A Model and Survey of Distributed Data-Intensive Systems

ALESSANDRO MARGARA, GIANPAOLO CUGOLA, NICOLÒ FELICIONI, and STEFANO CILLONI, Politecnico di Milano, Italy

Data is a precious resource in today's society, and it is generated at an unprecedented and constantly growing pace. The need to store, analyze, and make data promptly available to a multitude of users introduces formidable challenges in modern software platforms. These challenges radically impacted the research fields that gravitate around data management and processing, with the introduction of distributed data-intensive systems that offer innovative programming models and implementation strategies to handle data characteristics such as its volume, the rate at which it is produced, its heterogeneity, and its distribution. Each data-intensive system brings its specific choices in terms of data model, usage assumptions, synchronization, processing strategy, deployment, guarantees in terms of consistency, fault tolerance, and ordering. Yet, the problems data-intensive systems face and the solutions they propose are frequently overlapping. This article proposes a unifying model that dissects the core functionalities of data-intensive systems, and discusses alternative design and implementation strategies, pointing out their assumptions and implications. The model offers a common ground to understand and compare highly heterogeneous solutions, with the potential of fostering cross-fertilization across research communities. We apply our model by classifying tens of systems: an exercise that brings to interesting observations on the current trends in the domain of data-intensive systems and suggests open research directions.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Information systems** → **Parallel and distributed DBMSs**; **Middleware for databases**; **Stream management;**

Additional Key Words and Phrases: Data-intensive systems, distributed systems, data management, data processing, model, taxonomy

ACM Reference format:
Alessandro Margara, Gianpaolo Cugola, Nicolò Felicioni, and Stefano Cilloni. 2023. A Model and Survey of Distributed Data-Intensive Systems. *ACM Comput. Surv.* 56, 1, Article 16 (August 2023), 69 pages.
https://doi.org/10.1145/3604801

1 INTRODUCTION

As data guides the decision-making process of increasingly many human activities, software applications become data intensive [54]. They handle large amounts of data produced by disparate sources. They perform complex data analysis to extract valuable knowledge from the application environment. They take automated decisions in near real time. They serve content to a multitude of users, spread over wide geographical areas. The challenges for these applications come from data characteristics such as its volume, the rate at which it is generated, and its heterogeneity.

Authors' address: A. Margara, G. Cugola, N. Felicioni, and S. Cilloni, Politecnico di Milano, Piazza Leonardo da Vinci, 32, 20133, Milano, Italy; emails: alessandro.margara@polimi.com, gianpaolo.cugola@polimi.com, nicolo.felicioni@polimi.com, stefano.cilloni@mail.polimi.com.

0360-0300/2023/08-ART16 $15.00
https://doi.org/10.1145/3604801

This demands *distributed software systems* that could exploit the resources of interconnected computers to efficiently store, query, analyze, and serve data to customers at scale. Distribution brings along issues related to communication, concurrency and synchronization, deployment of data and computational tasks on physical nodes, replication and consistency, and handling of partial failures [54]. To address these issues, the past two decades have seen a flourishing of systems and execution models that abstract away some of the concerns related to distributed data management and processing. We collectively denote them as *distributed data-intensive systems*: they originate from research and development efforts in various communities, particularly those working on database and distributed systems.

Background: Different Research Lines Addressing Overlapping Problems. In the database community, the mutating requirements brought by data-intensive applications put the traditional (relational) data model and implementation strategies under question. The increasing complexity and volume of data demanded flexibility and scalability. Internet-scale applications demanded geographically replicated stores, supporting a multitude of users concurrently reading and updating data [6]. In these contexts, communication and synchronization costs may become the main bottleneck [15]. Workload characteristics changed as well: analytical tasks emerged and complemented query tasks [85]. In response to these challenges, researchers first investigated so-called NoSQL solutions [36] that trade strong consistency and transactional guarantees in favor of flexibility and scalability. More recently, the complexity of writing applications with weak guarantees inspired a renaissance of transactional semantics [86], coupled with programming, design, and implementation approaches to make transactions management more scalable [34, 88, 89, 92].

In parallel, within the distributed systems research community, the increasing centrality of data fostered the development of new systems that exploit the compute capabilities of cluster infrastructures to extract valuable information from this large amount of data. Pioneered by MapReduce [38], they organize the computation into a dataflow graph of operators that apply functional transformations on their input data. This dataflow model promotes distributed computations: operators may run simultaneously on different machines (task parallelism), and multiple instances of each operator may process independent portions of the input data in parallel (data parallelism). Developers only specify the behavior of operators, whereas the system automates their deployment, the exchange of data, and the re-execution of lost computations due to failures. Over the years, a multitude of systems adopted and revised this processing model in terms of programming abstractions (e.g., support for streaming data and iterative computations), as well as design and implementation choices (e.g., strategies to associate operators to physical machines and to exchange data across operators) [24, 99], whereas other systems brought alternative programming models suited to specific domains, such as graph processing [68].

In general, the challenging demands of data-intensive applications are continuously pushing researchers and practitioners to build novel solutions that go beyond traditional categories [87]. For instance, many dataflow platforms offer abstractions to process data through complex relational queries, thus crossing the boundaries of database technologies. Data stores such as VoltDB [89] and S-Store [27] aim to support near real-time processing of incoming data within a relational database core. Messaging services such as Kafka [55] offer persistency, querying, and processing capabilities [18].

Motivations. In summary, a multitude of distributed data-intensive systems proliferated over the years. The problems they face and the solutions they propose are frequently overlapping, but the commonalities in design principles and implementation choices often hide behind concrete realizations and heterogeneous terminologies adopted by different research communities. In this scenario, it is difficult for users to grasp the subtle differences between systems, evaluate their benefits and

limitations, capture their assumptions and guarantees, and select the most suitable ones for a given application. In addition, it is hard for researchers to get a coherent and comprehensive view of the area, identify salient design choices, and understand their consequences in terms of functionalities and performance.

Contributions and Methodological Approach. To address these issues, this article proposes a model for data-intensive systems that integrates key design and implementation choices into a unifying view. This model captures our view of a data-intensive system as a collection of abstract components that together provide the system functionalities. For each component, we define the following: ind (i) the assumptions it relies upon; iind (ii) the functionalities it provides; iiind (iii) the guarantees it offers; ivnd (iv) the possible strategies for its design and implementation, highlighting their implications. From these characteristics, we derive a list of classification criteria that we use to survey tens of state-of-the-art data-intensive systems, organized into a taxonomy that highlights their similarities and differences.

To develop our model and compile our taxonomy and survey, we started from works appearing in top tier conferences and journals in the area of database and distributed systems in the past 20 years. We then checked the systems they were citing, including commercial products not presented in scientific publications. To remain focused on our goal of surveying existing systems, we skipped works only presenting individual algorithms or mechanisms, and we concentrated on systems addressing data management and processing problems, not compute-intensive problems such as computer simulations. From this large background of material and our own past experience in the area, we derived our model and the relevant dimensions to classify systems. Without pretending to embrace all possible systems and issues, we are confident that this survey captures the key design strategies adopted in currently available distributed data-intensive systems.

Our work contributes to the research areas that gravitate around data-intensive systems in various ways: and (i) it enables an unbiased comparison of their features; and (ii) it offers a broad view of the research in the field; and (iii) it promotes cross-fertilization between communities, defining a vocabulary and conceptual framework that they can use to exchange ideas and solutions to common problems; and (iv) it highlights consolidated results and open challenges, and helps in identifying promising research directions.

ArticleOutline. The article is organized as follows. Section 2 presents our model and list of classification criteria. Based on these criteria, Section 3 proposes a taxonomy to organize data-intensive systems in a small set of classes. Next, we apply the model to describe these classes: pure **Data Management Systems (DMSs)** in Section 4, pure **Data Processing Systems (DPSs)** in Section 5, and other systems that propose new or hybrid programming and execution models in Section 6. Section 7 discusses key aspects that emerge from our analysis and points out future research directions. Finally, Section 8 concludes the article. In the appendices, we describe individual systems in detail; provide a summary of the terms and a map of the concepts that appear in the article, together with their mutual relations; and report related surveys and studies.

2 A UNIFYING MODEL FOR DATA-INTENSIVE SYSTEMS

This section presents a model that captures the core functionalities of data-intensive systems within a collection of abstract components. From the model, we derive fine-grained classification criteria that we summarize in Tables 1 through 8. We organize data-intensive systems into a taxonomy in Section 3, and we use the classification criteria to describe the systems within this taxonomy in Sections 4 through 6. In this section, coherently with the taxonomy in Section 3, when providing concrete examples of systems to explain the concepts in the model, we denote DMSs as those that are primarily designed to store some state and expose functions to query and mutate

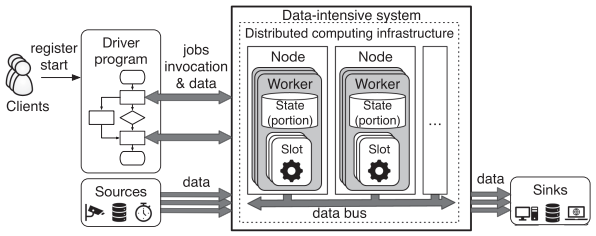

Fig. 1. Functional model of a data-intensive system.

Table 1. Classification Criteria: Functional Model

Driver exec	client-side/system-side
Driver exec time	on registration/on start
Invocation of jobs	synchronous/asynchronous
Sources	no/passive/active/both
Sinks	no/yes
State	no/yes
Deployment	cluster/wide-area/hybrid

such state, whereas we denote DPSs as those that enable expensive transformation and analysis of static (batch) or dynamic (streaming) data.

2.1 Functional Model

Figure 1 depicts the functional model of a data-intensive system. The resulting classification criteria are shown in Table 1. A data-intensive system offers data management and processing functionalities to external *clients*. Clients can register and start *driver programs*, which are the parts of the application logic that interact with the data-intensive system and exploit its functionalities. Specifically, during its execution, a driver program can invoke one or more *jobs*, which are the largest units of execution that can be offloaded onto the *distributed computing infrastructure* made available by the data-intensive system. Depending on the specific system, a driver program may execute *client-side* or *system-side*. Some systems decouple activation of driver programs from their registration; in this case, we say that driver execution time is *on start*, and otherwise we say that it is *on registration*. To exemplify, in a DMS, a driver program may be a stored procedure that combines code expressed in some general-purpose programming language with one or more queries (the jobs) expressed in the language offered by the engine (e.g., SQL). Stored procedures typically run system-side every time a client activates them (on start). Similarly, in a DPS, the driver program may be a piece of Java code that spawns one or more distributed computations (the jobs) written using the API offered by the data processing engine. In this context, the driver program will typically run system-side on registration.

The data-intensive system runs on the distributed computing infrastructure as a set of *worker* processes, hosted on the same or different *nodes* (physical or virtual machines). We model the processing resources offered by workers as a set of *slots*. Jobs are compiled into elementary units of execution that we denote as *tasks* and run sequentially on slots. Jobs consume input *data* and produce output data. Some systems also store some *state* within the distributed computing infrastructure: in this case, jobs may access (read and modify) the state during their execution. When present, state can be split (partitioned and replicated) across workers such that each of them is responsible for a state *portion*.

In our model, data elements are immutable and are distributed through communication channels (dark gray arrows in Figures 1 and 2) that we collectively refer to as the *data bus*. Notice that the data bus also distributes *jobs invocations*. Indeed, our model emphasizes the dual nature of invocations and data, which can both carry information and trigger jobs execution: invocations may transport data in the form of input parameters and return values, whereas the availability of new data may trigger the activation of jobs. Our model exploits this duality to capture the heterogeneity in activating jobs and exchanging data of the systems we surveyed.

Jobs invocations may be either *synchronous*, if the driver program waits for jobs completion before making progress, or *asynchronous*, if the driver program continues to execute after

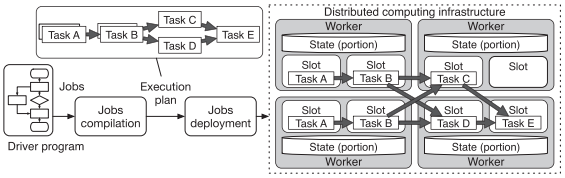

Fig. 2. Jobs definition, compilation, deployment, and execution.

Table 2. Classification Criteria: Jobs Definition, Compilation, Deployment, and Execution

Jobs Definition	
Jobs definition API	library/DSL
Exec plan definition	explicit/implicit
Task communication	explicit/implicit
Exec plan structure	dataflow/workflow
Iterations	no/yes
Dynamic creation	no/yes
Nature of jobs	one-shot/continuous
State management	absent/explicit/implicit
Data-parallel API	no/yes
Placement-aware API	no/yes
Jobs Compilation	
Jobs compil time	on driver registration/on driver execution
Use resources info	no/static/dynamic
Jobs Deployment and Execution	
Granularity of deployment	job level/task level
Deployment time	on job compilation/on task activation
Use resources info	static/dynamic
Management of res	system-only/shared

submitting the invocation. In both cases, invocations may return some result to the driver program, as indicated by the bidirectional arrows in Figure 1. In some systems, jobs also consume data from external *sources* and produce data for external *sinks*. We distinguish between *passive* sources, which consist of static datasets that jobs can access during their execution (e.g., a distributed filesystem), and *active* sources, which produce new data dynamically and may trigger job execution (e.g., a messaging system).

To exemplify, stored procedures (the driver programs) in a DMS invoke (synchronously or asynchronously, depending on the specific system) one or more queries (the jobs) during their execution. Invocations carry input data in the form of actual parameters. Queries can access (read-only queries) and modify (read-write queries) the state of the system, and return query results. In batch DPSs such as MapReduce, jobs read input data from passive sources (e.g., a distributed filesystem), apply functional transformations that do not involve any mutable state, and store the resulting data into sinks (e.g., the same distributed filesystem). In stream DPSs, jobs run indefinitely and make progress when active sources provide new input data. We say that input data *activates* a job, and in this case jobs may preserve some state across activations.

We characterize the distributed computing infrastructure based on its *deployment*. In a *cluster* deployment, all nodes belong to the same cluster or data center, which provides high bandwidth and low latency for communication. Conversely, in a *wide-area* deployment, nodes can be spread in different geographical areas, a choice that increases the latency of communication and may impact the possibility of synchronizing and coordinating tasks. For this reason, we also consider *hybrid* deployments, when the system adopts a hierarchical approach, exploiting multiple fully functional cluster deployments that are loosely synchronized with each other.

2.2 Jobs and Their Lifecycle: From Definition to Execution

This section concentrates on jobs, following their lifecycle from definition to execution (see Figure 2). Jobs are defined inside a driver program (Section 2.2.1) and compiled into an execution plan of elementary tasks (Section 2.2.2), which are deployed and executed on the distributed computing infrastructure (Section 2.2.3). The resulting classification criteria are presented in Table 2.

2.2.1 Jobs Definition. Jobs are defined inside driver programs. Frequently, driver programs include multiple jobs and embed the logic that coordinates their execution. For instance, stored procedures (driver programs) in DMSs may embed multiple queries (jobs) within procedural code. Similarly, DPSs invoke analytic jobs from a driver program written in a standard programming language with a fork-join execution model. Notably, some systems implement iterative algorithms by spawning a new job for each iteration and by evaluating termination criteria within the driver program.

Jobs are expressed using programming primitives (*jobs definition API*) with heterogeneous forms. For instance, relational DMSs rely on SQL (a **Domain-Specific Language (DSL)**), whereas DPSs usually offer *libraries* for various programming languages. Some systems support both forms.

Jobs are compiled into an *execution plan*, which defines the computation as a set of elementary units of deployment and execution called *tasks*. Tasks (i) run on slots, (ii) exchange data over the data bus (dark gray arrows in Figure 2) according to the communication schema defined in the execution plan, and (iii) may access the state portion of the worker they are deployed on. We say that the *execution plan definition* is *explicit* if the programming primitives directly specify the individual tasks and their logical dependencies. The definition is instead *implicit* if the logical plan is compiled from a higher-level, declarative specification of the job. To exemplify, the dataflow formalism adopted in many DPSs provides an explicit definition of the logical plan, whereas SQL, and most query languages, provide an implicit definition. With an explicit definition of the logical plan, the *communication* between tasks can itself be explicit or implicit. In the first case, the system APIs include primitives to send and receive data across tasks, whereas in the latter case, the exchange of data is implicit. The execution plan *structure* can be a generic *workflow*, where there are no restrictions to the pattern of communication between tasks, or a *dataflow*, where tasks need to be organized into an acyclic graph and data can only move from upstream tasks to downstream tasks. When present, we also highlight further structural constraints. For instance, the execution plan of the original MapReduce system forces data processing in exactly two phases: a map and a reduce.

Iterations within the execution plan may or may not be allowed. We say that a system supports *dynamic creation* of the plan if it enables spawning new tasks during execution. Dynamic creation gives the flexibility of defining or activating part of the execution plan at runtime, which may be used to support control flow constructs.

Jobs can be either *one-shot* or *continuous*. One-shot jobs are executed once and then terminate. We use the term *invoke* here: as invoking a program twice leads to two distinct processes, invoking a one-shot job multiple times leads to separate executions of the same code. For instance, queries in DMSs are typically one-shot jobs, and indeed multiple invocations of the same query lead to independent executions. Instead, continuous jobs persist across invocations. In this case, we use the term *activate* to highlight that the same job is repeatedly activated by the arrival of new data. This happens in stream DPSs, where continuous jobs are activated when new input data comes from active sources. As detailed in Section 2.4, the key distinguishing factor of continuous jobs is their ability to persist in some private task state across activations. By definition, this option is not available for one-shot jobs, since each invocation is independent from the other.

State management in jobs may be *absent*, *explicit*, or *implicit*. For instance, state management is absent in batch DPSs, which define jobs in terms of functional transformations that solely depend on the input data. State management is explicit when the system provides constructs to directly access state elements to read and write them. For instance, queries in relational DMSs provide `select` clauses to retrieve state elements and `insert` and `update` clauses to store new state elements and update them. State management is implicit when state accesses are implicit in job definition. For instance, stream DPSs manage state implicitly through ad hoc operators such as windows that record previously received data and use it to compute new output elements.

Another relevant characteristic of the programming model is the support for *data parallelism*, which allows defining computations for a single element and automatically executing them on many elements in parallel. Data parallelism is central in many systems, and particularly in DPSs, as it simplifies the definition of jobs by letting developers focus on individual elements. It promotes parallel execution, as the tasks operating on different elements are independent. Systems supporting data parallelism apply partitioning strategies to both data and state (when available), as we discuss in Section 2.3 and Section 2.4. As inter-task communication and remote data access may easily become performance bottlenecks, some systems aim to reduce inter-task communication and to promote local access to data and state by offering *placement-aware* APIs that enable developers to suggest suitable placement strategies based on the expected workload.

2.2.2 Jobs Compilation. The process of compiling jobs into an execution plan may either start *on driver registration* or *on driver execution*. The first case models situations where the driver program is registered in the system and can be executed multiple times, as in the case of stored procedures. The second case happens in DPSs, which usually offer a single command to submit and execute a program. Jobs compilation may *use resources information*—that is, information about the resources of the distributed computing infrastructure. The information is *static* if it only considers the available resources. For instance, data-parallel operators are converted into multiple tasks that run the same logical step in parallel: the concrete number of tasks is typically selected depending on the available processing resources (overall number of CPU cores). The information is *dynamic* if it also considers the actual use of resources. For instance, a join operation may be compiled into a distributed hash join or into a sort-merge join depending on the current cardinality and distribution of the elements to join.

