Distributed Mega-Datasets: The Need for Novel Computing Primitives

Niklas Semmler
SAP SE

Georgios Smaragdakis
TU Berlin

Anja Feldmann
Max Planck Institute for Informatics

***Abstract*—With the ongoing digitalization, an increasing number of sensors is becoming part of our digital infrastructure. These sensors produce highly, even globally, distributed data streams. The aggregate data rate of these streams far exceeds local storage and computing capabilities. Yet, for radical new services (e.g., predictive maintenance and autonomous driving), which depend on various control loops, this data needs to be analyzed in a timely fashion.**

In this position paper, we outline a system architecture that can effectively handle distributed mega-datasets using data aggregation. Hereby, we point out two research challenges: The need for (1) novel computing primitives that allow us to aggregate data at scale across multiple hierarchies (i.e., time and location) while answering a multitude of a priori unknown queries, and (2) transfer optimizations that enable rapid local and global decision making.

***Index Terms*—database, computer network, computer system**

I. Introduction

A few decades ago, *digitalization*[1] entered many processes of modern society, from public administration to e-commerce, from transportation to industrial automation, ultimately permeating our everyday lives. Now, we are in the middle of a digital revolution of unprecedented intensity where digitalization will force the future *convergence of the physical and digital worlds* with ubiquitous, novel, and disruptive applications. A prerequisite for the digital future is *timely* and *dependable* information from the physical world that allows the physical processes and their digital representations to interact through multiple real-time control loops at different levels of time, spatial scale, and detail.

We consider settings, e.g., smart factory and network management, where data, physical processes, and their models are distributed [15]. Consistency, real-time accuracy, data sharing, joined processing, and data privacy needs to be provided based on user and or process requirements. They are subject to the constraints of the available computing and bandwidth resources as well as security and data privacy restrictions.

Maintaining a digital representation of the real world in a consistent and timely manner is challenging. The challenges arise when handling data at different scales of time (from sub-millisecond to hours, and days, spanning easily 6-7 orders of magnitude) and space. Other challenges include handling heterogeneous, non-stationary, and non-uniform, distributed data streams. Furthermore, sensing must efficiently produce summaries of the physical world, while achieving specific (task-defined) approximation guarantees and respecting privacy during data integration and analysis.

Our vision is driven by the increasing deployment of sensors and their increasing resolution, particularly at the edge of the Internet [1], [3], [4]. These sensors produce data at a rate that outpaces the capacity growth of wide-area networks.

This trend has led to the creation of datasets that originate from many different sources. These datasets can no longer be fully stored and/or processed within a single computer system. We refer to them as *mega-datasets*. A mega-dataset can only be handled by a distributed, yet *local* system, e.g., a cluster of compute nodes. We define a *distributed mega-dataset* as a collection of physically distributed mega-datasets.

Processing the distributed mega-datasets, in a coordinated, yet distributed fashion in real-time requires novel *computing primitives*. Such primitives need to summarize data in a form that supports (a) support arbitrary queries on the data, (b) combining summaries gained from different locations or at different moments in time times, (c) adjusting the aggregation granularity, (d) adapting to variations in data and queries. Further, these primitives should (e) make use of domain knowledge to provide meaningful levels of aggregation. At the same time, they need to use minimal resources and address data lineage, quality criteria, and time constraints.

In this paper, we present our vision of how distributed mega-dataset can be successfully handled. We introduce a possible architecture and give two examples of computing primitives to handle them. Throughout the remainder of this paper, we use two cases, namely, Smart Factory [19], [23] and Network Monitoring [20].

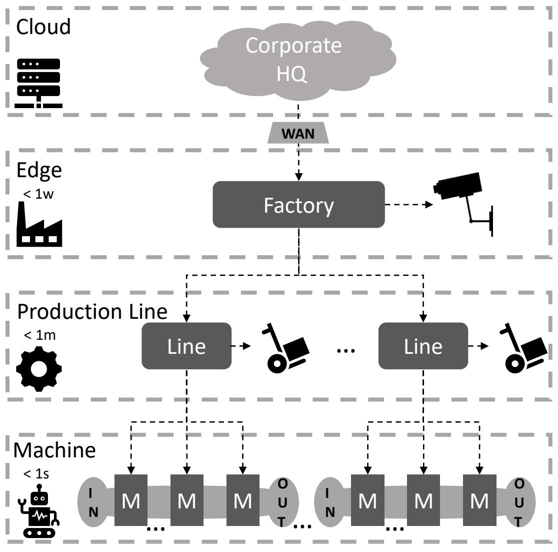
II. Use Cases

A. Use Case: Smart Factory

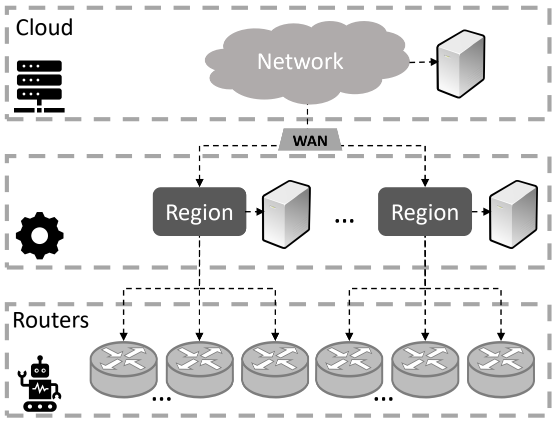
Traditionally, a factory consisted of single-purpose machines, which followed a rigid sequence of instructions and interacted with each other, e.g., by moving goods over a conveyor belt. Hereby, monitoring was limited to a small number of sensors with limited capabilities. This factory design limited the machine's range of operation and necessitated constant supervision and frequent interventions by human operators.

