time T2. Generally, intermediate results are the aggregate of all results up to and include the current time period. This process of determining new intermediate results at each subsequent time continues, until time Tn. At time Tn, execution of the query is complete. Additionally, at time Tn, RTTT 103 determines results 162 for execution of the query at time Tn. RTTT 103 generates final results 164 for query execution by combining intermediate results 160 (which is the aggregated results for all times prior to time Tn) with results 162. In this example, intermediate results 160 are the intermediate results at time Tn-1. Results 162 are the results at time Tn (e.g., the results at time Tn only).

Referring to FIG. 3, micrograph 200 is shown. In general, a micrograph is a specialized sub-graph of a dataflow graph as described above and is configured to be retrieved dynamically and embedded within a run-micrograph component. A system for executing such dynamically loaded graphs is described below. In some implementations, the micrograph may be precompiled.

In some examples, micrograph 200 is displayed in a user interface that allows the dataflow graph to be viewed, configured, and/or executed. Micrograph 200 represents a data processing operation that is performed by RTTT 103 on system 102, as shown in FIG. 1. In an example, a query worker engine executes micrograph 200 (or an instance of a micrograph).

5 Micrograph 200 runs a query on portions of a (e.g., real-time) data stream. For example, files (e.g., portions of the real-time data stream) arrive at system 102 at predefined time intervals (e.g., every five minutes). In response, system 102 keeps the files in a buffer cache of system 102, while multiple queries are run against the files. For the files received at the predefined intervals, system 102 executes a micrograph (e.g., micrograph 200) for each query to be run on the files.

10 In this example, micrograph 200 includes a graph component 202 representing an operation called “micrograph input,” which is referred to hereinafter as micrograph input component 202. Micrograph input operation receives (e.g., from a graph component in another dataflow graph that is calling micrograph 200 and thus passing data to micrograph 200) information specifying which query is to be executed and formatted data on which the query is
15 executed (e.g., a formatted file, a formatted work unit, and so forth). Micrograph 200 also includes component 204 representing an operation called “read multiple files,” which is referred to hereinafter as read multiple files component 204. In this example, data flows from micrograph input component 202 to read multiple files component 204. Read multiple files component 204 reads files from a database and adds them to the input information received via
20 micrograph input component 202. For example, read multiple files component 204 may read information specifying a data processing operation to be performed on the results of a certain query, e.g. how the results of the certain query are to be output (e.g. for user review), how the results are to be sorted, which types of data is to be filtered out of the query, and so forth. In an example, read multiple files component 204 reads in a data file that contact the query object for
25 the query to be executed. Using the query object, the query can then be executed or the results obtained in a manner consistent with the query object. Read multiple files component 204 outputs the results of adding the read files to the input data to rollup component 206.

A rollup component aggregates data, e.g., results of executing a query against data. In this example, rollup component 206 applies the query (e.g., the query terms or query string) to
30 the output of read multiple files component 204. Based on applying the query, rollup component 206 determines results of applying the query to the file(s) or the partitioned groupings of data.

The results include data that matches the query terms or that correspond to the types of data specified by the query string.

In this example, rollup component 206 aggregates the query execution results, e.g., for this particular execution of the query against the most recently received file or work unit. Rollup component 206 aggregates individual data items into a single data item or provides a count for the number of individual data items, depending on the configuration of rollup component 206. The output of rollup component 206 is the aggregated results of applying the query to the file or to a work unit. Rollup component 206 may also output the results in a format or manner specified by the query object.

Micrograph 200 also includes previous running aggregations component 210 that stores data indicative of prior aggregated results of applying the query to the previously received files. These prior aggregated results are stored in a per-query results file, e.g., a file that stores results (e.g., intermediate results) for a particular query. The per-query results file holds running totals, which can be reported back to the user or central server at pre-defined intervals (e.g., each five-minute interval, or some multiple of five-minute intervals). The per-query results file is stored in a data repository (e.g., control repository 112) and is accessed by previous running aggregations component 210.

Data flows from previous running aggregations component 210 and rollup component 206 to join component 208 that implements a join operation. The “join” operation combines two types of data, for example, one type of data included in one data source and another type of data contained in another data source. In this example, join component 208 joins the current results of application of the query (e.g., the output of rollup component 206) and the output of previous running aggregations component 210. Join component 208 writes a new per-query results file that combines (i) the current aggregate results for the current execution of the query, (ii) with the prior, aggregated results for prior executions of the query. Based on this combination, join component 208 produces a new aggregate result (that is the combination of the current and prior aggregated results). Join component 208 may store the new aggregate results in a new per-query results file in the data repository. These new aggregate results are intermediate results. These intermediate results may later be joined with subsequent results, e.g., at a later point in time.

Data (e.g., the intermediate results) flows from join component 208 to next running aggregations component 212. The next running aggregations component 212 updates the data

repository with the new per-query results file (that includes the intermediate results), to enable previous running aggregations component 210 to access this per-query results file, e.g., during the next execution of the query.

Each component of the micrograph 200 includes one or more connection ports through which the component can be connected by a dataflow connection (also referred to as a “connection” or “link”) to one or more other components of the dataflow graph. A connection port may be an input port for receiving data into a component or an output port through which data is output from a component. In some examples, a symbol that depicts a connection port may indicate whether the connection port is an input port or an output port. For instance, in the example of FIG. 3, read multiple files component 204 has input port 214 represented by an input port symbol and output port 216 represented by an output port symbol.

In this example, at least some of the components in micrograph 200 perform a checkpoint operation that saves in a buffer (e.g., of system 102) a local state of the graph component to enable recoverability of a state of the dataflow graph. The components in the other dataflow graphs described herein similarly perform checkpointing. As described above for the dataflow graph, the micrograph 200 may be implemented by a system, the system comprising: a data storage; a computing environment coupled to the data storage and configured to generate a data flow graph that implements a graph-based computation associated with a plurality of active queries to be executed against a file, wherein the graph-based computation includes executable computer code to implement the plurality of queries, the graph-based computation performed on data flowing from one or more input data sets through a graph of processing graph components to one or more output data sets, wherein the data flow graph is specified by data structures in the data storage, the dataflow graph having a plurality of vertices being specified by the data structures and representing the graph components connected by one or more links, the links being specified by the data structures and representing data flows between the graph components; a runtime environment that is coupled to the data storage and that is being hosted on one or more data processing devices may include a pre-execution module configured to read stored data structures specifying the micrograph 200 (i.e. the vertices representing the graph components 202, 204, 206, 208, 210, 212 and the links in between) and to allocate and configure computing resources for performing the computation of the graph components that are assigned to the micrograph 200 by the pre-execution module. The runtime environment may further

include an execution module to schedule and control execution of the assigned computations such that the operations described in context of Fig. 3 are executed.

Referring to FIG. 4, dataflow graph 300 executes queries against a file, e.g., starting with the execution in the moment in which the file arrives. In an example, a query driver executes dataflow graph 300. As described above, the dataflow graph 300 may be implemented by a system comprising: may be implemented by a system, the system comprising: a data storage; a computing environment coupled to the data storage and configured to generate a data flow graph that implements a graph-based computation associated with a plurality of active queries to be executed against a file, wherein the graph-based computation includes executable computer code to implement the plurality of queries, the graph-based computation performed on data flowing from one or more input data sets through a graph of processing graph components to one or more output data sets, wherein the data flow graph is specified by data structures in the data storage, the dataflow graph having a plurality of vertices being specified by the data structures and representing the graph components connected by one or more links, the links being specified by the data structures and representing data flows between the graph components; a runtime environment, coupled to the data storage and being hosted on one or more data processing devices, which may include a pre-execution module configured to read stored data structures specifying the dataflow graph 300 (i.e. the vertices representing the graph components and the links in between) and to allocate and configure computing resources for performing the computation of the graph components that are assigned to the dataflow graph 300 by the pre-execution module. The runtime environment may further include an execution module to schedule and control execution of the assigned computations such that the operations described in context of Fig. 4 are executed. These queries are executed by implementing a micrograph (e.g., micrograph 200) for each query. Dataflow graph 300 includes subscribe component 302 that subscribes to (e.g., receives data from) a source data reader. Through subscribe component 302, dataflow graph 300 accesses, e.g. in real-time, items of data included in a (e.g., real-time) data stream. In this example, subscribe component 302 receives a file (e.g., including thousands of records). Data flows from subscribe component 302 to join component 304. Join component 304 reads data (e.g., records) from a data source (or multiple data sources) in a sequence and arranges the input data into a sequence of discrete work units. As previously described, the work units may represent records stored in a predetermined format based on input records, for

example, or may represent transactions to be processed, for example. In some implementations, each work unit may be identified by a number that is unique within the batch, such as a count of work units processed. The work units are then passed in sequence to the next component in the dataflow graph 300.

5 Join component 304 also joins the file with currently active queries. Generally, a currently active query is a query that query driver has determined should be executed against a (e.g., real-time) data stream. In an example, join component 304 retrieves a list of currently active queries to be executed. This list is retrieved from either query driver 108 (FIG. 1), when stored locally on query driver 108, or from control repository 112. The list specifies query IDs
10 of queries to be executed. The currently active queries are determined based either on real-time or file time.

For example, a data repository includes a database table of currently active queries. The currently active queries are identified in the table by query IDs. For each currently active query, the database table also specifies a timestamp. The timestamp specifies at time at which query
15 driver 108 determines that the query should be executed. In some examples, join component 304 identifies, in the database table, queries with timestamps that occur within a current file time (e.g., a current time interval in which files are being currently received). In other examples, join component 304 identifies, in the database table, queries with timestamps that are close to a current time (e.g., timestamps for which a difference between the current time and a time
20 specified by the timestamp is less than a threshold amount of time). In still another example, join component 304 selects queries that are specified as being currently active queries. Join component 304 selects the currently active queries and joins these selected queries with the real-time data. Join component 304 also retrieves the query specifications for the selected queries, for use in processing the query.

25 Data flows from join component 304 to partition component 306, which partitions (e.g., allocates) the currently active queries to be processed (e.g., on the same file) to multiple component instances, e.g., instances 308a ... 308n of reformat component (hereafter reformat components 308a ... 308n) and instances 310a ... 310n of run micrograph component (hereafter run micrograph components 310a ... 310n). That is a partition represents a division of the
30 currently active queries across various instances of components. In this example, there are “n” currently active queries. In this example, the file is processed “n” times, once for each query.

By partitioning the queries, the queries may be in run in parallel with each other (and across various distributed systems). Partition component 306 partitions the queries in a round-robin manner, by assigning a first query to a first of the instances, a second query to a second of the instances, and so forth. In an example, a portion of the instances are executed on a system,
5 thereby distributing all instances across various systems.

When processing of a work unit is completed the results are passed to a gather component 338 that collects results from multiple instances and passes them to the next data processing component in the dataflow graph.

Data (e.g., data specifying the partitioning of the queries and the current file) flows from
10 partition component 306 to reformat components 308a ... 308n. Each of reformat components 308a ... 308n corresponds to one of the currently active queries. Reformat component represents a reformat process, so that when dataflow graph 300 is executed by system 102, the reformat process represented by a reformat component is executed. Generally, a reformat process includes a series of instructions for converting data from one format to another format. In this
15 example, a reformat component reformats the data records (e.g. in the current file) to a format that is readable by a dataflow graph (e.g. another dataflow graph or one or more of the micrographs 200 connected to dataflow graph 300).

The reformatted data flows from reformat components 308a ... 308n to run micrograph components 310a ... 310n, which execute a micrograph to execute a query (serially) on data
20 records in a file. In particular, run micrograph component 310 selects a micrograph (e.g., micrograph 200) to execute. For each query, a micrograph is executed, e.g., to serially process the queries against a file. In this example, each of run micrograph components 310a ... 310n corresponds to one of the “n” queries to be processed against the current file.

Dataflow graph 300 also includes gather component 312, which collects the results of
25 execution of the queries against the data records in the file. Generally, gather component 312 collects results from run micrograph components 310a ... 310n and passes them to the next data processing component in dataflow graph 300. Data flows from run micrograph components 310a ... 310n to gather component 312. Gather component 312 collects the results of execution of the queries, e.g., by accessing the per-query results files for the queries that are executed.
30 Data indicative of the results of the query executions flows from gather component to publish component 314, which publishes the results, e.g., in a results-viewing GUI that displays the

contents of the per-query results file for a given query ID or for multiple query IDs or to another dataflow graph that is configured to perform further graph-based computations to the published data. In an example, the per-query results file stores both the subtotals from the last query execution and the running grand totals. In this example, the GUI includes a dashboard that
5 displays results for the last five minutes and results since the query first started running. The dashboard also displays multiple sets of totals on different time scales, e.g., results for the last 5 minutes, the last 10 minutes, the last 30 minutes, and so forth. The dashboard displays varying levels of granularity, e.g., results for one or more particular queries and/or results for all queries.

Referring to FIG. 5A, dataflow graph 400 implements parallelism (e.g. provides pipeline
10 parallelism) within a given file/query combination, e.g., to both execute multiple queries in parallel and to further process, in parallel, the data records for a particular query. This process improves scalability for large files (e.g., files exceeding 2 gigabytes), thereby reducing processing time and increasing processing speed. For example, system 102 processes a file a predefined number of times (e.g., N times), because the file is processed for each relevant query.
15 In this example, when there are N relevant queries, system 102 processes the file N times. A query can be processed either serially or in parallel. When a query is processed serially, the query is serially applied to the data items in the file. Rather than serially processing a query, dataflow graph 400 processes the data for a particular query in parallel, e.g., by dividing the data in the current file into work units and then applying the query to be executed in parallel to the
20 work units.

As described above, the dataflow graph 400 may be implemented by a system comprising: a data storage; a computing environment coupled to the data storage and configured to generate a data flow graph that implements a graph-based computation associated with a plurality of active queries to be executed against a file, wherein the graph-based computation
25 includes executable computer code to implement the plurality of queries, the graph-based computation performed on data flowing from one or more input data sets through a graph of processing graph components to one or more output data sets, wherein the data flow graph is specified by data structures in the data storage, the dataflow graph having a plurality of vertices being specified by the data structures and representing the graph components connected by one
30 or more links, the links being specified by the data structures and representing data flows between the graph components; a runtime environment, coupled to the data storage and being

hosted on one or more data processing devices, which may include a pre-execution module configured to read stored data structures specifying the dataflow graph 400 (i.e. the vertices representing the graph components and the links in between) and to allocate and configure computing resources for performing the computation of the graph components that are assigned to the dataflow graph 400 by the pre-execution module. The runtime environment may further include an execution module to schedule and control execution of the assigned computations such that the operations described in context of Fig. 5A/B are executed.

Dataflow graph 400 includes subscribe component 402 (which performs the functionality described with regard to subscribe component 302 in FIG. 4) and join component 404 (which performs the functionality described with regard to join component 304 in FIG. 4). Data flows from join component 304 to normalize component 406, which divides the contents of the file into a series of row ranges for processing of the data in parallel. For example, the file includes rows of data, e.g., with each row representing a data record. Normalize component 406 assigns the rows to various row ranges (e.g., subsets of the rows). A row range represents a work unit. In an example, a file has ten thousand rows of transaction data. In this example, rows 1-1000 are assigned to row range I (e.g., a first work unit), rows 1001-2000 are assigned to row range II (e.g., a second work unit), and so forth.

Data flows from normalize component 406 to partition component 408 (which performs the functionality described with regard to partition component 306) to allocate instances of reformat 410 and run micrograph 412 components to each work unit (e.g., the reformat component 410 may comprise a plurality of instances of reformat components 308a...308n and the run micrograph component 412 may comprise a plurality of corresponding instances of run micrograph components 310a...310n as described in context of FIG. 4). As previously described, a partition represents a division of the currently active queries across various instances of components. In this example, each partition processes a different subset of the rows (or work unit). Partition component 408 allocates the queries to be executed (e.g., the currently active queries) to each work unit. For example, for each work unit, a total number of queries to be executed are allocated to various instances of reformat 410 and run micrograph 412 for that work unit.

Data indicative of the partitions (for each work unit) flows from partition component 408 to instances of reformat component 410 to reformat the data in the work units for input into a

dataflow graph (e.g., a micrograph graph 200). Generally, a reformat process includes a series of instructions for converting data from one format to another format. In this example, a reformat component reformats the work units to a format that is readable by the respective micrograph.

The reformatted data flows to instances of run micrograph component 412 that computes partial
5 sums for execution of the queries against a particular work unit. Reformat component 410 implements the functionality that was described with regard to reformat component 308a-n. Run micrograph component 412 implements the functionality that was described with regard to run micrograph component 310a-n. Each instance of run micrograph component 412 executes a particular query for a particular work unit, as described in further detail below for FIG. 5B. That
10 is, if there are “p” queries to be executed, each work unit is processed “p” times. The work units are processed in parallel. For each work unit, the queries are processed in parallel. Generally, a partial sum is data indicative of results of execution of the query on a portion of data (e.g., a portion of a file or a particular work unit). For each work unit, an instance of run micrograph component 412 is executed for each currently active query. To do so, system 102 generates p
15 different instances of run micrograph component 412, with each instance of run micrograph component 412 being for one of the “p” currently active queries.

An aspect of the partition component 408 may be that the partition component 408 performs a pre-processing prior to the actual allocation of the instances of reformat 410 and run micrograph 412 to each work unit. The pre-processing may include: determining the computing
20 resources likely to be needed for the execution of each of the queries (e.g., each of queries 1 to p) assigned to the respective instance (e.g., instances 1 to p) and distributing computing resources among the instances according to the determined need of computing resources for execution of each of the queries, wherein more computing resources are allocated to instances (e.g., instance 1), for which it is determined that the execution of the associated query (e.g., query 1) will likely
25 need more computing resources, than to instances (e.g., instance 2), for which it is determined that the execution of the associate query (e.g., query 2 needs less computing resources than query 1) will likely need less computing resources. The pre-processing may additionally or alternatively include: determining the data format of the work unit received by the partition component 408; and allocating the work unit to one of the instances based on the determined
30 data format of the work unit. For example, partition component 408 may determine the computing resources and/or the time needed to convert the work unit from the determined data

format to a data format required by each of the instances, in particular by each of the micrographs, and may allocate the work unit to the instance with which the execution of the query to be executed requires the least amount of computing resources and/or conversion time.

The pre-processing may further increase the speed of data processing and may further
5 reduce the time needed for the execution of the queries. This may be beneficial also in networks that have to authenticate users, transactions or pieces of data in a very fast manner to ensure data security, even though a distributed network and/or large amounts of data are involved (e.g. authentication systems related to credit card processing, banking, or air travel).