2.2.3 Jobs Deployment and Execution. Jobs deployment is the process of allocating the tasks of a job execution plan onto slots. For instance, the execution plan in Figure 2 consists of seven tasks, and each of them is deployed on a different slot. Tasks tagged A and B exemplify data-parallel operations, each executed by two tasks in parallel. Deployment can be performed with *job-level granularity* or *task-level granularity*. Job-level granularity is common when the deployment takes place *on job compilation*, whereas task-level granularity is used when the deployment (of individual tasks) takes place *on task activation*. It is important to note that the preceding classification is orthogonal to the nature of jobs (one-shot or continuous) as defined earlier. One-shot jobs may be (i) entirely deployed either on job compilation or (ii) progressively, as their input data is made available by previous tasks in the execution plan. The first choice is frequent in DMSs, whereas the latter characterizes several DPSs. Similarly, continuous jobs may be (i) fully deployed on compilation, with their composing tasks remaining available onto slots, ready to be activated by incoming data elements, or (ii) their tasks may be deployed individually when new input data becomes available and activates them. In this case, the same task is deployed multiple times, once for each activation: systems that follow this strategy minimize the overhead of deployment by accumulating input data into batches and deploying a task only once for the entire batch, as well exemplified by the micro-batch approach of Spark Streaming [98]. As we discuss in Section 2.3, task-level deployment requires a persistent data bus, to decouple task execution in time. If the data bus is not persistent, all tasks in the execution plan need to be simultaneously deployed to enable the exchange of data.

The deployment process always exploits *static* information about the resources available in the computing infrastructure, like the address of workers and their number of slots. Some systems also exploit *dynamic* information, such as the current load of workers and the location of data. This is typically associated with task-level scheduling on activation, where tasks are deployed when their input data is available and they are ready for execution. Finally, the deployment process may

Table 3. Classification Criteria: Data Management

Elements structure	no/general /domain specific
Temporal elements	no/yes
Bus connection type	direct/mediated
Bus implementation	
Bus persistency	persistent/ephemeral
Bus partitioning	no/yes
Bus replication	no/yes
Bus interaction model	push/pull/hybrid

Table 4. Classification Criteria: State Management

Elements structure	no/general /domain specific
Storage medium	memory /disk/hybrid/service
Storage structure	
Task state	no/yes
Shared state	no/yes
Partitioned	no/yes
Replication	no/backup-only/yes
Replication consist.	weak/strong
Replication protocol	leader/consensus /conflict resolution
Update propagation	state/operations

Table 5. Classification Criteria: Tasks Grouping

Group Atomicity	
Causes for aborts	system/job
Protocol	blocking/coord free
Assumptions	
Group Isolation	
Level	blocking/coord free
Implementation	lock/timestamp
Assumptions	

have a *system-only* or *shared management of resources*. System-only management only considers the resources occupied by the data-intensive system. Shared management takes global decisions in the case in which multiple software systems share the same distributed computing infrastructure. For instance, it is common to use external resource managers such as Yarn for task deployment in cluster environments.

2.3 Data Management

This section studies the characteristics of data elements and the data bus used to distribute them. The resulting classification criteria are shown in Table 3. Recall that in our model, data elements are immutable, meaning that once they are delivered through the data bus, they cannot be later updated. In addition, they are used to represent both data and invocations, as they carry some payload and may trigger the activation of tasks. Data elements may be *structured*, if they have an associated schema determining the number and type of fields they contain, or *unstructured*, otherwise. Structured data is commonly found in DPSs, when input datasets or data streams are composed of tuples with a fixed structure. The structure of elements may reflect on the data bus, with assumptions of homogeneous data elements (same schema) in some communication channels. For instance, DPSs typically assume homogeneous input and homogeneous output data for each task. We further distinguish between systems that accept *general* structured data, when the developers are free to define their custom data model, and systems that assume a *domain-specific* structure, when developers are constrained to a specific data model, as in the case of relational data, time series, or graph-shaped data. Finally, data may or may not have a *temporal* dimension: this is particularly relevant for stream DPSs, where it is used for time-based analysis. Section 2.6 will detail how the temporal dimension influences the order in which tasks analyze data elements.

The data bus can either consist of *direct* connections between the communicating parties or can be *mediated* by some middleware service. Accordingly, the actual *bus implementation* may range from TCP links (direct connection) to various types of middleware systems (mediated connection), like message queuing or distributed storage services.[1] Whereas direct connections are always *ephemeral*, various mediated connections are *persistent*. In the first case, receivers need to be active when the data is transmitted over the bus and they cannot retrieve the same elements later in time. In the second case, elements are preserved inside the bus and receivers can access them multiple times and at different points in time. For instance, DPSs that implement job-level deployment usually adopt a direct (and ephemeral) data bus made of TCP connections among tasks. Conversely, DPSs that deploy tasks independently (task-level deployment) require a

[1]When the values associated with a classification criterion are specific to individual systems, as in the case of bus implementation (see Table 3), we do not report a list of values in the corresponding table.

persistent and mediated data bus (e.g., a distributed filesystem or a persistent messaging middleware) where intermediate tasks can store their results for downstream tasks.

In many systems, the data bus provides communication channels where data elements are logically *partitioned* based on some criterion—for instance, based on the value of a given field. The use of a partitioned data bus is common in DPSs, where it is associated with data-parallel operators: the programmer specifies the operator for the data elements in a single partition, but the operator is concretely implemented by multiple (identical) tasks that work independently on different partitions. A persistent data bus may also be *replicated*, meaning that the same data elements are stored in multiple copies. Replication may serve two purposes: improving performance, such as by enabling different tasks to consume the same data simultaneously from different replicas, or tolerating faults, to avoid losing data in the case in which one replica becomes unavailable. We will discuss fault tolerance in greater detail in Section 2.7. Among the systems we analyze in this article, all those that replicate the data bus implement a single-leader replication schema, where one *leader* replica is responsible for receiving all input data and for updating the other f (*follower*) replicas. The update is synchronous (or semi-synchronous), meaning that the addition of an input data element to the data bus completes when the data element has been successfully applied to all f follower replicas (or to $r < f$ replicas, if the update is semi-synchronous). The data bus offers an *interaction model* that is *push* if the sender delivers data to recipients or *pull* if receivers demand data to senders. *Hybrid* approaches are possible in the presence of a mediated bus, a common case being a push approach between the sender and the data bus and a pull approach between the data bus and the recipients.

2.4 State Management

After discussing data, we focus on state, deriving the classification criteria listed in Table 4. In absence of state, tasks results (data written on the bus) only depend on their input (data read from the bus), but many systems support stateful tasks whose results also depend on some mutable state that they can read and modify during execution. This marks a difference between data and state elements: the former are immutable, whereas the latter may change over time. As for their structure, state elements resemble data elements: they may be *unstructured* or *structured*, and in the second case, they may rely on *domain-specific* data models.

When present, state may be stored on different *storage media*: (i) many systems store the entire state *in-memory* and replicate it to disk only for fault tolerance; (ii) other systems use *disk* storage, or (iii) *hybrid* solutions, where state is partially stored in memory for improved performance and flushed to disk to scale in size; and (iv) some systems rely on a storage *service*, as is common in cloud-based systems that split their core functionalities into independently deployed services. Some recent work investigates the use of persistent memory [61], but these solutions are not employed in currently available systems.

The *storage structure* indicates the data structure used to represent state on the storage media. This structure is heavily influenced by the expected access pattern. For instance, relational state may be stored row-wise, to optimize access element by element (common in data management workloads), or column-wise, to optimize access attribute by attribute (common in data analysis workloads, e.g., to compute an aggregation function over an attribute). Many DMSs use indexed structures such as B-trees or **Log-Structured Merge (LSM)** trees to rapidly identify individual elements.

Data-intensive systems may support two types of state: *task state*, which is private to a single task, and *shared state*, which can be accessed by multiple tasks. The availability of these types of state deeply affects the design and implementation of the system. Shared state is central in DMSs, where two tasks (e.g., an insert and a select query) can write and read simultaneously from the

same state (e.g., a relational table). Conversely, most DPSs avoid shared state to simplify parallel execution. Frequently, batch DPSs do not offer any type of state, whereas stream DPSs only offer task state, which does not require any concurrency control mechanism, as it is accessed only by one task (sequential unit of execution). Notice that task state is only relevant in continuous jobs, where it can survive across multiple activations of the same task. Indeed, it is used in DPSs to implement stateful operators like windows.

In our model, workers are responsible for storing separate portions of the shared state. Tasks have local access to elements stored (in memory or on disk) on the shared state portion of the worker they are deployed on. They can communicate with remote tasks over the data bus to access shared state portions deployed on other workers. For systems that rely on a storage service, we model the service as a set of workers that are only responsible for storing shared state portions and offer remote access through the data bus. Splitting of the shared state among workers may respond to a criterion of *partitioning*. For instance, partitioning enables DMSs to scale beyond the memory capacity of a single node, but also to run tasks belonging to the same or different jobs (queries) in parallel on different partitions. Besides partitioning, many data-intensive systems adopt replication. As in the case of data bus replication, state replication may also serve two purposes: (i) reduce read access latency, by allowing multiple workers to store a copy (replica) of the same state elements locally, and (ii) provide durability and fault tolerance, avoiding potential loss in the case of failures. We return on the specific use of replication for fault tolerance in Section 2.7. Here, we consider if the replication is *backup-only*, meaning that replicas are only used for fault tolerance and cannot be accessed by tasks during execution, or not.

If tasks can access state elements from multiple replicas, different replication *consistency* models are possible, which define which state values may be observed by tasks when accessing multiple replicas. Replication models have been widely explored in database and distributed systems theory. For the goal of our analysis, we only distinguish between *strong* and *weak* consistency models, where the former require synchronous coordination among the replicas while the latter do not. This classification approach is also in line with the recent literature that denotes models that do not require synchronous coordination as being highly available [15]. Intuitively, strong consistency models are more restrictive and use coordination to avoid anomalies that may arise when tasks access elements simultaneously from different replicas. In practice, most data-intensive systems that adopt a strong consistency model provide *sequential* consistency, a model that ensures that accesses to replicated state are the same as if they were executed in some serial order. This simplifies reasoning on the state of the system, as it hides concurrency by mimicking the behavior of a sequential execution. In terms of implementation, we distinguish two main classes of mechanisms to achieve strong consistency: in *leader*-based algorithms, all state updates are delivered to a single replica (leader) that decides their order, and in *consensus*-based algorithms, replicas use quorum-based or distributed consensus protocols to agree on the order of state accesses. Systems that adopt a weak consistency model typically provide *eventual* consistency, where updates to state elements are propagated asynchronously, which may lead to (temporary) inconsistencies between replicas. For this reason, weak consistency is typically coupled with automated *conflict resolution* algorithms, which guarantee that all replicas solve conflicts in the same way and eventually converge to the same state. A popular approach to conflict resolution is conflict-free replicated data types, which expose only operations that guarantee deterministic conflict resolution in the presence of simultaneous updates [81].

Finally, replication protocols may employ two approaches to propagate updates: *state-based* or *operation-based* (also known as active replication). In state-based replication, when a task updates a value in a replica, the new state is propagated to the other replicas. In operation-based replication, the operation causing the update is propagated and re-executed at each replica: this approach

may save bandwidth, but it may spend more computational resources to re-execute operations at each replica.

2.5 Tasks Grouping

Several systems offer primitives to identify groups of tasks and provide additional guarantees for such groups, which we classify as *group atomicity* (Section 2.5.1) and *group isolation* (Section 2.5.2). The resulting classification criteria are presented in Table 5. Atomicity ensures no partial failures for a group of tasks: they either all fail or all complete successfully. Isolation limits the ways in which running tasks can interact and interleave with each other. In DMSs, these concerns are considered part of transactional management, together with consistency and durability properties. In our model, we discuss consistency constraints as part of group atomicity in the next section, whereas we integrate durability with fault tolerance and discuss it in Section 2.7.

2.5.1 Group Atomicity. Atomicity ensures that a group of tasks either entirely succeeds or entirely fails. We use the established jargon of database transactions and say that a task (or group of tasks) either *commits* or *aborts*. If the tasks commit, all the effects of their execution, and particularly all their changes to the shared state, become visible to other tasks. If the tasks abort, none of the effects of their execution becomes visible to other tasks. We classify group atomicity along two dimensions. First, we consider the possible *causes for aborts* and distinguish between *system-driven* or *job-driven*. System-driven aborts (e.g., a worker running out of memory) derive from non-deterministic hardware or software failures, whereas job-driven aborts (e.g., database integrity constraints) are part of a job definition and are triggered if job completion may lead to a logic error. Second, we consider how systems implement group atomicity. Atomicity is essentially a consensus problem [65], where tasks need to agree on a common outcome: commit or abort. The established protocol to implement atomicity is *two-phase commit*. In this protocol, one of the participants (tasks) takes the role of a coordinator, collects all votes (commit or abort) from participants (phase 1), and distributes the common decision to all of them (phase 2). Notice that this protocol is not robust against failures of the coordinator; however, data-intensive systems typically adopt orthogonal mechanisms to deal with failures, as discussed in Section 2.7. Most importantly, two-phase commit is a *blocking* protocol as participants cannot make progress before receiving the global outcome from the coordinator. For these reasons, some systems adopt simplified, *coordination-free* protocols, which reduce or avoid coordination under certain assumptions. Being specific to individual systems, we discuss such protocols in Section 4.

2.5.2 Group Isolation. Group isolation constrains how tasks belonging to different groups can interleave with each other and is classically organized into *levels* [4]. The stronger, serializable isolation, requires the effects of execution to be the same as if all groups were executed in some serial order, with no interleaving of tasks, whereas weaker levels enable some disciplined form of concurrency that may lead to anomalies in the results that clients observe. In line with the approach adopted for replication consistency and for atomicity, in this work we consider only two broad classes of isolation levels: those that require *blocking coordination* between tasks and those that are *coordination free* (referred to as being highly available in the literature [15]). This is also motivated by the systems under analysis, which either provide strong isolation levels (typically, serializable) or do not provide isolation at all.

Implementation-wise, strong isolation is traditionally achieved with two classes of coordination protocols: *lock-based* and *timestamp-based*. With lock-based protocols, tasks acquire non-exclusive or exclusive locks to access shared resources (shared state in our model) in read-only or read-write mode. Lock-based protocols may incur distributed deadlocks: to avoid them, protocols implement detection or prevention schemes that abort and restart groups in the case of deadlock.

Table 6. Classification Criteria: Delivery and Order

Delivery guarantees	at most /at least/exactly once
Nature of timestamps	n.a./ingestion/event
Order guarantees	n.a./eventually/always

Table 7. Classification Criteria: Fault Tolerance

Detection	leader-worker /distributed
Scope	comput - task state - shared state
Computation recovery	absent/job/task
State recovery	checkp (indep./coord/per-activ) /log (WAL/CL)/repl
Guarantees for state	none/valid/same
Assumptions	

Table 8. Classification Criteria: Dynamic Reconfiguration

Goal	
Automated	no/yes
Mechanisms	state migr/task migr
Restart	no/yes

Timestamp-based protocols generate a serialization order for groups before execution, then the tasks need to enforce that order. Pessimistic timestamp protocols abort and re-execute groups when they try to access shared resources out of order. Multi-version concurrency control protocols reduce the probability of aborts by storing multiple versions of shared state elements and allowing tasks to read old versions when executed out of order. Optimistic concurrency control protocols allow out-of-order execution of tasks but check for conflicts before making the effects of a group of tasks visible to other tasks. Finally, a few systems adopt special protocols that reduce or avoid coordination under certain assumptions: as in the case of group atomicity, we discuss these protocols in Section 4.

2.6 Delivery and Order Guarantees

Delivery and *order* guarantees define how external actors (driver programs, sources, and sinks) observe the effects of their actions (submitting input data and invocations). Both topics are crucial for distributed systems and have been widely explored in the literature. Here, we focus on the key concepts that characterize the behavior of the systems we analyzed, and we offer a description that embraces different styles of interaction, from invocation-based (as in DMS queries) to data-driven (as in stream DPSs). The resulting classification criteria are presented in Table 6.

Delivery focuses on the effects of a single input I (data element or invocation). Under *at most once* delivery, the system behaves as if I was either received and processed once or never. Under *at least once* delivery, the system behaves as if I was received and processed once or more than once. Under *exactly once* delivery, the system behaves as if I was received and processed once and only once. A well-known theoretical result in the area of distributed systems states the impossibility to deliver an input exactly once in a distributed environment where components can fail. Nevertheless, a system can behave as if the input was processed exactly once under some assumptions: the most common are that driver programs and sources can resubmit the input upon request (to avoid loss of data), whereas sinks can detect duplicate output results and discard them (to avoid duplicate processing and output). To exemplify, DMSs offer exactly once delivery when they guarantee group atomicity through transactions: in this case, a job entirely succeeds or entirely fails, and in the case of a failure, the system either notifies the driver program (that may retry until success) or internally retries, allowing the jobs to be executed exactly once. DPSs offer exactly once delivery by replaying data from sources (or from intermediate results stored in a persistent data bus) in the case of a failure. In the presence of continuous jobs (stream processing), systems also need to avoid duplicating the effects of processing on task state when replaying data: to do so, they often discard the effects of previous executions by reloading an old task state from a checkpoint (see also the role of checkpoints on fault tolerance in Section 2.7).

Order focuses on multiple data elements or invocations and defines in which order their effects become visible. Order ultimately depends on the *nature of timestamps* physically or logically

attached to data elements. In some systems, no timestamp is associated with data elements; in these cases, no ordering guarantees are provided. Conversely, when data elements represent occurrences of events in the application domain, they have an associated timestamp that can be set by the original source or by the system when it first receives the element. We rely on established terminology and refer to the former case as *event time* and the latter case as *ingestion time* [8]. When a timestamp is provided, systems may ensure that the associated order is guaranteed *always* or *eventually*. Systems in the first class wait until all data elements before a given timestamp become available and then process them in order. To do so, they typically rely on a contract between the sender components and the data bus, where sender components use special elements (denoted as *watermarks*) to indicate that all elements up to a given time t have been produced, and the data bus delivers elements up to time t in the correct order. Systems in the second class execute elements out of order, but they *retract* previously issued results and correct them when they receive new data with an older timestamp. Thus, upon receiving all input data up to time t, the system eventually returns the correct results. Notice that this mechanism requires the elements receiving output data to tolerate temporarily incorrect results. According to our preceding definitions, retraction is not compatible with exactly once delivery, as it changes the results provided to sinks after they have already been delivered, thus breaking the illusion that they have been produced once and only once.

2.7 Fault Tolerance

Fault tolerance is the ability of a system to tolerate failures that may occur during its execution. We consider hardware failures, such as a disk failing or a node crashing or becoming unreachable, and non-deterministic software failures, such as a worker exhausting a node's memory. We assume that the logic of jobs is correct, which guarantees that the re-execution of a failed job does not deterministically lead to the same failure. Our minimal unit of fault is the worker, and we assume the typical approach to tolerate failures that involves first detecting the failure and then recovering from it. The classification criteria for fault tolerance are presented in Table 7.