Drastic changes have happened since then, e.g., the introduction of collaborative robotic arms [17] and autonomous forklifts [5]. Robotic arms extend the range of movements that a machine can perform and simplifies the interaction with

[1]Digitalization here refers to the process of leveraging digitization (the conversion of analog media into digital form) to improve business processes.



(a) Smart Factory.



(b) Network Monitoring.

Fig. 1: Two settings with distributed mega-datasets: Smart Factory (left) and Network Monitoring (right). The dotted lines represent the lines of (partial) control from higher to lower hierarchy levels.

human operators. Autonomous forklifts extend the reach of factory automation into the warehouse. Both are examples of recent innovations, which are enabled by a combination of high-resolution sensors (e.g., 3D cameras) and a multitude of lower-resolution sensors. As a consequence, their data rates have exploded, e.g., a single 3D camera can produce 52 GB/h of uncompressed data and a high-resolution camera can produce 17.5 GB/h of uncompressed data. The data rates have increased to a degree that they often require dedicated data processing equipment, which enables new more complex behavior and richer interactions. Still, these innovations are constrained by limited compute and storage resources in the vicinity of the machine and limited sharing of data across machines.

The next step in the evolution of factories will be "the" fully automated factory consisting of autonomously operating parts. It will rely on *rapid local decision making* while respecting today's factories setups (particularly at the machine and production line level) and at the same time it will be able to *constantly adapt* to insights gained across *factory lines* and *different factory locations*. This adaptation will enable improvements to the efficiency of existing processes, e.g., adjustments to degrading machine mechanics, as well as enable new processes, e.g., for mass customization.

In Figure 1, we illustrate (on the left) the typical hierarchical structure of a factory. Machines connected by a conveyor belt or related technology (the production line) are located at the lowest level. The controller of the production line is one level above. Besides the control of the respective machines, this level may also control supporting processes, such as the movement of materials and products. All production lines are monitored and managed on the factory level, which may leverage additional data (e.g., from factory cameras). Some part of the gathered data/information may be exported into the Cloud resp. dedicated datacenters, where it can be combined with additional resources (e.g., for Enterprise Resource Planning or ERP).

Given the ever-increasing data flood from all sensors, *mega-datasets* arise at different points in the factory. As a whole, they form a *distributed mega-dataset*. Different applications require this data to be processed in different ways. While some require the data to be processed immediately as data streams, others require it to be stored, either temporarily or almost permanently, to answer (interactive) queries at a later point, e.g., on the history of produced goods for supply-chain management. Similarly, applications have different requirements: they differ in the degree of precision that they require, e.g., in terms of measurements over time; whether they require data from a single mega-dataset or multiple mega-datasets; what timeliness they require, e.g, decision making at the machine resp. factory level may require results between 1 second and 1 minute respectively.

There are many applications that can be enabled by better use of the factories' data. To name but a few: (a) *predictive maintenance*, the analysis of operational data belonging to a type or class machines of machines to predict failures and schedule maintenance operations accordingly; (b) *supply chain management*, procedures for tracing product failures back to the material used in the production steps or to variations in the production process itself; (c) *process mining*, the review of production processes attained by combining operational data and enterprise data to identify sources for efficiency gains.

#	Challenge	Smart Factory	Network Monitoring
1	Increasing computation requirements	High-resolution camera feed	High-speed traffic inspection
2	Large number of devices producing data streams	Streams of sensor data	Streams of flow data
3	Massive combined data rates	Machine and factory-level sensors	flow exports from switches, routers, etc.
4	Rapid local decision making	Machine control	Repair of network failures
5	High data variability	Differing sensor types	Logs, flows, packets
6	Analytics require full knowledge	Predictive maintenance	Traffic engineering & provisioning
7	Hierarchical structure	Machines, production lines, factories	Devices, regions, networks
8	Varying requirements across applications	Maintenance *vs.* process optimization	Attack mitigation *vs.* load balancing
9	A priori unknown queries	state of production	network state

TABLE I: Challenges of distributed mega-datasets and examples from both use cases

B. Use Case: Network Monitoring

Today's de facto communication medium is the Internet, a network of networks. To cope with the increasing demand and complexity, network operators have to manage their networks. Network management requires an accurate view of the network, based on the continuous analysis of network data, i.e., network monitoring.

In the past a coarse-grained view of the network was sufficient. Today, network operators must have a fine-grained view of their networks. They have to continuously keep track of their network activity both over relatively large time windows, e.g., days or busy hour (6pm to 12am), for network provisioning or to make informed peering decisions as well as over smaller time windows, e.g., minutes, to identify and rectify unusual events, e.g., attacks or network disruptions. To that end, they typically rely on either flow-level or packet-level captures from routers within their network. As gathering such data for every packet is often too expensive at high-speed links, packets are sampled, e.g., 1 of every 10K packets [7]. Still, even the resulting datasets can exceed multiple Terabytes per day and router. Thus, sending this data to a central location may or may not be possible, e.g., due to data protection regulations, or desirable, due to bandwidth restrictions. Rather, each dataset produced by a set of routers forms a "local/regional" *mega-dataset*. These datasets together with additional network configuration data, e.g., topology, network element state, and routing configuration, form a *distributed mega-dataset*.