Dataflow graph 400 includes gather component 414, which gathers together, for each of
10 the work units, the partial sums for execution of the queries against that particular work unit. In this example, data (e.g., data indicative of the partial sums) flows from gather component 414 to run micrograph component 416, which computes global sums. Generally, a global sum is data indicative of final results of execution of a query against a file (e.g., execution of a query against all the row ranges (work units) for the file). To compute the global sums, run micrograph
15 component 416 executes another micrograph (e.g. micrograph 200) that aggregates the results for execution of the query on each work unit. In an example, each query has an associated instance of run micrograph component 416 to compute global sums for that query (e.g., results of executing that query against all the work units). That is, run micrograph component 416 computes a global sum (e.g., for a particular query) by aggregating the partials sums (across the
20 different work units) for that particular query. Data indicative of the global sums flows from run micrograph component 416 to publish component 418 to publish the global sums (e.g., the results of executing the queries against the file). The functionality of publish component 418 includes the functionality previously described for publish component 314.

Referring to FIG. 5B, diagram 500 illustrates an example of results of execution of
25 components 406, 408, 410, 412 and 414 in FIG. 5A. In this example, normalize component 406 divides a file into “q” work units, e.g., work units 502a ... 502q. In this example, there are also “p” currently active queries to be executed against the file. The file contains data, which is then distributed across the work units by the normalize component 406. As previously described, partition component 408 is applied to each of work units 502a ... 502q. Application of partition
30 component 408 to each of work units 502a ... 502q partitions the “p” queries across “p” different instances of run micrograph component 412 (e.g. in addition to associated different instances of

reformat component 410), for each work unit. In this example, partition component 408 generates, for work unit 502a, instances 506a ... 506p of run micrograph component 412, with each instance corresponding to a query. Partition component 408 generates, for work unit 502q, instances 508a ... 508p of run micrograph component 412, with each instance corresponding to a query. Instances 310a ... 310n of run micrograph component may be used for or may be identical with instances 506a ... 506p of run micrograph component 412 and may communicate with respective instances 308a ... 308n of reformat component 410. Alternatively, a single reformat component 410 may be used with the plurality of instances of run micrograph component 412.

Partition component 408 may also generate, for each work unit, q different instances of reformat component 410, to reformat the data in the work unit prior to forwarding the work unit to the micrograph. Following execution of the instances of run micrograph component 412 for each work unit, gather component 414 generates partial results for the work unit. For example, following execution of instances 506a ... 506p of run micrograph component 412, gather component 414 generates partial results 510a for work unit 502a. Partial results 510a include the results of executing queries 1 ... p against the data in the work unit 502a. Following execution of instances 508a ... 508p of run micrograph component 412, gather component 414 generates partial results 510q for work unit 502q. Partial results 510q include the results of executing queries 1 ... p against the data in the work unit 502q. Gather component 414 transmits partial results 510a ... 510q to run micrograph component 416, which computes global sums for execution of each query against all the work units in the file.

The dataflow graphs (and micrographs) described herein are recoverable, e.g., from an error event. To ensure recoverability, a component (in the dataflow graph or micrograph) maintains a local state by checkpointing. The local state specifies a location in the stream of data processing has occurred, e.g., up to which point in the stream of data has processing occurred. The local state also specifies the results of processing up to that point. In an example, a file is arranged into a sequence of work units. That is, the data records included in a file are arranged into a sequence of work units. Each work unit has an assigned identifier (ID). As a dataflow graph starts processing the work units, one or more components of the dataflow graph store the IDs, to specify which of the work units have been processed. The component also performs checkpointing to store a result of performing an action (as specified by the component) on the

work unit. This result is stored in association with the ID. The checkpointing effectively inserts a marker into the stream of data to specify which data in the stream has already been processed and results of processing that data. Because of this recoverability via checkpointing, if a dataflow graph or micrograph crashes or otherwise encounters an error, the dataflow graph or micrograph can recover by determining up to which unit it has already processed and continue processing on from that work unit, e.g., rather than having to starting processing the stream again.

In an example, system 102 maintains the file in on disk and/or in a memory cache (e.g., but does not store the file in a data repository). In this example, a query worker engine sends a message to a query driver that the query worker engine has finished processing a query object on the file. When the query driver receives this message from all the query worker engines that are processing the file, the file is archived and/or the file deleted from the cache and/or the disk. In this example, archival of the file does not result in the file being stored in a data repository.

Referring to FIG. 6, system 102 (or RTTT 103 on system 102) implements process 600 in selecting a query for execution on data items appearing at different times in a stream of real-time data. In operation, system 102 accesses (602) the data items of the stream, in real-time with respect to each of the data items. For example, system 102 accesses the data items by receiving the stream of real-time data from a data queue, a data repository (e.g., that is internal to system 102 or that is external to system 102), or a data feed (e.g., real-time data stream 104 in FIG. 1). In an example, system 102 access two or more data items in near real-time with respect to each of the two or more data items, with each of the two or more data items being located at different places in the stream, e.g., by one item of data occurring before another item of data in the stream.

System 102 determines (604) attributes of the data items, e.g., data types or formats of the received data items, e.g., by analyzing contents of the data items. System 102 accesses, in a control repository, pre-defined queries that are candidates for execution on the stream of real-time data. System 102 compares (606) the attributes to query criteria for the various queries that are candidates for execution. Based on the comparison, system 102 selects (608) a query for execution on two or more of the data items that appear at different times and/or are located at different places in the stream. For example, system 102 selects a query with query criteria that are satisfied by one or more of the attributes of the data items. System 102 also selects the query

at different times (and/or for data items appears in different portions of the stream), e.g., based on when data items that are relevant to the query are received.

In a variation, system 102 also selects, based on one or more attributes of the data items, another query for execution on two or more of the data items that appear at different times in the stream. System 102 executes both of these selected queries in real-time with respect to the data items of the stream. In still another variation, system 102 determines which of the queries to execute, by selecting the active query worker engines for that time, e.g., based on the user specified start time of the query. As previously described, a query worker engine is programmed to become active at the query start time. System 102 (or query driver) calls one instance of the query worker engine for every query to be executed for the current unit of work.

Referring to FIG. 7, system 102 (or RTTT 103 on system 102) implements process 700 in executing a query on data items appearing at different times in a stream of real-time data to provide (e.g., near-real time) intermediate results for the query, as the query continues to be executed (e.g., on subsequently appearing data items). In operation, system 102 executes (702), from time to time, the query on two or more of the data items that appear at different times in the stream. For example, system 102 executes the query on (i) first data items in the stream of real-time data, with the first data items being received at a first time, and (ii) second data items in the stream of real-time data, with the second data items being received at a second time.

System 102 executes the query by intermittently (or periodically) executing a dataflow graph as described above that implements the query. In this example, the dataflow graph includes executable computer code to implement the query. Based on execution of the query, system 102 generates (704) initial results, e.g., information indicative of results of executing the query. System 102 determines (706) if the query will execute at a subsequent time. For example, system 102 makes this determination based on a query stop time in the query specification. If system 102 determines that the query will not execute at a subsequent time, system 102 sets (708) the initial results of the query execution to be final results. If system 102 determines that the query will execute at a subsequent time, system 102 re-executes (710) the query. Based on re-execution of the query, system 102 generates (712) intermediate results by aggregating the results of the re-execution with prior results (e.g., the initial results) of executing the query on data items that previously appeared in the stream of real-time data. In an example, as the query is being executed, system 102 generates the intermediate results, e.g., to later be

aggregated with subsequent results. In this example, a user (e.g., a business analyst) may want to see and review the intermediate results, e.g., to determine which cards or transactions are at risk. In this example, system 102 transmits to a client device of the user the intermediate results of query execution, prior to completion of execution of the query.

5 System 102 determines (714) if the query will execute at a subsequent time. If system 102 determines that the query will not execute at a subsequent time, system 102 sets (716) the intermediate results of the query execution to be final results. If system 102 determines that the query will execute at a subsequent time, system 102 repeats actions 710, 712, 714, 716, as appropriate. Based on repeating these actions, system 102 aggregates (at a subsequent point in
10 time) the intermediate results with results for executing the query at the subsequent point in time, e.g., to generate final results. In this example, system 102 stores the final results in a data repository and discards the intermediate results. Based on the final results (or the intermediate results), system 102 generates a real-time alert to alert a user of detection of a pre-defined condition.

15 Using the techniques described herein, RTTT executes concurrent queries on data items in a stream of real-time data, as the data is received and without accessing stored data from a database (which increases latency). By executing the queries in real-time as the data is received, RTTT generates real-time results (e.g., both intermediate and final) for query execution as the queries are executed. To receive results, a user does not need to wait for all the data to be
20 received. Rather, the user can receive the results in real-time, as the queries are being executed.

 Additionally, user-defined queries are run for a predefined monitoring period in the future. Data arrives continuously during the monitoring period and incoming data during the monitoring period is utilized in generating results. Rather than a traditional database process model of transient queries on static data, RTTT implements static queries on transient data. The
25 queries are created by the users in advance and they are dormant in the system till the monitoring start time. The users use a simple interface to specify the query predicates such as attributes required, filtering, aggregation, sorting criteria, query control parameters like start/stop time, optional alerting conditions and a result recipient list. The queries 'listen' to the continuous data feed (e.g., message queues or messages on a port), 'wake up' at the monitoring start time,
30 perform computations on data in the stream and sleep if/when there is no data in the stream. The

intermediate results are available throughout the ‘life’ of the query and the final results are available immediately at the end time of the monitoring period for the query.

Referring to FIG. 8, diagram 800 conceptually displays the processing of data records in a file via five concurrent queries (e.g., that are executed via five different micrographs, e.g., micrograph 200), e.g. by the system underlying the dataflow graph described in context of FIG. 5A. In this example, RTTT 103 periodically receives files 802a ... 802n (which, in turn, are included in real-time data stream 801). RTTT 103 receives files 802a ... 802n at time intervals T_1 ... T_n , respectively. In this example, each of files 802a ... 802n are located at different places in stream 801. For example, file 802a is located in a place in stream 801 (e.g., a particular portion of stream 801) that comes before another place (e.g., portion) in stream 801 that includes file 802b. That is, one portion of stream 801 includes file 802a, another portion includes file 802b, and so forth. The illustrative example shown in FIG. 8 illustrates particular points in time in which queries 806, 808, 810, 812, 814 execute on files 802a ... 802n. That is, files 802a ... 802n represent files in stream 801 that are received during particular points in time (e.g., points in time in which queries 806, 808, 810, 812, 814 are simultaneously executing). In this example, queries 806, 808, 810, 812, 814 may also execute on other files (included in stream 801) that are not shown in FIG. 8, because one or more of queries 806, 808, 810, 812, 814 execute for different time intervals. In this example, results of execution against these other files may be used in determining the final results.

In this example, RTTT 103 receives file 802a at time 08:00:30, file 802b at time 08:01:00 and so forth. RTTT 102 receives a new file every thirty seconds. File 802 includes records 804a ... 804n. In this example, there are 250, 000 records in file 802. In this example, each record includes the following types of data: date (dd/mm/yyyy), time (hh:mm:ss), a product identifier (“prod id.”), a country identifier (“country id.”), a state identifier (“state id.”) and a sales amount. The other files (e.g., files 802b ... 802n) also include individual data records.

In this example, a running shoe company has a business manager that wants to watch sales of sneakers (e.g., following release of new marketing campaign) and generates questions – dynamically, e.g. in real time, such as what is the amount of uptick (e.g., increase) in sneaker sales, e.g., immediately after and during the marketing campaign.

In this example, the business manager uses RTTT 103 to create and execute queries 806, 808, 810, 812, 814 against files 802a ... 802n. In this example, because queries 806, 808, 810,

812, 814 execute for different periods of time and/or start execution at different times, each of queries 806, 808, 810, 812, 814 may only process a portion of files 802a ... 802n, depending on when the files are received and when the queries start and stop execution.

In this example, the running shoe company is launching two world-wide campaigns on two different running shoe brands, Acme sneakers and cool running sneakers. The campaign for Acme sneakers is launched at 08:00:00 (local time for each country in which the campaign is launched) on November 14, 2014 (14/NOV/2014). The campaign for cool running sneakers is launched at 13:00:00 (local time for each country in which the campaign is launched) on November 14, 2014 (14/NOV/2014). Each of queries 806, 808, 810, 812, 814 generally follows a format of "Show me the total sales and/or count of transaction for product X in country Y (or for a specific state in a country) for a period of N hours starting DD/MM/YYYY." In this example, query 806 is a query to count a number of transactions (Tx) for Acme sneakers sold Singapore starting 14/Nov/2014 08:00:00 for a period of 10 hours. Query 808 is a query to count a number of transactions (Tx) for Acme sneakers sold in Georgia, USA starting 14/Nov/2014 08:00:00 for a period of 10 hours. In this example, query 808 is a more granular than query 806, as query 808 is directed to metrics for a particular state, rather than an entire country. Query 810 is a query to count total sales for Acme sneakers sold in the USA starting 14/Nov/2014 08:00:00 for a period of 10 hours. Query 812 is a query to count total sales for cool running sneakers sold in the Japan starting 14/Nov/2014 13:00:00 for a period of 24 hours. Query 814 is a query to count total sales and number of transaction for cool running sneakers sold in the USA starting 14/Nov/2014 13:00:00 for a period of 24 hours.

Using the techniques described herein, RTTT 103 passes records (e.g., records 804a ... 804n of file 802a and records in files 802b ... 802n) through queries 806, 808, 810, 812, 814, concurrently, e.g. and in real/near-real time, without storing the records to an electronic data warehouse. As queries 806, 808, 810, 812, 814 process records (e.g., 804a ... 804n), RTTT 103 generates results 816, 818, 820, 822, 824, respectively, to provide live time results of query execution, as the queries are being processed. In this example, RTTT 103 processes approximately 500,000 records per minute and RTTT 103 receives multiple files per minute. In this example, RTTT 103 executes queries 806, 808, 810, 812, 814 continuously and processes files 802a ... 802n, e.g. in near real time, as files 802a ... 802n are received. For example, at time 08:00:00, queries 806, 808, 810, 812, 814 are executed against file 802a. At time 08:00:30,

queries 806, 808, 810, 812, 814 are executed against file 802b, and so forth (e.g., in the event that files are received every thirty seconds).

Based on execution of query 806, RTTT generates intermediate (incremental) results every time after it has processed new incoming data file (e.g., when the data is received as files periodically). In this example, intermediate results are produced as soon as the current set of input data is processed where there is a record matching the query condition. The files arrive every 30 seconds and the intermediate results are produced or updated at 08:00:30, 08:01:00 and so forth for queries 806, 808 and 810 which start at 08:00:00. In this example, at least one pair of Acme shoes are sold immediately in Singapore and Georgia at 8:00:00, when the first data file for these queries arrive. If not, the output will be produced anytime after 8AM, when the first pairs are sold in these places.

In this example, RTTT 103 generates results 816 that display the number of transactions for Acme shoes in Singapore. In this example, results 816 include intermediate (e.g., partial) results 816a, 816b and final results 816c. As shown by intermediate results 816a, 816b, a first pair of Acme shoes are sold in Singapore between 08:00:00 and 08:00:30 and a second pair of Acme shoes are sold in Singapore between 08:00:30 and 08:01:00, respectively. In this example, the intermediate results are updated (in real-time), when RTTT detects a record matching the query criteria. In this example, the intermediate results 816a, 816b display the time at which the intermediate results are produced, e.g., at 08:00:30, 08:01:00 and so forth. In this example, the intermediate results display the time at which the transaction occurred (or the time at which the incremental or additional transaction occurred).

In this example, each of the displayed final results are aggregated results, from when the query started execution. For example, the final results 816c shown for 18:00:00 (e.g., 157,692 transactions) are a number of transactions from 08:00:00 to 18:00:00. In this example, the results shown at 18:00:00 are results of processing multiple files. Each of these results are provided live, in near real-time as the files are received by RTTT 103. Using the techniques described here, there is no data storage as queries operate on data, in real-time and as it is received. The queries are forward looking as they operate on data that will arrive in future and not historical data. Queries can be set to run for a duration ranging from seconds to weeks or months. RTTT also provides an ability for users to receive intermediate results during the monitoring period. The data is read once and distributed to query workers, which independently

apply the individual query logic. For example, a system could include hundreds of query workers active in the system at a time. The latency of information delivered to the business is of the order of seconds.

Referring to FIG. 9, diagram 900 provides a visual illustration of dynamic execution of multiple queries over a twenty-four hour time period from 8am, November 14, 2015 to 8am, November 15, 2015. Queries 902, 904 are schedule for execution for 10 hours, starting at 8am on November 14, 2015. Query 906 is schedule for execution for twenty-four hours, also starting at 8am on November 14, 2015. At 8am (or at a pre-specified time interval before 8am), query driver 108 (FIG. 1) polls control repository 112 for active queries. Based on the polling, query driver 108 identifies queries 902, 904, 906 as scheduled for execution at 8am and instantiates query worker engines to execute queries 902, 904, 906. Queries 902, 904, 906 are executed against near real-time data stream 911.

In this example, query 908 is scheduled for execution for ten hours, starting at 1pm on November 14, 2015 and ending at 11pm on November 14, 2015. Query 910 is scheduled for execution for twenty-four hours, starting at 1pm on November 14, 2015 and ending at 1pm the next day (November 15, 2014). During an intermittent polling of control repository 112, query driver 108 identifies queries 908, 910 as active queries for execution and causes queries 908, 910 to begin execution at 1pm, while queries 902, 904, 906 are still executing. In this example, queries 908, 910 are dynamically added for execution, concurrent with other queries (i.e., queries 902, 904, 906) already being executed. At 6pm, queries 902, 904 complete execution, as queries 906, 908, 910 continue with execution. At 11pm, query 908 completes execution, as queries 906, 910 continue with execution. At 8am the next day, query 906 completes execution, while query 910 still proceeds with its execution.

FIG. 9 also shows files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n, which are files received by RTTT 103 in near real-time data stream 911 in a twenty-four hour time period (between 8am, November 14, 2014 and 8am, November 15, 2014). In this example, RTTT 103 receives files 912a ... 912n from 8am-1pm, files 914a ... 914n from 1pm-6pm, files 916a ... 916n from 6pm-11pm and files 918a ... 918n from 11pm-8am the next day. In this example, files included in near real-time data stream 911 are received by RTTT 103 at various time intervals, including, e.g., every thirty seconds. Appropriate queries are executed against files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n (as each of the files is received) and

intermediate (or final) results for these queries are updated based on processing files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n. Queries 902, 904, 906, 908, 910 execute on different combinations of files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n, depending on the start and stop times of the queries.