Fault detection is usually addressed as a problem of group membership: given a group of workers, determine the (sub)set of those active and available to execute jobs. Systems address this problem either using a *leader-worker* approach, which assumes one entity with a special role (leader) that cannot fail and can supervise normal workers, or using a *distributed* protocol, like gossip-based protocols.

After a failure is detected, *fault recovery* brings the system into a state from which it can resume with the intended semantics. We describe the recovery process by focusing on five aspects: scope, computation recovery, state recovery, guarantees, and assumptions. Depending if tasks are stateless or stateful and if they can share state or not, the *scope* of recovery may involve recovering the *computation* of failing tasks, the *task state* of failing tasks, and/or the *shared state* portions held by failing workers.

Computation recovery may be *absent*, in which case failing jobs are simply discarded and the system offers at most once delivery (see Section 2.6). Otherwise, the system recovers the computation by restarting it: we distinguish between systems that need to restart an entire *job* and systems that can restart individual *tasks*. DMSs typically restart entire jobs to satisfy transactional (atomicity) guarantees that require a job to either entirely succeed or entirely fail. Some DPSs (those using a persistent data bus to save the intermediate results of tasks) may restart only failed tasks. Restarting a computation requires that input data and invocations are persisted and replayable, and that duplicate output data can be detected and discarded by sinks (if the system wants to ensure exactly once delivery; see Section 2.6). To replay input data and invocations, systems either rely on replayable sources, such as persistent message services, or keep a log internally (see the discussion on logging in the following).

To recover state, systems may rely on *checkpointing*, *logging*, and *replication*. Frequently, they combine these mechanisms. Checkpointing involves saving a copy of the entire state to durable storage. When a failure is detected, the last available checkpoint can be reloaded and the execution of jobs may restart from that state. Different workers may take *independent* checkpoints, or they may *coordinate*, such as by using the distributed snapshot protocol [28] to periodically save a consistent view of the entire system state. A third alternative (*per-activation* checkpoint) is sometimes used for continuous jobs to save task state: at each activation, a task stores its task state together with the output data. This approach essentially transforms a stateful task into a stateless task, where state is encoded as a special data element that the system receives in input at the next activation. In practice, per-activation checkpoint is used in presence of a persistent data bus that stores checkpoints. Frequent checkpointing may be resource consuming and affect the response time of the system. *Logging* is an alternative approach that saves either individual operations or state changes rather than the entire content of the state. These two forms of logging are known in the database literature as follows: (i) the **Command Logging (CL)** persists input data and invocations, and in the case of failure re-processes the same input to restore state [67], and (ii) **Write Ahead Log (WAL)** persists state changes coming from tasks to durable storage before they are applied, and in the case of a failure reapplies the changes in the log to restore the state . As logs may grow unbounded with new invocations entering the system, they are always complemented with (infrequent) checkpoints. In the case of failure, the state is restored from the last checkpoint and then the log is replayed from the checkpoint on. Finally, systems may *replicate* state portions in multiple workers. In this case, a state change performed by a task succeeds only after the change has been applied to a given number of replicas r. This means that the system can tolerate the failure of $r - 1$ replicas without losing the state and without the need to restore it from a checkpoint. As already discussed in Section 2.4, the same replicas used for fault tolerance may also be used by tasks during normal processing to improve state access latency.

The preceding recovery mechanisms provide different *guarantees* on the state of the system after recovery. It can be *any* state, a *valid* system state, or the *same* state the system was before failing. In *any* state recovery, the presence of a failure may bring the system to a state that violates some of the invariants for data and state management that hold for normal (non failing) executions. For instance, the system may drop some input data. A *valid* state recovery mechanism brings the system to a state that satisfies all invariants, but it may differ from those states traversed before the failure. For instance, a system that provides serializability for groups of tasks may recover by re-executing groups of tasks in a different (but still serial) order. Depending on the system, clients may be able or not to observe the differences between the two states (before and after the failure). For instance, in a DMS, two read queries before and after the failure may observe different states. A *same* state recovery mechanism brings the system in the same state it was before the failure. Replication, write-ahead logging, and per-activation checkpointing bring the system to the same state it was prior to fail, whereas independent checkpointing only guarantees to bring the system back to a valid state, as the latest checkpoints of each task may not represent a consistent cut of the system. The same happens for CL, due to different interleavings in the original execution and in the recovery phase that may lead to different (albeit valid) states.

A final aspect related to recovery is the *assumptions* under which it operates, like assuming no more than k nodes can fail simultaneously as in the case of replication.

2.8 Dynamic Reconfiguration

With dynamic reconfiguration, we refer to the ability of a system to modify the deployment and execution of jobs on the distributed computing infrastructure at runtime. The corresponding

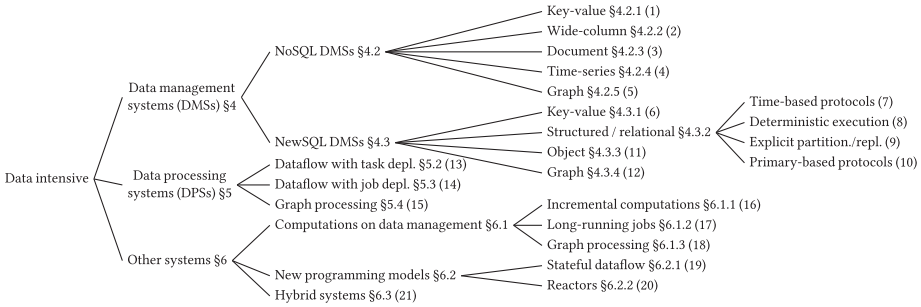

Fig. 3. A taxonomy of data intensive systems. List of systems:
(1) Dynamo, DynamoDB, Redis, Voldemort, Riak KV, Aerospike, PNUTS, Memcached, (2) BigTable, Cassandra, (3) MongoDB, CouchDB, AsterixDB, (4) InfluxDB, Gorilla, Monarch, Peregreen, (5) TAO, Unicorn, (6) Deuteronomy, FoundationDB, Solar, (7) Spanner, CockroachDB, (8) Calvin, (9) VoltDB, (10) Aurora, Socrates, (11) Tango, (12) A1, (13) MapReduce, Dryad, HaLoop, CIEL, Spark, Spark Streaming, (14) MillWheel, Flink, Storm, Kafka Streams, Samza/Liquid, timely dataflow, (15) Pregel, GraphLab, PowerGraph, Arabesque, G-Miner, (16) Percolator, (17) F1, (18) Trinity, (19) SDG, TensorFlow, Tangram, (20) ReactDB, and (21) S-Store, SnappyData, StreamDB, TSpoon, Hologres.

classification criteria are shown in Table 8. Reconfiguration may be driven by different *goals*, which may involve providing some minimum quality of service, such as in terms of throughput or response time and/or minimizing the use of resources to cut down operation costs. It may be activated manually or be *automated*, if the system can monitor the use of resources and make changes that improve the state of the system with respect to the intended goals. The reconfiguration process may involve different mechanisms: *state migration* across workers, such as to rebalance shared state portions if they become unbalanced, and *task migration*, to change the association of tasks to slots, including the addition or removal of slots, to add computational resources if the load increases or release them when they are not necessary. State migration is common in DMSs, where the distribution of shared state across workers may affect performance. Task migration is instead used in DPSs in the presence of continuous jobs, where tasks are migrated across invocations. In both cases, the migration may adapt the system to the addition or removal of slots. Some systems can continue operating during a reconfiguration process, whereas other systems need to temporarily stop and *restart* the jobs they are running: this approach appears in some systems that adopt job-level deployment; in this case, reconfiguration takes place by saving the current state, restarting the whole system, and restoring the last recorded state.

3 SURVEY OF DATA-INTENSIVE SYSTEMS

To keep our survey compact and general, we decided not to survey data-intensive systems one by one (this is provided in appendix), but to group them into the taxonomy of Figure 3 and to discuss systems class by class in Sections 4 through 6. The caption of Figure 3 lists all systems we grouped in each class.

Our taxonomy groups existing systems in a way that emphasizes their commonalities with respect to our classification criteria (see Tables 1–8) while also capturing pre-existing classifications widely adopted by experts. The top-level distinction between DMSs (Section 4) and DPSs (Section 5) is well known to researchers and practitioners, and it is also well captured by our classification criteria, with all DMSs providing shared state but not task state and DPSs having opposite characteristics: this distinction impacts on the value of most fields of Table 4.

Table 9. Survey of Systems: Functional Components; Jobs Definition, Compilation, Deployment, and Execution; Data Management

	Data Management		Data Processing			Other		
	NoSQL	NewSQL	Dataflow Task Depl	Dataflow Jobs Depl	Graph	Comput on Data Manag	New Prog Models	Hybrid
Functional Components								
Driver Exec	client+sys^{SP}	client+sys^{SP}	sys/conf	sys dep	sys	client*	sys*	sys*
Driver Exec Time	reg+start^{SP}	reg+start^{SP}	reg	reg	reg	reg*	reg*	reg*
Invocation of Jobs	sync (+async)	sync (+async)	sys dep	async*	sys dep	sys dep	sys dep	sys dep
Sources	sys dep	no	pass^{B} / act^{S}	pass^{B} / act^{S}	pass	sys dep	sys dep	sys dep
Sinks	sys dep	no*	yes	yes	yes	yes*	yes*	yes
State	yes	yes	no^{B} / yes^{S}	no^{B} / yes^{S}	yes	yes	yes	yes
Deployment	sys dep	sys dep	cluster	cluster	cluster	cluster*	cluster	cluster
Jobs Definition								
Jobs Def API	class dep	class dep	lib (+DSL)	lib (+DSL)	lib	sys dep	lib	lib (+DSL)
Exec Plan Def	impl*	impl*	expl (+impl)	expl (+impl)	expl	impl (+expl)	sys dep	expl (+impl)
Task Comm	impl	impl*	impl	impl	expl	impl (+expl)	impl*	impl
Exec Plan Struct	worfkl*	worfkl*	dataflow	dataflow	graph	sys dep	sys dep	sys dep
Iterations	no*	no*	sys dep	sys dep	yes	no*	yes	no*
Dyn Creation	no*	no*	no*	no	no*	no	no*	no
Nature of Jobs	one-shot	one-shot	one-shot^{B} cont^{S}	one-shot^{B} cont^{S}	cont	one-shot	one-shot*	one-shot^{B} cont^{S}
State Man	expl	expl	absent^{B} / impl^{S}	absent^{B} / impl^{S}	expl	expl	expl	expl*
Data Par API	no	no*	yes	yes	yes	no*	yes*	yes*
Placem-Aware API	no	no*	no	no	yes	no	no*	sys dep
Jobs Compilation								
Jobs Compil Time	on exec*	on exec*	on exec	on exec	on exec	on exec*	on exec*	on exec*
Use Resources Info	static	sys dep	dynamic	static	static	static	static*	static
Jobs Deployment and Execution								
Granular of Depl	job*	job	task	job	sys dep	job	sys dep	sys dep
Depl Time	compil*	compil	activ	compil	sys dep	compil	sys dep	sys dep
Use Resources Info	static	static	dynamic	static*	sys dep	static	static*	sys dep
Managem of Res	sys*	sys*	shared*	sys*	sys*	sys*	sys*	sys
Data Management								
Elem Struct	class dep	class dep	gen (+spec)	gen (+spec)	graph	sys dep	sys dep	spec*
Temp Elem	sys dep	no	no^{B} yes^{S}	no^{B} yes^{S}	no	no*	no	sys dep
Bus Conn Type	direct	direct	mediated*	direct*	sys dep	direct	direct*	direct*
Bus Impl	net chan	net chan*	fs (+RAM)	net chan / msg service	net chan / mem	net chan	net chan*	net chan*
Bus Persist	ephem	ephem	pers*	ephem*	sys dep	ephem	ephem*	ephem*
Bus Partition	yes	yes	yes	yes	yes	yes	yes	yes
Bus Repl	no	no	no	no*	no	no	no*	no*
Bus Interact	pull	pull*	hybrid*	push*	sys dep	push*	push*	push*

Sys dep, system dependent; class dep, differences captured by the specific sub-classes in our taxonomy (see Figure 3); *, with few system-specific exceptions; ^{SP}, in the case of stored procedures; ^{B}, in the case of batch processing; ^{S}, in the case of stream processing.

Within the class of DMSs, the ability to offer strong guarantees in terms of consistency (see Table 4), group atomicity, and group isolation (see Table 5) draws a sharp distinction between those systems usually known as "NoSQL" and those known as "NewSQL." NoSQL and NewSQL systems can be further classified looking at the data model they offer (the field "elements structure" in Table 3).

Within the class of DPSs, the criterion "execution plan structure" (see Table 2) differentiates dataflow and graph processing systems, whereas the criterion "granularity of deployment" (see Table 2) further separates dataflow systems into those offering task-level deployment and those offering job-level deployment.

Finally, other systems (Section 6) that do not clearly fall into the two main categories of DMSs and DPSs include those that implement data processing (as DPSs) but on top of shared state abstractions usually offered by DMSs only, those that aim to offer new programming models, and hybrid systems that explicitly try to integrate data processing and management capabilities within a unified solution.

Table 9 and Table 10 show how the classes of systems at the second level of our taxonomy map on the classification criteria of our model. The next sections describe each class in detail, explaining the values in these tables and making practical examples that refer to specific systems.

Table 10. Survey of Systems: State Management, Group Atomicity and Isolation, Delivery and Order, Fault Tolerance, and Dynamic Reconfiguration

	Data Management		Data Processing			Other		
	NoSQL	NewSQL	Dataflow Task Depl	Dataflow Jobs Depl	Graph	Comput on Data Manag	New Prog Models	Hybrid
State Management								
Elem Struct	class dep	class dep	–	–	graph	sys dep	gen*	spec*
Stor Medium	sys dep	sys dep	–	–	mem	sys dep	mem	mem*
Stor Struct	sys dep	sys dep	–	–	user-def	sys dep	sys dep	sys dep
Task State	no	no	no^{B} / yes^{S}	no^{B} / yes^{S}	yes	no	no	sys dep
Shared State	yes	yes	no	no	no	yes	yes	yes*
Partitioned	yes	yes	–	–	–	yes	yes	yes*
Replication	yes / backup	yes / backup	–	–	–	yes / backup	no	sys dep
Repl Consist	weak / conf	strong	–	–	–	sys dep	–	–*
Repl Protocol	lead / cons +confl res	lead / cons	–	–	–	lead.*	–	–
Upd Propag	op*	op*	–	–	–	op*	–	–*
Group Atomicity								
Aborts	–*	job+sys*	–	–	–	job+sys*	–*	job+sys*
Protocol	–*	blocking*	–	–	–	blocking*	–*	blocking*
Assumptions	–	– / DC / 1W	–	–	–	no	–	no / DC *
Group Isolation								
Level	– / coord free*	blocking*	–	–	–	blocking	sys dep	sys dep
Implement	– / SEQ	ts / lock	–	–	–	ts / lock*	sys dep	sys dep
Assumptions	– / 1P	no / DC / 1W	–	–	–	no / 1P	no	no / DC *
Delivery and Order								
Delivery Guar	most / exact*	exact*	exact	exact / least	exact	exact*	exact	exact
Nature of Ts	no / event	no	no^{B} / event^{S}	no / event	no	no*	no	sys dep
Order Guar	–	–	–^{B} / alw^{S}	alw / event*	–	–	–	– / alw
Fault Tolerance								
Detection	sys dep	lead-work*	lead-work	lead-work	lead-work	lead-work*	lead-work*	lead-work*
Scope	shared st	shared st	comp +task st^{S}	comp +task st^{S}	comp+task st	shared st (+comp)	shared st (+comp)	sys dep
Comput Recov	–	–	task	job*	job*	sys dep	sys dep	sys dep
State Recov	log (+repl)	log (+repl)	–^{B} / checkp^{S}	checkp*	checkp	sys dep	checkp*	log+checkp
Guar for State	none/conf*	same/conf	–^{B} valid / same^{S}	valid / same	valid / same	same*	valid	valid / same*
Assumptions	STOR / REPL	STOR / REPL (+DC)	REPLAY	REPLAY	–	STOR	sys dep	sys dep
Dynamic Reconfiguration								
Goal	avail +load bal+elast	change schema +load bal+elast	– / elast	– / load bal+elast	load bal	sys dep	– / elast	– / load bal+elast
Automated	sys dep	sys dep	– / yes	sys dep	yes	yes	– / yes	sys dep
State Migr	yes	yes	–^{B} / yes^{S}	yes	yes	yes	– / yes	– / yes
Task Migr	–	–	–^{B} / yes^{S}	yes	yes	sys dep	– / yes	– / yes
Add/Rem Slots	yes	yes	– / yes	– / yes	no	yes	– / yes	sys dep
Restart	no	no*	– / no	sys dep	no	no	– / no	sys dep

Sys dep, system dependent; class dep, differences captured by the specific sub-classes in our taxonomy (see Figure 3); *, with few system-specific exceptions; ^{B}, in the case of batch processing; ^{S}, in the case of stream processing; DC, jobs are deterministic; SEQ, jobs are executed sequentially, with no interleaving; 1W, a single worker handles all writes; 1P, jobs access a single state portion; STOR, storage layer is durable; REPL, replicated data is durable; REPLAY, sources are replayable.

4 DATA MANAGEMENT SYSTEMS

DMSs offer the abstraction of a mutable state store that many jobs can access simultaneously to query, retrieve, insert, and update elements. Differently from DPSs, they mostly target lightweight jobs, which do not involve computationally expensive data transformations and are short lived. Since their advent in the 1970s, relational databases represented the standard approach to data management, offering a uniform data model (relational), query language (SQL), and execution semantics (transactional). Over the past two decades, new requirements and operational conditions brought the idea of a unified approach to data management to an end [87, 88]: new applications emerged with different needs in terms of data and processing models, for instance, to store and retrieve unstructured data; scalability concerns related to data volume, number of simultaneous users, and geographical distribution pointed up the cost of transactional semantics. This state of things fostered the development of the DMSs presented in this section. Section 4.1 discusses the

aspects in our model that are common to all such systems. Then, following an established terminology, we organize them in two broad classes: NoSQL databases [36] (Section 4.2) emerged since the early 2000s, providing simple and flexible data models such as key-value pairs, and trading consistency guarantees and strong (transactional) semantics for horizontal scalability, high availability, and low response time; NewSQL databases [86] (Section 4.3) emerged in the late 2000s and take an opposite approach: they aim to preserve the traditional relational model and transactional semantics by introducing new design and implementation strategies that reduce the cost of coordination.

4.1 Overview

Functional Model. All DMSs provide a global shared state that applications can simultaneously access and modify. In the more traditional systems, there is a sharp distinction between the application logic (the driver, executed client-side on registration) and the queries (the jobs, executed by the DMS). Recent systems increasingly allow to move the part of the application logic that orchestrates jobs execution within the DMS, in the form of stored procedures that run system-side on start. Stored procedures may bring two advantages: (i) reducing the interactions with external clients, thus improving latency, and (ii) moving part of the overhead for compiling jobs from driver execution time to driver registration time.