Many problems that network operators face can be resolved by analyzing such a distributed mega-dataset or a subset of mega-datasets. It can help to (a) determine network trends, e.g., popular network applications or traffic sources; (b) compute traffic matrices, for planning network upgrades, (c) investigate performance and/or DDoS incidents, i.e., identify affected network parts and possible sources, (d) perform dynamic traffic engineering, by aggregating flow statistics across time and sites, (e) answer interactive queries on the state of the network.

III. Architecture

In both use cases distributed mega-datasets arise and similar challenges have to be addressed. We summarize 9 key challenges in Table I and associate each of these challenges with the operation of smart factories (middle column) and network monitoring (right column). These challenges motivate us to propose the following novel data processing architecture.

The design of our architecture leverages the following observations: (a) data has to be filtered and aggregated prior to computation (Challenge 3), (b) data from different data streams, with varying rates and characteristics, have to be combined (Challenge 2), (c) processing has to enable local (Challenge 4) and far-reaching control loops (Challenge 6), (d) processing has to be spread out over a hierarchy of physical processes, resources, and restrictions (Challenges 1, 7), (e) computation has to be modular to include different aggregation, analytics, and application logic (Challenges 8 and 5) and (f) data has to be stored at different aggregation granularity for future, yet unknown, queries (Challenges 9).

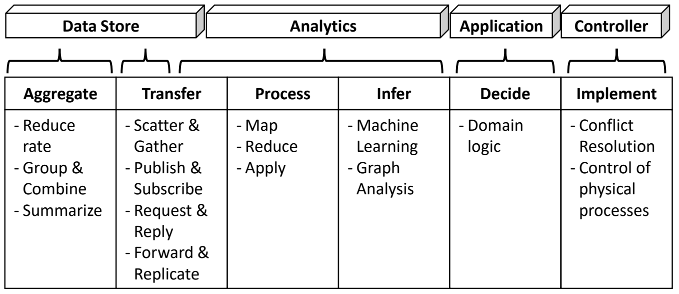
Flexible yet resource efficient data summarization is at the heart of extracting timely information from distributed mega-datasets. For this purpose, we propose novel *computing primitives* that construct summaries of the data, which can be combined across the hierarchy.

A. Building Blocks

We base our architecture on four building blocks, see Figure 2a. These are *Data Stores* ("collect & aggregate"), *Analytics* ("transfer & process"), *Applications* ("model & learn"), and *Controllers* ("resolve conflicts & decide"). In the following, we describe each building block in more detail. For an illustration of the architecture see Figure 2b.

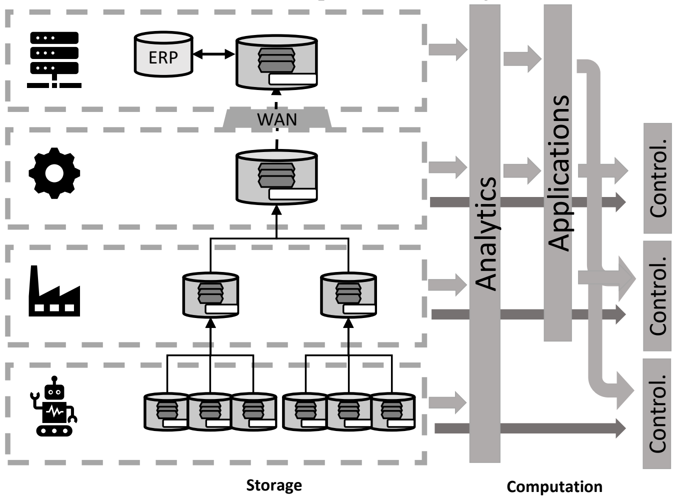
Data Store: A data store aggregates data, using one or multiple instances of *computing primitives*, which we refer to as aggregators. We describe data stores in depth in Section IV and computing primitives in Section V.

Applications: To satisfy the varying needs of the users of the distributed mega-datasets, we envision a range of different applications. Each application embodies the decision logic for a single purpose. Applications can be long-running processes or enable short-term queries. They function as an interface to the users to gather information from the data stores. Thus, they can serve monitoring or reporting purposes. We envision that most applications run on a compute cluster, either at the edge or in a datacenter. Applications can be purely local, e.g., one that supervises the temperature of all machines of a specific type, or global ones, e.g., one that tracks the efficiency of different factories in different countries. Other application examples are process mining and predictive maintenance. Applications have two ways to interact with the physical world. They can either contact the controller, to manipulate



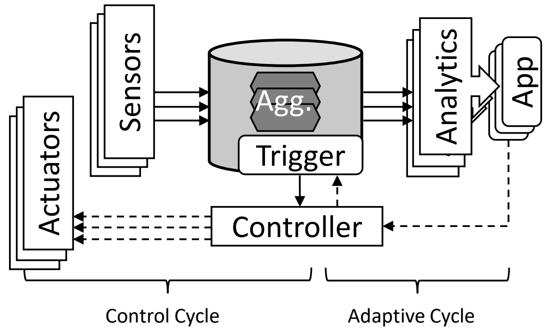
Data Store		Analytics		Application	Controller
Aggregate	**Transfer**	**Process**	**Infer**	**Decide**	**Implement**
- Reduce rate - Group & Combine - Summarize	- Scatter & Gather - Publish & Subscribe - Request & Reply - Forward & Replicate	- Map - Reduce - Apply	- Machine Learning - Graph Analysis	- Domain logic	- Conflict Resolution - Control of physical processes