5 In this example, RTTT 103 receives files 912a ... 912n, 914a ... 914n between 8am-6pm. Queries 902, 904 process files 912a ... 912n, 914a ... 914n, in near real-time as each of files 912a ... 912n, 914a ... 914n is received, to generate intermediate and final results for execution of queries 902, 904.

10 RTTT 103 receives files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n during the twenty-four hour time period between 8am (11/14/2014)-8am (11/15/2014). Query 906 processes files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n, in near real-time as each of files 912a ... 912n, 914a ... 914n, 916a ... 916n, 918a ... 918n is received, to generate intermediate and final results for execution of query 906. RTTT 103 receives files 914a ... 914n, 916a ... 916n during the ten hour time period between 1pm-11pm. Query 908 processes files 15 914a ... 914n, 916a ... 916n, in near real-time as each of files 914a ... 914n, 916a ... 916n is received, to generate intermediate and final results for execution of query 908. RTTT 103 receives files 914a ... 914n, 916a ... 916n, 918a ... 918n during 1pm-8am the next day. Query 910 is scheduled to run for twenty-four hours starting at 1pm on November 14, 2014. Query 910 processes files 914a ... 914n, 916a ... 916n, 918a ... 918n, in near real-time as each of files 914a 20 ... 914n, 916a ... 916n, 918a ... 918n is received, to generate intermediate results for execution of query 910. Query 910 continues processing other files (which are not shown) received after 8am (on 11/15/2015) until completion of query 910.

25 In an example, a data processing environment includes an environment for developing a query as a dataflow graph that includes vertices (representing data processing components or datasets) connected by directed links (representing flows of work elements, i.e., data) between the vertices.

Overview

FIG. 10A is a block diagram of one embodiment of the invention showing the interrelationship of principal elements. A graphic development environment (GDE) 1002 30 provides a user interface for creating executable graphs and defining parameters for the graph

components. The GDE may be, for example, the CO>OPERATING SYSTEM® GDE available from the assignee of the present invention. The GDE 1002 communicates with a repository 1004 and a parallel operating system 1006. Also coupled to the repository 1004 and the parallel operating system 1006 are a Web Interface 1008 and an executive 1010.

5 The repository 1004 preferably is a scalable object-oriented database system designed to support the development and execution of graph-based applications and the interchange of metadata between the graph-based applications and other systems (*e.g.*, other operating systems). The repository 1004 is a storage system for all kinds of metadata, including (but not limited to) documentation, record formats, transform functions, graphs, jobs, and monitoring information.
10 Repositories are known in the art; see, for example, U.S. Patent Nos. 5,930,794; 6,032,158; 6,038,558; and 6,044,374.

 The parallel operating system 1006 accepts the representation of a data flow graph generated in the GDE 1002 and generates computer instructions that correspond to the processing logic and resources defined by the graph. The parallel operating system 1006 then
15 typically executes those instructions on a plurality of processors (which need not be homogeneous). A suitable parallel operating system is the CO>OPERATING SYSTEM® available from the assignee of the present invention.

 The Web Interface 1008 provides a web-browser-based view of the contents of the repository 1004. Using the Web Interface 1008, a user may browse objects, create new objects,
20 alter existing objects, specify application parameters, schedule jobs, *etc.* The Web Interface 1008 automatically creates a forms-based user interface for a parameterized graph based on information stored in the repository 1004 for the graph's runtime parameters.

 The executive 1010 is an optional repository-based job scheduling system accessed through the Web Interface 1008. The executive 1010 maintains jobs and job queues as objects
25 within the repository 1004, and the Web Interface 1008 provides a view of and facilities to manipulate jobs and job queues.

 FIG. 10B shows a simple data flow graph 1020 with an input dataset 1022 connected by a flow 1024 to a filter component 1026. The filter component 1026 is connected by a flow 1028 to an output dataset 1030. A dataset can include, for example, a file or a database table that
30 provides data (*e.g.*, an input dataset) or receives data (*e.g.*, an output dataset) for a computation performed by a data flow graph.

The flow of data represented by a “flow” in a data flow graph can be organized into discrete data elements. For example, the elements can include records from a dataset that is organized into records (or rows) and fields (or columns). Metadata describing the sequence of fields and data types corresponding to values in a record is called a “record format.”

5 Components and datasets in a graph have input and/or output ports for connecting to flows. The “source ends” of the flows 1024 and 1028 interface with an output port of the input dataset 1022 and with an output port of the filter component 1026, respectively. The “sink ends” of the flows 1024 and 1028 interface with an input port of the filter component 1026 and with an input port of the output dataset 1030, respectively. An input or output port of a dataset or
10 component is associated with metadata, such as a record format for the data flowing into or out of the port.

 A parameter including a record format for a port or other metadata associated with a component is bound to a value according to rules for parameter scoping. A parameter can be bound to a value at design time or at runtime (i.e., a “runtime parameter,” as described below).
15 The value of a parameter can be defined, for example, by a user over a user interface (e.g., in response to a prompt), defined from a file, or defined in terms of another parameter in the same context or a in different context. For example, a parameter can be exported from a different context (e.g., a parameter evaluated in the context of a different component) by designating the parameter to have a “same as” relationship to another parameter.

20 A component used in a graph can be implemented using other components that are interconnected with flows forming a “sub-graph.” Before a sub-graph is used as a component in another graph, various characteristics of the component are defined such as the input and/or output ports of the component. In some cases, characteristics of a component having to do with relationships among sub-graph components should be specified before the component is used in
25 a graph. For example, a prompting order for runtime parameters of sub-graph components may need to be selected. An approach for selecting a prompting order for runtime parameters of components in a graph is described in more detail below.

Metadata Propagation

 The value of metadata associated with a port, such as a record format parameter, can be
30 obtained by “propagation.” Metadata propagation can occur “externally” or “internally.” For

external metadata propagation, the value of a record format parameter for a port of a first component can obtain a value by propagating a record format value for a port of a second component that is connected to the first component by a flow. The value is able to propagate either downstream from the source end to the sink end of a flow or upstream from the sink end to the source end of a flow. Metadata propagates from a port that has defined metadata to a port that does not have defined metadata.

For internal metadata propagation, metadata defined for one port of a component propagates to another port of that component based on a sub-graph that implements the component. In some cases, internal metadata propagation occurs over “non-transforming” internal data paths. For example, a user may provide metadata for the input port of a sort component that specifies the data type of records flowing into the sort component. Since the sort component re-orders but does not transform the records, the data type is not changed by the sort component and the data type propagates unchanged to the output port of the sort component accurately describing the data type of the records flowing out of the sort component.

Some components do transform (or optionally transform) data flowing through them. For example, a user may provide metadata for the input port of a filter component that specifies the fields of records flowing into the filter component. The filter component may remove values of a given field from each record. A metadata definition can be used to specify that the metadata for the output port of the filter component is related to the metadata of the input port according to the filter action of the component. For example, the filtered field may be removed from the metadata specifying the record fields. Such a metadata definition can be supplied even before the input port metadata is known. Therefore, metadata can propagate even over transforming internal data paths by allowing metadata associated with a port to be specified as a function of one or more parameters, including metadata for another port, as described in more detail below.

This internal and external metadata propagation can optionally be configured to occur at design time while a graph is being constructed and a user supplies metadata for some ports of some components in the graph. Alternatively, metadata propagation can occur after a graph is constructed, including at or just before runtime.

Runtime Parameters

A runtime parameter allows an application builder to defer the value of a parameter setting (*e.g.*, the key parameter of a sort function, file names, record formats, transform functions, *etc.*) to runtime (*e.g.*, the time a program is executed or soon to be executed on a computer system). The values of runtime parameters may be supplied by the end user or be

Runtime parameters add a certain amount of flexibility to an application. Additional flexibility is achieved by using those parameters to compute metadata (data formats or *types*, and program logic or *transforms*) on demand. Types and transforms may be synthesized from other types and transforms, user-supplied parameter values, and stored objects (*e.g.*, from a repository). This makes it possible to build “generic” applications that work on input data of any type, or that produce data through a series of transforms whose construction is controlled, directly or indirectly, through runtime parameter values.

In some implementations, when creating or editing a runtime parameter, a developer may specify a prompt for each parameter and the conditions for displaying the prompt. The system interprets the prompting directives to present, if conditions are met, a graphical user interface (GUI) control for receiving the parameter value.

Designation of Runtime Parameters

Runtime parameters provide a mechanism for a developer to modify the behavior of a graph based on external input at graph execution time (*i.e.*, runtime). In the preferred embodiment, these external values are provided by direct user input. However, these external values also may come from a number of different sources, including environment variables and command line parameters. The GDE 1002 generates the correct code to handle all of these situations as well as prompting the developer for test values when the graph is executed directly from the GDE. Using runtime parameters, a developer can, for example, explicitly declare that the path of an input file will be provided by an environment variable with a particular name; that environment variable then becomes a known part of the graph’s interface. Thus, there is a well-defined interface to such parameters. There is no need, for example, to read a generated shell script and search it for references to environment variables and command-line arguments to find the set of parameters that control the execution of a particular graph.

FIG. 11 is a block diagram of a typical graph 2000 having a rollup component 2002 and a sort component 2004 with designated runtime parameters. The runtime parameters (a key for the sort component 2004 and rules for the rollup component 2002) would be presented to a user in an interface 2006 for input. The following sections describe how to designate a runtime parameter, and create an integrated user interface for presentation of runtime parameters prompting for user input.

A runtime parameter may be designated or defined in a number of ways. One way is by use of a runtime parameters grid displayed in the GDE 1002.

FIG. 12 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 3000 that would be associated with a graph. A new runtime parameter is created by simply filling in the appropriate fields. An object associated with each runtime parameter is created in the repository 1004 and linked to all graph components that utilize the parameter. For example, if a sort key for a graph sort component is defined as a runtime parameter, an object representing the sort key parameter is stored in the repository 1004 and linked to the associated sort component. An alternative way of defining a runtime parameter is to specially flag an existing parameter of a graph component and make it “visible” (export it) to other components. A combination of these methods may be used. For example, when creating a component, a developer may designate a particular parameter of that component as a runtime parameter. The developer may then use a parameter grid to set default values and other characteristics of all of the runtime parameters for a graph, and define new runtime parameters.

When the graph is run, the parameters are processed to obtain values for each parameter from user input or from external programmatic sources (*e.g.*, command line parameters or environmental variables). In the illustrated embodiment, the runtime parameters grid 3000 includes the following fields:

Name 3002 – This field contains the name of the runtime parameter. “Score_threshold” is the example shown for a name.

Type 3004 – This field contains the type of value to be allowed in the runtime parameter. “Integer” is the example shown for a type. Supported types in the illustrated embodiment are:

- boolean – value can be either True or False;
- choice – value is one of a list of values;

- collator – a key parameter value;
- dataset – an external data file name and location;
- date – a date value;
- expression – an arithmetic, logical, and/or conditional expression (*e.g.*, a select expression);
- float – a floating point number;
- integer – an integer number;
- layout – a parallel or serial layout definition;
- record format – a record description or a file containing a record description;
- string – an arbitrary character string;
- transform – a transform description or a file containing a transform description.

Location (Loc) 3006 – This field is used with record format and transform types. It specifies whether the type field 3004 describes a file location or whether it contains an embedded description. Supported locations are:

- Embedded – the parameter will contain the record or transform description;
- Host – the parameter will contain a reference to a file on a host machine;
- Local – the parameter will contain a reference to a file on a local machine;
- Repository – the parameter will contain a reference a repository transform or record format.

Default Value 3008 – This field contains either (1) the default value for the runtime parameter which will be used if no other value is provided from an external programmatic source, or (2) a rule or expression describing how to derive the runtime value from user input or how to obtain that information interactively from the user executing the graph. In the latter case, a second default value field (not shown) may be used to provide a value for the runtime parameter if the user does not provide an input value. For types of “boolean” and “choice”, this field limits the user to the valid choices. For “layout” types, this field is read-only and displays

the currently defined layout definition. For all other types, this field preferably is a simple text editor into which the user may type a valid string.

Edit 3010 – Clicking on the edit space 3010 (or an icon; for example, a pencil icon) in a parameter row will bring up a more advanced edit window, which walks a user through the various options for editing the default value field 3008. In the illustrated embodiment, the following editors are available for their associated types:

- Single line edit – for integer, float, date and string types;
- Choice dialog – for boolean and choice types;
- Key Editor – for a collator type;
- File Browser – for a dataset type and for record format and transform types where the location is not embedded;
- Transform Editor – for a transform type with a location of Embedded;
- Record Format Editor – for a record format type with a location of Embedded;
- Expression Editor – for an expression type;
- Layout Editor – for a layout type.

The above editors are launched unless the Kind field value (see below) is “PL” (for Parameter Language). In this case the user is presented with an editor with which to define the rules for deriving or prompting for the parameter value at graph execution time.

Description 3012 – This is a free format field in which a developer describes the expected values of the runtime parameter. It is used as a prompt at runtime if the default value contains a rule for asking the user for an input value.

Kind 3014 – This field defines where a graph is to obtain the value for the associated parameter at graph execution time. Supported kind field 3014 values are:

- Environment – The value for the runtime parameter is expected to be found in an environment variable of the same name. If the environment variable is not defined, then the value in the default value field 3008 is used. If the parameter is required (*i.e.*, an

exported parameter), and the default value field 3008 is empty, then a runtime error will be generated and graph execution will stop.

- Positional – The value for the runtime parameter is expected at its relative position on a command line invoking the application. For example, if a runtime parameter is the third positional runtime parameter defined, then its parameter value will be expected as the third positional command line argument in an execution script. Any specified positional parameters must be provided and a runtime error will be generated if one is missing.
- Keyword – The value for the runtime parameter is expected as a keyword command line parameter. In the illustrated embodiment, keyword parameters are of the form:

$$- \langle \text{parameter name} \rangle \quad \langle \text{parameter value} \rangle.$$

Keyword parameters are optional and a runtime error will only be generated if the keyword parameter is not provided and the default value field 3008 is blank and a corresponding exported parameter is required.

- Fixed – The runtime value for the parameter is always the default value. This is useful for sharing a constant value between two or more runtime parameters.
- PL – The default value of the runtime parameter contains a PL expression which will be interpreted at graph execution to either derive the value of the runtime parameter from other parameters or prompt the user for additional input. The Component Description Language that is selected for use with any particular embodiment of the invention may be any suitable scripting language, such as the publicly available object-oriented scripting language “Python”. Such scripts can construct metadata (types and transforms) under program control, and perform conditional tests, comparisons, data transformations, arithmetic and logical operations, string and list manipulations, and other functions on user input, externally programmatically supplied input, and other runtime parameters to generate a final value for any runtime parameter.

In the illustrated embodiment, a useful convention for referencing a runtime parameter that has been created directly on the runtime parameters grid 3000 is to simply enter the parameter name preceded by the dollar sign “\$”. For example, *\$key* references a runtime variable named *key*. In the illustrated embodiment, new runtime parameters default to a type of “string”

and a default kind based on the value in the advanced options dialog for the default runtime kind (the default runtime kind is “Environment”).

Because runtime parameter values can be determined at runtime, and PL scripts can provide conditional testing, “conditional” runtime parameters can be created. A conditional
5 runtime parameter causes a prompt to be generated for user input only if all of the conditions for the parameter – determined at runtime – are enabling. Thus, for example, if a user responds to a first prompt requesting whether a data set is to be sorted with “NO”, a second, conditional prompt that requests a sort key need not be displayed.

Thus, during a design phase (“design time”), a developer designates a particular
10 parameter of a graph component as a “runtime” parameter. An object associated with that graph component is then stored with the relevant parameter data (*e.g.*, the types of information from the parameters grid 3000 of FIG. 11).

FIG. 13 is a flowchart that summarizes the process of using a runtime parameter. During runtime, parameter objects corresponding to an application to be executed are retrieved (*e.g.*,
15 from a repository) (STEP 4000). A determination is made for each such object as to whether user input is indicated (STEP 4002). If so, a determination is made as to whether any condition for display of the prompt has been met (STEP 4003), which may include evaluation of user input to prior prompts. If not, a default value is used (STEP 4008). Alternatively, the parameter value may not be needed (*e.g.*, a sort key would not be needed if the user did not choose to activate a
20 sort function), and thus may be ignored. Otherwise, a prompt is generated for user input (STEP 4004).

If the user does not input a value for a particular parameter (STEP 4006), the default value for the parameter may be selected (STEP 4008). Alternatively, an error condition may be raised to indicate the lack of user input. In any event (assuming no error condition because of a
25 lack of user input), a determination is made of the final value for the parameter, taking into account transformations of the input and dependencies and conditions based on other parameters (STEP 4010).

If a determination is made that user input is not indicated for a particular parameter (STEP 4002), a determination is then made as to whether the parameter value is to be externally
30 supplied programmatically, such as by an environment variable or a command line parameter (STEP 4012). If not, the default value for the parameter is selected (STEP 4014). Alternatively,

an error condition may be raised to indicate the lack of available input of the specified type. In any event (assuming no error condition because of a lack of external input), a determination is made of the final value for the parameter, taking into account transformations of the input and dependencies and conditions based on other parameters (STEP 4010).

5 Once the final parameter values are determined, as an optional step all conditional components (discussed below) can be removed either completely or replaced by flows (*i.e.*, a graph link or edge), according to the specified conditions and the rules outlined above (STEP 4016). Once the operational graph structure is finalized and the final parameter values are determined, the graph is executed in conventional fashion (STEP 4018).

10 *Test Values*

In order to support a developer during the creation and testing of graphs with runtime parameters, the preferred embodiment of the GDE 1002 also supports test values for runtime parameters. When a developer runs a graph with runtime parameters or wants to view the underlying code affecting a graph component, the GDE 1002 displays an associated test
15 parameters grid where the user can enter new test values for one or more runtime parameters. Preferably, the last set of test values used is remembered and saved with the graph.

For each runtime parameter, the developer enters a desired test value in a test value column. An edit field may be associated with each test value column. The test value field and edit field behave the same as the default value field and edit field in the runtime parameters grid
20 2000 except when the parameter kind is PL.