Being conceived for interactive use, all DMSs offer synchronous APIs to invoke jobs from the driver program. Many also offer asynchronous APIs that allow the driver to invoke multiple jobs and receive notifications of their results when they terminate. A common approach to reduce the cost of communication when starting jobs from a client-side driver is batching multiple invocations together, which is offered in some NoSQL systems such as MongoDB [32] and Redis [64].

Several DMSs can interact with active sources and sinks. Active sources push new data into the system, leading to insertion or modification of state elements. Sinks register to state elements of interest (e.g., by specifying a key or a range of keys) and are notified upon modification of such elements.

DMSs greatly differ in terms of deployment strategies, which are vastly influenced by the coordination protocols that govern replication, group atomicity, and isolation. NoSQL systems do not offer group atomicity and isolation. Those designed for cluster deployment typically use blocking (synchronous or semi-synchronous) replication protocols. Those that support wide-area deployments either use coordination-free (asynchronous) replication protocols that reduce durability and consistency guarantees or employ a hybrid strategy, with synchronous replication within a data center and asynchronous replication across data centers. Many NewSQL systems claim to support wide-area deployments while offering strong consistency, group atomicity, and isolation. We review their implementation strategies to achieve this result in Section 4.3.

Jobs. All DMSs implement one-shot jobs with explicit state management primitives to read and modify a global shared state. Jobs definition APIs greatly differ across systems, ranging from pure key-value stores that offer CRUD (create, read, update, and delete) primitives for individual state elements to expressive domain-specific libraries (e.g., for graph computation) or languages (e.g., SQL for relational data). In almost all DMSs, the execution plan and the communication between tasks are implicit and do not include iterations or dynamic creation of new tasks. A notable exception are graph databases, which support iterative algorithms where developers explicitly define how tasks update portions of the state (the graph) and exchange data. The structure of the execution plan also varies across systems: common structures include the use of a single task that implements CRUD primitives, workflows orchestrated by a central coordinator task, or hierarchical structures. Jobs are compiled on driver execution, except for those systems where part of the

driver program is registered server-side (as stored procedures). Job compilation always uses static information about resources, such as the allocation of shared state portions onto nodes. Some structured NewSQL DMSs such as Spanner [14] and CockroachDB [90] also exploit dynamic information about resources—for instance, to configure a given task (e.g., select a sort-merge or a hash-based strategy for a join task) depending on some resource utilization or statistics about state (e.g., cardinality of the tables to join).

All DMSs perform deployment at the job level, when the job is compiled, with the only exception of AsterixDB [9], which compiles jobs into a dataflow plan and deploys individual tasks when they are activated [19]. Deployment is always guided by the location of the state elements to be accessed, which we consider as a static information. Indeed, under normal execution, shared state portions do not move, and our model captures dynamic relocation of shared state portions (e.g., for load balancing) as a distinct aspect (*dynamic reconfiguration*). In addition, we keep saying that deployment is based on static information even for those systems that exploit dynamic information (e.g., the load of workers) but only to deploy the tasks that manage the communication between the system and external clients. Finally, DMSs are not typically designed to operate in scenarios where the compute infrastructure is shared with other software applications. This is probably due to the interactive and short-lived nature of jobs, which would make it difficult to predict and adapt the demand of resources in those scenarios. The only case in which we found explicit mention of a shared platform is in the description of the Google infrastructure, where DMS components (BigTable [29], Percolator [76], Spanner [34]) share the same physical machines and their scheduling, execution, and monitoring is governed by an external resource management service.

Data and State Management. The data model (i.e., the structure of data and state elements) is a key distinguishing characteristic of DMSs, which we use to organize our discussion in Sections 4.2 and 4.3. Some systems explicitly consider the temporal dimension of data elements. For instance, some wide columns stores associate timestamps to elements and let users store multiple versions of the same element, whereas time-series databases are designed to efficiently store and query sequences of measurements over time. DMSs differ in terms of storage medium and structure. We detail the choices of the different classes of systems in Section 4.2 and Section 4.3, but we can identify some common concerns and design strategies. First, the representation of state on storage is governed by the data model and the expected access pattern: relational tables are stored by row, whereas time series are stored by column to facilitate computations on individual measurements over time (e.g., aggregation, trend analysis). Second, there is a tension between read and write performance: read can be facilitated by indexed data structures such as B-trees, but they incur higher insertion and update costs. Hierarchical structures such as LSM trees improve write performance by buffering updates in higher-level logs (frequently in-memory) that are asynchronously merged into lower-level indexed structured, at the expense of read performance, due to the need to navigate multiple layers. Third, most DMSs exploit main memory to reduce data access latency. For instance, systems based on LSM trees store the write buffer in memory. Similarly, most systems that use disk-based storage frequently adopt some in-memory caching layer. Finally, some systems adopt a modular architecture that supports different storage layers. This is common in DMSs offered as a service in public cloud environments (e.g., Amazon Aurora [95]) or in private data centers (e.g., Google Spanner [34]), where individual system components (including the storage layer) are themselves services.

Tasks always communicate using direct, ephemeral connections, which implement a partitioned, non-replicated data bus. Shared state is always partitioned across workers to enable concurrent execution of tasks. Most DMSs also adopt state replication to improve read access performance, with different guarantees in terms of consistency between replicas: NoSQL databases provide weak (or

configurable) consistency guarantees to improve availability and reduce response time. NewSQL databases provide strong consistency using leader-based or consensus protocols and restricting the types of transactions (jobs) that can read from non-leader replicas—typically snapshot transactions, which are a subset of read-only transactions that read a consistent version of the state, without guarantees of it being the most recent one. Group atomicity and isolation are typically absent or optional in NoSQL databases, or restricted to very specific cases, such as jobs that operate on single data elements. Instead, NewSQL databases provide strong guarantees for atomicity and isolation, at the cost of blocking coordination. The transactional semantics of NewSQL systems ensures exactly once delivery. Indeed, transactions (jobs) either complete successfully (and their effects become durable) or abort, in which case they are either automatically retried or the client is notified and can decide to retry them until success. Conversely, NoSQL systems frequently offer at most once semantics, as they do not guarantee that the results of job execution are safely stored on persistent storage or replicated. In some cases, users can balance durability and availability by selecting the number of replicas that are updated synchronously. Finally, systems that support timestamps use event time semantics, where timestamps are associated with state elements by clients, whereas none of the systems provides order guarantees. Even in the presence of timestamps, DMSs do not implement mechanisms to account for elements produced or received out of timestamp order.

Fault Tolerance. Frequently, DMSs offer multiple mechanisms for fault tolerance and durability that administrators can enable and combine depending on their needs. Fault detection can be centralized (leader-worker) or distributed, depending on the specific system. Fault recovery mostly targets the durability of shared state. Since jobs are lightweight, DMSs simply abort them in the case of failure and do not attempt to recover (partial) computations. Transactional systems guarantee group atomicity: in the case of failure, none of the effects of a job become visible. To enable recovery of failed jobs, they either notify the clients about an abort, allowing them to restart the failed job, or restart the job automatically. Almost all DMSs adopt logging mechanisms to ensure that the effects of jobs execution on shared state are durable. Logging enables changes to be recorded on some durable append-only storage before being applied to shared state: most systems adopt a WAL that records changes to individual data elements, whereas few others adopt a command log that stores the operations (commands) that perform the change. Individual systems make different assumptions on what they consider as durable storage: in some cases the logs are stored on a single disk, but more frequently they are replicated (or saved on third-party log services that are internally replicated). Logging is frequently used in combination with replication of shared state portions on multiple workers: in these cases, each worker stores its updates on a persistent log to recover from software crashes, whereas replication on other workers may avoid unavailability in the case of hardware crashes or network disconnections. In addition, most systems offer geo-replication for disaster recovery, where the entire shared state is replicated in a different data center and periodically synchronized with the working copy. Similarly, many systems provide periodic or manual checkpointing to store a copy of the entire database at a given point in time. Depending on the specific mechanisms adopted, DMSs provide either no guarantees for state, such as in the case of asynchronous replication, or same state guarantees, such as in the case of persistent log or consistent replication.

Dynamic Reconfiguration. Most DMSs support adding and removing workers at runtime, and migrating shared state portions across workers, which enable dynamic load balancing and scaling without restarting the system. All systems support dynamic reconfiguration as a manual administrative procedure, and some can also automatically migrate state portions for load balancing. A special case of reconfiguration for NewSQL systems that rely on structured state (in particular,

relational systems) involves changing the state schema: many systems support arbitrary schema changes as long-running procedures that validate state against the new schema and migrate it while still serving clients using the previous schema. Instead, systems such as VoltDB [89] rely on a given state partitioning scheme and prevent changes that violate such scheme.

4.2 NoSQL Systems

Using an established terminology, we classify as NoSQL all those DMSs that aim to offer high availability and low response time by relinquishing features and guarantees that require blocking coordination between workers. In particular, they typically do the following. First, they avoid expressive job definition APIs that may lead to complex execution plans (as in the case of SQL, hence the name) [85]. In fact, the majority of the systems we analyzed focus on jobs comprising a single task that operates on an individual element of the shared state. Second, they use asynchronous replication protocols: depending on the specific protocol, this may affect consistency, may affect durability (if replication is used for fault tolerance), and may generate conflicts (when clients are allowed to write to multiple replicas). Third, they abandon or restrict group guarantees (atomicity and isolation) when jobs with multiple tasks are supported. In our discussion, we classify systems by the data model they offer.

4.2.1 Key-Value Stores. Key-value stores offer a very basic API for managing shared state: (i) shared state elements are represented by a key and a value; (ii) elements are schema-less, meaning that different elements can have different formats, such as free text or JSON objects with heterogeneous attributes; (iii) the key-space is partitioned across workers; and (iv) jobs consist of a single task that retrieves the value of an element given a key (`get`) or insert/update an element given its key (`put`).

Individual systems differ in the way they physically store elements. For instance, Dynamo [39] supports various types of physical storage, DynamoDB uses B-trees on disk but buffers incoming updates in main memory to improve write performance, and Redis [64] stores elements in memory only. Keys are typically partitioned across workers based on their hash (hash partitioning), but some systems also support range partitioning, where each worker stores a sequential range of keys, as in the case of Redis. Given the focus on latency, some systems cache the association of keys to workers client-side, allowing clients to directly forward requests to workers responsible for the key they are interested in.

Some systems provide richer APIs to simplify the interaction with the store. First, keys can be organized into tables, mimicking the concept of a table in a relational database. For instance, DynamoDB and PNUTS [33] let developers split state elements into tables. Second, elements may have an associated structure. For instance, PNUTS lets developers optionally define a schema for each table, DynamoDB specifies a set of attributes but does not constrain their internal structure, and Redis provides built-in data types to define values and represent them efficiently in memory. Third, most systems provide functions to iterate (`scan`) on the key-space or on individual tables (range-based partitioning may be used to speedup such range-based iterations), as it happens for DynamoDB, PNUTS, and Redis. Fourth, some systems provide query operations (`select`) to retrieve elements by value and in some cases these operations are supported by secondary indexes that are automatically updated when the value of an element changes, as in DynamoDB.

All key-value stores replicate shared state to increase availability but use different replication protocols. Dynamo, Voldemort, and Riak KV use a quorum approach, where read and write operations for a given key need to be processed by a given number of replica workers responsible for that key. After a write quorum is reached, updates are propagated to remaining replicas asynchronously. A larger number of replicas and a larger write quorum better guarantee durability and consistency

at the cost of latency and availability. However, being designed for availability, these systems adopt mechanisms to avoid blocking on write when some workers are not responsive. For instance, other workers can supersede and store written values on their behalf. In some cases, these mechanisms can lead to conflicting simultaneous updates: Dynamo tracks causal dependencies between writes to solve conflicts automatically whenever possible, and stores conflicting versions otherwise, leaving manual reconciliation to the users. DynamoDB, Redis, and Aerospike [84] use single-leader protocols where all writes are processed by one worker and propagated synchronously to some replicas and asynchronously to others, depending on the configuration. DynamoDB also supports consistent reads that are always processed by the leader at a per-operation granularity. Redis supports multi-leader replication in the case of wide-area scenarios, using conflict-free replicated data types for automated conflict resolution.

In summary, key-value stores represent the core building block of a DMS. They expose a low-level but flexible API to balance availability, consistency, and durability, and to adapt to different deployment scenarios. Systems that adopt a modular implementation can use key-value stores as a storage layer or a caching layer [73] and build richer job definition API, job execution engines, protocols for group atomicity, group isolation, and consistent replication on top.

4.2.2 Wide-Column Stores. Wide-column stores organize shared state into tables (multi-dimensional maps), where each row associates a unique key to a fixed number of column families, and each column family contains a value, possibly organized into more columns (attributes). State is physically stored per column family and keys need not have a value for each column family (the table is typically sparse). One could define the wide-column data model as middle ground between the key-value and the relational model: it is similar to the key-value model but associates a key to multiple values (column families), and it defines tables as the relational model, but tables are sparse and lack referential integrity. The main representative systems of this class are Google BigTable [29], with its open source implementation HBase, and Apache Cassandra [57]. As the official documentation of Cassandra explains, the typical use of wide-column systems is to compute and store answers to frequent queries (read-only jobs) for each key, at insertion/update time, within column families. In contrast, relational databases normalize tables to avoid duplicate columns and compute results at query time (rather then insertion/update time) by joining data from multiple tables. In fact, wide-column stores offer rich API to scan, select, and update values by key but do not offer any join primitive. To support the preceding scenario, wide-column systems (i) aim to provide efficient write operations to modify several column families (i.e., both BigTable and Cassandra adopt LSM trees for storage and improve write latency by buffering writes in memory) and (ii) provide isolation for operations that involve the same key. These two design choices allow users to update all entries for a given key (answers to queries) efficiently and in isolation.

BigTable and Cassandra have different approaches to replication. BigTable uses replication only for fault tolerance and executes all tasks that involve a single key on the leader worker responsible for that key. It also supports wide-area deployment by fully replicating the data store in additional data centers: replicas in these data centers can be used for fault tolerance but also to perform jobs, in which case they are synchronized with eventual consistency. Cassandra uses quorum replication as in Dynamo, and allows users to configure the quorum protocols to trade consistency and durability for availability.

4.2.3 Document Stores. Document stores represent a special type of key-value stores where values are structured documents, such as XML or JSON objects. Document stores offer an API similar to key-value stores, but they can exploit the structure of documents to update only some of their fields. Physical storage solutions vary across systems, ranging from disk-based to memory solutions to hybrid approaches and storage-agnostic solutions. In most cases, document stores

support secondary indexes to improve retrieval of state elements using criteria different from the primary key. Most document stores offer group isolation guarantees for jobs that involve a single document. This is the case of MongoDB [32] and AsterixDB [9]. Recent versions of MongoDB also implement multi-document atomicity and isolation as an option, using blocking protocols.

MongoDB supports replication for fault tolerance or also to serve read-only jobs. It implements a single-leader protocol with semi-synchronous propagation of changes, where clients can configure the number of replicas that need to synchronously receive an update, thus trading durability and consistency for availability and response time. CouchDB [10] offers a quorum-based replication protocol and allows for conflicts in the case a small write quorum is selected. In this case, conflict resolution is manual. AsterixDB does not currently support replication.

Several document stores support some form of data analytic jobs. MongoDB offers jobs in the form of a pipeline of data transformations that can be applied in parallel to a set of documents. CouchDB focuses on Web applications and can start registered jobs when documents are added or modified to update some views. AsterixDB provides a declarative language that integrates operators for individual and for multiple documents (like joins, group by), and compiles jobs into a dataflow execution plan.

4.2.4 Time-Series Stores. Time-series stores are a special form of wide-column stores dedicated to store sequences of values over time, like measurements of a numeric metric such as the CPU utilization of a computer over time. Given the specific application scenario, this class of systems stores data by column, which brings several advantages: (i) together with the use of an in-memory or hybrid storage layer, it improves the performance of write operations, which typically append new values (measurements) to individual columns; (ii) it offers faster sequential access to columns, which is common in read-only jobs that perform aggregations or look for temporal patterns over individual series; and (iii) it enables a higher degree of data compression, such as by storing only the difference between adjacent numerical values (delta compression), which is small if measurements change slowly.

Among the time-series stores we analyzed, InfluxDB is the most general one. It provides a declarative job definition language that supports computations on individual columns (measurements). Gorilla [75] is used as an in-memory cache to store monitoring metrics at Facebook. Given the volume and rate at which metrics are produced, Facebook keeps the most recent data at a very fine granularity within the Gorilla cache and stores historical data at a coarser granularity in HBase. Peregreen [96] follows a similar approach and optimizes retrieval of data through indexing. It uses a three-tier data indexing, where each tier pre-computes aggregated statistics (minimum, maximum, average, etc.) for the data it references. This allows to quickly identify chunks of data that satisfy some conditions based on the pre-computed statistics and to minimize the number of interactions with the storage layer. Monarch [3] is used to store monitoring data at Google. It has a hierarchical architecture: data is stored in the zone (data center) in which it is generated and sharded (by key ranges, lexicographically) across nodes called *leaves*. Jobs are evaluated hierarchically: nodes are organized in three layers (global, zone level, leaves), and the job plan pushes tasks as close as possible to the data they need to consume. All time-series stores we analyzed replicate shared state to improve availability and performance of read operations. To avoid blocking write operations, they adopt asynchronous or semi-synchronous replication, thus reducing durability guarantees. This is motivated by the specific application scenarios, where losing individual measurements may be tolerated.

4.2.5 Graph Stores. Graph stores are a special form of key-value stores specialized in graph-shaped data, meaning that shared state elements represent entities (vertices of a graph) and their relations (edges of the graph). Despite that researchers widely recognized the importance

of large-scale graph data structures, several graph data stores do not scale horizontally [80]. A prominent example of distributed graph store is TAO [20], used at Facebook to manage the social graph that interconnects users and other entities such as posts, locations, and actions. It builds on top of key-value stores with hybrid storage (persisted on disk and cached in memory), asynchronously replicated with no consistency or grouping guarantees.

A key distinguishing factor in graph stores is the type of queries (read-only jobs) they support. Indeed, a common use of graph stores is to retrieve sub-graphs that exhibit certain patterns of relations. For instance, in a social graph, one may want to retrieve people (vertices) that are direct friends or have friends in common (friendship relation edges) and like the same posts. This problem is denoted as *graph pattern matching*, and its general form can only be solved by systems that can express iterative or recursive jobs, as it needs to traverse the graph following its edges. These types of vertex-centric computations have been first introduced in the Pregel DPS [66], also discussed in Section 5.

Efficient query of graph stores can also be supported by external systems. For instance, Facebook developed the Unicorn [35] system to store indexes that allow to quickly navigate and retrieve data from a large graph. Indexes are updated periodically using an external compute engine. Unicorn adopts a hierarchical architecture, where indexes (the shared state of the system) are partitioned across servers and the results of index lookups (read jobs) are aggregated first at the level of individual racks and then globally to obtain the complete query results. This approach aggregates results as close as possible to the servers producing them to reduce network traffic. Unicorn supports graph patterns queries by providing an `apply` function that can dynamically start new lookups based on the results of previous ones: our model captures this feature by saying that jobs can dynamically start new tasks.