(a) The feedback loop in four building blocks

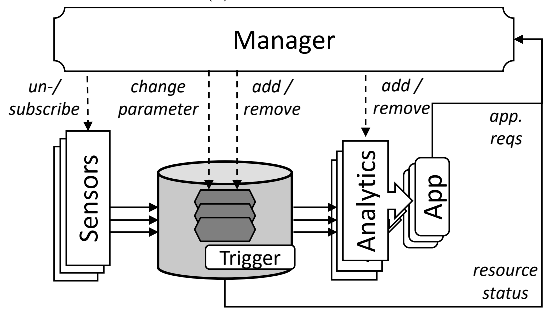


(b) Architecture (view of the full hierarchy)

Fig. 2: Breakdown of processing



(a) Data Plane



(b) Control Plane

Fig. 3: Architecture (view from the perspective of a level)

physical processes directly, or install *triggers* in the data store, to influence future behavior. As the name suggests, triggers are triggered by events and then signal a controller. We envision that the latter can be used for simple conditions that need real-time reactions while the former is used to detect complex situations and may require complex actions, e.g., an update of the controller's logic.

Analytics: Many applications require processing beyond the computing primitives capabilities that can be installed within the data stores. They may require computation ranging from big data analytics, e.g., primitives that exploit "embarrassingly parallel" computation (MapReduce, etc.), which can run on various devices, to more complex computations, even those that require specific hardware (e.g., GPUs, TPUs or FPGAs). In principle, we envision Analytics as a toolset that includes machine learning, graph processing, as well as big data systems such as Flink [16], Spark [14], etc. But, it does not have to stop here. Rather, it can also include visualization and statistics toolkits, e.g., R [10] or MATLAB [21], or tools to build interactive data visualizations, using e.g., Shiny [2].

Controller: For operating at production speed, machines may not be able to wait for input from applications. Yet, some validation may be necessary to avoid failures, e.g., raising a robot arm beyond its highest point. Thus, we envision a local control logic, the controller. The controller monitors the machine, e.g., using a set of patterns, which can be installed at the data stores as triggers. When a trigger matches the controller is activated and regulates the machine accordingly. The logic for the controller is installed and updated by individual applications but are checked for conflicts by the controller prior to installation.

B. Data and Control flow

The Manager component manages the architecture. We describe both the data and the control plane in turn.

Data flow: Having described the components of our system, we now turn to explain how data flows through the system and fires consecutive actions. Data is first pushed from the sensor (based on a request or a prior subscription) to the data store. In the data store, data is aggregated according to a chosen set of *computing primitives*. If any registered trigger matches a data summary (including possibly raw storage of the data), it activates the controller which regulates the respective machine(s). Independent from whether or not triggers are activated, the data store sends data to any registered Analytics pipeline. The pipeline performs pre-processing (e.g., using MapReduce), data transfer (scatter and gather semantics) and inference (e.g., using a Machine Learning algorithm). A pipeline feeds the processed data to one or possibly many applications. The applications, in turn, decide whether to install new rules in the controller and can also forward the data for monitoring or reporting purposes. Conflicts between rules are resolved locally at the controller.

Control flow/Manager: Its configuration is a major challenge in realizing the architecture. The control plane which we envision is shown in Figure 3b. The Manager assigns and adapts resources according to the varying application needs. For each application, it records the application requirements in terms of the required data source and aggregation format (e.g., sample or histogram) and the required precision (e.g., sample rate or bin size). The manager then uses this information to decide (a) what data should be kept from which sensors (b) what computing primitive should be installed, (c) how the computing primitives should be configured and (d) what analytics is deployed within the infrastructure (from the level of the machine up to the datacenter). Besides the storage within the data stores (shown in Figure 3b), the Manager tracks the availability of network bandwidth and computing nodes across the architecture. In summary, the manager controls all components of the architecture.

Hierarchy: So far, we described the architecture that handles one data store, which corresponds to one single mega-dataset. In the case of distributed mega-datasets, each mega-dataset is stored in its own data store. Further data stores exist to merge and aggregate data from multiple mega-datasets, depending on the need of the applications. We have depicted an example of a hierarchy in Figure 2b. In this example, the data stores responsible for combining data are located closer to a center of the infrastructure, i.e., closer to compute clusters, while the other data stores are located closer to the majority of physical processes. The manager decides what data stores should be deployed based on the needs of the applications and connects the Analytics pipelines with the respective data stores.

C. Security, Privacy, Lineage, and Integration

Security & Privacy: In our architecture, privacy can be enforced, by limiting what summaries can be shared with the analytics component and at what granularity. Other summaries and more precise data may still be used by a local Controller. Security can be achieved, by encrypting data along the Analytics pipelines, requiring updates to the Controller to be certified to ensure authenticity, and by requiring authorization prior to interaction with the manager.