If a PL expression indicates that the user is to be prompted for a value for a particular runtime parameter, then the test value field and the edit behavior are based on the interpretation of the associated PL expression. If the PL expression simply derives a value based on other input, then in normal mode the runtime parameter is not visible in the test values grid.

25 *Specifying How Runtime Parameters Get Their Values*

After a parameter has been designated as a runtime parameter, a corresponding object is created in the repository 1004. If the runtime parameter has a kind field 2014 value of “PL”, the default value field 3008 for the parameter includes a `prompt_for` pseudo-function with the following preferred form:

`prompt_for "prompt-kind[modifiers]" options ...`

As indicated above, the `prompt_for` pseudo-function may be part of a conditional expression that determines whether a prompt is to be displayed based on prior input.

For such objects, a user interface is used to present direct entry runtime parameters to a user. In the preferred embodiment, the Web Interface 1008 provides this function. In particular, during runtime, each `prompt_for` pseudo-function of each runtime parameter object is parsed by the Web Interface 1008 to generate a web page (*e.g.*, in HTML) having a corresponding user prompt. (Alternatively, such web pages can be generated before runtime and simply presented at runtime. However, runtime generation of such web pages provides greater flexibility. In particular, the contents of a page can depend on prior user input.) The Web Interface 1008 is used in conjunction with a conventional web browser that can display such web pages and receive user input.

The `prompt_for` pseudo-function indicates to the Web Interface 1008 how to prompt for a parameter value. In particular, the `prompt-kind` parameter, a string constant, indicates what kind of user interface (UI) element to present (text box, dropdown list, *etc.*). The *modifiers* part of the string, a comma-separated list of keywords, provides some options common for various kinds of prompts. In the illustrated embodiment, space is not significant within the modifiers string. Modifier keywords are interpreted as follows:

- The keyword `in place` declares that the element should be presented directly at the summary level user interface for an application, allowing the value to be supplied without “drilling in” to a lower level. If `in place` is not specified, a simple “edit” button is presented at the summary level interface which will takes a user to another page to supply the parameter value.
- The keyword `blank ok` declares that a user need not supply a value; the application will deal with the default value in a reasonable way. If `blank ok` is not specified, then the user will not be able to execute the application without supplying some value.

Following are some examples of `prompt_for` calls with different kinds of modifiers:

```
${prompt_for "text,inplace"}
```

```
{prompt_for "filter, in place", $input_type}
```

```
{prompt_for "radio, blankok, in place", ${list 1, 2, 3}}
```

The remainder of this section lists a variety of `prompt-kinds` and their corresponding options and explains how each would appear in a web page generated by the Web Interface

5 1008.

`text [size]` – Presents a conventional single-line text box *size* characters wide (if *size* is not supplied it defaults to the browser’s default size for text boxes).

`radio choice-list [description-list]` – Presents a conventional “choose one” prompt in the form of a set of radio buttons, one button for each element of the *choice-list*. If *description-list* is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the *choice-list*.

10

`radioplus choice-list [description-list]` – Like `radio`, but presents an additional button next to a text box, to allow a user to choose a “write-in” value not in the *choice-list*.

`checkbox choice-list [description-list]` – Presents a conventional “choose zero or more” prompt in the form of a set of check boxes, one button for each element of the *choice-list*. If *description-list* is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the *choice-list*.

15

`dropdown choice-list [description-list, size]` – Presents a conventional “choose one” prompt in the form of a dropdown list for the elements of the *choice-list*. If *description-list* is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the *choice-list*. If *size* is supplied, that many choices will be visible at once; otherwise, only one will be visible.

20

`multidropdown choice-list [description-list, size]` – Presents a conventional “choose zero or more” prompt in the form of a dropdown list for the elements of the *choice-list*. If *description-list* is supplied, each choice is labeled with the corresponding description; otherwise, the choices are labeled with the string form of the corresponding item from the *choice-list*. If *size* is supplied, that many choices will be visible at once; otherwise, the browser’s default number of items is shown.

25

key type-obj [size] – Presents a prompt for a key (also known as a collator) made up of fields from the given *type-obj*. The key can have as many as *size* parts, which defaults to the number of fields in *type-obj*.

FIG. 14 is a diagram of one embodiment of a graphical dialog 5000 generated by the *key* prompt. Following is an example of the script text for a 3-entry *key* prompt, where the file */datasets/fixed* defines the contents of the available keys shown in the drop down boxes 5002:

```
${prompt_for "key", ${dataset_type "/datasets/fixed"},3}
```

In the illustrated embodiment, the normal collation order is ascending, but a user can select a descending collation order for a key by checking an associated check box 5004.

filter type-obj – Presents a prompt for a filter expression made up of conditions on each field of the given *type-obj*. The *blank ok* modifier has no effect for filters; a blank filter yields a “True” expression.

FIG. 15 is a diagram of one embodiment of a graphical dialog 6000 generated by the *filter* prompt. The available field names 6002 associated with each expression text edit box 6004 are defined by *type-obj*. Comparison values are entered into the text edit boxes 6004, and a comparison operator (*e.g.*, equal, greater than, less than or equal to) is selected from a corresponding dropdown list control 6006.

flexifilter type-obj – Similar to the *filter* prompt, but presents a prompt for a filter expression made up of conditions on each field of the given *type-obj* where the field name on each line is selectable from a dropdown list. This permits using the same field for multiple conditions (*e.g.*, *field STATE = MA OR field STATE = CA*).

rollup type-obj key [size] – Presents a prompt for a rollup computation based on the fields of the given *type-obj* being rolled up by the given key. The rollup can have as many as *size* rules, which defaults to the number of fields in *type-obj*. The *blank ok* modifier has no effect for rollups; a blank rollup yields a package that provides just the key value for each group.

FIG. 16 is a diagram of one embodiment of a graphical dialog 7000 generated by the *rollup* prompt. In the illustrated embodiment, a column of dropdown boxes 7002 defines the available rollup computation functions (*e.g.*, sum, minimum, maximum). The available field names 7004 associated with each computation are defined by *type-obj*. Each rollup rule has an associated text edit box 7006 for user definition of a desired expression, a “where” text edit box 7008 for defining (through a boolean expression) criteria under which the source value will take

part in the computation, and an output field text edit box 7010 for designating a field that will receive the computation result. In cases where it can be unambiguously derived, the name of the output field need not be specified.

5 `reformat type-obj [size]` – Presents a prompt for a reformat computation based on the fields of the given *type-obj*. The reformat can have as many as *size* rules, which defaults to the number of fields in *type-obj*.

FIG. 17 is a diagram of one embodiment of a graphical dialog 8000 generated by the `reformat` prompt. In the illustrated embodiment, the `reformat` prompt includes a section 8002 for simply copying input fields to like-named output fields (either selected/deselected individually using checkbox controls or collectively by using Select All or Select None buttons).
10 A second section of the prompt includes a column of text edit boxes 8004 that allow definition of reformatting expressions (*e.g.*, `total = revenue_1 - revenue_2`). Each rule has an associated output field text edit box 806 for designating a field that will receive the reformatted result.

15 `outputspec` – Presents a prompt for an output dataset specification. The displayed control includes a dropdown control for presenting available format options, and a text edit box for entering the name of a specific instance of the output dataset. The `blank ok` modifier has no effect for output dataset specifications.

20 `fpath starting-point` – Presents a prompt for a file path. The prompt is essentially a text box, but has a “Browse” button next to it that will cause a popup window to appear for browsing for a file path. If the text box is non-blank, then it will be used as the starting point for the browsing operation; if it is blank, the *starting-point* argument is used.

25 `rpath starting-point` – Presents a prompt for a repository path. The prompt is essentially a text box, but has a “Browse” button next to it that will cause a popup window to appear for browsing. If the text box is non-blank, then it will be used as the starting point for the browsing operation; if it is blank, the *starting-point* argument is used.

`radiofpath choice-list [description-list]` – Like `radioplus`, but presents an `fpath`-style box-plus-browse-button in the “write-in” slot.

30 `radiorpath choice-list [description-list]` – Like `radioplus`, but presents an `rpath`-style box-plus-browse-button in the “write-in” slot.

Conditional Components

Some implementations include a conditional components mechanism that permits changes to the structure of the components and flows of a graph based on parameter values and computed metadata. Each component of a graph has a condition which controls whether or not that component will appear in the graph at runtime. The condition can be computed directly or indirectly through runtime parameters. Conditional components can be used for a variety of purposes, such as to optimize or specialize graphs. For optimization, an application might omit processing of certain datasets if values from them will not be used, thus allowing the graph to run more efficiently. For specialization, an application might condition the production of several different output datasets based on the level of detail desired, or allow execution of one of several optional portions of a graph.

FIG. 18A is a block diagram of a first graph in which a MergeJoin component 9000 joins data from files A and B and outputs the result to an output file 9002.

FIG. 18B is a block diagram of a second graph in which a Rollup component 9004 aggregates data from file A and outputs the result to an output file 9002.

FIG. 18C is a block diagram of a graph in which a MergeJoin component 9006 joins data from files A and B, and a Rollup component 9008 aggregates the resulting data and outputs a final result to an output file 9002. Using conditional components, these three graphs can be combined into a single graph that initially looks like the graph of FIG. 18C, but the exact structure of which is not determined until runtime. By setting appropriate conditions, the Rollup component 9008 can be replaced by a connection (flow), resulting in a runtime graph similar to the graph of FIG. 18A. Similarly, by setting appropriate conditions, the MergeJoin component 9006 can be replaced by a connection (flow) to file A, resulting in a runtime graph similar to the graph of FIG. 18B.

In the illustrated embodiment, a conditional component can be any graph component that defines a vertex (*i.e.*, a dataset component such as an input/output file, a processing component such as a reformat or sort component, or other graphs, known as subgraphs). In the preferred embodiment, a conditional component is controlled by two special parameters: a *Condition* and a *Condition-interpretation*. A Condition is a boolean expression or value whose evaluation is deferred until runtime. In the illustrated embodiment, the values “false” and “0” specify a false condition, all other values (including empty) indicate a true condition. A Condition-interpretation

parameter has two allowed mutually exclusive values: *Remove Completely* and *Replace With Flow*.

FIG. 19 is a diagram of one embodiment of a graphical dialog 1900 presenting a Condition 1902 having a Condition-interpretation control 1904. The Condition-interpretation control 1904 allows selection of either a *Remove Completely* interpretation 1906 or a *Replace With Flow* interpretation 1908.

Remove Completely: With this interpretation, if the Condition is met, the component and all of its connected flows (*i.e.*, graph links or edges) are to be removed from the graph. An active *Remove Completely* condition functionally removes the component and all its directly connected flows from a graph. *Remove Completely* conditions can be used on any component.

A conditional component that is removed from a graph can “poison” other connected components that depend on the presence of the conditional component, causing their removal.

FIG. 20 is a diagram of a graph 2100 showing a situation in which such poisoning arises. If the condition on the Input File component 2102 indicates removal and its corresponding condition-interpretation is *Remove Completely*, then both the Input File component 2102 and its connected flow are removed from the graph 2100. This in turn poisons the Sort component 2104, causing it to be removed because its input is a required input port, but there are no longer any data flows connected to it. This in turn poisons the Rollup component 2106, causing it to be removed because its input is a required input port, but there are no longer any data flows connected to it. The only thing that stops this “poison of disappearance” is connection to an optional or counted port of a downstream component. Thus, the entire sort-rollup graph branch 2108 is effectively removed from the graph 2100 when the condition on the Input File component 2102 indicates removal. The result in FIG. 20 is that the nominally 3-input Join component 2110 of the original graph structure becomes a 2-input Join component at runtime.

In one implementation, the detailed semantics of poisoning (also known as “implied conditions”) are as follows:

- If a component has a required port and there are no live flows connected to it, the component and all flows connected to it are removed from the graph.
- If a component is removed completely from a graph, then all flows connected to its ports are removed from the graph.

- If a component is replaced with a flow, then all flows connected to all ports other than that component's *designated input port* and *designated output port* are removed from the graph.
- If a required indexed port has no live flows connected to it, then for each corresponding optional indexed port with the same index, any flows connected to that corresponding port are removed from the graph.

There are some surprising consequences of these rules. For example, a component with only optional ports can never be removed because of poisoning. Therefore, it must be explicitly removed if desired.

FIG. 21 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Remove Completely* conditional component. If the Condition-interpretation is *Remove Completely* and the Condition is not met (STEP 2200), then the conditional COMPONENT is not removed from the graph (STEP 2202). If the Condition is met (Step 2200), then the conditional component is removed from the graph, along with all flows connected to that component (STEP 2204). All "poisoned" components and flows are then removed from the graph, in accordance with the rules set forth above (STEP 2206).

Replace With Flow: With this interpretation, if the Condition is met, the component is to be replaced with a flow (*i.e.*, a graph edge). A *Replace With Flow* condition-interpretation needs additional information. Referring to FIG. 19, the user designates an input port 1910 (or a family of counted ports) and an output port 1912 (or a family of counted ports) through which to make connections when the component is removed from a graph. By default, if there is exactly one required input port or counted port, and exactly one required output port or counted port, those are the designated flow-through connection ports (termed the *designated input port* and the *designated output port*, respectively). A required port is one that requires at least one flow to be connected.

FIG. 22 is a flowchart that summarizes the process of runtime preparation of a graph that includes a *Replace With Flow* conditional component for a particular embodiment of the invention. Because of the dependency of some components on certain available inputs and outputs in the illustrated embodiment (which is based on components available in the CO>OPERATING SYSTEM®), several rules apply to this implementation and use of a *Replace With Flow* condition:

- If the Condition-interpretation is *Replace with Flow* and the Condition is not met (STEP 3300), then the conditional component is not removed from the graph (STEP 3302).
- A component with a designated input port and a designated output port can be replaced with a flow only if there is exactly one live straight flow connected to its designated input port, and exactly one live straight flow connected to its designated output port (a “live” flow is a flow that has not been removed at run-time) (STEP 3304). If so, the component itself is removed from the graph, and the straight live flow connected to its designated input port and the straight live flow connected to its designated output port are linked together (STEP 3306). Any other flows directly linked to the removed component’s other ports (*i.e.*, any ports other than the specially designated input and output ports) are removed from the graph. Any “poisoned” components and flows that were connected to the removed component are removed, as described above (STEP 3308).
- If a component with a *Replace With Flow* condition has live flows attached to more than one *designated input port* in a family of counted inputs (STEP 3310), then it is not removed from a graph, because the component is needed to make the graph valid (STEP 3312).
- Components that have live fan-in-flows on required inputs require special handling. A “live fan-in flow” means either the component has a live fan-in or all-to-all flow connected to a required input port, or it has more than one live straight flow connected to a single required input port. For such components, interpreting a *Replace With Flow* condition should replace the conditional component with a gather component which gathers all of live input flows (STEP 3314). Any “poisoned” flows and components that were connected to the replaced component are then removed, as described above (STEP 3316).

Aspects of Metadata Propagation

Metadata for a graph can be supplied, for example, by a graph developer, by a graph user, or by propagation from another portion of the graph. Various kinds of metadata can be propagated, including metadata associated with the data or computations on the data such as: a record format for a port (e.g., sequence of fields and data types of records flowing into or out of a

port), sortedness, compression method, character set, binary representation (big-endian, little-endian), partitioning, what computing resources (e.g., processor, temporary disk space) the component may use, data transformations, and amounts of memory the component may use. Various aspects of graph construction can affect the propagation of metadata. Two of these
5 aspects are described below.

Propagation After Component Removal

In some implementations, when a flow is generated after the removal of a graph component, a choice must be made as to how metadata defining the data in such flow should propagate in the revised graph. Metadata may be available from either end of the flow. In some
10 implementations, the metadata from the upstream end of the flow is preferred.

If the upstream end of the flow is a removed component (or a component that has been replaced by a gather component), then the GDE 1002 finds metadata for the flow by “walking” upstream in the graph until it finds a component that has not been removed. The metadata exposed by that upstream component is used to define the characteristics of the data for the
15 generated flow.

Propagation of Transformed Metadata

As described above, metadata can propagate even over transforming internal data paths by allowing metadata associated with a port to be specified as a function of one or more parameters, including metadata for another port. For example, FIG. 32A shows a graph 31300
20 that computes a join operation on data from data set 31302 and data set 31304. In this example, a graph developer supplies metadata at output ports of the data sets. This metadata is then propagated to a “smart join” component 31306 that computes a join operation on the records of the input data sets. For example, metadata propagates from output port 31308 to input port 31310. The metadata is then transformed by the “smart join” component 31306 and propagated
25 to an input port 31317 of a filter component 31318 from an output port 31316 of the “smart join” component 31306.

FIG. 32B shows a sub-graph implementing the “smart join” component 31306. The component 31306 uses a *key_field* parameter whose value represents the key field of the join operation performed by a join component 31350. The component 31306 also uses the *key_field*

parameter as a condition for including conditional sort components 31354 and 31356. If the records flowing into the input port 31310 are already sorted on the *key_field*, then the sort component 31354 is conditioned out. Similarly, if the records flowing into the input port 31314 are already sorted on the *key_field*, then the sort component 31356 is conditioned out. If either
 5 flow of input records are not already sorted on the *key_field*, then the sort components 31354 and 31356 sort the records before they flow into the join component 31350.

To enable propagation of transformed metadata through this “smart join” component, a graph developer defines the metadata (e.g., metadata for describing the fields) for the output port 31316 of the “smart join” component 31306 as a function of metadata for the first input port
 10 31310 *input0.metadata*, metadata for the second input port 31314 *input1.metadata*, and the key field parameter *key_field*:

output.metadata = metadata_join(key_field, input0.metadata, input1.metadata)

The output port metadata is determined by binding the function arguments to values (with respect to the appropriate context) and performing the function *metadata_join* on the results. In
 15 this example, since metadata for the ports 31310 and 31314 are undefined, propagated metadata are bound to the metadata parameters *input0.metadata* and *input1.metadata*. A user supplies metadata for the output port 31308 that specifies fields “A” and “B” for records flowing from port 31308 to input port 31310 of the “smart join” component 31306. The user also supplies metadata for the output port 31312 that specifies fields “A” and “C” for records flowing from
 20 port 31312 to input port 31314 of the “smart join” component 31306. This user-supplied metadata propagates to the ports 31310 and 31314. The key field for the join operation is field A, so the “formal parameter” *key_field* is bound to the value “A.”

The function *metadata_join* determines the output metadata by first determining whether the value of the *key_field* parameter is a member of both sets of fields specified by
 25 *input0.metadata* and *input1.metadata*. If so, the output metadata is the union of the two sets of fields. If not, the output metadata indicates an empty set of fields.