4.3 NewSQL Systems

NewSQL systems aim to provide transactional semantics (group atomicity and isolation), durability (fault tolerance), and strong replication consistency while preserving horizontal scalability. Following the same approach we adopted in Section 4.2, we organize them according to their data model.

4.3.1 Key-Value Stores. NewSQL key-value stores are conceived as part of a modular system, where the store offers transactional guarantees to read and update a group of elements with atomicity and isolation guarantees, and it is used by a job manager that compiles and optimizes jobs written in some high-level declarative language. A common design principle of these systems is to separate the layer that manages the transactional semantics from the actual storage layer, thus enabling independent scaling based on the application requirements. Deuteronomy [59] implements transactional semantics using a locking protocol and is storage agnostic. FoundationDB [100] uses optimistic concurrency control with a storage layer based on B-trees. Solar [101] also uses optimistic concurrently control with LSM trees.

4.3.2 Structured and Relational Stores. Stores for structured and relational data provide the same data model, job model, and job execution semantics as classic non-distributed relational databases. As we clarify in the following classification, they differ in their protocols for implementing group atomicity, group isolation, and replication consistency, which reflects on their architectures.

Time-Based protocols. Some systems exploit physical (wall-clock) time to synchronize nodes. This approach was pioneered by Google's Spanner [34]. It adopts standard database techniques: two-phase commit for atomicity, two-phase locking and multi-version concurrency control for

isolation, and single-leader synchronous replication of state portions. Paxos consensus is used to elect a leader for each state portion and to keep replicas consistent. The key distinguishing characteristic of Spanner is the use TrueTime, a clock abstraction that uses atomic clocks and GPS to return physical time within a known precision bound. In Spanner, each job is managed by a transaction coordinator, which assigns jobs with a timestamp at the end of the TrueTime clock uncertainty range and waits until this timestamp is passed for all nodes in the system. This ensures that jobs are globally ordered by timestamp, thus offering the illusion of a centralized system with a single clock (external consistency). Spanner is highly optimized for workloads with many read-only jobs. Indeed, multi-version concurrency control combined with TrueTime allows read-only jobs to access a consistent snapshot of the shared state without locking and without conflicting with in-progress read-write jobs, as they will be certainly be assigned a later timestamp. More recently, Spanner has been extended with support for distributed SQL query execution [14]. CockroachDB [90] is similar to Spanner but uses an optimistic concurrency control protocol that, in the case of conflicts, attempts to modify the timestamp of a job to a valid one rather than re-executing the entire job. Like Spanner, CockroachDB supports distributed execution plans. It supports wide-area deployment and allows users to define how data is partitioned across regions, to promote locality of data access or to enforce privacy regulations.

Deterministic Execution. Calvin [92] builds on the assumption that jobs are deterministic and achieves atomicity, isolation, and consistency by ensuring that jobs are executed in the same order in all replicas. Determinism ensures that jobs either succeed or fail in any replica (atomicity), and interleave in the same way (global order ensures isolation), leading to the same results (consistency). Workers are organized into three layers. The first is a sequencing layer that receives jobs invocations from clients, organizes them into batches, and orders them consistently across replicas. Ordering of jobs is the only operation that requires coordination and takes place before jobs execution. Calvin provides both synchronous (Paxos) and asynchronous protocols for ordering jobs, which bring different tradeoffs between latency and cost of recovery in the case of failures. The second is a scheduler layer that executes tasks onto workers in the defined global order. In cases where it is not possible to statically determine which shared state portions will be involved in the execution of a job (i.e., in the case of state-dependent control flow), Calvin uses an optimistic protocol and aborts jobs if some of their tasks are received by workers out of order. The third is a storage layer that stores the actual data. In fact, Calvin supports any storage engine providing a key-value interface.

Explicit Partitioning and Replication Strategies. VoltDB [88, 89] lets users control partitioning and replication of shared state, so they can optimize most frequently executed jobs. For instance, users can specify that `Customer` and `Payment` tables are both partitioned by the attribute (column) `customerId`. Jobs that are guaranteed to access only a shared state portion within a given worker are executed sequentially and atomically on that worker. For instance, a job that accesses tables `Customer` and `Payment` to retrieve information for a given `customerId` can be fully executed on the worker with the state portion that includes that customer. Every table that is not partitioned is replicated in every worker, which optimizes read access from any worker at the cost of replicating state changes. In the case in which jobs need to access state portions at different workers, VoltDB resorts to standard two-phase commit and timestamp-based concurrency control protocols. Differently from Spanner and Calvin, VoltDB provides strong consistency only for cluster deployment: geographical replication is supported but only implemented with asynchronous and weakly consistent protocols.

Primary-Based Protocols. Primary-based protocols are a standard approach to replication used in traditional transactional databases. They elect one primary worker that handles all read-write jobs

and acts as a coordinator to ensure transactional semantics. Other (secondary) workers only handle read-only jobs and can be used to fail over if the primary crashes. Recently, the approach has been revamped by DMSs offered as services on the cloud. These systems adopt a layered architecture that decouples jobs execution functionalities (e.g., scheduling, managing atomicity and isolation) from storage functionalities (durability): the two layers are implemented as services that can scale independently from each other. The execution layer still consists of one primary worker and an arbitrary number of secondary workers, which access shared state through the storage service (although they typically implement a local cache to improve performance). Amazon Aurora [95] implements the storage layer as a sequential log (replicated for availability and durability), which offers better performance for write operations. Indexed data structures that improve read performance are materialized asynchronously without affecting write latency. The storage layer uses a quorum approach to guarantee replication consistency across workers. Microsoft Socrates [11] adopts a similar approach but further separates storage into a log layer (that stores write requests with low latency), durable storage layer (that stores a copy of the shared state), and a backup layer (that periodically copies the entire state).

4.3.3 Objects Stores. Object stores became popular in the early 1990s, inheriting the same data model as object-oriented programming languages. We found one recent example of a DMS that uses this data model, namely Tango [16]. In Tango, clients store their view of objects locally, in-memory, and this view is kept up-to-date with respect to a distributed (partitioned) and durable (replicated) log of updates. The log represents the primary replica of the shared state that all clients refer to. All updates to objects are globally ordered on the log through sequence numbers that are obtained through a centralized sequencer. Total order guarantees isolation for operations on individual objects: Tango also offers group atomicity and isolation across objects using the log to store information for an optimistic concurrency control protocol.

4.3.4 Graph Stores. We found one example of a NewSQL graph store, named *A1* [22], which provides strong consistency, atomicity, and isolation using timestamp-based concurrency control. Its data model is similar to that of NoSQL distributed graph stores, and jobs can traverse the graph and read and modify its associated data during execution. The key distinguishing characteristic of A1 is that it builds on a distributed shared memory abstraction that uses RDMA (remote direct memory access) implemented within network interface cards [41].

5 DATA PROCESSING SYSTEMS

DPSs aim to perform complex computations (long-lasting jobs) on large volumes of data. Most of today's DPSs inherit from the seminal MapReduce system [38]: to avoid the hassle of concurrent programming and to simplify scalability, they organize each job into a dataflow graph where vertices are functional operators that transform data and edges are the flows of data across operators. Each operator is applied in parallel to independent partitions of its input data, and the system automatically handles data partitioning and data transfer across workers. Following an established terminology, we denote as *batch processing* systems those that take in input static (finite) datasets, and *stream processing* systems those that take in input streaming (potentially unbounded) datasets. In practice, many systems support both types of input, and we do not use the distinction between batch and stream processing as the main factor to organize our discussion. Instead, after discussing the aspects in our model that are common to all DPSs (Section 5.1), we classify dataflow systems based on the key aspect that impacts their implementation: if they deploy individual tasks on activation (Section 5.2) or entire jobs on registration (Section 5.3). Finally, we present systems designed to support computations on large graph data structures. They evolved in parallel with respect to

dataflow systems, which originally were not suited for iterative computations that are typical in graph algorithms (Section 5.4).

5.1 Overview

Functional Model. Most DPSs use a leader-workers architecture, where one of the processes that compose the system (denoted the leader) has the special role of coordinating other workers. Such systems always allow submitting the driver program to the leader for system-side execution. Some of them also allow client-side driver execution, such as Apache Spark [99] and Apache Flink [24]. Other systems, such as Kafka Streams [18] and timely dataflow [71], are implemented as libraries where client processes also act as workers. Developers start one or more client processes, and the library handles the distributed execution of jobs onto them. Stream processing systems support asynchronous invocation of (continuous) jobs, whereas batch processing systems may offer synchronous or asynchronous job invocation APIs, or both. All DPSs support sources and sinks, as they are typically used to read data from external systems (sources), perform some complex data analysis and transformation (jobs), and store the results into external systems (sinks). Sources are passive in the case of batch processing systems and active in the case of stream processing systems. Most batch processing systems are stateless: output data is the result of functional transformations of input data. Stream processing systems can persist a (task) state across multiple activations of a continuous job. We model iterative graph algorithms as continuous jobs where tasks (associated with vertices, edges, or sub-graphs) are activated at each iteration and store their partial results (values associated with vertices, edges, or sub-graphs) in task state. DPSs assume a cluster deployment, as job execution typically involves exchanging large volumes of data (input, intermediate results, and final results) across workers.

Jobs. All dataflow systems provide libraries to explicitly define the execution plan of jobs. Increasingly often, they also offer higher-level abstractions for specific domains, such as relational data processing [2, 12, 23], graph computations [46], or machine learning [69]. Some of these APIs are declarative in nature and make the definition of the execution plan implicit. Task communication is always implicitly defined and controlled by the system's runtime. Concerning jobs, dataflow systems differ with respect to the following aspects. The first is generality. MapReduce [38] and some early systems derived from it only support two processing stages with fixed operators, whereas later systems like Spark support any number of processing stages and a vast library of operators. The second is support for iterations. Systems like HaLoop [21] extended MapReduce to efficiently support iterations by caching data accessed across iterations in workers. Spark inherits the same approach and, together with Flink, supports some form of iterative computations for streaming data. Timely dataflow [71] generalizes the approach to nested iterations. The third is dynamic creation of tasks. Among the systems we analyzed, only CIEL [72] enables dynamic creation of tasks depending on the results of processing.

Data parallelism is key to dataflow systems, and all operators in their jobs definition API are data parallel. Jobs are one-shot in the case of batch processing and continuous in the case of stream processing. In the latter case, jobs may implicitly define some task state, such as by expressing computations that operate on a recent portion (window) of data rather than on individual data elements. Jobs cannot control task placement explicitly, but many systems provide configuration parameters to guide placement decisions—for instance, to force or inhibit the colocation of certain tasks. An exception to the preceding rules is represented by graph processing systems, which are based on a programming model where developers define the behavior of individual vertices [68]: the model provides explicit primitives to access the state of a vertex and to send messages between vertices (explicit communication).

All DPSs compile jobs on driver execution. For other characteristics related to jobs compilation, deployment, and execution, we distinguish between systems that perform task-level deployment (discussed in Section 5.2) and systems that perform job-level deployment (discussed in Section 5.3).

Data and State Management. Deployment and execution strategies affect the implementation of the data bus. In the case of job-level deployment, the data bus is implemented using ephemeral, push-based communication channels between tasks (e.g., direct TCP connections). In the case of task-level deployment, the data bus is mediated and implemented by a persistent service (e.g., a distributed filesystem or a persistent message queuing system) where upstream tasks push the results of their computation and downstream tasks pull them when activated. A persistent data bus can be replicated for fault tolerance, as in the case of Kafka Streams, which builds on replicated Kafka topics. CIEL [72] and Dryad [51] support hybrid bus implementations, where some connections may be direct while others may be mediated. Data elements may range from arbitrary strings (unstructured data) to specific schemas (structured data). The latter offer opportunities for optimizations in the serialization process, such as allowing for better compression or for selective deserialization of only the fields that are accessed by a given task. In general, DPSs do not provide shared state. Stream processing and graph processing systems include a task state to persist information across multiple activations of a continuous job (i.e., windows). In the absence of shared state, DPSs do not provide group atomicity or isolation properties. Almost all systems provide exactly once delivery, under the assumption that sources can persist and replay data in the case of failure and sinks can distinguish duplicates. The concrete approaches to provide such guarantee depend on the type of deployment (task level or job level) and are discussed later in Sections 5.2 and 5.3. Order is relevant for stream processing systems: with the exception of Storm [94], all stream processing systems support timestamped data (event or ingestion time semantics). Most systems deliver events in order, under the assumptions that sources either produce data with a pre-defined maximum delay or inform the system about the progress of time using special metadata denoted as watermark. Kafka Streams takes a different approach: it does not wait for out-of-order elements and immediately produces results. In the case in which new elements arrive out of order, it retracts updates the previous results.

Fault Tolerance. All DPSs detect faults using a leader-worker architecture, and in absence of a shared state, they recover from failures through the mechanisms that guarantee exactly once delivery.

Dynamic Reconfiguration. DPSs use dynamic reconfiguration to adapt to the workload by adding or removing slots. Systems that adopt task-level deployment can decide how to allocate resources to individual tasks when they are activated, whereas systems that adopt job-level deployment need to suspend and resume the entire job, which increases the overhead for performing a reconfiguration. The mechanisms that dynamically modify the resources (slots) available to a DPS can be activated either manually or by an automated service that monitors the utilization of resources and implements the allocation and deallocation policies. All commercial systems implement automated reconfiguration, frequently by relying on external platforms for containerization, such as Kubernetes, or for cluster resources management, such as YARN. The only exceptions for which we could not find official support for automated reconfiguration are Storm [94] and Kafka Streams [18].

5.2 Dataflow with Task-Level Deployment

Systems that belong to this class deploy tasks on activation, when their input data becomes available. Tasks store intermediate results on a persistent data bus, which allows to selectively restart them in the case of failure. This approach is best suited for long running batch jobs and was

pioneered in the MapReduce batch processing system [38]. It has been widely adopted in various extensions and generalizations. HaLoop optimizes iterative computations by caching loop-invariant data and by co-locating tasks that reuse the same data across iterations [21]. Dryad [51] generalizes the programming model to express arbitrary dataflow plans and enables developers to flexibly select the concrete channels (data bus in our model) that implement the communication between tasks. CIEL [72] extends the dataflow model of Dryad by allowing tasks to create other tasks dynamically, based on the results of their computation. Spark is the most popular system of this class [99]: it inherits the dataflow model of Dryad and supports iterative execution and data caching like HaLoop. Spark Streaming [98] implements streaming computations on top of Spark by splitting the input stream into small batches and by running the same job for each batch. It implements task state using native Spark features: the state of a task after a given invocation is implicitly stored as a special data item that the task receives as input in the subsequent invocation.

In systems with task-level deployment, job compilation considers dynamic information to create tasks. For instance, the number of tasks instantiated to perform a data-parallel operation depends on how the input data is partitioned. Similarly, the deployment phase uses dynamic information to submit tasks to workers running as close as possible to the their input data. Hadoop (the open source implementation of MapReduce) and Spark adopt a delay scheduling [97]. They put jobs (and their tasks) in a FIFO queue. When slots become available, the first task in the queue is selected: if the slot is located near to the input data for the task, then the task is immediately deployed, and otherwise each task can be postponed for some time to wait for available slots closer to their input data. Task-level deployment enables sharing of compute resources with other applications: in fact, most of the systems that use this approach can be integrated with cluster management systems.

Task-level deployment also influences how systems implement fault tolerance and ensure exactly once delivery of results. Batch processing systems simply re-execute the tasks involved in the failure. In absence of state, the results of a task depend only on input data and can be recomputed at need. Intermediate results may be persisted on durable storage and retrieved in the case of a failure or recomputed from the original input data. Spark Streaming [98] adopts the same fault tolerance mechanism for streaming computations. It segments a stream into a sequence of so-called micro-batches and executes them in order. Task state is treated as a special form of data; it is periodically persisted to durable storage and is retrieved in the case of a failure. Failure recovery may require activating failed tasks more than once, to recompute the task state from the last state persisted before the failure.

Dynamic reconfiguration is available in all systems that adopt task-level deployment. Systems that do not provide any state abstraction can simply exploit new slots to schedule tasks when they become available and remove workers when idle. In the presence of task state, migrating a task involves migrating its state across activations: as in the case of fault tolerance, this is done by storing task state on persistent storage.

5.3 Dataflow with Job-Level Deployment

In the case of job-level deployment, all tasks of a job are deployed onto the slots of the computing infrastructure on job registration. As a result, this class of systems is better suited for streaming computations that require low latency: indeed, no scheduling decision is taken at runtime and tasks are always ready to receive and process new data. Storm [94] and its successor Heron [56] are stream processing systems developed at Twitter. They offer a lower-level programming API than dataflow systems discussed previously, asking developers to fully implement the logic of each processing step using a standard programming language. Flink [24] is a unified execution engine for batch and stream processing. In terms of a programming model, it strongly resembles Spark, with a core API to explicitly define job plans as a dataflow of functional operators, and

domain-specific libraries for structural (relational) data, graph processing, and machine learning. One notable difference involves iterative computations: Flink supports them with native operators (within jobs) rather than controlling them from the driver program. Timely dataflow [71] offers a lower-level and more general dataflow model than Flink, where jobs are expressed as a graph of (data-parallel) operators and data elements carry a logical timestamp that tracks global progress. Management of timestamps is explicit, and developers control how operators handle and propagate them, which enables various execution strategies. For instance, developers may choose to complete a given computation step before letting the subsequent one start (mimicking a batch processing strategy as implemented in MapReduce or Spark), or they may allow overlapping of steps (as it happens in Storm or Flink). The flexibility of the model allows for complex workflows, including streaming computations with nested iterations, which are hard or even impossible to express in other systems. The preceding systems rely on direct and ephemeral channels (typically, TCP connections) to implement the data bus. Kafka Streams [18] and Samza [74], instead, build a dataflow processing layer on top of Kafka [55] durable channels. In systems that adopt job-level deployment, job compilation and deployment only depend on static information about the computing infrastructure. For instance, the number of tasks for data-parallel operations only depends on the total number of slots made available in workers. As a result, this class of systems does not support sharing resources with other applications: all resources need to be acquired at job compilation, which prevents scheduling decisions across applications at runtime.

We observed three approaches to implement fault tolerance and delivery guarantees. First, systems such as Flink and MillWheel [7] periodically take a consistent snapshot of the state. The command to initiate a snapshot starts from sources and completes when it reaches the sinks. In the case of failure, the last completed snapshot is restored and sources replay data that was produced after that snapshot, in the original order. If sinks can detect and discard duplicate results, this approach guarantees exactly once delivery. Second, Storm acknowledges each data element delivered between two tasks: developers decide whether to use acknowledgements (and retransmit data if an acknowledgement is lost), providing at least once delivery, or not, providing at most once delivery. Third, Kafka Streams relies on the persistency of the data bus (Kafka): it stores the task state in special Kafka topics and relies on two-phase commit to ensure that upon activation a task consumes its input, updates its state, and produces results for downstream tasks atomically. In the case of failure, a task can resume from the input elements that were not successfully processed, providing exactly once delivery (unless data elements are received out of order, in which case it retracts and updates previous results leading to at least once delivery). These three mechanisms are also used for dynamic reconfiguration, as they allow a system to resume processing after a new deployment.