Lineage: An unavoidable problem in systems interacting with and processing sensor data is faulty or missing data. To address this problem we need to track data as it moves through and is transformed by the system. This process is referred to as "lineage". Data lineage can, e.g., be used to identify faulty sensors or retract erroneous rules. Data lineage [6], [22] can be differentiated into schema-level and instance-level lineage. Schema-level lineage tracks the transformation of data from sensors to applications with respect to changing formats and the locations of transformations. It can help to identify how data came to its current format, but it cannot give information on specific results. Instance-level lineage tracks individual items as they move through the system. It can be used to see how faulty data propagates, but it usually comes at a high cost (high overhead). The identification of a lineage mechanism for each computing primitive and analytic component at the envisioned data rate and with the flexibility of this system is an unsolved challenge.

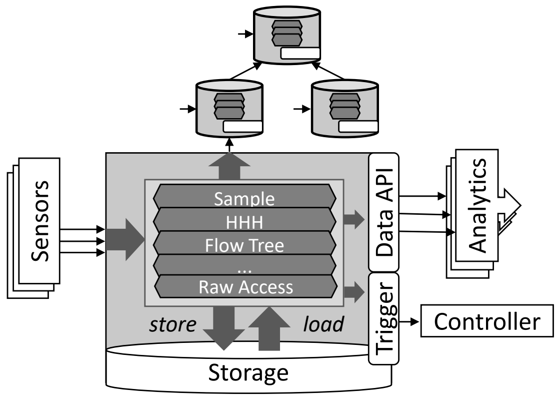


Fig. 4: Datastore

Integration: Another problem that we have not touched upon is the integration of data beyond the operational data. We envision that applications will also make use of business data, e.g., the Enterprise Resource Planning database. This can be realized by the interaction between the business data and a data store located at a central data center.

IV. Data Store

We show our vision for the data store in Figure 4. The data store selects and collects data from sensors and then feeds their data into aggregators, instances of computing primitives, that have subscribed to the respective data streams. Queries received by the data store are broken into sub-queries and are forwarded to the respective aggregator. Sub-queries for aggregators stored at other data stores are forwarded or resolved on a local replicate. The main responsibility of the data store is to ensure that the storage and network resources are efficiently used:

Storage: Note that the data store is the only entity in our architecture, which stores data. All other elements might cache data or intermediate results (depending on their implementation), but they are not expected to persistently store raw or summarized data. Hence, when a data store chooses to delete data, it cannot be recovered. Thus, storage space has to be carefully allocated and managed.

We identify three basic strategies for storing data in the data store: (1) storage with predefined expiration, (2) storage using a round-robin mechanism, and (3) storage using a round-robin mechanism and hierarchical aggregation. The first strategy gives application developers the guarantee that data is stored for a fix time duration. Note, choosing the time period optimally in advanced may be difficult. The second strategy optimizes the use of storage in the sense that it fully utilizes the storage. In this case, the duration that the data is stored depends on the data rate. The third strategy is a combination of the first and second one in the sense that older data is not expired but aggregated to a coarser granularity with a smaller footprint and then stored. This guarantees long-term storage but at the price of reduced detail due to aggregation. More sophisticated strategies may regard stored data as a storage investment that has to be traded off against future queries that it will help answer. These strategies may compress or delete data that is deemed of lesser interest in the long-term.

Network Transfers: Different data stores gather data at different locations. Yet, some analysis requires their joint processing. Thus, data in one data store may have to be combined with data from other data stores to answer queries across the distributed mega-dataset. In this case, the data store has the choice of (1) shipping the query to the data or (2) *replicating* the respective aggregator(s). A basic strategy for this decision is to replicate the data produced by an aggregator when the data it holds has been accessed at least n number of times by a remote data store, when at least b of its bytes have been transferred or when it has created a transfer volume of at least p percent of its own storage volume. Each of these strategies is heuristic in nature. More sophisticated strategies can be developed using predictions of future accesses. We address this problem in more detail in Section VII.

V. Computing Primitives

A major challenge in realizing our architecture is the efficient summarization of data across multiple sources and locations to answer a priory unknown queries. This is accomplished by computing primitives, which can be used by the individual data store to create data summaries, aggregates of raw data, from the incoming data streams. Computing primitives can use aggregation methods from simple statistics over time bins (e.g., sum, mean, median, and standard deviation) and sampling methods to more complicated streaming algorithms (e.g., heavy hitter detection or even hierarchical heavy hitter detection).

Yet, none of the above methods are suited to unleash the full capabilities of our proposed architecture as they do not support any of the following: support arbitrary queries, enable the combination of data summaries, have an adjustable level of aggregation, self-adapt to incoming data and queries and take domain knowledge into account, to create more meaningful summaries. Thus, we need novel computing primitives. In the following, we discuss these desired properties in more detail.

A. Design properties

Support arbitrary queries: Each computing primitive needs to enable arbitrary queries, particular a priori unknown queries, on its respective data summaries. While the format of the queries depends on the particular data organization, each primitive should permit flexibility, e.g., in the degree of precision.

Can combine summaries: For integration into the hierarchy of data stores, computing primitive should be able to combine data summaries. Each summary represents a single time interval and a collection of data streams at a single location. Hence, a combined data summary can answer queries over data from multiple locations (including differences between the locations).

Adjust the level of aggregation granularity: In most cases, the raw data is produced at rates that are too high for storage or timely processing. Therefore, computing primitives should aggregate data into summaries that can be stored and timely processed. Furthermore, to deal with volatility in the rate of incoming data streams, computing primitives should be able to adjust their level of aggregation granularity over time.