After the metadata propagates to the input ports of the “smart join” component 31306 (or is otherwise supplied, for example, by a user), the transformed metadata for the output port of the “smart join” component 31306 includes fields A, B and C. This transformed metadata can then
 30 be propagated to other components. In this example, the transformed metadata propagates to the filter component 31318.

Metadata, whether supplied by a user or propagated between ports, can be displayed to the user. For example, the user can use an input device (e.g., a mouse) to select a portion of a component for which to view metadata values. The metadata propagation can also be triggered in response to such a user selection.

5 *Exemplary metadata propagation process*

FIG. 33 shows a flowchart for an exemplary metadata propagation process 22400. The process 22400 can be performed, for example, each time there is a change in a graph, in response to a user action, and/or just before the graph is run. The process 22400 generates 22402 a worklist with each component in the graph ordered according to the partial ordering determined by the flows (e.g., component A comes before component B if there is a flow from component A to component B). Where flows do not determine a unique order between two components, alphabetic order of component labels may be used as a tie-breaker. This provides a stable ordering for the components in the worklist (assuming the component labels are unique). If the propagation process 22400 is repeated for a graph (e.g., after the addition of a new component), the new worklist preserves the same order between components previously in the worklist.

The process 22400 starts at the beginning of the worklist and, for each component in the worklist, the process 22400 propagates metadata internally 22404 within the component (e.g., from an input port to an output port, or from an output port to an input port) based on a specification of the sub-graph implementing the component (e.g., an data flow in the sub-graph). This internal metadata propagation includes transferring metadata untransformed between ports on either end of an non-transforming data path. Internal metadata propagation also includes deriving metadata for a port that has a metadata definition that refers to parameters of the graph and/or metadata for other port(s), as described above. When the process 22400 encounters such a metadata definition, the process 22400 evaluates any parameters whose values are needed to derive the metadata.

After performing internal metadata propagation for a component on the worklist, the process 22400 propagates metadata externally 22406 from each port of the component that has metadata to a port of a related component that does not have metadata. Any component that acquires metadata by this external propagation is moved 22408 to the end of the worklist. The process 22400 terminates 22410 after the last component on the worklist is processed.

One type of relationship between components that supports this type of external metadata propagation is a data flow link between ports of two components (e.g., from an input port to an output port, or from an output port to an input port).

Another type of relationship between components that supports this type of external metadata propagation is a link indicating that metadata for one port may also be used for another port. This type of “metadata link” does not necessarily correspond to a data flow link. For example, a port can have a metadata link to metadata in a graph that is not associated with any port in particular.

Runtime Parameters in Componentized Sub-Graphs

Before a sub-graph is “componentized” to be used as a component in another graph, various characteristics of the component are defined, such as the input and/or output ports of the component. For a sub-graph that includes components with runtime parameters, a prompting order for the runtime parameters should be selected. Since components in a graph are not necessarily sequentially ordered, there can be multiple possible global orderings of the runtime parameters for prompting a user. Some of the global orderings are not as consistent with the original orderings associated with each component. It is useful to generate a global ordering for prompting that preserves as much as possible the orderings of the parameters in each component, while reordering when appropriate to take dependencies into account. For example, a component may order a prompt asking “what data would you to process?” before a prompt asking “where would you like to store the processed data?” Even though it may be possible to provide the prompts in either order, it may be desirable to provide the prompts in this order.

Since it may be necessary to evaluate non-prompted runtime parameters in the process of evaluating prompted runtime parameters, the prompting order is obtained from an evaluation order for all of the runtime parameters. One approach for determining an evaluation order for the runtime parameters of a graph (including parameters for the graph that are not associated with any component) includes performing a topological sort based on one or more directed acyclic graphs representing dependencies among the parameters. However, some topological sort algorithms may reorder parameters unnecessarily, resulting in an undesirable prompting order for runtime parameters.

Sorting Example 1

In a first example, a parameter sorting process provides an initial list of parameters for parameters of two graph components: Component I, and Component II connected to Component I. In this example, the parameters have only “intra-component” dependencies. That is,
 5 parameters of a component depend only on other parameters in the same component. The parameters are defined as follows.

Component I includes the following parameters:

$x = \{\text{prompt_for "text"}\}$

$y = x + \{\text{prompt_for "text"}\}$

10 $z = x + y + \{\text{prompt_for "text"}\}$

$q = \{\text{prompt_for "text"}\}$

Component II includes the following parameters:

$a = \{\text{prompt_for "text"}\}$

$b = a + \{\text{prompt_for "text"}\}$

15 $c = \{\text{prompt_for "text"}\}$

The order in which the parameters are listed define a desired order in which to prompt a user for values. The initial list of parameters maintains this “initial ordering” for each component. An “ordinal” is assigned to each parameter to indicate that parameter’s place in the initial ordering. The following table lists the parameters in this initial ordering.

Parameter	Ordinal	Dependencies
x	0	
y	1	x
z	2	x, y
q	3	
a	4	
b	5	a
c	6	

20

The “dependencies” column indicates other parameters on which the listed parameter depends. The dependencies impose an ordering constraint on the evaluation of the parameters: a parameter needs to be defined before it is used (e.g., referenced) by another parameter.

A “common topological sort” algorithm passes through the list transferring parameters with zero dependencies into an ordered output list on each pass. After each pass, any transferred parameters are removed from the dependencies column. This process repeats until all parameters have been transferred. The order of parameters in the output list represents the “final ordering”
 5 such that parameters that depend on other parameters are evaluated after those other parameters have been evaluated.

In this example, on the first pass, the parameters x, q, a and c are transferred into the output list. On the second pass, the parameters y and b are transferred into the output list. On the third and final pass, parameter z is transferred into the output list. Thus, the final ordering for the
 10 parameters is: x, q, a, c, y, b, z. While this ordering does satisfy the ordering constraint imposed by the parameter dependencies, it unnecessarily reorders the parameters. In this example, the initial ordering also satisfies the ordering constraint imposed by the parameter dependencies.

Other approaches for determining an evaluation order for the parameters of a graph that satisfies the ordering constraint do respect the initial ordering. For example, some approaches
 15 order the parameters to satisfy the ordering constraint, choosing the ordering according to a criterion based on the initial ordering. The criterion can include any of a variety of criteria that give preference to keeping the order close to the initial ordering (e.g., minimize a metric based on changes to the initial ordering). In some cases, there may not be a unique “best” ordering, since multiple orderings may satisfy a given criterion equally well according to the criterion.

An example of an approach that respects the initial ordering is a “modified topological sort” approach. In this approach, the criterion based on the initial ordering is to minimize the number of parameters that are transferred from the initial list before a preceding parameter that does not depend on any untransferred parameter is transferred. In other words, the “modified topological sort” removes a transferred parameter from the dependencies column before
 20 transferring the next parameter with zero dependencies. For the example above, the “modified topological sort” approach generates a final ordering that is the same as the initial ordering: x, y, z, q, a, b, c.

Modified Topological Sort Process Respecting Initial Ordering

Pseudocode is given below for two exemplary “modified topological sort” processes that
 30 both respect initial ordering as determined by an assigned ordinal for each parameter. The second

process includes an optimization to improve time efficiency for some cases. The processes manipulate data structures generated from input data for the parameters.

Assuming there are N parameters to be ordered, the input data includes a list of N triples consisting of a unique parameter name, a set of parameters upon which the named parameter depends (called a “dependency set”) and an optional attribute data object storing information related to the named parameter.

Associated with this input data are one or more directed acyclic graphs that represent the dependencies among the parameters, called “dependency graphs.” Each unique parameter name corresponds to a node in a dependency graph, and the associated dependency set corresponds to a set of links from other nodes to that node. So a link points from a first node for a first parameter to a second node for a second parameter that depends on the first parameter. Alternatively, the correspondence between the link direction and parameter dependency could be reversed.

An output data structure *result_list* includes a list of the N parameters from the input data reordered (if necessary) so that a parameter is evaluated before it is used for evaluating another parameter while giving preference to keeping the order close to the initial ordering. To generate the output data structure *result_list*, the processes “eliminate” parameters by transferring parameters one at a time from a working data structure *param_list* to the output data structure *result_list*. The output data structure is complete after all parameters have been eliminated.

A first “modified topological sort” process includes two phases. In the first phase, the process builds working data structures based on the input data for use in generating the sorted output data structure. In the second phase, the process iteratively sorts and eliminates parameters according to the dependency constraint represented by these working data structures.

Some of the working data structures that the process builds in the first phase are dictionaries, which are data structures based on hashing. Items in dictionaries can be accessed effectively in $O(\log N)$ time. The following exemplary data structures are built in the first phase:

param_list[index]: an ordered list of non-eliminated parameter names, indexed by a number *index* (where *index* = 0 corresponds to the first item in the list). This data structure is “dynamic” (i.e., changes during the execution of the process). The list is indexed by position, such that if an item is removed from the middle of the list, then the index of items after the removed item are shifted accordingly.

n_dependencies_dict[name]: a dictionary keyed by a parameter name (*name*), whose entries contain the number of parameters on which the keyed parameter depends. This dictionary is dynamic.

5 *dependers_dict[name]*: a dictionary keyed by a parameter name (*name*), whose entries are dictionaries (also keyed by parameter name), representing the set of parameters that depend on the keyed parameter. This dictionary is “static” (i.e., does not change during execution of the process).

10 *order_dict[name]*: a dictionary keyed by a parameter name (*name*), storing the ordinal position, an integer ranging from 0 to $N-1$, of the parameter in the initial ordering. This dictionary is static.

attribute_dict[name]: a dictionary keyed by a parameter name (*name*), storing the optional attribute data object for the keyed parameter. This dictionary is static.

15 *result_list[index]*: an ordered list of parameter names and attributes representing the output of the process, indexed by a number *index* (where *index* = 0 corresponds to the first item in the list). This data structure is initially empty. This data structure is dynamic.

For the purposes of analyzing the time efficiency of the processes, the average “degree” (or number of links from a node) of the dependency graphs is assumed to be z . Building these data structures take $O(N)$ time, except for *n_dependencies_dict* and *dependers_dict*, which take $O(N*z)$ time.

20 In the second phase, the process sorts the parameters in the *param_list* data structure according to a sort criterion *by_n_deps_and_order* that orders parameters first by the number of non-eliminated parameters on which they depend (i.e., by their value of *n_dependencies_dict*), from lowest to highest, and then by their ordinal (i.e., by their value of *order_dict*), from lowest to highest. The process then eliminates the first parameter in the sorted *param_list*. The value of *n_dependencies_dict* for this parameter should be zero. (If the value of *n_dependencies_dict* for the first parameter in the sorted *param_list* is not zero, then an error is flagged.)

25 To eliminate a parameter, the process appends it to *result_list* (along with any corresponding attributes) and decrements the dependency count (i.e., the value of *n_dependencies_dict*) of all of its dependers (i.e., parameters in *dependers_dict*) by one. Finally, 30 the parameter is deleted from *param_list*. This sorting and eliminating of the resulting first parameter is repeated until all parameters have been eliminated.

The following is a pseudocode definition for an *eliminate* procedure:

```
def eliminate(list, index):
    result_list.append( (list[index], attribute_dict[list[index]]) )
    for depender in dependers_dict[list[index]]:
5       n_dependencies_dict[depender] = n_dependencies_dict[depender] - 1
    delete list[index]
```

The arguments of the *eliminate* procedure are *list* (whose value is, e.g., *param_list*) and *index*. The function *result_list.append* appends the indicated *list* item at position *index* along with its associated attribute to *result_list*. Then, the procedure decrements the value of
10 *n_dependencies_dict* for each parameter *depender* that is a member of the *dependers_dict* data structure, keyed on the parameter being eliminated. Then, the procedure deletes the parameter from *list*. The run time for the *eliminate* procedure is $O(z\log N)$.

The following is pseudocode for a sort/eliminate loop for the first “modified topological sort” process:

```
15 while param_list is not empty:
    param_list.sort(by_n_deps_and_order)
    while param_list is not empty and n_dependencies_dict[param_list[0]] == 0:
        eliminate(param_list, 0)
        param_list.sort(by_n_deps_and_order)
20 if param_list is not empty and n_dependencies_dict[param_list[0]] > 0:
    delete param_list[0]
    < record a circularity error and continue >
```

The process first performs an initial sorting of *param_list* using the function *param_list.sort(by_n_deps_and_order)* that orders parameters of *param_list* according to the sort
25 criterion *by_n_deps_and_order* described above. The process then performs the *eliminate* procedure followed by another sorting of *param_list* until *param_list* is empty. The process checks to make sure that the number of dependencies for the first parameter (with *index* = 0) in *param_list* is zero. If not, the process removes the parameter, records a circularity error, and continues. The sort takes $O(M\log N)$ and the loop range is *N*, so the estimate for the overall run
30 time for the loop is $O(N^2\log N)$.

A second “modified topological sort” process takes advantage of the cases in which the dependency graphs are sparse, such that $z \ll N$. After one initial sort, the process can maintain the sortedness of a list *candidates* of parameters that do not depend on any other parameters. This reduces this expected run time as described below.

5 The following is pseudocode for the second “modified topological sort” process:

```

    parm_list.sort(by_n_deps_and_order)
    while parm_list is not empty:
        # section 1
        candidates = []
10      for p in parm_list:
            if n_dependencies_dict[p] == 0:
                candidates.append(p)
        # section 2
        while candidates is not empty and n_dependencies_dict[candidates[0]] == 0:
15          this_parm = candidates[0]
            eliminate(candidates, 0)
            idx = parm_list.index(this_parm)
            delete parm_list[idx]
            tmp = get_new(this_parm)
20          candidates = merge(candidates, tmp)
        # section 3
        if parm_list is not empty:
            parm_list.sort(by_n_deps_and_order)
            if n_dependencies_dict[parm_list[0]] > 0:
25          delete parm_list[0]
                < record a circularity error and continue >

```

The process first performs an initial sorting of *param_list* using the function *parm_list.sort(by_n_deps_and_order)* that orders parameters of *param_list* according to the sort criterion *by_n_deps_and_order* described above. The process then performs a loop having three
30 sections (labeled “# section 1,” “# section 2,” and “# section 3”).

In section 1, the process builds a *candidates* list that contains only parameters with zero dependencies. The process scans all of the parameters in *parm_list* and appends them to *candidates*, preserving their relative ordering.

In section 2, the process performs a loop in which parameters from *candidates* are
 5 eliminated and new parameters are merged into *candidates*. The first parameter in *candidates*, saved as *this_parm*, is eliminated from *candidates* and deleted from *parm_list*. A function *get_new(this_parm)* returns a list of names of parameters that are members of *dependers_dict* for the newly eliminated *this_parm* and have zero dependencies left. These parameters, representing
 10 parameters that have had their last dependency removed, are then sorted according to *by_n_deps_and_order* (to ensure they ordered according to their respective ordinals) and merged into *candidates*. Thus, the *candidates* list remains a list of zero-dependency parameters sorted by ordinal.

Section 3 is only entered if there is a “circularity error” caused, for example, when two parameters are defined in terms of each other. In this case, the process sorts *parm_list* again, and
 15 if the first parameter in *parm_list* has nonzero dependencies it is deleted and the loop repeats with section 1.

Assuming there are no circularity errors, the *N*-parameter list *parm_list* is sorted only at the beginning, resulting in a sorting time of $O(N \log N)$. Thereafter, sorting only occurs on the much smaller list of newly generated zero-dependency parameters resulting from eliminating the
 20 parameter at the head of the *candidates* list. The size of this list is less than *z* (on average), resulting in a sorting time of $O(z \log z)$ and a merging time of $O(z)$. Thus, one iteration of the loop is $O(z \log z)$ and the overall time is $O(Nz \log z + N \log N)$. For the cases in which *z* does not grow with increasing *N*, this time is effectively $O(N \log N)$.

Sorting Example 2

25 In another example, a parameter sorting process (e.g., the first or second “modified topological sort” process) determines an initial list of runtime parameters for a graph 22500 having graph components 22502, 22504 and 22506, as shown in FIG. 34A. The graph 22500 also has runtime parameters associated with an output port 22508 of an input data set 22510 and an input port 22512 of an output data set 22514. In this example, the parameters have both
 30 “intra-component” dependencies and “inter-component” dependencies. That is, parameters of a

component depend on parameters in the same component and parameters in other components. In this example, the inter-component dependencies come about due to flows between components that enable propagation of metadata upon which some parameters depend.

Dependencies are indicated in FIG. 34A by a dotted arrow from a first parameter or port to a second parameter or port. An arrow to a port indicates that the value of the linked parameter propagates from that port to a downstream port. An arrow from a port indicates that a value is propagated to the linked parameter from an upstream port. An arrow from a first parameter to a second parameter indicates that the value of the second parameter depends on (e.g., references) the value of the first parameter.

FIG. 34B shows a dependency graph 22550 that represents an ordering constraint among parameters p0, p1, p2, p4, p5 and p6 based on the graph 22500.

FIG. 34C shows a dependency graph 22552 that represents an ordering constraint among parameters p3, p7, p8 and p9 based on the graph 22500.

The parameter sorting process assigns an ordinal to each of ten parameters p0, p2,..., p9 for various graph elements according to the order of placement of the elements in the graph 22500. In FIG. 34A, the first graph element added to the graph 22500 (e.g., by a user using the GDE 1002) is component 22502 having parameter p0, p1 and p2. The second element added is component 22506 having parameters p3, p4 and p5. The third element added is data set 22510 having parameter p6. The fourth element added is data set 22514 having parameter p7. The last element added is data set 22516 having no runtime parameters. The following table lists the parameters in the initial ordering defined by the assigned ordinals.

Parameter	Ordinal	Dependencies
p0	0	
p1	1	p0, p6
p2	2	p6
p3	3	p8
p4	4	p1
p5	5	p1
p6	6	
p7	7	p3
p8	8	
p9	9	p8

The following listings of the parameters in *param_list* and *result_list* at various stages of processing correspond to the first “modified topological sort” process described above. The *param_list* is shown sorted according to the sort criterion *by_n_deps_and_order* at each stage.