5.4 Graph Processing

Early dataflow systems were not well suited for iterative computations, which are common in graph processing algorithms. To overcome this limitation, an alternative computational model was developed for graph processing, known as vertex-centric [68]. In this model, pioneered by the Google Pregel system [66], jobs are iterative: developers provide a single function that encodes the behavior of each vertex v at each iteration. The function takes in input the current (local) state of v and the set of messages produced for v during the previous iteration; it outputs the new state of v and a set of messages to be delivered to connected vertices, which will be evaluated during the next iteration. The job terminates when vertices do not produce any message at a given iteration. Vertices are partitioned across workers and each task is responsible for a given partition. Jobs are continuous, as tasks are activated multiple times (once for each iteration) and store the vertex state across activations (in their task state). Tasks only communicate by exchanging data (messages

between vertices) over the data bus, which is implemented as direct channels. One worker acts as a leader and is responsible for coordinating the iterations within the job and for detecting possible failures of other workers. Workers persist their state (task state and input messages) at each iteration: in the case of a failure, the computation restarts from the last completed iteration. Several systems inherit and improve the original Pregel model in various ways: (i) by using a persistent data bus, where vertices can pull data when executed, to reduce the overhead for broadcasting state updates to many neighbor vertices [62]; (ii) by decoupling communication and processing in each superstep, to combine messagess and reduce the communication costs [45]; (iii) by allowing asynchronous execution of supersteps, to reduce synchronization overhead and inactive time [62]; (iv) by optimizing the allocation of vertices to tasks based on topological information, to reduce the communication overhead; (v) by dynamically migrating vertices between tasks (dynamic reconfiguration) across iterations, to keep the load balanced or to place frequently communicating vertices on the same worker [30]; and (vi) by offering sub-graph centric abstractions, suitable to express graph mining problems that aim to find sub-graphs with given characteristics [91]. For the sake of space, we do not discuss all systems derived from Pregel here, but the interested reader can refer to the detailed survey by McCune et al [68].

6 OTHER SYSTEMS

This section includes all systems that do not clearly fall in either of the two classes identified previously. Due to their heterogeneity, we do not provide a common overview, but we organize and discuss them within three main classes: (i) systems that support analytical jobs on top of shared state abstractions, (ii) systems that propose new programming models, and (iii) systems that integrate concepts from both DMSs and DPSs in an attempt to provide a unifying solution.

6.1 Computations on DMSs

DMSs are designed to execute lightweight jobs that read and modify a shared state. We identified a few systems that support some form of heavy-weight job.

6.1.1 Incremental Computations. Percolator [76] builds on top of the BigTable column store and incrementally updates its shared state. It adopts observer processes that periodically scan the shared state: when they detect changes, they start a computation that may update other tables with its results. In Percolator, computations are broken down into a set of small updates to the current shared state. This differentiates it from DPSs, which are not designed to be incremental. For instance, Percolator can incrementally update Web search indexes as new information about Web pages and links become available. Percolator jobs may involve multiple shared-state elements, and the system ensures group atomicity using using two-phase commit and group isolation using a timestamp-based protocol.

6.1.2 Long-Running Jobs. F1 [83] implements a SQL query executor on top of Spanner. It supports long-running jobs, which are compiled to execution plans where the tasks (or at least part of them) are organized into a dataflow to enable distributed execution as in DPSs. F1 also introduces optimistic transactions (jobs), which consist of a read phase to retrieve all the data needed for the computation and a write phase to store the results. The read phase does not block other concurrent jobs, so they can run for long time (as in the case of analytical jobs). The write phase completes only if no conflicting updates from other jobs occurred during the read phase.

6.1.3 Graph Processing. In graph data stores, long-running jobs appear as computations that traverse multiple hops of the graph (e.g., jobs that search for paths or patterns in the graph) or as iterative analytical jobs (e.g., vertex-centric computations). Trinity [80] inherits the same model of

NoSQL graph stores such as TAO but implements features designed specifically to support long-running jobs. It lets users define the communication protocols that govern the exchange of data over the data bus, to optimize them for each specific job. For instance, data may be buffered and aggregated at the sender or at the receiver. It checkpoints the intermediate state of a long-running jobs to resume it in the case of failure.

6.2 New Programming Models

6.2.1 Stateful Dataflow. The absence of shared mutable state in the dataflow model forces developers to encode all information as data that flows between tasks. However, some algorithms would benefit from the availability of state that can be modified in-place, such as machine learning algorithms that iteratively refine a set of parameters. Thus, several systems propose extensions to the dataflow programming model that accommodate shared mutable state. In **Stateful Dataflow Graphs (SDGs)** [43], developers write a driver program using imperative (Java) code that includes state and methods to access and modify it. Code annotations are used to specify state access patterns within methods. The resulting jobs are compiled into a dataflow graph where operators access the shared state. If possible, state elements are partitioned across workers, and otherwise they are replicated in each worker and the programming model supports user-defined functions to merge changes applied to different replicas. Deployment and execution rely on a DPS with job-level deployment [26].

Tangram [50] implements task-based deployment and allows tasks to access and update an in-memory key-value store as part of their execution. By analyzing the execution plan, Tangram can understand which parts of the computation depend on mutable state and which parts do not, and optimizes fault tolerance for the job at hand.

TensorFlow [1] is a library to define machine learning models. Jobs represent models with transformations (tasks) and variables (shared state elements). As strong consistency is not required for the application scenario, tasks can execute and update variables asynchronously, with only barrier synchronization at each step of an iterative algorithm. TensorFlow was conceived for distributed execution, whereas other machine learning libraries such as PyTorch were initially designed for single-machine execution and later implemented distributed training using the same approach as TensorFlow.

6.2.2 Relational Actors. ReactDB [79] extends the actor-based programming model with data management concepts such as relational tables, declarative queries, and transactional semantics. It builds on logical actors that embed state in the form of relational tables. Actors can query their internal state using a declarative language and asynchronously send messages to other actors. ReactDB lets developers explicitly control how the shared state is partitioned across actors. Jobs are submitted to a coordinator actor that governs their execution. The system guarantees transactional semantics for the entire duration of the job, across all actors that are directly or indirectly invoked by the coordinator.

6.3 Hybrid Systems

Several works aim to integrate data management and processing within a unified solution. S-Store [27] integrates stream processing capabilities within a transactional database. It uses an in-memory store to implement the shared state (visible to all tasks), the task state (visible only to individual tasks of stream processing jobs), and the data bus (where data flowing from task to task of stream processing jobs is temporarily stored). S-Store uses the same concepts as VoltDB [89] to offer transactional guarantees with low overhead. Data management and stream processing tasks are scheduled on the same engine in an order that preserves transactional semantics and

is consistent with the dataflow. S-Store unifies input data (for streaming jobs) and invocations (of data management jobs, in the form of stored procedures): this is in line with the conceptual view we provide in Section 2.

SnappyData [70] has a similar goal to S-Store but a different programming and execution model. It builds on Spark and Spark Streaming, and augments them with the ability to access a key-value store (shared state) during their execution. In the attempt to efficiently support heterogeneous types of jobs, SnappyData lets developers select how to organize the shared state, such as in terms or format (row oriented or column oriented), partitioning, and replication. It supports group atomicity and isolation using two-phase commit and multi-version concurrency control, and integrates fault detection and recovery mechanisms for Spark tasks and their effects on the shared state.

StreamDB [31] and TSpoon [5] take the opposite approach with respect to S-Store by integrating data management capabilities within a stream processor. StreamDB models database queries as stream processing jobs that receive updates from external sources and output new results to sinks. Stream processing tasks can read and modify portions of a shared state: all database queries that need to access a given portion will include the task responsible for that portion. StreamDB ensures group atomicity and isolation without explicit locks: invocation of jobs are timestamped when first received by the system, and each worker executes tasks from different jobs in timestamp order. TSpoon does not provide a shared state but enriches the dataflow model with (i) the ability to read (query) task state on demand and (ii) transactional guarantees in the access to task state. Developers can identify portions of the dataflow graph (denoted as transactional sub-graphs) that need to be read and modified with group atomicity and isolation. TSpoon implements atomicity and isolation by decorating the dataflow graph with additional operators that act as transaction managers. It supports different levels of isolation (from read committed to serializable) with different tradeoffs between guarantees and overhead.

Hologres [53] is used within Alibaba to execute both analytical jobs and interactive jobs. The system is designed to support high volume ingestion data from external sources and continuous jobs that derive information to be stored in the shared state or to be presented to external sinks. The shared state is partitioned across workers. A worker stores an in-memory representation of the partition it is responsible for and delegates durability to an external storage service. The distinctive features of the system are (i) a structured data model where state is represented as tables that can be physically stored row-wise or column-wise depending on the access pattern, and (ii) a scheduling mechanism where tasks are deployed and executed onto workers based on load balancing and prioritization of jobs that require low latency.

7 DISCUSSION

In building and discussing our model and taxonomy, we derived several observations. We report them in this section, pointing out ideas for future research.

State and Data Management. The dichotomy between DMSs and DPSs is frequently adopted in the literature but not defined in precise terms. Our model makes the characteristics that contribute to this dichotomy clear and explicit, introducing the state component and a sharp distinction between shared and task state. DMSs offer primitives to read and modify a mutable shared state, whereas DPSs target computationally expensive data transformations and do not support state at all, or support it only within individual tasks. This distinction brings together other differences (which we made explicit with the classification criteria in Table 4) such that the two classes of systems complement each other and are often used in combination to support heterogeneous workloads.

This complementarity pushed researchers to extend their DMSs or DPSs to break the dichotomy, adding features typical of the other class. For instance, some DMSs, such as AsterixDB, support long-lasting queries using the dataflow processing model typical of DPSs, whereas recent versions

of stream DPSs, such as Flink and Kafka, started to offer primitives to access their task state with read-only queries (one-shot jobs). This triggered interesting research on declarative APIs that integrate streaming data and state changes into a unifying abstraction [78]. A few systems, such as S-Store and TSpoon, pushed this effort even further, integrating transactional semantics within stream DPSs.

Future research efforts could continue to explore approaches that extend the capabilities of individual systems, with the goal of better supporting hybrid workloads that demand both state management and data processing capabilities, reducing the need to deploy many different systems, thus simplifying the overall architecture of data-intensive applications.

Coordination Avoidance. In distributed scenarios, the coordination between workers may easily become a bottleneck. Avoiding or reducing coordination is a recurring principle we observed in all data-intensive systems. Most DPSs circumvent this problem by forcing developers to think in terms of functional and data-parallel transformations. As state is absent or local to tasks, tasks may freely proceed in parallel. Coordination, if present, is limited to barrier synchronization in systems that support iterative jobs (e.g., iterative dataflow systems and graph processing systems).

Conversely, DMSs require coordination to control concurrent access to shared state from multiple jobs. Indeed, the approach to coordination is the main criterion we used to classify them in Section 4. NoSQL systems partition state by key: they either only support jobs that operate on individual keys or relinquish group guarantees for jobs that span multiple keys, effectively treating accesses to different keys as if they came from independent jobs that do not coordinate with each other. NewSQL systems do not entirely avoid coordination but try to limit the situations in which it is required or its cost. In our analysis, we identified four main approaches to reach this goal: (i) use of precise clocks [34], (ii) pre-ordering of jobs and deterministic execution [92], (iii) explicit partitioning strategies to maximize jobs executed (sequentially) in a single slot [89], and (iv) primary-based protocols that delegate the scheduling of all read-write jobs to a single worker [95]. In addition, all DMSs adopt strategies that optimize the execution of read-only jobs and minimize their impact on read-write jobs. They include the use of replicas to serve read-only jobs and multi-version concurrency control to let read-only jobs access a consistent view of the state without conflicting with read-write jobs.

An open area of investigation for future research is a detailed study of the assumptions and performance implications of coordination avoidance strategies under different workloads. This study could guide the selection of the best strategies for the scenario at hand and open the room for dynamic adaptation mechanisms.

Architectures for Data-Intensive Applications. Data-intensive applications typically rely on complex software architectures that integrate different data-intensive systems to harness their complementary capabilities [37]. For instance, many scenarios require integrating OLTP (online transaction processing) workloads, which consist of read-write jobs that mutate the state of an application (e.g., user requests in an e-commerce portal), and OLAP (online analytical processing) workloads, which consist of read-only analytic jobs (e.g., analysis of sales segmented by time, product, and region). To support these scenarios, software architectures typically delegate OLTP jobs to DMSs that efficiently support concurrent read-only and read-write queries (e.g., relational databases), and use DMSs optimized for read-only queries (e.g., wide-column stores) for OLAP jobs. The process of extracting data from OLTP systems and loading it into OLAP systems is denoted as ETL (extract, transform, load), and is handled by DPSs that pre-compute and materialize views to speedup read queries in OLAP systems (e.g., by executing expensive grouping, joins, and aggregates, as well as building secondary indexes). Traditionally, ETL was executed periodically by batch DPSs, with the downside that analytical jobs do not always access the latest available

data, whereas recent architectural patterns (e.g., lambda and kappa architectures [60]) advocate the use of stream DPSs for this task.

In general, the architectural patterns of data-intensive applications are in continuous evolution [37], and our study highlights a vast choice of diverse data-intensive systems, with partially overlapping features. Future research could build on our model and classification to simplify the design of applications. Indeed, although the primary goal of our model was to present the key characteristics of data-intensive systems to researchers and practitioners with diverse backgrounds, it may inspire high-level modeling frameworks to capture the requirements of data-intensive applications and guide the design of a suitable software architecture for the specific scenario at hand. Recent work already explored similar model-driven development in the context of stream processing applications [47].

Modular Implementations. Several data-intensive systems have a modular design, where the functionalities of the system are implemented by distinct components that can be developed and deployed independently. This approach is well suited for cloud environments where individual components are offered as services and can be scaled independently depending on the workload. In addition, the same service can be used in multiple products—for example, storage services, log services, lock services, and key-value stores may be used as stand-alone products or adopted as building block of a relational DMS. We observed this strategy in systems developed at Google [29, 34], Microsoft [11], and Amazon [95].

Future research could bring this idea forward, proposing more general component models that promote reusability and adaptation to heterogeneous scenarios. Our work may guide the identification of the abstract components that build data-intensive systems, the interfaces they offer, the assumptions they rely on, and the functionalities they provide. These research efforts may complement the aforementioned study of architectural patterns, promoting the definition of complex architectures from pre-defined components.

Wide Area Deployment. The systems we analyzed are primarily designed for cluster deployment. In DPSs, tasks exchange large volumes of data over the data bus and the limited bandwidth of wide-area deployment may easily become a bottleneck. Some DMSs support wide-area deployment through replication, but in doing so they either drop consistency guarantees or implement mechanisms to reduce the cost for updating remote replicas. For instance, deterministic databases [92] define an order for jobs and force all replicas to follow this order, with no need to explicitly synchronize job execution.

However, increasingly many applications work at a geographical scale and the edge computing paradigm [82] is emerging to exploit processing and storage resources at the edge of the network, close to the end users. Designing data-intensive systems that embrace this paradigm is an important topic of investigation.

New and Specialized Hardware. The use of specialized hardware to improve the performance of data-intensive systems is an active area of research. Recent works study hardware acceleration for DPSs [49] and DMSs [42, 58] using GPUs or FPGAs. Offloading of tasks to GPUs is also supported in recent versions of DPSs, such as Spark, and is a key feature for systems that target machine learning problems, such as TensorFlow [1].

Open research problems in the area include devising suitable programming abstractions to simplify the deployment of tasks onto hardware accelerators, building new libraries of tasks that may run onto hardware accelerators, and exploring new types of accelerators. More in general, the availability of new hardware solutions stimulates the definition of design choices that better exploit the characteristics of those solutions. In the context of DMSs, non-volatile memory offers durability at nearly the same performance as main memory. The interested reader can refer to

the work by Arulraj and Pavlo [13] that discusses the use of non-volatile memory to implement a database system. As pointed out in recent studies, another area of investigation consists of using remote memory access to better exploit data locality and reduce data access latency. The potential of remote memory access has been pointed out in recent studies both in the domain of DMSs [102] and in the domain of DPSs [40].

Dynamic Adaptation. Our model captures the ability of some systems to adapt to mutating workload conditions (see Table 8). Many works use this feature to implement automated control systems for DPSs that monitor the use of resources and adapt the deployment to meet the quality of service specified by the users while using the minimum amount of resources. The interested reader can refer to recent work on dynamic adaptation for batch [17] and stream [25] DPSs.

Future studies in the area of dynamic adaptation could intersect with topics already presented in this section: in particular, they may consider the availability of geographically distributed processing, memory, and storage resources, as well as heterogeneous and specialized hardware platforms.

8 CONCLUSION

This article presented a unifying model for distributed data-intensive systems, which defines a system in terms of abstract components that cooperate to offer the system functionalities. The model precisely captures the possible design and implementation strategies for each component, with the assumptions they rely on and the guarantees they provide. From the model, we derived a list of classification criteria that we use to organize state-of-the-art systems into a taxonomy and survey them, highlighting their commonalities and distinctive features. Our work can be useful not only for engineers who need to deeply understand the range of possibilities to select the best systems for their application but also to researchers and practitioners who work on data-intensive systems, to acquire a wide yet precise view of the field.

APPENDICES

A SUMMARY OF TERMS AND CONCEPTUAL MAP

This section presents additional resources to help the reader navigate through the concepts discussed in the article.

Figure 4 organizes the entities introduced in our model into a map, which highlights their relations. The map adopts a UML-like notation, where entities are characterized by a (possibly empty) list of attributes and may be connected to each other through different types of relations: (i) *specialization* (empty arrow), when an entity is a specialization of a more general entity; (ii) *aggregation* (empty diamond), when an entity is included into a more comprehensive entity; (iii) *composition* (filled diamond), when an entity is constituting part of another entity; and (iv) *dependency* (dashed arrow), when an entity depends or uses another entity, in which case we also denote the type of use as a label of the arrow.

Table 11 complements this map by reporting the definition of all the terms we introduced in our unifying model for data-intensive systems (Section 2) and used throughput the article to classify and describe individual systems.