Self-adaptive: Every summary produced by a computing primitive increases the storage and processing footprint of the data store. To limit the occupied storage, the summary should continuously re-organize the data it stores and its level of aggregation granularity according to the incoming data streams and queries. If the manager were to know all future requests in advance, e.g., because the set of application is fixed, it would be straightforward for the manager to choose the appropriate aggregation level for the computing primitive. Yet, most of the time this information is not available. Therefore, computing primitive should ideally be able to adjust the granularity on demand. Where this is not possible due to the aggregation method, the applications may be forced to specify at which aggregation level they want to operate.

Uses domain knowledge: Knowledge of the data domain can help create more meaningful computing primitives where aggregation has a semantic relationship to the data. Such computing primitive can enable queries to express their desired level of aggregation in terms of the domain.

B. Toy Example

A toy example of a computing primitive can be an aggregator that uses random sampling. This primitive can produce a data summary in the form of a sampled time series and has the following properties:

- **Query:** It enables queries on a time series, e.g., by selecting all data points in a given time frame that exceed a given value.
- **Combine:** Two summaries, i.e., time series, can be combined by combining individual data points from the respective time series.
- **Aggregate:** The level of aggregation can be changed by adjusting the sampling rate of the time series.
- **Self-adapt:** The time granularity required by incoming queries and the rate of the incoming data can be used to adjust the sampling rate.
- **Domain knowledge:** This computing primitive using random sampling is an example of aggregation without domain knowledge.

VI. Example: Computing primitives for Network Monitoring

Here, we revisit the use-case of network monitoring and present a computing primitive for this use case. In the case of network monitoring, data summaries across the hierarchy should enable queries ranging from network troubleshooting to routine summaries on a set of distributed flow features vectors. (Recall, a flow feature vector is, typically, a 5-tuple and summarizes traffic information per flow: packet and byte count.) The challenge here is that we need succinct and space

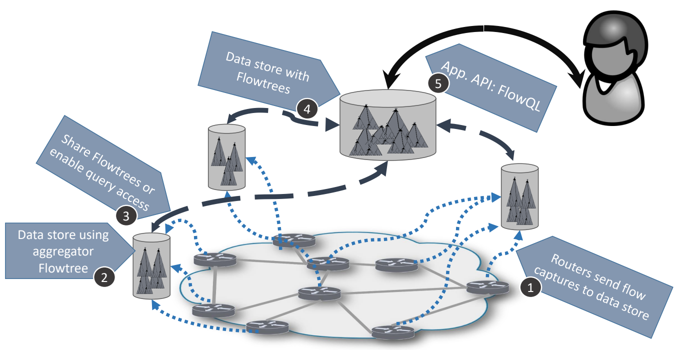

Data store with Flowtrees
4
App. API: FlowQL
5
Share Flowtrees or enable query access
3
Data store using aggregator Flowtree
2
Routers send flow captures to data store
1

Fig. 5: Flowstream system overview

Operator	Description
Merge	Join two Flowtrees into one (requires either shared time or location).
Compress	Summarize the lower level nodes of a Flowtree.
Diff	Subtract the popularity scores from flows appearing in one tree from the other.
Query	Return the popularity score of a single flow.
Drilldown	Return the flows and popularity scores that are children of a single flow.
Top-k	Show the k flows with the highest popularity score.
Above-x	Show all flows that have a popularity score above x.
HHH	Return all flows across the Flowtree that have a substantial popularity score.

TABLE II: Flowtree operators

efficient data summaries of network flows to *accurately* and *quickly* answer queries and tackle network management tasks that involve *multiple sites* and/or span *multiple time periods*.

Our current proposal is to rely on the concept of generalized flows as aggregation unit: Flows summarize related packets over time at a specific aggregation granularity. Possible flow types include "5-feature" flows, i.e., protocol, source and destination IP, source and destination port. Other flow types are "2-feature" flows, e.g., source and destination IP or destination IP and destination port. Each feature can be generalized by using a mask, e.g., by moving from an IP to a prefix. By generalizing from flows in this way, we arrive at hierarchies of features, whereby an IP a.b.c.d is part of the prefix a.b.c.d/n_1 and a.b.c.d/n_1 is a more specific of a.b.c.d/n_2 if $n_1 > n_2$.

We propose a novel computing primitive, namely Flowtree, for computing generalized flows. Flowtree is a self-adjusting data structure which uses *existing network traces* as input and works on the fly. Since the input data is often heavily sampled prior to ingestion, the Flowtree does *not* provide *exact summaries*. Rather, it allows us to distinguish heavy hitters from non-popular flows and summarize data across different network sites and time.

Flowtree takes advantage of the fact that flows are part of a tree, where each observed flow and each generalized flow thereof is a node. One node is a parent of the other when its flow is the most specific generalized flow of the other. We annotate each node with a popularity score, which can be either its packet count, flow count, byte count, or combinations thereof. The popularity score of a node is the sum of its own popularity score plus the popularity scores of the children.

Flowtree supports a number of operators, which we have described in Table II. *Merge* and *Compress* enable us to compute efficient summaries across time and/or space. In effect, they allow us to add the *time* and *location* as features. For example, given two instances of the data structure A_1 for time period t_1 (location l_1) and A_2 for t_2 (l_2) we construct a summary for the joined time period (both locations) by setting $A_{12} = \text{compress}(A_1 \cup A_2)$. For more details see [20]. Flowtree is an example of a novel computing primitive. It supports queries, enables the combination (*Merge* and *Diff*) of data summaries, includes an adjustable level of aggregation and adapts the level of aggregation to the incoming data using its tree structure. Furthermore, it is modeled closely after the data domain, which enables aggregation along the level of subnets.