5	<i>param_list</i>	<i>result_list</i>
	p0 p6 p8 p2 p3 p4 p5 p7 p9 p1	<i>empty</i>
	p6 p8 p1 p2 p3 p4 p5 p7 p9	p0
	p1 p2 p8 p3 p4 p5 p7 p9	p0 p6
	p2 p4 p5 p8 p3 p7 p9	p0 p6 p1
10	p4 p5 p8 p3 p7 p9	p0 p6 p1 p2
	p5 p8 p3 p7 p9	p0 p6 p1 p2 p4
	p8 p3 p7 p9	p0 p6 p1 p2 p4 p5
	p3 p9 p7	p0 p6 p1 p2 p4 p5 p8
	p7 p9	p0 p6 p1 p2 p4 p5 p8 p3
15	p9	p0 p6 p1 p2 p4 p5 p8 p3 p7
	<i>empty</i>	p0 p6 p1 p2 p4 p5 p8 p3 p7 p9

The following listings of the parameters in *candidates* and *result_list* at various stages of processing correspond to the second “modified topological sort” process described above. It is not necessary to sort *candidates* between stages since the parameters remain in the same order at

each stage.

	<i>candidates</i>	<i>result_list</i>
	p0 p6 p8	<i>empty</i>
	p6 p8	p0
25	p1 p2 p8	p0 p6
	p2 p4 p5 p8	p0 p6 p1
	p4 p5 p8	p0 p6 p1 p2
	p5 p8	p0 p6 p1 p2 p4
	p8	p0 p6 p1 p2 p4 p5
30	p3 p9	p0 p6 p1 p2 p4 p5 p8
	p7 p9	p0 p6 p1 p2 p4 p5 p8 p3

p9	p0 p6 p1 p2 p4 p5 p8 p3 p7
<i>empty</i>	p0 p6 p1 p2 p4 p5 p8 p3 p7 p9

Thus, referring to FIG. 35, the “modified topological sort” process 22600, takes as input a desired first ordering 22602 in which to prompt a user for values of runtime parameters, and an ordering constraint 22604 for the parameters (e.g., dependency graphs 22550 and 22552). The process 22600, provides the new ordering 22606 of the set of parameters that satisfies the ordering constraint according to the desired first ordering 22602.

Typical Usage

Typically, a user sits in front of the Web Interface 1008 and finds in the repository 1004 the graph of an application the user would like to run. By scanning all of the objects associated with the application graph, the Web Interface 1008 generates web page forms that allow the user to specify values for the runtime parameters of the application. Once all runtime parameters have been specified, the combination of the application and the parameter settings are brought together as a job, which is scheduled for execution by the executive 1010. When it comes time to run the job, the executive 1010 queues the application for execution under the parallel operating system 1006, in known fashion. The parallel operating system 1006 collects tracking information and job status and stores this information in the repository 1004 so that users and administrators can track the progress and performance of jobs.

Examples

FIG. 23 is a diagram of a graph 1400 representing a rollup application without runtime parameters. This graph computes the number of accounts of each kind and writes the results to an output file. Every aspect of this application has been determined by the developer who created the graph: the name of the input file component 1402, the format of the input data, the key and transform rules used to roll up the data in a HashRollup component 1404, the output format, and the name of the output file component 1406. A user can only execute this graph exactly as defined.

FIG. 24 is a diagram of a graph 1500 representing a runtime parameterized version of the rollup application of FIG. 23. The dataflow graph structure of this application is very similar to the non-runtime parameterized version, but the application is much more flexible. Through

runtime parameters, an end user may specify the name of the abstracted input dataset 1502 (a repositied object from which the input file name and format will be derived), the rollup key and rollup rules for the HashRollup component 1504, and the name of the output file component 1506.

FIG. 25 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 1600 for the example application of FIG. 24. This is a filled in version of the parameters grid shown in FIG. 11. Note that a number of default parameters are defined using the `prompt_for` pseudo-function, as described above, and thus require user input through the Web Interface 1008. While the appearance of this graph differs little from the non-runtime parameterized application graph, one or more parameter grids (or other suitable control) enable a developer to completely track all parameters that control the execution of the graph.

FIG. 26A is a diagram of one embodiment of a graphical dialog representing a form 1700 generated by the Web Interface 1008 from the information in the parameters grid 1600 of FIG. 25. In this example, the form 1700 presents four runtime parameters for user input: an input dataset repository path 1702, a rollup key 1704, rollup rules 1706, and an output path 1708.

FIG. 26B is a diagram of the form 1700 of FIG. 26A filled in by a user with parameter values. Using direct entry and/or edit or browser control buttons associated with the runtime parameters 1702-1708, a user provides corresponding parameter values 1710-1716 for executing the associated graph.

FIG. 27 is a diagram of a graph 1800 representing a runtime parameterized rollup and join application.

FIG. 28 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 1900 for the example application of FIG. 27. Here, some aspects of the application have been parameterized, but most, including the join key and the input datasets, remain fixed.

FIG. 29 is a diagram of one embodiment of a graphical dialog representing a form 2220 generated by the Web Interface 1008 from the information in the parameters grid 1900 of FIG. 28. Note that since the input type to the rollup is known at the time the top-level form is displayed, the rollup rules 2222 can be prompted for in-place.

FIG. 30 is a diagram of a graph 22110 representing a runtime parameterized rollup-join-sort application. While similar to the example in FIG. 27, a conditional sort component 22112 has been added to the graph 22110.

FIG. 31 is a diagram of one embodiment of a graphical dialog representing a runtime parameters grid 32200 for the example application shown in FIG 30. The `sort_key` runtime parameter 32202 is prompted for only if the user indicates that sorting is desired. To get this effect, a developer puts a `prompt_for` pseudo-function within an `if` conditional test for the default value 32204 of the `sort_key` runtime parameter 32202. The `if` conditional test references a second runtime parameter, `do_sort` 32206. The default value field 32208 and description field 32210 of the `do_sort` parameter 32206 are defined to generate a radio prompt asking the user for a true/false or yes/no answer to the text prompt “Should the data be sorted?”. If the value provided for the `do_sort` parameter 32206 is “true”, the sort component 22112 will be included as part of the graph at runtime. Otherwise, the sort component 22112 will be removed completely from the graph or replaced with flow, depending on its specified condition interpretation.

Script Implementation

While the GDE 1002 facilitates construction of parameterized graphs, sometimes there are non-graph programs for which one would like to provide a forms-based interface. Using application-level PL and the repository 1004, one can parameterize arbitrary shell scripts. For example, the description of an application can be written to a file with a structure similar to the following:

```

application AppName(
    description("One-line Description"),
    comment("Longer description"),
5    parameter ParmName1(
        string, kind(keyword), required,
        description("Short prompt for top-level form"),
        comment("Longer prompt for out-of-line form"),
        default(${prompt_for ...})
10    ),

    parameter ParmName2(
        type, kind(derived),
        default(PL-expression)
15    ),

    . . . more parameters . . .

    script(=scriptname.ksh)
20 )

```

General Computer Implementation

The invention may be implemented in hardware or software, or a combination of both (e.g., programmable logic arrays). Unless otherwise specified, the algorithms included as part of the invention are not inherently related to any particular computer or other apparatus. In particular, various general purpose machines may be used with programs written in accordance with the teachings herein, or it may be more convenient to construct more specialized apparatus to perform the required method steps. However, preferably, the invention is implemented in one or more computer programs executing on one or more programmable computer systems each comprising at least one processor, at least one data storage system (including volatile and non-volatile memory and/or storage elements), at least one input device or port, and at least one output device or port. The program code is executed on the processors to perform the functions described herein.

Each such program may be implemented in any desired computer language (including machine, assembly, or high level procedural, logical, or object oriented programming languages) to communicate with a computer system. In any case, the language may be a compiled or interpreted language.

Each such computer program is preferably stored on a storage media or device (*e.g.*, solid state, magnetic, or optical media) readable by a general or special purpose programmable computer, for configuring and operating the computer when the storage media or device is read by the computer system to perform the procedures described herein. The inventive system may
5 also be considered to be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer system to operate in a specific and predefined manner to perform the functions described herein.

A number of embodiments of the invention have been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope
10 of the invention. For example, a number of the function steps described above may be performed in a different order without substantially affecting overall processing. For example, STEPS 4002 and 4012 in FIG. 13 may be performed in reverse order. Accordingly, other embodiments are within the scope of the following claims.

FIG. 36 shows an exemplary data processing system 1100 in which the dynamic loading
15 techniques can be used. The system 1100 includes a data source 1101 that may include one or more sources of data such as storage devices or connections to online data streams, each of which may store data in any of a variety of storage formats (*e.g.*, database tables, spreadsheet files, flat text files, or a native format used by a mainframe). An execution environment 1104 includes a pre-execution module 1105 and an execution module 1106. The execution
20 environment 1104 may be hosted on one or more general-purpose computers under the control of a suitable operating system, such as the UNIX operating system. For example, the execution environment 1104 can include a multiple-node parallel computing environment including a configuration of computer systems using multiple central processing units (CPUs), either local (*e.g.*, multiprocessor systems such as SMP computers), or locally distributed (*e.g.*, multiple
25 processors coupled as clusters or MPPs), or remotely, or remotely distributed (*e.g.*, multiple processors coupled via a local area network (LAN) and/or wide-area network (WAN)), or any combination thereof.

The pre-execution module 1105 is configured to perform various tasks in preparation for
executing dataflow graphs and other executable programs such as compiling dataflow graphs,
30 storing/loading compiled dataflow graphs to/from a data storage system 1107 accessible to the execution environment 1104, or resolving parameter values and binding the resolved values to

parameters. In some cases, the pre-execution module 1105 performs tasks (e.g., loading compiled dataflow graphs) in response to data from the data source 1101. Storage devices providing the data source 1101 may be local to the execution environment 1104, for example, being stored on a storage medium connected to a computer running the execution environment
5 1104 (e.g., hard drive 1102), or may be remote to the execution environment 1104, for example, being hosted on a remote system (e.g., mainframe 1103) in communication with a computer running the execution environment 1104, over a remote connection.

The execution module 1106 uses the compiled dataflow graphs generated by the pre-execution module 1105 to generate output data, which can be provided back to the data source
10 1101 and/or stored in the data storage system 1107. The data storage system 1107 is also accessible to a development environment 1108 in which a developer 1109 is able to design dataflow graphs. Data structures representing the dataflow graphs can be serialized and stored in the data storage system 1107.

The execution module 1016 can receive data from a variety of types of systems of the
15 data source 1101 including different forms of database systems. The data may be organized as records having values for respective fields (also called “attributes” or “columns”), including possibly null values. When reading data from a data source, an executing dataflow graph may include components that handle initial format information about records in that data source. In some circumstances, the record structure of the data source may not be known initially and may
20 instead be determined after analysis of the data source. The initial information about records can include the number of bits that represent a distinct value, the order of fields within a record, and the type of value (e.g., string, signed/unsigned integer) represented by the bits.

Referring to FIG. 37, an example of a dataflow graph 1155 that is executed by the execution module 1106 allows data from an input data source 1110 to be read and processed as a
25 flow of discrete work elements. Different portions of the computations involved with processing the work elements are performed in components 1120, 1130 that are represented as the vertices (or nodes) of the graph, and data flows between the components that are represented by the links (or arcs, edges) of the graph, such as the link 1125 connecting components 1120 and 1130. A system that implements such graph-based computations is described in U.S. Patent 5,566,072,
30 EXECUTING COMPUTATIONS EXPRESSED AS GRAPHS, incorporated herein by reference. Dataflow graphs made in accordance with this system provide methods for getting

information into and out of individual processes represented by graph components, for moving information between the processes, and for defining a running order for the processes. This system includes algorithms that choose interprocess communication methods (for example, communication paths according to the links of the graph can use TCP/IP or UNIX domain sockets, or use shared memory to pass data between the processes).

The process of preparing an uncompiled dataflow graph for execution involves various stages. An uncompiled representation of the dataflow graph is retrieved along with any parameters associated with the dataflow graph that provide values used in the compilation process. During a static parameter resolution phase, static parameters (whose values are designated for resolution before run-time) are resolved and the resolved values are bound to the static parameters. In some cases, in order to resolve the values of the static parameters, calculations are performed to derive certain values (e.g., metadata values, as described in U.S. Publication No. 2006/0294150 entitled “MANAGING METADATA FOR GRAPH-BASED COMPUTATIONS,” incorporated herein by reference). Some parameters may be designated as dynamic parameters that are left unresolved to be resolved later at or just before run-time. Components designated as conditional components are removed from the graph (e.g., by being replaced by a dataflow link) if a predetermined condition is not met (or is met), for example, based on a static parameter value. During the compilation phase, data structures representing the dataflow graph, including its components and links, to be used during execution are generated. Compiling can also include compiling embedded scripts in scripting languages into bytecode or machine code. At run-time, any dynamic parameters associated with the dataflow graph are bound to resolved values, and the data structures of the compiled dataflow graph are launched by starting one or more processes, opening any needed files (e.g., files identified by dynamic parameters), and/or linking any dynamic libraries. The processes also perform tasks to set up data flows represented by the links (e.g., allocating shared memory, or opening TCP/IP streams). In some cases, the data structures are configured to execute multiple components in a single process, as described in U.S. Publication No. US 2007/0271381 entitled “MANAGING COMPUTING RESOURCES IN GRAPH-BASED COMPUTATIONS,” incorporated herein by reference.

Dataflow graphs can be specified with various levels of abstraction. A “subgraph” which is itself a dataflow graph containing components and links can be represented within another

containing dataflow graph as a single component, showing only those links which connect the subgraph to the containing dataflow graph. In some cases, subgraphs are used to hide the components and links of a containing dataflow graph within the development environment 108, but the data representing the components and links are already integrated within the containing dataflow graph. In some embodiments, subgraphs are not initially included in a containing dataflow graph, but are later derived from a “micrograph” that starts as a compiled dataflow graph that includes the data structures representing the components and links to be loaded into the containing dataflow graph, as described in more detail below.

Referring again to FIG. 37, the dataflow graph 1155 is able to accept a continuous flow of input data from a data source, which in this example is represented by the dataset component 1110. The flow of data is continuous in the sense that, even though it may wax and wane during the operation of the dataflow graph 1155, the flow of data does not necessarily have a distinguishable beginning or end, for example, a flow of credit card transactions or orders received in entry systems. Additionally, the dataflow graph 1155 is able to provide a continuous flow of output data to an output data repository, which in this example is represented by the dataset component 1140. Some dataflow graphs are designed for execution as continuous dataflow graphs that process continuous flows of data for an indefinite period of time, and some dataflow graphs are designed for execution as batch dataflow graphs that begins execution to process a discrete batch of data and then terminates execution after the batch has been processed. The data processing components 1120 and 1130 of a dataflow graph may contain a series of instructions, a sub-graph, or some combination thereof.

In some arrangements, a specialized component 1130 may load a micrograph 1160, which is a specialized kind of subgraph configured to be retrieved dynamically and embedded within a containing dataflow graph. A micrograph can be dynamically embedded with a containing component of the containing dataflow graph, such as the specialized component 1130. In some arrangements, a micrograph 1160 is derived from a data flow graph that was previously compiled and stored in the data storage system 1107. In some arrangements, a micrograph 1160 remains in an un-compiled form when loaded from the data storage system 1107. For example, instead of loading a compiled dataflow graph, the specialized component 1130 may initiate a graph compilation procedure to enable the dataflow graph containing the specialized component 1130 to be able to execute micrographs that have been compiled just before use. The graph

compilation procedure may be performed external to the specialized component 1130, or internal to the specialized component 1130. In some arrangements, the micrograph 1160 is serialized prior to being stored in the data storage system 1107. In general, serialization is a process by which a dataflow graph, in a compiled or uncompiled representation, is translated into a binary stream of zeroes and ones so that the dataflow graph is in a form that can easily be stored in persistent memory or in a memory buffer. In implementations in which fast loading of micrographs is desirable, typically the dataflow graph is serialized in a compiled representation with data structures and stored in the data storage system 1107, so that the serialized compiled dataflow graph can be easily retrieved and the data structures de-serialized and loaded dynamically at run-time. Compilation is the process by which a computer program, including a program expressed as a dataflow graph, is prepared to be executed by a computer. Compilation may result in the generation of machine code, or instructions ready to be executed on a computer, or in intermediate code which is executed by a virtual machine executing on a computer, for example, Java byte code. In the case of dataflow graphs, compilation includes generation of data structures representing the components and links of the dataflow graph in a form ready to be executed on a computer.

Data flowing through the dataflow graph 1155 along the incoming link 1125 of the specialized component 1130 is temporarily stored in a buffer while the specialized component 1130 loads the micrograph 1160 and integrates the micrograph 1160 into the dataflow graph 1155. Similarly, the data flow is allowed to accumulate in a buffer while a micrograph is detached and unloaded from the dataflow graph 1155. In some arrangements, a detached micrograph may remain loaded in memory to be accessed later (for example, using a caching mechanism).

The buffering can be configured to use a flow control mechanism that is implemented using input queues for the links providing an input flow of work elements to a component. This flow control mechanism allows data to flow between the components of a dataflow graph without necessarily being written to non-volatile local storage, such as a disk drive, which is typically large but slow. The input queues can be kept small enough to hold work elements in volatile memory, typically smaller and faster than non-volatile memory. This potential savings in storage space and time exists even for very large data sets. Components can use output buffers instead of, or in addition to, input queues. When two components are connected by a

flow, the upstream component sends work elements to the downstream component as long as the downstream component keeps consuming the work elements. If the downstream component falls behind, the upstream component will fill up the input queue of the downstream component and stop working until the input queue clears out again.

5 In some arrangements, the micrograph 1160 is executed on a scheduled basis by a scheduling process, as part of a batch dataflow graph, or from the command line, and is not necessarily loaded into a specialized component of a containing dataflow graph. Alternatively, a scheduler can use a specialized component to launch a micrograph for batch processing by sending a message to a queue that provides a continuous flow of messages to a running dataflow
10 graph containing a specialized component that will load the appropriate micrograph in response to the message from the queue.

1 Specialized Component

Referring to FIG. 38A, in one embodiment, a specialized component 38200 is configured to accept multiple inputs 38205, 38210, 38215 and deliver multiple outputs 38220, 38225,
15 38230, and 38235. The inputs include data inputs 38205, 38210, and a control input 38215. The outputs include data outputs 38220, 38225, 38230, and a status output 38235. The control input 38215 accepts an identifier of a micrograph 38240 to run (e.g., within a received control element), and optionally a set of parameters used to run the micrograph 38240. In general, the specialized component 38200 may accept zero or more flows of input work elements over
20 respective data input ports, such as data inputs 38205, 38210. The status output 38235 produces a status record which includes exit status and tracking information from the execution of the micrograph 38240. Additionally, the specialized component produces zero or more flows of output work elements over respective data output ports, such as data outputs 38220, 38225, and 38230. In one embodiment, the specialized component 38200 produces one status record and
25 accepts one control record during a normal execution.