Table 11. Summary of Terms Introduced in the Unifying Model for Data-Intensive Systems (Section 2)

Term	Definition
Client	Software component that exploits the functionalities offered by a data-intensive system by registering and starting driver programs.
Driver program	Part of the application logic that interacts with the data-intensive system and exploits its functionalities by invoking one or more jobs.
Node	Physical or virtual machine.
Worker	Process running on a node.
Slot	Processing resource unit offered by a worker.
Distributed computing infrastructure	Set of nodes on top of which a distributed data-intensive system runs.
Job	Largest unit of execution that can be offloaded onto the distributed computing infrastructure.
One-shot job	Job that is executed once and terminates. Invoking the same jobs multiple times leads to separate executions of the same code.
Continuous job	Job that persists across invocations (in this case, we call them *activations* of the same job). It may persist in some state across activations.
Task	Elementary unit of execution. Tasks derive from the compilation of a job. They are executed sequentially on a slot.
Data (elements)	Immutable units of information. Delivered through the data bus.
State (elements)	Mutable units of information. Split into state portions. Stored within workers.
Task state	State that is private to / accessible from a single task.
Shared state	State that can be accessed simultaneously from multiple tasks, belonging to the same or different jobs.
Data bus	Communication channel that distributed data elements and jobs invocations.
Source	Component that provides input data for the data-intensive system.
Passive source	Source that provides a static input dataset.
Active source	Source that provides a dynamic input dataset—that is, continuously produces new data.
Sink	Component that consumes output data from the data-intensive system.
Execution plan	A workflow of tasks: it is the result of the compilation of a job.
Data-parallel API	API where a computation is defined for a single data element but executed in parallel on multiple elements.
Placement-aware API	API where developers can control or influence the placement of tasks onto slots.
Resources information	Information about the resources of the distributed computing infrastructure and their use.
Static resources information	Resources information that only considers the resources available in the distributed computing infrastructure.
Dynamic resources information	Resources information that considers the use of the resources available in the distributed computing infrastructure.
Group atomicity	Property of a group of tasks: ensures that either all of the tasks complete successfully or none of them (and none of their effects become visible).
Group isolation	Property that constrains how tasks belonging to different groups can interfere with each other.
Delivery guarantees	Define how external components (driver programs, sources, sinks) observe the effects of a single input data element (or invocation).
Order guarantees	Define the order in which external components (driver programs, sources, sinks) observe the effects of multiple data element (or invocation).
Event time	Timestamp associated with a data element by the original source.
Ingestion time	Timestamp associated with data elements when they first enter the data-intensive system.
Watermark	Special input element containing a timestamp t. It is delivered by input components (e.g., sources) and indicates that the components will not produce any further data element with a timestamp lower than t.
Checkpointing	Process that stores a copy of state on durable storage.
Logging	Process that stores individual operations or state changes on durable storage.
Dynamic reconfiguration	Ability of a system to modify the deployment and execution of jobs at runtime.

B DATA MANAGEMENT SYSTEMS

This section details individual Data Management Systems (DMSs). Tables 12, 13, 14, 15, 16, 17, 18, 19 summarize the characteristics of these systems with respect to the classification criteria presented in Section 2.

B.1 NoSQL Systems

B.1.1 Key-Value Stores Dynamo. Dynamo [39] is a NoSQL key-value store used by Amazon to save the state of its services. State elements are arbitrary values (typically, binary objects) identified by a unique key. Jobs consist of operations on individual state elements: retrieve the value associated with a key (get) or insert/update a value with a given key (put). Dynamo builds a distributed hash table: workers have an associated unique identifier, which determines the portion of shared state (set of keys) they are responsible for. Shared state is replicated, so multiple workers are responsible for the same key. When a worker receives a client request for a key, it forwards the request to one of the workers responsible for that key. Clients may also be aware of the distribution of shared state portions across workers and use this information to better route their requests. Dynamo uses a quorum-based approach where read and write operations on a key need to be processed by a quorum of the replicas responsible for that key. Users can set the number of replicas for each key, the read quorum, and the write quorum to trade consistency and durability for performance and availability. After a write quorum is reached, updates are asynchronously propagated to remaining replicas. In the case of transient unavailability of one replica, another

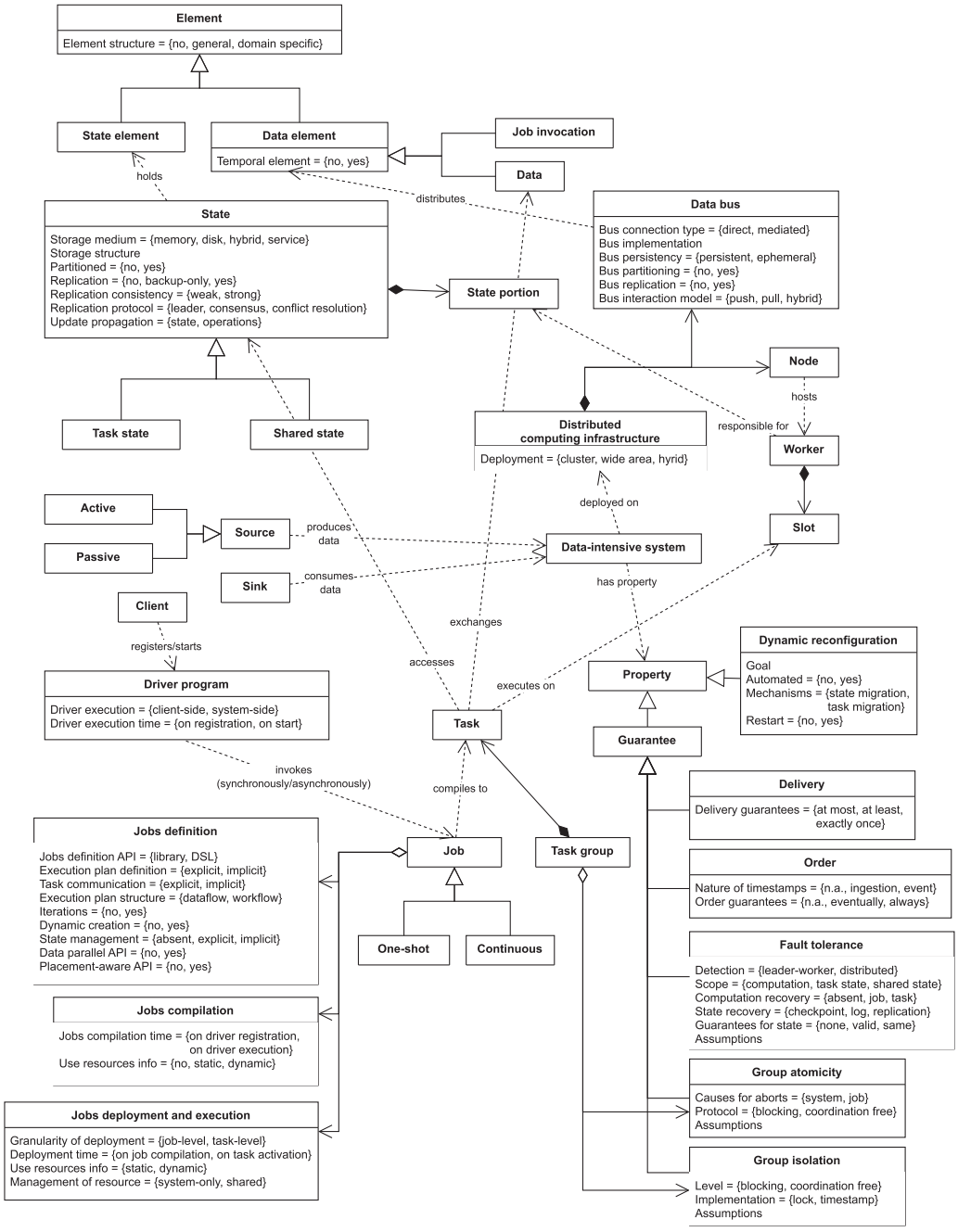

Fig. 4. Conceptual map of all the concepts introduced in the unifying model for data-intensive systems (Section 2) and their relations.

Table 12. Data Management Systems: Functional Model

	Driver Exec	Driver Exec Time	Invoc of Jobs	Sources	Sinks	State	Deployment
NoSQL Systems							
Dynamo	client	reg	sync	no	no	yes	wide
DynamoDB	client	reg	sync	no	yes	yes	wide
Redis	client+sys^{SP}	reg+start^{SP}	sync	no	yes	yes	hybrid
BigTable	client+sys^{SP}	reg+start^{SP}	sync	no	no	yes	hybrid
Cassandra	client	reg	sync+async	no	yes	yes	wide
MongoDB	client	reg	sync	active	yes	yes	cluster
CouchDB	client	reg	sync	no	yes	yes	hybrid
AsterixDB	client	reg	sync	both	no	yes	cluster
InfluxDB	client+sys^{SP}	reg+start^{SP}	sync+async	active	yes	yes	cluster
Gorilla	client	reg	unknown	active	no	yes	cluster
Monarch	client+sys^{SP}	reg+start^{SP}	unknown	active	yes	yes	wide
Peregreen	client	reg	unknown	no	no	yes	cluster
TAO	client	reg	unknown	no	no	yes	wide
Unicorn	client+sys^{SP}	reg	sync	no	no	yes	cluster
NewSQL Systems							
Deuteronomy	client	reg	sync	no	no	yes	cluster+wide
FoundationDB	client	reg	sync	no	no	yes	cluster+wide
SolarDB	client	reg	sync	no	no	yes	cluster+wide
Spanner	client	reg	sync+async	no	no	yes	cluster+wide
CockroachDB	client	reg	sync	no	no	yes	wide
Calvin	client	reg	implem dep	no	no	yes	cluster+wide
VoltDB	sys^{SP}+client	start^{SP}(+reg)	sync+async	no	yes	yes	hybrid
Aurora	client	reg	sync	no	no	yes	cluster+wide
Socrates	client	reg	sync	no	no	yes	cluster
Tango	client	reg	sync	no	no	yes	cluster
A1	client	reg	sync	no	no	yes	cluster

^{SP}, stored procedures.

node can temporarily store writes on its behalf, which avoids blocking write operations while preserving the desired degree of replication for durability. When Dynamo is configured to trade consistency for availability, replicas may diverge due to concurrent writes: to resolve conflicts, Dynamo adopts a versioning system, where multiple versions of a given key may exist at different replicas. Upon write, clients specify which version they want to overwrite: Dynamo uses this information to trace causality between updates and automatically resolves conflicts when it can determine a unique causal order of updates. In the case of concurrent updates, Dynamo preserves conflicting versions and presents them to clients (upon a read) for semantic reconciliation. Dynamic reconfiguration is a key feature of Dynamo. Nodes can be added and removed dynamically, and shared state portions automatically migrate. Dynamo adopts a distributed (gossip based) failure detection model, whereas failure recovery takes place by simply redistributing state portions over remaining nodes.

DynamoDB. DynamoDB[2] is an evolution of Dynamo that aims to simplify operational concerns: it is fully managed and offered as a service. It exposes the same data model as Dynamo but uses a single-leader replication protocol, where all writes for a key are handled by the leader for that key, which propagates them synchronously to one replica (for durability) and asynchronously to another replica. Read operations can be either strongly or weakly consistent: the former are always processed by the leader, whereas the latter may be processed by any replica, even if the replica lags behind of updates. Replicas can be located in different data centers to support wide-area deployments. DynamoDB stores data on disk using B-trees and buffers incoming write requests in a WAL. It supports secondary indexes that are asynchronously updated from the log upon write. Recent versions of DynamoDB enable automated propagation of changes to external sinks. They also offer an API to group individual operations in transactions that provide group atomicity and configurable group isolation. Transactions can be configured to be idempotent, thus offering exactly once delivery. To detect failures, the leader of each partition periodically sends heartbeats

[2]http://aws.amazon.com/dynamodb/.

Table 13. Data Management Systems: Jobs Definition

	Jobs Def API	Exec Plan Def	Task Comm	Exec Plan Struct	Iter	Dyn Creat	Nature of Jobs	State Man	Data Par API	Placem -Aware API
NoSQL systems										
Dynamo	crud	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
DynamoDB	crud+scan	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
Redis	crud+scan +ops on vals	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
BigTable	crud+scan	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
Cassandra	crud+scan +ops on vals	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
MongoDB	crud+scan +ops on vals+aggs	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
CouchDB	crud+scan +ops on vals	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
AsterixDB	SQL-like	impl	impl	dataflow	no	no	one-shot	expl	no	no
InfluxDB	crud+scan +aggs	impl	impl	workflow	no	no	one-shot	expl	no	no
Gorilla	crud+scan	expl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
Monarch	SQL-like	impl	impl	workflow (tree)	no	no	one-shot	expl	no	no
Peregreen	crud	impl	impl	workflow (tree)	no	no	one-shot	expl	no	no
TAO	crud+relations	impl	impl	task (put / get)	no	no	one-shot	expl	no	no
Unicorn	search	impl	impl	hierar workflow	yes	yes	one-shot	expl	no	no
NewSQL systems										
Deuteronomy	crud+scan	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
FoundationDB	crud+scan	impl	impl	1 task (put / get)	no	no	one-shot	expl	no	no
SolarDB	crud+scan	impl	impl	1 task (put / get)	no	no / yes	one-shot	expl	no	no
Spanner	declar DSL	impl	impl	workflow	no	no	one-shot	expl	no	no
CockroachDB	SQL	impl	impl	dataflows+coord	no	no	one-shot	expl	no	yes
Calvin	API agnostic	impl	impl	workflow	no	no	one-shot	expl	no	no
VoltDB	Java + SQL	impl	impl	workflow	no	no	one-shot	expl	no	yes
Aurora	SQL	impl	impl	workflow	no	no	one-shot	expl	no	no
Socrates	SQL	impl	impl	workflow	no	no	one-shot	expl	no	no
Tango	library	impl	impl	workflow	no	no	one-shot	expl	no	no
A1	library	expl	expl	graph	yes	no	one-shot	expl	yes	no

Table 14. Data Management Systems: Jobs Compilation and Execution

	Jobs Comp Time	Use Resources Info (Comp)	Granul of Depl	Depl Time	Use Resources Info (Depl)	Manag of Res
NoSQL Systems						
Dynamo	exec	static	job	job comp	static	sys-only
DynamoDB	exec	static	job	job comp	static	sys-only
Redis	exec	static	job	job comp	static	sys-only
BigTable	exec	static	job	job comp	static	shared
Cassandra	exec	static	job	job comp	static	sys-only
MongoDB	exec	static	job	job comp	static	sys-only
CouchDB	exec	static	job	job comp	static	sys-only
AsterixDB	exec	static	task	task activ	static	sys-only
InfluxDB	reg	static	job	job comp	static	sys-only
Gorilla	exec	static	job	job comp	static	sys-only
Monarch	reg	static	job	job comp	static	sys-only
Peregreen	exec	static	job	job comp	static	sys-only
TAO	exec	static	job	job comp	static	sys-only
Unicorn	exec	static	job	job comp	static	sys-only
NewSQL Systems						
Deuteronomy	exec	static	job	job comp	static	sys-only
FoundationDB	exec	static	job	job comp	static	sys-only
SolarDB	exec	static	job	job comp	static	sys-only
Spanner	exec	static	job	job comp	dynamic	shared
CockroachDB	exec	static	job	job comp	dynamic	sys-only
Calvin	exec	static	job	job comp	static	sys-only
VoltDB	reg	static	job	job comp	static	sys-only
Aurora	exec	static	job	job comp	static	sys-only
Socrates	exec	static	job	job comp	static	sys-only
Tango	exec	static	job	job comp	static	sys-only
A1	exec	static	job	job comp	static	sys-only

to all replicas. After some heartbeats are lost, the remaining nodes use the Paxos consensus algorithm to elect a new leader. As an additional fault tolerance mechanism, B-trees and logs are periodically checkpointed to durable storage. Amazon offers automatic scaling of DynamoDB as a service: users can select the expected read and write throughput for a given table (blocks of

Table 15. Data Management Systems: Data Management

	Elem Struct	Temp Elem	Bus Conn	Bus Impl	Bus Persist	Bus Partition	Bus Repl	Bus Inter
NoSQL Systems								
Dynamo	key-value	no	direct	net chan	ephemeral	yes	no	pull
DynamoDB	key-value	no	direct	net chan	ephemeral	yes	no	pull
Redis	key-value	no	direct	net chan	ephemeral	yes	no	pull
BigTable	wide-col	yes	direct	net chan	ephemeral	yes	no	pull
Cassandra	wide-col	no	direct	net chan	ephemeral	yes	no	pull
MongoDB	document	no	direct	net chan	ephemeral	yes	no	pull
CouchDB	document	no	direct	net chan	ephemeral	yes	no	pull
AsterixDB	document	no	direct	net chan	ephemeral	yes	no	pull
InfluxDB	time series	yes	direct	net chan	ephemeral	yes	no	pull
Gorilla	time series	yes	direct	net chan	ephemeral	yes	no	pull
Monarch	time series	yes	direct	net chan	ephemeral	yes	no	pull
Peregreen	time series	yes	direct	net chan	ephemeral	yes	no	pull
TAO	typed graph	yes	direct	net chan	ephemeral	yes	no	pull
Unicorn	typed graph	yes	direct	net chan	ephemeral	yes	no	pull
NewSQL Systems								
Deuteronomy	key-value	no	direct	net chan	ephemeral	yes	no	pull
FoundationDB	key-value	no	direct	net chan	ephemeral	yes	no	pull
SolarBD	key-value	no	direct	net chan	ephemeral	yes	no	pull
Spanner	semi-relational	no	direct	net chan	ephemeral	yes	no	pull
CockroachDB	relational	no	direct	net chan	ephemeral	yes	no	pull
Calvin	structure agnostic	no	direct	net chan	ephemeral	yes	no	push
VoltDB	relational	no	direct	net chan	ephemeral	yes	no	pull
Aurora	relational	no	direct	net chan	ephemeral	yes	no	pull
Socrates	relational	no	direct	net chan	ephemeral	yes	no	pull
Tango	object	no	direct	net chan	ephemeral	yes	no	pull
A1	typed graph	no	direct	remote mem	ephemeral	yes	no	pull

Table 16. Data Management Systems: State Management

	Elem Struct	Stor Medium	Stor Struct	Task St	Shared St	St Part	Repl	Repl Consist	Repl Prot	Update Propag
NoSQL systems										
Dynamo	key-value	agnostic	agnostic	no	yes	yes	yes	conf	quorum +confl	op
DynamoDB	key-value	disk	B-trees	no	yes	yes	yes	conf	leader	unknown
Redis	key-value	mem	user-def	no	yes	yes	yes	weak	leader (clus) confl (wide)	op
BigTable	wide-col	hybrid	LSM trees	no	yes	yes	backup (clus) yes (wide)	weak	confl	op
Cassandra	wide-col	hybrid	LSM trees	no	yes	yes	yes	conf	quorum +confl	op
MongoDB	document	agnostic	agnostic	no	yes	yes	yes	conf	leader	op
CouchDB	document	disk	B-trees	no	yes	yes	yes	conf	quorum +confl	op
AsterixDB	document	hybrid	LSM trees	no	yes	yes	no	n.a.	n.a.	n.a.
InfluxDB	time series	hybrid	LSM trees	no	yes	yes	yes	weak	leader	state
Gorilla	time series	mem	TSMap	no	yes	yes	yes	weak	leader	state
Monarch	time series	mem	user-def	no	yes	yes	yes	weak	leader	state
Peregreen	time series	agnostic	agnostic	no	yes	yes	yes	strong	leader	unknown
TAO	graph	serv	MySQL+ memcache	no	yes	yes	yes	weak	leader	op
Unicorn	graph	mem	indexes	no	yes	yes	backup	n.a.	n.a.	n.a.
NewSQL Systems										
Deuteronomy	key-value	agnostic	agnostic	no	yes	yes	yes	strong	leader	n.a.
FoundationDB	key-value	disk	B-trees	no	yes	yes	yes	strong	leader	state
SolarDB	key-value	hybrid	LSM trees	no	yes	yes	backup	n.a.	leader	op
Spanner	relational	disk	B-trees	no	yes	yes	yes	strong	leader	op
CockroachDB	relational	hybrid	LSM trees	no	yes	yes	yes	strong	cons	op
Calvin	agnostic	agnostic	agnostic	no	yes	yes	yes	strong	leader or cons	op
VoltDB	relational	mem	B-trees	no	yes	yes	yes	strong (clus) weak (wide)	cons (clus) +confl (wide)	op
Aurora	relational	disk	log+ B-trees	no	yes	yes	yes	strong	leader	op
Socrates	relational	serv+disk	log	no	yes	yes	yes	strong	leader	op
Tango	object	serv	log	no	yes	yes	backup	n.a.	n.a.	n.a.
A1	graph	mem	user-def	no	yes	yes	backup	n.a.	n.a.	n.a.