We propose to use Flowtree as computing primitive per flow within the data store. Parameters at each data store include feature sets as well as time and location granularity that are kept at the data store. The resulting Flowtrees can (a) be exported to data stores, e.g., to be build trees at larger time or location granularity, (b) be exported to an analytic engine, which we refer to as FlowDB, or (c) can be queried directly by an application. FlowDB takes flow summaries as input, stores, and indexes them while using them to answer FlowQL queries. FlowQL is an SQL-like query language which uses Flowtrees operators to answer network management questions. More specifically, with FlowQL the user chooses his operator via a `SELECT` clause, one or multiple time periods via a `FROM` clause, and the feature set via a `WHERE` clause. Moreover, he can use the `WHERE` clause to add restrictions, e.g., source IP = a.b.c/24.

Together, the resulting system, called Flowstream, see Figure 5, is an instantiation of the architecture outlined in Section III. Thus, the router sends its raw flow data to a data store ①. The data store uses Flowtree as its aggregator to compute summaries ② and potentially exports these to other data stores ③. The data store can either further aggregate them or use them ④ to answer user queries via the FlowQL API ⑤.

VII. Transfer optimization

In Section V we described how data stores have to exchange data to support queries across locations. These data exchanges require careful optimization of the available resources. Particularly, because the resources of a data store will often be shared between multiple instances of computing primitives. In this section, we frame the problem of network optimization as a trade-off between the cost of shipping query results and of replication (illustrated in Figure 6) and give an example of a possible mechanism.

When a data store receives a query for combined data 🅐, it must first ensure that the respective data is locally available. When this is not the case, it has to query the respective data store 🅑 and collect the result 🅒 before it can process 🅓 and answer the query 🅔. The time spent on retrieving data from another data store increases the latency of the individual

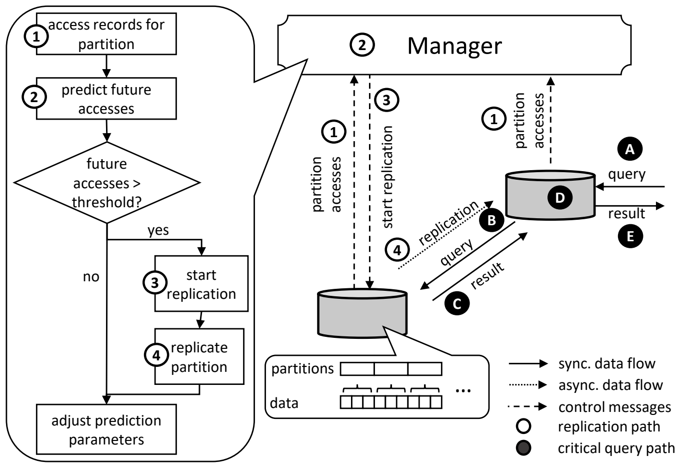


Fig. 6: Optimizing data transfers with adaptive replication

query and, when done repeatedly, increases the system's use of the network. The former can become a problem for applications that have low latency requirements, while the latter can degrade the performance.

The performance can be improved both by reactively caching [18] earlier results and by proactively *replicating* data to the local data store. Of the two, caching is the more constrained approach, as it can only help for repeat queries. For this reason, we focus on replication. (Note, that the approaches are not mutually exclusive, but can be combined.) Due to the size of the mega-datasets, full replication of data between data stores introduces massive transfer and storage cost. Instead, the data maintained by a data store can be partitioned to allow partial replication. Then, the data store can trade off the cost of replicating individual partitions against their estimated future access pattern. In some use cases, this access pattern may be known in advance, e.g., when applications use predefined queries. When it is not known in advance, which is more often the case, access patterns have to be estimated from previous accesses.

In our architecture (see Section III), the accesses of partitions ① can be recorded by the manager. From these accesses, the manager can record, for every partition, the time at which it is accessed and the data volume of query results. The manager can use this information to classify partitions and predict further data transfers that a partition is involved in ②. A threshold can be used to balance the predicted future access of a partition against the cost of replicating the partition. If the predictions for a partition exceed a threshold, the manager can initiate its replication ③. The replication is then executed between the two data stores ④. We refer to the replication of data partitions on the basis of prediction of their future accesses as *adaptive replication*.

In the remainder of the section, we describe one example for an adaptive replication mechanism. As a simple approach, we use the aggregated data volume of past query results of one partition to predict its expected number of future accesses.

To this end, we rely on results derived for the classical ski rental problem [8], [12]. In this theoretical problem, a skier is faced with a choice between buying a ski-set for a large one-time cost and renting the ski-set for a small daily payment. He will face the same decision choice every day, until he either chooses to buy the ski or his skiing career ends. The length of his skiing career is unknown in advance and so he has to choose in the face of uncertainty. This problem is similar to choosing the right moment of replicating a single partition, in that renting the ski-set corresponds to shipping queries, buying the ski-set corresponds to replication and the days to the point in time where queries for the given partition appear. Work by Karlin et al. [12] showed that the best (deterministic) worst-case solution for this problem is to buy the ski-set when money equal to the price of buying has been spent on rent.