The control input 38215 can receive a series of multiple control elements that each identifies a corresponding micrograph to be loaded. Each control input element is associated with a different subset of work elements in the flow(s) of work elements received over the data input(s) that represent a unit of work to be processed by the identified micrograph. In some
30 cases, the control element identifying the micrograph is generated based on analyzing one or

more work elements in the unit of work to select the appropriate micrograph to process that unit of work. In some cases, the control element identifying the micrograph and the work elements representing the unit of work to be processed by the identified micrograph are received independently and are matched to each other using any of a variety of techniques. For example, the control element identifying the micrograph is received first and the work elements representing the unit of work to be processed by the identified micrograph are determined by matching a key value appearing in at least one work element of a unit of work to a corresponding control element (e.g., just the first work element in the unit of work, or every work element in the unit of work). Alternatively, work elements called “delimiter work elements” can function as delimiters that separate different sequences of work elements belonging to the same unit of work. Alternatively, the specialized component 200 is configured to receive a predetermined number of work elements to belong to successive units of work to be associated with respective control elements.

The micrograph 38240 can be selected from a collection of micrographs that have been designed to be compatible with the specialized component 38200. For example, the number of input ports and output ports of the micrograph 38240 may match the number of input ports and output ports of the specialized component 38200. In this example, the micrograph 38240 has two input ports and three output ports, which could be located on two different components of the micrograph 38240 that are configured to receive input flows and three different components of the micrograph 38240 that are configured to provide output flows. Alternatively, multiple input or output ports of the micrograph 38240 could be located on the same component.

In some embodiments, the specialized component 38200 monitors the micrograph 38240 for predefined conditions and may respond to those conditions. For example, the specialized component 38200 may use a separate process to monitor the process that executes the components of the micrograph 38240 to determine if the latency during execution of the micrograph 38240 exceeds a maximum threshold or for a timeout condition. In response, the specialized component 38200 may respond to the condition by, for example, loading a second instance of the micrograph 38240. Similarly, error conditions are monitored. In response to detecting an error condition, the specialized component 38200 may log the error, redirect the unit of work based on the error condition, and, if necessary, may restart a micrograph 38240 and report the error via the status output 38235. The restarting of the micrograph 38240 does not

need to interrupt any other components of the containing dataflow graph in which the specialized component 38200 is contained.

In some embodiments, the specialized component 38200 analyzes the data flow on an input port to determine which micrograph 38240 to execute. In other embodiments, the name or other identifying information of the micrograph 38240 to run is supplied to the specialized component 38200 as part of the data flow. In still other embodiments, the information identifying the micrograph 38240 is supplied through the control input 38215 of the specialized component 38200.

The specialized component 38200 loads the micrograph 38240 from the data storage system 1107, embeds the micrograph 38240 into the dataflow graph containing the specialized component 38200, as described in more detail below, and allows the micrograph 38240 to process the data flow.

When the operation is complete, the specialized component 38200 removes the micrograph 38240. In some embodiments, the specialized component 38200 may store the micrograph 38240 in a micrograph cache stored in a storage location from which it is relatively more easily accessible than from the data storage system 1107, for later access. In some embodiments, the specialized component 38200 may buffer the incoming data while the micrograph 38240 is being loaded, integrated, and removed.

2 Micrograph Structure

Referring to FIG. 38B, an example of micrograph 38240 includes data processing components 38305, 38310, 38315, and 38320 which perform operations on input work elements in data flows arriving at one or more input ports linked to upstream components, and produce output work elements in data flows leaving one or more output ports linked to downstream components. The micrograph 38240 also includes components 38300A, 38300B and 38302A, 38302B, 38302C that are configured to facilitate the process of embedding the micrograph 38240 into the specialized component 38200, called “interface components.” Interface components also enable the micrograph to be run as an independent dataflow graph that does not need to be embedded into a specialized component to run.

Interface components are used to connect a micrograph to the ports of the containing specialized component. Embedding a micrograph into a specialized component involves

determining how to pair each interface component with the correct port of the specialized component. Input interface components (38300A, 38300B) have an output port, and output interface components (38302A, 38302B, 38302C) have an input port. When an input interface component is paired with an input port of the specialized component, a dataflow link is generated
5 between that input port and the port to which the output port of the input interface component is linked. Similarly, when an output interface component is paired with an output port of the specialized component, a dataflow link is generated between the port to which the input port of the output interface component is linked and that output port.

The correct pairing of interface components with their respective specialized component
10 input and output ports can be determined as follows. The interface components and the ports of the specialized component can optionally be labeled with identifiers. If there is a match between an identifier assigned to an input/output port of the specialized component and an identifier assigned to an input/output interface component, then that port and that interface component will be paired. Matches between identifiers can be exact, or inexact (e.g., finding a match between
15 some prefix or postfix of an identifier). In one pairing procedure, after exact matches are determined, interface components with numeric suffixes are matched to ports with matching numeric suffixes (e.g., an “out1” port of a specialized component will be paired with a “Micrograph Output-1” interface component).

When a micrograph is executed outside of a specialized component (e.g., in response to a
20 command from the development environment 1108 for testing purposes) the interface components provide the functionality of dataset components, providing a source or sink of work elements in a data flow over the link connected to the interface component’s output or input port. The appropriate functionality is determined based on whether the micrograph has been embedded within a specialized component. The interface components each include instructions
25 for sending work elements between the micrograph and the specialized component when the micrograph is embedded within a specialized component, and instructions for sending work elements between the micrograph and a storage location when the micrograph is not embedded within a specialized component. The storage location can be a file that is read to provide input work elements or written to receive output work elements.

30 In some embodiments, the micrograph 38240 is configured by placing certain constraints on the functionality of the data processing components that can be included in the micrograph (in

this example, components 38305, 38310, 38315, and 38320). For example, in some embodiments, the data processing components 38305, 38310, 38315, and 38320 of the micrograph 38240 may be required to be able to be run within a single process (e.g., by being folded into a single process as described in more detail in U.S. Publication No. 2007/0271381, incorporated herein by reference). In some embodiments, a micrograph does not support subscriber components that receive data from a subscribed source (such as a queue). In some embodiments, a micrograph may be required to be configured as a batch dataflow graph. In some embodiments, any transactional operations executed by the micrograph 38240 must fit into a single transaction. In other embodiments, the transactional aspects of the micrograph 38240, for example checkpoints, transactional context, and multi-phase commits is controlled through a control input 38215.

In dataflow graph processing, the continuous flow of data can affect traditional transactional semantics. A checkpoint operation involves storing sufficient state information at a point in the data flow to enable the dataflow graph to restart from that point in the data flow. If checkpoints are taken too often, performance degrades. If checkpoints are taken too infrequently, recovery procedures in the case of a transactional failure become more complex and resource intensive. A transactional context may be used to inform the micrograph that its operations are part of a larger transaction. This transaction may be a larger transaction encompassing multiple components acting against a single data source, or may include information necessary to coordinate the transaction across multiple data source, for example, in a two phase commit operation.

3 Micrograph Management

Referring to FIG. 39, in one embodiment, a specialized component 39405 may have more than one micrograph loaded into a cache accessible to the specialized component (e.g., stored in a local memory) at a time. In this example, one micrograph 39430 is connected into the dataflow graph that includes the specialized component 39405. The input 39440 of the specialized component 39405 is connected by a link to the input 39445 of the micrograph 39430 and the output 39450 of the micrograph 39430 is connected by a link to the output 39455 of the specialized component 39405. The input 39440 and output 39450 of the micrograph 39430

represent input and output interface components, for example, or any other mechanism for embedding a micrograph into a specialized component.

The cache storing the other micrographs 39410, 39415, 39420, and 39425 can be located in the same memory that stores the connected micrograph 39430. In some embodiments, the micrographs 39410, 39415, 39420, 39425, and 39430 are able to execute as threads within the same process that runs the specialized component 39405, or as child processes of that process. Alternatively, the micrographs 39410, 39415, 39420, 39425, and 39430 may be executed within the same main thread of that process. In some embodiments, the specialized component 39405 runs multiple copies of the micrographs 39410, 39415, 39420, 39425, and 39430. In some embodiments, the specialized component 39405 uses the copy of a selected micrographs that is stored in the cache before accessing the original micrograph stored in the data storage system 1107 unless that copy is marked as “dirty” (indicating that there has been a change in the original micrograph). When a copy of the selected micrograph is not in the cache, the specialized component 39405 loads the micrograph from the data storage system 1107. In some embodiments, the selected micrograph is indicated (e.g., by an entry in the cache) as being in an “offline” state, indicating that the selected micrograph is currently unavailable (e.g., the micrograph may be in the process of being modified by a developer). In response, the specialized component can indicate an error in loading the selected micrograph, or can load a substitute micrograph (e.g., a different micrograph that has similar capabilities).

Referring to FIG. 40A, in one scenario, a single specialized component 40505 has multiple micrographs 40510, 40515, and 40520 integrated into the data flow at the same time. The input port 40525 of the specialized component 40505 is connected to the inputs 40530, 40545, 40555 of the micrographs 40520, 40515, and 40510, respectively, using a partition component 40557 that partitions work elements from an input data flow into multiple data flows. The outputs 40535, 40550, and 40560 of the micrographs are connected to the output 40540 port of the specialized component 40505, using a component 40562 (called a “gather component”) that gathers work elements from multiple data flows and merges them into a single output data flow. In this configuration, the specialized component 40505 is able to route the incoming data to the micrographs 40510, 40515, and 40520. For example, when the components are separate copies of identical micrographs, the specialized component 40505 may utilize load balancing algorithms (e.g. round robin, least utilized, etc...) when performing the partitioning.

Referring to FIG. 40B, in another scenario, a dataflow graph 40565 includes multiple specialized components, enabling flexible combinations of micrographs to be arranged in a highly customizable container dataflow graph. In this example, work elements from a dataset component 40570 (which may represent a batch of input data or a continuous stream of input data) are first processed by a data processing component 40572 (e.g., to reformat the work elements) and then sent over a link 40574 to the first specialized component 40576. In response to detecting a particular type of work element, the component 40576 loads a micrograph 40578 configured for processing work elements of the detected type. The processed output work elements from the first specialized component 40576 are then sent to a second specialized component 40580, which loads a micrograph 40582. In this arrangement, the micrograph 40582 that is selected for loading into the second specialized component 40580 can depend on results of the first selected micrograph 40578. Resulting output work elements are sent to a dataset component 40584. A large number of combinations of different micrographs can be dynamically loaded using specialized components in strategic locations within a dataflow graph. In this simple example, if there are 10 different possible micrographs that can be loaded into the first specialized component 40576 and 10 different possible micrographs that can be loaded into the second specialized component 40580, there are as many as 100 different dataflow graphs that can be dynamically constructed on demand while a data flow is being processed, with potentially much fewer resources needed compared to partitioning a dataflow to 100 different running dataflow graphs, and with potentially much faster latency compared to starting up one of 100 different dataflow graphs for each unit of work.

4 Pre-processing and Post-processing

A specialized component can include other components in addition to the components within the loaded micrograph. Referring to FIG. 41, in one embodiment, the specialized component 41605 includes pre-processing before the micrograph is executed, and post-processing after the micrograph is executed (represented in the figure as pre-processing component 41610 and post-processing component 41620 surrounding the micrograph 41615). Pre and post processing activities may pertain to, for example, transaction management. In some embodiments, each micrograph 41615 represents a separate transaction, in which case the pre-processing may start a transaction and the post processing may end the transaction. In other

embodiments, the micrograph 41615 may represent a checkpoint in a longer transaction. For example, the micrograph 41615 may be part of a longer transaction using a two phase commit protocol. Such a transaction may be processed using multiple different micrographs or multiple executions of the same micrograph, for example.

5 The pre-processing component 41610 may load the micrograph 41615 from a data store (not shown) and potentially store the loaded micrograph in a data cache for later access or access a loaded version of the micrograph from an in memory cache (not shown) and integrate it into the specialized component 41605. The post-processing component 41620 may remove a loaded micrograph 41615 from its integration with the data flow of the specialized component 41605. A
10 variety of other pre and post processing functions can be performed including, for example, pre-processing by preparing records within a data flow for processing by the micrograph 41615 (e.g., reformatting the records), and post-processing by preparing records received from the micrograph 41615 for processing by components connected to the output of the specialized component 41605 (e.g., by reformatting the records).

15 5 Database Connection Management

Referring to FIG. 42, in some scenarios, different components are required to be able to access different types of databases. For example, component 42702 is required to access a database 42706 provided by one vendor, while another component 42704 is required to access a database 42704 provided by another vendor. Generally, a component 42702 will access a
20 database 42706 by accessing a library 42710 integrated into the component, for example, a library supplied by the vendor of the database 42706. Similarly, component 42704 will access the database 42708 by accessing another library 42712 integrated into the component, for example, a library provided by the vendor of database 42708. Libraries can be compiled into a particular component, or can be dynamically linked to a particular component.

25 Referring to FIG. 43, a specialized component 43802 can include multiple different micrographs 43804, 43806, and 43808. In this example, micrograph 43808 is embedded within the specialized component 43802, and micrographs 43804 and 43806 are loaded into an accessible cache to be dynamically embedded as necessary. Some of the micrographs may access one database 43706 and other micrographs may access another database 43708.

30 Traditionally, accessing the two databases may require a library 43710 to support the first

database 43706 and another library 43712 to support the other database 43708 to be integrated with the specialized component 43802. Integrating multiple different database libraries can lead to an increased size in the binaries associated with the specialized component 43802, as well as unpredictable behavior if one library 43710 supplied by a vendor is incompatible with another
5 library 43712 supplied by the different vendor. For example, incompatibilities may include conflicting symbol names or different compilation models.

One method of avoiding incompatibilities and/or decreasing the size of the binaries is to remove the libraries from the specialized component and, instead, have them accessed by a separate computer process from a process executing the specialized component. Referring to
10 FIG. 44A, the database libraries 44710 and 44712 are removed from the specialized component 44902 (e.g., not compiled or dynamically linked to the component 44902). Micrograph components 44904, 44906, and 44908 access the databases 44706, 44708 by accessing the libraries 44710, 44712 over a client/server interface. In order to access the libraries 44710, 44712, the specialized component 44902 uses an integrated client stub 44910 which
15 communicates with a paired server stub 44912 running in an external process using inter-process communication. For example, the external processes and the specialized component 44902 may exchange data by accessing a shared memory segment. Similarly, to access the external process running the server stub 44920, the specialized component 44902 uses another integrated client stub 44918 which communicates with the server stub 44920 running in an external process using
20 inter-process communication.

The server stub 44912 integrates with the database library 44710 in order to provide access to the database 44706. Similarly, the server stub 44920 integrates with the database library 44712 in order to provide access to the database 44708. Generally, the client stubs 44910, 44918 have a smaller memory footprint than the database libraries 44710, 44712 and
25 therefore allow the specialized component to use less memory resources. Additionally, because the database libraries have been moved out of the specialized component 44902, there is no risk of incompatibility between the database libraries 44710, 44712.

Referring to FIG. 44B, in some embodiments, the client stubs 44910, 44918 and server stubs 44912, 44920 are configured to closely reflect the respective application programming
30 interfaces (APIs) of the database libraries 44710, 44712. In order to isolate the micrographs 44904, 44906, and 44908 from differences in library APIs, an abstraction layer 44930 is

integrated into the specialized component 44902. The abstraction layer 44930 provides multiple different components within a micrograph that may need to access different databases with a single API with which to perform standard database actions independent of differences between the database libraries for those databases. In some embodiments, the abstraction layer 44930 translates the requests from the components of the micrographs into specific calls to the library-specific APIs of the client stubs 44910, 44918. In some embodiments, the abstraction layer 44930 passes requests from the micrograph components as generic calls, and each client stub is configured to perform the translation from generic calls into the library-specific calls to that respective server stub's APIs.

6 Compilation and Parameters

Referring to FIG. 45, in one embodiment, a dataflow graph compilation and execution system 451000 includes a dataflow graph compiler 451015, a parameter analyzer 451030, a dataflow graph loader 451035, and a dataflow graph execution engine 451040. The compiler 451015 processes uncompiled dataflow graphs, including micrographs, from an un-compiled dataflow graph data store 451010 based on parameters in parameter sets from a parameter set data store 451005. The value associated with a given parameter can determine any of a variety of characteristics of a dataflow graph. The compiler 451015 compiles the dataflow graphs using values from a parameter set to generate a compiled dataflow graph. The compiled dataflow graph is stored in a compiled dataflow graph data store 451020. The data stores 451010 and 451020 can be hosted, for example, within the data storage system 1107. The compiler 451015, parameter analyzer 451030, and loader 451035 can be implemented, for example, as part of the pre-execution module 1105. The execution engine 451040 can be implemented as part of the execution module 1106.

The parameter set data store 451005 contains sets of parameters and each parameter set can be associated with a dataflow graph. A parameter set includes a group of parameter elements. These elements contain the name of a parameter and an expression that when evaluated (e.g., by performing computations, and in some cases, by finding values of other referenced parameters) are resolved into a value that is bound to the parameter. Each uncompiled dataflow graph can be associated with one or more parameter sets. Some parameters can affect the compilation process. For example, some parameters can affect whether certain

conditional components are included in a compiled version of a dataflow graph. Some parameters can affect the loading and launching of a compiled dataflow graph. For example, the value of a parameter can be a path to a file stored on a computer, or the name of a storage location to be associated with a dataset component (e.g., a storage location representing a table in a database) that contains input data or is the target for output data. The value of a parameter can determine how many ways parallel a given component needs to run. The value of a parameter can determine whether a data flow of a link between components crosses a boundary between different processors and/or computers running the linked components, and if so, causing a TCP/IP flow to be allocated rather than a shared memory flow. In some scenarios, the values of the parameters may be dependent upon other parameters. For example, the name of a currency conversion lookup file may be dependent upon a parameter which specifies a date.

In general, a parameter is bound to a value according to rules for parameter scoping based on contexts. A given parameter can have one value in a first context and a different value in another context. A parameter can be bound to a value during compilation (e.g., by the compiler 451015 if the parameter could affect the compilation process), at run-time (e.g., when the loader 451035 loads the compiled dataflow graph data structures into memory for the execution engine 451040), while the dataflow graph is being executed (e.g., delaying a parameter that provides a file name from being resolved until just before the file is read or written), or, in some cases, a combination of different times. The value of a parameter can be defined, for example, by a user over a user interface (e.g., in response to a prompt), defined from a file, included in a data source, or defined in terms of another parameter in the same context or in different context. For example, a parameter can be imported from a different context (e.g., a parameter evaluated in the context of a different component) by designating the parameter to have a “same as” relationship to another parameter.