Our problem is similar to a variant of the ski rental problem [9], [11], [13] where the probability for the number of skiing days is known to follow a given probability distribution. Under this condition a better solution, i.e., a better threshold can be found for the average case [9], [13]. In our scenario, the probability of future accesses is not known, but the aggregate result size for older partitions are from a distribution that can be used to predict future access for partitions created at a later date. We are currently evaluating this method and variations thereof on an enterprise-level query trace.

In summary, adaptive replication is an essential mechanism to reduce the data volume transferred across the network.

VIII. Summary

Digitalization goes along with data floods from many distributed sources. Using two use cases, smart factory and network monitoring we point out why and how distributed mega-datasets arise. These datasets cannot be stored or processed locally. Yet, they need to be processed to support the correct interactions with and the necessary control loops of the real world. Indeed, an accurate reflection of the digital world with bounded capacities for communication, storage, computation, and accuracy is challenging.

To address this challenge, we propose an architecture for handling mega-datasets which consist of a hierarchy of data stores, applications to address the needs of the users via data analytics, controllers to facilitate interactions with the physical world as directed to by the applications, and a manager that controls the data flow.

Further, we highlight the need for novel computing primitives to efficiently summarize data across different units of time and across the many data sources. We list five essential properties: (1) the support for arbitrary queries, (2) the production of combinable data summaries, (3) an adjustable level of aggregation, (4) self-adaptation of the data summary to the incoming data streams and queries and (5) use of domain knowledge to define meaningful aggregation levels.

We apply our architecture to the network monitoring problem and show how Flowtree, an example of a novel computing primitive, can efficiently handle a mega-dataset (network flow data). Finally, we show how computing primitives can be used to optimize data transfers and improve the performance of our architecture.

Acknowledgments

Georgios Smaragdakis was supported by the European Research Council (ERC) grant ResolutioNet (ERC-StG-679158).

References

[1] B. Baumley. Deutsche Telekom completes world's first public mobile edge network powered by MobiledgeX Edge-Cloud R1.0. https://www.globenewswire.com/news-release/2019/02/19/1734346/0/en/deutsche-telekom-completes-world-s-first-public-mobile-edge-network-powered-by-mobiledgex-edge-cloud-r1-0.html. published: 2019-02-19, accessed: 2019-04-17.

[2] C. Beeley and S. R. Sukhdeve. *Web Application Development with R Using Shiny: Build stunning graphics and interactive data visualizations to deliver cutting-edge analytics*. Packt Publishing Ltd, 2018.

[3] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. In *Big data and internet of things: A roadmap for smart environments*, pages 169–186. Springer, 2014.

[4] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[5] A. Correa, M. R. Walter, L. Fletcher, J. Glass, S. Teller, and R. Davis. Multimodal Interaction with an Autonomous Forklift. In *Proceedings of the 5th ACM/IEEE International Conference on Human-robot Interaction*, HRI '10, pages 243–250, Piscataway, NJ, USA, 2010. IEEE Press. event-place: Osaka, Japan.

[6] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *The VLDB JournalThe International Journal on Very Large Data Bases*, 12(1):41–58, 2003.

[7] N. Duffield, C. Lund, and M. Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *ACM SIGCOMM*, 2003.

[8] A. Fiat. Online algorithms: The state of the art (lecture notes in computer science). 1998.

[9] H. Fujiwara and K. Iwama. Average-case competitive analyses for ski-rental problems. *Algorithmica*, 42(1):95–107, 2005.

[10] R. Ihaka and R. Gentleman. R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3):299–314, 1996.

[11] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 41–55. ACM, 1991.

[12] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

[13] A. Khanafer, M. Kodialam, and K. P. Puttaswamy. The constrained ski-rental problem and its application to online cloud cost optimization. In *INFOCOM, 2013 Proceedings IEEE*, pages 1492–1500. IEEE, 2013.

[14] M. Zaharia and M. Chowdhury and M. J. Franklin and S. Shenker and I. Stoica. Spark: Cluster Computing with Working Sets. In *USENIX conference on Hot topics in cloud computing*, 2010.

[15] D. Mc Hugh. Volkswagen to network factories in the cloud with Amazon. https://www.reuters.com/article/us-volkswagen-amazon-cloud/vw-to-improve-production-with-amazon-cloud-to-network-its-factories-idUSKCN1R80RZ. published: 2019-03-27, accessed: 2019-04-17.

[16] P. Carbone and A. Katsifodimos and S. Ewen and V. Markl and S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[17] M. Peshkin, J. Colgate, W. Wannasuphoprasit, C. Moore, R. Gillespie, and P. Akella. Cobot architecture. *IEEE Transactions on Robotics and Automation*, 17(4):377–390, Aug 2001.

[18] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[19] A. Radziwon, A. Bilberg, M. Bogers, and E. S. Madsen. The smart factory: exploring adaptive and flexible manufacturing solutions. *Procedia engineering*, 69:1184–1190, 2014.

[20] S. J. Saidi, D. Foucard, G. Smaragdakis, and A. Feldmann. Flowtree: Enabling Distributed Flow Summarization at Scale. In *Proceedings of ACM SIGCOMM 2018*, Budapest, Hungary, August 2018.

[21] C. Thompson and L. Shure. *Image Processing Toolbox: For Use with MATLAB;[user's Guide]*. MathWorks, 1995.

[22] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 91–102. IEEE, 1997.

[23] D. Zuehlke. SmartFactory Towards a factory-of-things. *Annual Reviews in Control*, 34(1):129–138, 2010.