Parameters for a dataflow graph can be bound before any input data has been received such as during compilation (e.g., by the compiler 451015). Such parameters that are bound before or during compilation of a dataflow graph are called “static parameters.” Parameters for a dataflow graph can also be bound in response to receiving new input data such as just before run-time (e.g., by the loader 451035 in response to receiving a new batch of data or an initial unit of work within a flow of data), or during run-time (e.g., by the execution engine 451040 in response to loading a new micrograph to handle a new unit of work within a flow of data). Such

parameters that are bound after compilation of the dataflow graph and closer to run-time are called “dynamic parameters.” In some cases, dynamic parameters do not need to be re-evaluated for each new batch of data or unit of work within a flow of data. Static parameters are typically evaluated while a graph is being compiled and can affect the compiled dataflow graph.

5 However, static parameters can also be evaluated at run-time if, for example, compilation does not occur until run-time. Because dynamic parameters are evaluated at run-time and may contribute to the overall latency of starting up a dataflow graph, dynamic parameter evaluation can, in some embodiments, be optimized by limiting the capabilities of dynamic parameters. For example, dynamic parameters may be limited to specific data types (e.g. strings), they may not
10 be referenced within certain expressions, and may have more restricted bindings (e.g., not referenced directly or indirectly by a “same as” binding.) However, in other embodiments, dynamic parameters may not be restricted in their functionality and are utilized just like other parameters.

There are various ways to enable the compiler 451015 to distinguish between static and
15 dynamic parameters during the compilation process. One technique uses a flag or special syntax for dynamic parameters in a parameter set, signaling to the compiler that the dynamic parameter should be left unbound. Another technique separates static and dynamic parameters of a parameter set into different subsets, and only providing the subset of static parameters to the compiler 451015. The subset of dynamic parameters are then provided to the loader 451035 at
20 run-time. Even though the compiler 451015 does not bind the dynamic parameters to resolved values, the compiler 451015 can still be configured to check dynamic parameters (e.g., for illegal syntax or other errors) during compilation. A given expression may include both static and dynamic parameter references. For example, a static directory name parameter can be resolved by the compiler 451015, but a dynamic file name parameter can be left unresolved and the
25 compiler 451015 can preserve the unresolved dynamic parameter reference during the compiling process because it is recognized as a dynamic parameter.

In some scenarios, the existence or value of a parameter may affect the topology and/or connectivity of components within a dataflow graph. A parameter may indicate that the operations performed by one or more components are not to be executed during a particular
30 execution of the graph. This may be especially relevant when the same dataflow graph is used on two different data sources. For example, one data source may be in a first format (e.g., UTF-

8) and another source may contain text in a second format that uses different encodings for at least some characters. A dataflow graph that processes both data sources may need to convert text in the second format into the UTF-8 format. However, when accessing a UTF-8 data source, no conversion would be necessary. A parameter could be used to inform the dataflow graph that the data is already in UTF-8 format and that a conversion component may be bypassed. In some arrangements, the exclusion of a conditional component based on a parameter value may result in the conditional component being removed and replaced with a dataflow in the compiled dataflow graph. Additional description of conditional components can be found in U.S. Patent No. 7,164,422, incorporated herein by reference.

Referring again to FIG. 45, the compiler 451015 obtains an uncompiled dataflow graph from the uncompiled dataflow graph data store 451010. The compiler 451015 obtains the parameter set that is to be used for compiling the dataflow graph from the parameter set data store 451005. In some cases, multiple different parameter sets could be used for a given dataflow graph, and for each parameter set, the graph compiler 451015 is able to compile a corresponding version of the uncompiled dataflow graph. Each compiled version of the dataflow graph may include or exclude some components or other executable statements based on the values of the parameters in the parameter set. The compiled dataflow graph is associated with the bound parameter values from the parameter set that was used to generate the compiled dataflow graph, for example, when the compiled dataflow graph is serialized. The compiled dataflow graph is associated with the parameter values from the associated parameter set using any of a number of different mechanisms (e.g., a lookup table, a foreign key to primary key relationship in a database, etc. ...). The compiled dataflow graph data store 451020 can be implemented, for example, using any file system or database capable of read and write operations.

During dataflow graph execution (at “run-time”), data enters the system from an input data source 451025. The input data source 451025 can include a variety of individual data sources, each of which may have unique storage formats and interfaces (for example, database tables, spreadsheet files, flat text files, or a native format used by a mainframe). The individual data sources can be local to the system 451000, for example, being hosted on the same computer system (e.g., a file), or can be remote to the system 451000, for example, being hosted on a remote computer that is accessed over a local or wide area data network.

The parameter analyzer 451030 and loader 451035 enable a dataflow graph to be quickly loaded from a stored compiled dataflow graph, avoiding the potentially lengthy compilation process, while still allowing flexibility at run-time by selecting among different compiled versions of a dataflow graph in response to a received input data. The parameter analyzer 5 451030, in response to receiving input data from the input data source 451025, analyzes the input data, and potentially other values that may not be known until run-time, to determine values for one or more parameters (potentially including both static parameters and dynamic parameters) to be used with a target dataflow graph. The target dataflow graph is an uncompiled dataflow graph that has been compiled into different versions stored in the compiled dataflow graph data 10 store 451020 using different respective parameter sets. The loader 451035 compares any values of static parameter provided by the parameter analyzer 451030 with any values of those same parameters that may have been used to generate any of the compiled versions of the target dataflow graphs to find a match. If the loader 451035 finds a match in the static parameters, then the loader 451035 can resolve and bind the dynamic parameters, and load the resulting dataflow 15 graph to be launched by the execution engine 451040. If the loader 451035 does not find a match in the static parameters, the loader 451035 can route the input data to an error processor (not shown), or the loader 451035 may have a version of the target dataflow graph available which is capable of handling different possible parameter values as a non-optimized default. Alternatively, the loader can initiate the compiler 451015 to compile a new version of the target 20 dataflow graph with the appropriate static parameter values. In these cases, compilation can be done on the fly as needed. The first time a target dataflow graph is run, compilation is done at run-time and the resulting compiled dataflow graph with bound static parameters is saved. Then the next time the target dataflow graph is run, it is only recompiled if a compiled version with the desired static parameter values is not found.

25 For example, for a parameter set including parameters A, B, C (with A and B static, and C dynamic), consider a first compiled version of a target dataflow graph that has been compiled using A = True and B = False, and a second compiled version of the same target dataflow graph that has been compiled using A = True and B = True. If the parameter analyzer 451030 determines that a received unit of work is to be processed using the target dataflow graph with A 30 = True, B = True, and C = True, then the second compiled version can be loaded and the dynamic parameter C is bound by the loader 451035. If the parameter analyzer 451030

determines that a received unit of work is to be processed using the target dataflow graph with A = False, B = True, and C = True, then a new version of the target dataflow graph is compiled by the compiler 451015 with A = False, B = True and the dynamic parameter C is bound by the loader 451035. The newly compiled version can also be stored in the compiled dataflow graph data store 451020 for later use.

The loader 451035 loads the selected compiled dataflow graph into a memory accessible to the graph execution engine 451040 to be executed to process the flow of input data from the input data source 451025. In some embodiments, the function of the loader 451035 is performed by a specialized component in a running dataflow graph and the loaded dataflow graph is a micrograph embedded into the specialized component. The loader 451035 may access a previously loaded version of the selected compiled dataflow graph which remains cached in memory (after determining that the appropriate static parameter values were used) without necessarily needing to access the compiled dataflow graph data store 451020. The loaded dataflow graph is then executed by the execution engine 451040. Once the input data has been processed by the target dataflow graph, the dataflow graph may either be unloaded from the system, or may be cached for later access.

In general, compilation of a dataflow graph is the process by which the graph is transformed into an executable format. The executable format can be in a platform specific form (e.g., machine code) or in an intermediate form (e.g., byte code). In some embodiments, the compiler 451015 resolves the static parameters, traverses the dataflow graph, and reduces it to a set of data structures that are prepared to be executed. The transformation from a dataflow graph, which is represented as vertices and links, to machine code may include several steps. One of these steps can include dynamic code generation where the dataflow graph is transformed into a third generation programming language (e.g. C, C#, C++, Java, etc. ...). From the third generation language, machine readable code or byte code can be generated using a standard compiler.

In some embodiments, whether a parameter is treated as a static parameter or a dynamic parameter is not determined until compilation. Parameters that are evaluated and their values hard coded into the compiled dataflow graph area treated as static parameters. Whereas, dynamic parameters are generally not evaluated at compile time, but are instead evaluated during graph loading or execution. As described above, the values determined by the parameter set are

used for the purposes of preparing different compiled versions of dataflow graphs for fast loading and execution at run-time. In the case where the value of the parameter from the parameter set definitively defines the only possible value that is valid for the compiled graph, the value is coded into the compiled dataflow graph, and the parameter is treated as a static
5 parameter. In other cases, where the value of the parameter from the parameter set provides a range of possible values, the parameter may not be evaluated at compile time as a static parameter, but instead may be evaluated at load-time or run-time as a dynamic parameter.

Also during the compilation process the compiler may optimize the dataflow graph, for example, by eliminating unnecessary executable statements. For example, dataflow graphs may
10 contain conditional components. Conditional components may include a series of executable statement which are either included in or excluded from the compiled dataflow graph based on the value of one of more parameters. Conditional components can be used for a variety of purposes, such as graph optimization or specialization. For graph optimization, an application may omit processing or creation of datasets if values from them will not be used, thus allowing
15 the graph to run more efficiently. For graph specialization, an application might condition the production of several different output datasets based on the level of detail desired, or allow execution of one of several optional portions of a graph.

The techniques described above can be implemented using software for execution on a computer. For instance, the software forms procedures in one or more computer programs that
20 execute on one or more programmed or programmable computer systems (which can be of various architectures such as distributed, client/server, or grid) each including at least one processor, at least one data storage system (including volatile and non-volatile memory and/or storage elements), at least one input device or port, and at least one output device or port. The software can form one or more modules of a larger program, for example, that provides other
25 services related to the design and configuration of dataflow graphs. The nodes or components and elements of the graph can be implemented as data structures stored in a computer readable medium or other organized data conforming to a data model stored in a data repository.

The software can be provided on a storage medium and/or a hardware storage device, such as a CD-ROM, readable by a general or special purpose programmable computer, or
30 delivered (encoded in a propagated signal) over a communication medium of a network to a storage medium of the computer where it is executed. All of the functions can be performed on a

special purpose computer, or using special-purpose hardware, such as coprocessors. The software can be implemented in a distributed manner in which different parts of the computation specified by the software are performed by different computers. Each such computer program is preferably stored on or downloaded to a storage media or device (e.g., solid state memory or media, or magnetic or optical media) readable by a general or special purpose programmable computer, for configuring and operating the computer when the storage media or device is read by the computer system to perform the procedures described herein. The inventive system can also be considered to be implemented as a computer-readable storage medium, configured with a computer program, where the storage medium so configured causes a computer system to operate in a specific and predefined manner to perform the functions described herein.

A number of embodiments of the invention have been described. Nevertheless, it will be understood that various modifications can be made without departing from the spirit and scope of the invention. For example, some of the steps described above can be order independent, and thus can be performed in an order different from that described.

It is to be understood that the foregoing description is intended to illustrate and not to limit the scope of the invention, which is defined by the scope of the appended claims. For example, a number of the function steps described above can be performed in a different order without substantially affecting overall processing. Other embodiments are within the scope of the following claims.

CLAIMS

1. A computer-implemented method for executing a dataflow graph that represents a query on data items in a stream of near real-time data to provide, as the dataflow graph is being executed, intermediate results for the query, the method including:

receiving a stream of near real-time data having data items located in different places in the stream;

between a first time and a second time, intermittently executing the dataflow graph that represents the query multiple times, with the dataflow graph being executed by one or more computer systems in near real-time with respect to receipt of the stream of near real-time data and being executed upon the stream of near real-time data for two or more of the data items, with the dataflow graph including computer code to implement the query, and with the dataflow graph receiving as input query specifications for the query;

generating, during execution of the dataflow graph, one or more query results that satisfy the query;

generating intermediate results from the one or more query results, as the dataflow graph intermittently executes between the first time and the second time, by aggregating the one or more query results with one or more prior query results of one or more prior executions of the dataflow graph on the stream of near real-time data for the two or more of the data items that previously appeared in the stream of near real-time data; and

transmitting to one or more client devices the intermediate results during intermittent execution of the dataflow graph, prior to completion of execution of the dataflow graph.

2. The computer-implemented method of claim 1, further including:
at a subsequent point in time, aggregating the intermediate results with results for executing the dataflow graph at the subsequent point in time to generate final results.

3. The computer-implemented method of claim 2, further including:
storing the final results and discarding the intermediate results.

4. The computer-implemented method of claim 1, wherein intermittently executing the dataflow graph includes:

executing the dataflow graph on a first one of the data items in the stream of near real-time data located in a first portion of the stream of near real-time data, and

executing the dataflow graph on a second one of the data items in the stream of near real-time data located in a second portion of the stream.

5. The computer-implemented method of claim 1, wherein the dataflow graph includes components that represent operations to be performed in execution of the query, and wherein the method further includes:

for a component:

performing a checkpoint operation that saves a local state of the component to enable recoverability of a state of the dataflow graph.

6. The computer-implemented method of claim 1, wherein the dataflow graph is executed on data items that appear in the stream of near real-time data during a period of time the end of which is unknown at a start of executing the dataflow graph.

7. The computer-implemented method of claim 1, wherein an amount of data items in the stream of near real-time data on which the dataflow graph is executed is unknown at a start of executing the dataflow graph.

8. The computer-implemented method of claim 6, further including:
accessing information indicative of user-defined custom operations for data transformation on the aggregated results;

executing the user-defined custom operations on the aggregated results; and
transforming the aggregated results in accordance with the user-defined custom operations.

9. The computer-implemented method of claim 1, further including:

generating, based on the aggregated results, a near real-time alert to alert a user of detection of a pre-defined condition.

10. The computer-implemented method of claim 1, wherein the stream of near real-time data includes a stream of near real-time data in which data items are (i) periodically received at different times or (ii) continuously received at different times.

11. The computer-implemented method of claim 1, further including receiving the stream of near real-time data from a data queue, a data repository, or a data feed.

12. The computer-implemented method of claim 1, wherein the dataflow graph is a first dataflow graph, with the method further including:

selecting a second dataflow graph for execution on two or more of the data items that appear at different locations in the stream of near real-time data; and

executing the first and second dataflow graphs in near real-time with respect to the data items of the stream of near real-time data.

13. The computer-implemented method of claim 1, further including:
generating information for a user interface that when rendered on a display device includes:

input fields for input of information defining dataflow graphs to be executed on the stream of data.

14. The computer-implemented method of claim 1, further including:
accessing, in a control repository, pre-defined dataflow graphs that are candidates for execution on the stream of near real-time data.

15. A system for executing a dataflow graph that represents a query on data items in a stream of near real-time data to provide, as the dataflow graph is being executed, intermediate results for the query, the system including:

one or more processing devices; and

one or more machine-readable hardware storage devices storing instructions that are executable by the one or more processing devices to perform operations including:

- receiving a stream of near real-time data having data items located in different places in the stream;

- between a first time and a second time, intermittently executing the dataflow graph that represents the query multiple times, with the dataflow graph being executed by one or more computer systems in near real-time with respect to receipt of the stream of near real-time data and being executed upon the stream of near real-time data for two or more of the data items, with the dataflow graph including computer code to implement the query, and with the dataflow graph receiving as input query specifications for the query;

- generating, during execution of the dataflow graph, one or more query results that satisfy the query;

- generating intermediate results from the one or more query results, as the dataflow graph intermittently executes between the first time and the second time, by aggregating the one or more query results with one or more prior query results of one or more prior executions of the dataflow graph on the stream of near real-time data for the two or more of the data items that previously appeared in the stream of near real-time data; and

- transmitting to one or more client devices the intermediate results during intermittent execution of the dataflow graph, prior to completion of execution of the dataflow graph.

16. The system of claim 15, wherein the operations further include:
at a subsequent point in time, aggregating the intermediate results with results for executing the dataflow graph at the subsequent point in time to generate final results.

17. The system of claim 16, wherein the operations further include:
storing the final results and discarding the intermediate results.

18. The system of claim 15, wherein intermittently executing the dataflow graph includes:

executing the dataflow graph on a first one of the data items in the stream of near real-time data located in a first portion of the stream of near real-time data, and

executing the dataflow graph on a second one of the data items in the stream of near real-time data located in a second portion of the stream.

19. The system of claim 15, wherein the dataflow graph includes components that represent operations to be performed in execution of the query, and wherein the operations further include:

for a component:

performing a checkpoint operation that saves a local state of the component to enable recoverability of a state of the dataflow graph.

20. The system of claim 15, wherein the dataflow graph is executed on data items that appear in the stream of near real-time data during a period of time the end of which is unknown at a start of executing the dataflow graph.

21. The system of claim 15, wherein an amount of data items in the stream of near real-time data on which the dataflow graph is executed is unknown at a start of executing the dataflow graph.

22. One or more machine-readable hardware storages for executing a dataflow graph that represents a query on data items in a stream of near real-time data to provide, as the dataflow graph is being executed, intermediate results for the query, the one or more machine-readable hardware storages storing instructions that are executable by one or more processing devices to perform operations including:

receiving a stream of near real-time data having data items located in different places in the stream;

between a first time and a second time, intermittently executing the dataflow graph that represents the query multiple times, with the dataflow graph being executed by one or more

computer systems in near real-time with respect to receipt of the stream of near real-time data and being executed upon the stream of near real-time data for two or more of the data items, with the dataflow graph including computer code to implement the query, and with the dataflow graph receiving as input query specifications for the query;

generating, during execution of the dataflow graph, one or more query results that satisfy the query;

generating intermediate results from the one or more query results, as the dataflow graph intermittently executes between the first time and the second time, by aggregating the one or more query results with one or more prior query results of one or more prior executions of the dataflow graph on the stream of near real-time data for the two or more of the data items that previously appeared in the stream of near real-time data; and

transmitting to one or more client devices the intermediate results during intermittent execution of the dataflow graph, prior to completion of execution of the dataflow graph.