Table 17. Data Management Systems: Group Atomicity, Group Isolation, Delivery, and Order

	Aborts	Protocol	Assumptions	Level	Impl	Assumptions	Delivery	Nature of Ts	Order
NoSQL Systems									
Dynamo	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	conf (most / exact)	no	n.a.
DynamoDB	sys	unknown	cluster deployment	conf	unknown	cluster depl	conf (most / exact)	no	n.a.
Redis	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most	no	n.a.
BigTable	n.a.	n.a.	n.a.	coord free	SEQ	1P	exact	event	n.a.
Cassandra	n.a.	n.a.	n.a.	coord free	SEQ	1P	conf (most / exact)	no	n.a.
MongoDB	sys+job	blocking	n.a.	blocking	ts	none	conf (most / exact)	no	n.a.
CouchDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	conf (most / exact)	no	n.a.
AsterixDB	n.a.	n.a.	n.a.	blocking	lock	1P	exact	no	n.a.
InfluxDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	event	n.a.
Gorilla	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most	event	n.a.
Monarch	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most	event	n.a.
Peregreen	n.a.	n.a.	n.a.	coord free	SEQ	1P	most	no	n.a.
TAO	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most	event	n.a.
Unicorn	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most	event	n.a.
NewSQL Systems									
Deuteronomy	sys+job	blocking	none	blocking	lock+ts	none	exact	no	n.a.
FoundationDB	sys+job	blocking	none	blocking	ts (OCC)	none	exact	no	n.a.
SolarDB	sys+job	blocking	none	blocking	ts (OCC)	none	exact	no	n.a.
Spanner	sys+job	blocking	none	blocking	lock+ts	none	exact	no	n.a.
CockroachDB	sys+job	blocking	none	blocking	lock+ts	none	exact	no	n.a.
Calvin	job	blocking	DC	blocking	ts	DC	exact	no	n.a.
VoltDB	sys+job	blocking	DC	blocking	ts	DC	exact	no	n.a.
Aurora	sys+job	coord free	1W	coord free	ts	1W	exact	no	n.a.
Socrates	sys+job	coord free	1W	coord free	ts	1W	exact	no	n.a.
Tango	sys+job	blocking	none	blocking	ts	none	exact	no	n.a.
A1	sys+job	blocking	none	blocking	ts	none	conf (most / exact)	no	n.a.

DC, jobs are deterministic; SEQ, jobs are executed sequentially, with no interleaving; 1W, a single worker handles all writes; 1P, jobs access a single state portion; OCC, optimistic concurrency control.

Table 18. Data Management Systems: Fault Tolerance

	Detection	Scope	Comput Recov	State Recov	Guarantees for State	Assumptions
NoSQL Systems						
Dynamo	p2p	shared st	n.a.	repl	none	none
DynamoDB	lead-work	shared st	n.a.	log+checkp+repl	same	REPL
Redis	p2p	shared st	n.a.	log+checkp+repl	conf (none or same)	REPL
BigTable	lead-work	shared st	n.a.	log+repl	same	STOR
Cassandra	p2p	shared st	n.a.	log+repl	none	none
MongoDB	lead-work	shared st	n.a.	log+repl	conf (none or same)	REPL
CouchDB	manual	shared st	n.a.	log+repl	none	none
AsterixDB	n.a.	shared st	n.a.	log+checkp	same	STOR
InfluxDB	lead-work	shared st	n.a.	log+repl	none	none
Gorilla	lead-work	shared st	n.a.	log+repl	none	none
Monarch	unknown	shared st	n.a.	log+repl	none	none
Peregreen	p2p	shared st	n.a.	repl	same	REPL
TAO	unknown	shared st	n.a.	repl	none	none
Unicorn	unknown	shared st	n.a.	repl	none	none
NewSQL Systems						
Deuteronomy	lead-work	shared st	n.a.	log+checkp+repl	same	STOR
FoundationDB	lead-work	shared st	n.a.	log+repl	same	STOR
SolarDB	lead-work	shared st	n.a.	log+checkp+repl	same	STOR
Spanner	lead-work	shared st	n.a.	log+checkp+repl	same	STOR
CockroachDB	lead-work	shared st	n.a.	log+repl	same	STOR
Calvin	n.a.	shared st	n.a.	log+checkp+repl	same	DC
VoltDB	p2p	shared st	n.a.	log+checkp+repl	same (cluster) / none (hybrid)	DC
Aurora	lead-work	shared st	n.a.	log+checkp	same	STOR
Socrates	lead-work	shared st	n.a.	log+checkp	same	STOR
Tango	lead-work	shared st	n.a.	log+repl	same	REPL
A1	lead-work	shared st	n.a.	repl	conf (none or same)	STOR

STOR, storage layer is durable; REPL, replicated data is durable.

key-value pairs), and DynamoDB automatically increases the amount of resources dedicated to that table to meet the requirements.

Redis. Redis [64] is a single-node in-memory key-value store. Since version 3.0, Redis Cluster provides a distributed implementation of the store. In terms of a data model, Redis differs from other key-value stores in that it provides typed values and optimized operations for those types. For instance, a value may be declared as a list, which supports appending new elements without overwriting the entire list. Redis provides a scripting language to express driver programs (stored pro-

Table 19. Data Management Systems: Dynamic Reconfiguration

	Goal	Automated	State Migration	Task Migration	Add / Del Slots	Restart
NoSQL systems						
Dynamo	load balan+elast	yes	yes	n.a.	yes	no
DynamoDB	load balan+elast	yes	yes	n.a.	yes	no
Redis	load balan	no	yes	n.a.	yes	no
BigTable	load balan	yes	yes	n.a.	yes	no
Cassandra	load balan	yes	yes	n.a.	yes	no
MongoDB	load balan	yes	yes	n.a.	yes	no
CouchDB	load balan	no	yes	n.a.	yes	no
AsterixDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
InfluxDB	load balan	no	yes	n.a.	yes	no
Gorilla	load balan	yes	yes	n.a.	yes	no
Monarch	load balan	yes	yes	n.a.	yes	no
Peregreen	load balan+elast	unknown	yes	n.a.	yes	no
TAO	unknown	unknown	unknown	n.a.	unknown	unknown
Unicorn	unknown	unknown	unknown	n.a.	unknown	no
NewSQL systems						
Deuteronomy	elast	no	yes	n.a.	yes	no
FoundationDB	elast	no	yes	n.a.	yes	no
SolarDB	elast	no	yes	n.a.	yes	no
Spanner	change schema+load balan	yes	yes	n.a.	yes	no
CockroachDB	elast	yes	yes	n.a.	yes	no
Calvin	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
VoltDB	n.a	n.a.	n.a.	n.a.	n.a.	n.a.
Aurora	elast	yes	yes	n.a.	yes	no
Socrates	elast	no	yes	n.a.	yes	no
Tango	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
A1	unknown	unknown	unknown	n.a.	unknown	unknown

cedures) that run system-side. Redis supports data dispatching to sinks in the form of a publish-subscribe service: clients can subscribe to a given key and be notified about the changes to that key. Users can configure their desired level of durability (for fault tolerance), with options ranging from no persistence to using periodic checkpointing to CL. Redis Cluster partitions data by key and uses single-leader asynchronous replication. Alternative solutions are available for wide-area deployments, where redirecting all writes to a single leader may be unfeasible. For instance, Redis CRDTs offer multi-leader replication with automated conflict resolution based on conflict-free replicated data types. Dynamic reconfiguration with migration of shared state portions is supported but not automated.

Other Key-Value Stores. Several other stores implement the key-value model with design and implementation strategies that are similar to those presented earlier. For the sake of space, we only discuss their key distinguishing factors. Voldemort[3] and Riak KV[4] follow the same design as Dynamo. Like Redis, Aerospike [84] adopts single-leader replication within a single data center and multi-leader replication for wide-area deployments. It focuses on both horizontal scalability (with workers on multiple nodes) and vertical scalability (with multiple workers on the same node). It adopts a hybrid storage model where key indexes are kept in memory but concrete values can be persisted on disk, and a threading model that reduces locking and contention by assigning independent shared state portions to threads. PNUTS [33] provides single-leader replication in wide-area deployments: it uses an intermediate router component to dispatch jobs invocations to responsible workers, and relies on an external messaging system to implement the data bus that propagates updates to replicas. The same service can also notify external sinks. Thanks to their simple interface, key-value stores are sometimes used as building blocks in distributed software architectures: Memcached is used as a memory-based distributed cache to reduce data access latency at Facebook [73]. RocksDB[5] is used to persist task state in the Flink stream processing system (also discussed later).

[3] https://www.project-voldemort.com/.
[4] https://riak.com/products/riak-kv/.
[5] https://rocksdb.org.

Other data stores offering a key-value model are presented in the following as part of NewSQL databases.

B.1.2 Wide-Column Stores BigTable. BigTable [29] and its open source implementation HBase[6] use a wide-column data model: shared state is organized in tables, where each row is associated with a fixed number of column families. A column family typically contains multiple columns that are frequently accessed together. Tables associate a value (binary object) to a row and a column (within a column family), and they are range partitioned across workers by row and physically stored (compressed) per column family. Jobs can read and update individual values and perform table scans. Rows are units of isolation: accesses to columns of the same row from different jobs are serialized, and this is the only task grouping guarantee that BigTable offers. BigTable adopts a leader-worker deployment, where a leader is responsible for assigning shared state portions to workers. Initially, each table is associated with a single worker, but it is automatically split when it increases in size. Clients retrieve and cache information about state distribution and submit their requests (tasks) involving a given state portion to the worker responsible for that state portion. BigTable can store multiple versions for each value. Versions are also visible to clients, which can retrieve old versions and control deletion policies. Writes always append new versions of a value, which improves write throughput to support frequent updates. Workers store recent versions in memory in LSM trees and use an external storage service (the GFS distributed filesystem) for durability. Background compaction procedures prune old versions from memory and from the storage. BigTable uses replication only for fault tolerance. Specifically, it relies on the replication of the GFS storage layer, where it also saves a command log. In the case of failure of a worker, the latest snapshot of its shared state portion is restored from GFS and from the commands in the log that were not part of the snapshot. BigTable supports wide-area deployments by fully replicating the data store in each data center. Replicas in other data centers may be only used for fault tolerance or they can serve client invocations, in which case they provide eventual consistency. Similar to key-value stores, BigTable provides dynamic reconfiguration by migrating state across available workers.

Cassandra. Cassandra [57] combines the wide-column data model of BigTable and the distributed architecture of Dynamo. Like BigTable, Cassandra uses LSM trees with versioning and background compaction tasks to improve the performance of write-intensive workloads. It offers a richer job definition language that includes pre-defined and user-defined types and operations. It supports task grouping only for compare-and-swap operations within a single partition. Like Dynamo, it uses a distributed hash table to associate keys to workers, and provides replication both in cluster and wide-area deployments, using a quorum-based approach for consistency. The quorum protocol can be configured to trade consistency and durability for performance, setting the number of local replicas (within a data center) and global replicas (in the case of wide-area deployments) that need to receive and approve a task before the system returns to the client. In the case of weak (eventual) consistency, Cassandra uses an anti-entropy protocol to periodically and asynchronously keep replicas up-to-date, using automated conflict resolution (last write wins).

B.1.3 Document Stores MongoDB. MongoDB [32] is representative of document stores, an extension of key-value stores where values are semi-structured documents, such as binary JSON in the case of MongoDB. MongoDB jobs can insert, update, and delete entire documents but also scan, retrieve, and update individual values within documents. Recent versions of MongoDB support simple data analytic jobs expressed as pipelines of data transformations. External systems can register as sinks to be notified about changes to documents. Shared state partitioning can be either

[6] https://hbase.apache.org.

hash-based or range-based. Clients are oblivious of the location of state portions and interact with the data store through a special worker component that acts as a router. Shared state portions can be replicated for fault tolerance only or also to serve read queries. Replication is implemented using a single-leader protocol with semi-synchronous propagation of changes, where clients can configure the number of replicas that need to synchronously receive an update, thus trading durability and consistency for availability and response time. By default, only single-document jobs are atomic. Recent versions also support distributed transactions using two-phase commit for atomicity and multi-version concurrency control for snapshot isolation.

CouchDB. CouchDB [10] adopts the same document data model as MongoDB. Early versions only support complete replication of shared state, allowing clients to read and write from any replica to improve availability. Replicas are periodically synchronized and conflicts are handled by storing multiple versions of conflicting documents or fields, delegating resolution to users. Since version 2.0, CouchDB provides a cluster mode with support for shared state partitioning and quorum-based replication, where users can configure the number of replicas for shared state portions, and quorum for read and write operations, thus balancing availability and consistency. CouchDB is conceived for Web applications and provides a synchronous HTTP API for job invocation. It makes the list of changes (change feed) to a document available for external components, which can consume it either in pull mode or in push mode (thus representing the sink components in out model). CouchDB lets users define multiple views for each document. In our model, we can see views as the results of registered jobs that are triggered by changes to documents. Computations that create views are restricted to execute on individual documents, but their results can then be aggregated by key (mimicking the MapReduce programming model). CouchDB supports dynamic reconfiguration with addition and removal of nodes, and redistribution of shared state portions across nodes. However, reconfiguration is a manual procedure.

AsterixDB. AsterixDB[7] is a semi-structured (document) store born as a research project that integrates ideas from NoSQL databases and distributed processing platforms. AsterixDB offers an SQL-like declarative language that integrates operators for individual documents as well as for multiple document (like joins, group by). Jobs are converted into a dataflow format and run on the Hyracks data-parallel platform [19]. Interestingly, the platform deploys jobs task by task but does not rely on a persistent data bus. Rather, when all input data for a task become ready, it dynamically establishes network connections from upstream tasks that deliver the results of their computation before terminating. AsterixDB supports querying shared state as well as external data from active and passive sources. Like BigTable and Cassandra, it stores state in LSM trees, which improve the performance of write operations [9]. It supports partitioning but does not currently support replication. As part of its shared state, AsterixDB can store indexes to simplify accessing external data. It offers isolation only for operations on individual values, using locking to update indexes. Its data structure offer fault tolerance through logging, but it does not currently implement fault detection nor dynamic reconfiguration mechanisms.

B.1.4 Time-Series Stores InfluxDB. InfluxDB[8] is a DMS for time-series data. Shared state is organized into measurements, which are sparse tables resembling wide columns in BigTable and Cassandra. Each row maps a point in time (primary key) to the values of one or more columns. Developers need to explicitly state which columns are indexed and which are not, thus balancing read and write latency. InfluxDB uses a storage model (time structured merge trees) that derives from LSM trees. It stores measurements column-wide and integrates disk-based storage and an

[7] https://asterixdb.apache.org.
[8] https://www.influxdata.com.

in-memory write cache to improve write performance. Jobs are defined in the InfluxQL declarative language and are restricted to single measurements. InfluxDB supports active sources, and mimics continuous jobs through periodic execution of one-shot jobs, which write their results inside the database and/or send them to sinks. InfluxDB partitions and replicates shared state across workers. Write operations are propagated semi-synchronously across replica workers for fault tolerance. Read operations can access any of the replica workers that contain the requested data, leading to weak consistency. InfluxDB adopts a leader-worker approach, where leader nodes store meta-data about membership, data partitioning, continuous queries, and access rights, and worker nodes store the actual data. Leader nodes are replicated with strong consistency for fault tolerance, using the Raft consensus protocol. InfluxDB offers dynamic reconfiguration to migrate data and add new workers, but the process is manual.

Gorilla. Gorilla [75] is an in-memory time-series store that Facebook uses as a cache to an HBase data store. HBase stores historical data compressed with a coarser time granularity, whereas Gorilla persists the most recent data (26 hours) in memory. Gorilla uses a simple data model where values for each measure always consist of a 64-bit timestamp and a 64-bit floating point number. It uses an encoding scheme based on bit difference that reduces the data size by 12 times on average. Jobs in Gorilla only perform simple read, write, and scan operations. A few ad hoc jobs have been implemented to support correlation between time series and in-memory aggregation, which is used to compress old data before writing it to HBase. Gorilla supports geo-replication for disaster recovery but trades durability for availability. Written data is asynchronously replicated, and it can be lost before it is made persistent. Gorilla supports dynamic adaptation by redistributing the key space across workers.

Monarch. Monarch [3] is a geo-distributed in-memory time-series store designed for monitoring large-scale systems and used within Google. Its data model stores time-series data as schematized tables. Each table consists of multiple key columns that form the time-series key, and a value column, which stores one value for each point in the history of the time series. Key columns include a target field, which is the entity that generates the time series, and a metrics field, which represents the aspect being measured. Monarch has a hierarchical architecture. Data is stored in the zone (data center) in which it is generated and sharded (by key ranges, lexicographically) across nodes called *leaves*. Data is stored in main memory and asynchronously persisted to logs on disk, trading durability to reduce write delay. Monarch offers a declarative, SQL-like language to express jobs, which can be either one-shot or continuous. In the latter case, they are evaluated periodically and store their results in new derived tables (materialized views). Jobs are evaluated hierarchically: nodes are organized in three layers (global, zone level, leaves), and the job plan pushes tasks as close as possible to the data they need to consume. Each level also stores an approximate view (compressed index) of what the nodes in the lower-level store. This enables optimizing communication across levels by avoiding pushing tasks to nodes that have no data related to that task. Monarch supports dynamic reconfiguration: it monitors lower-level nodes and redistributes data (key ranges) across nodes to adapt to changes in the load.

Peregreen. The design of the Peregreen [96] time-series database aims to satisfy the following requirements. First, cloud deployment of large volumes of historical data: as the scale of data is prohibitive for in-memory solutions, Peregreen relies on storage services such as distributed filesystems or block storage. It limits a raw data footprint by supporting only numeric values and by representing them in a compressed columnar format: each column is split into chunks and data in each chunk is represented using differences between adjacent values (delta encoding), further compressed to form a single binary array. Second, fast retrieval of data through indexing: Peregreen uses a three-tier data indexing, where each tier pre-computes aggregated statistics

(minimum, maximum, average, etc.) for the data it references. This allows to quickly identify chunks of data that satisfy some conditions based on the pre-computed statistics and to minimize the number of interactions with the storage layer. Peregreen jobs can only insert, update, delete, and retrieve data elements. Retrieval supports limited conditional search (only based on pre-computed statistics) and data transformation. Chunks are versioned, and a new version is created in the case of modifications. Peregreen is designed for cluster deployments and uses the Raft algorithm to reach consensus on the workers available at any point in time and the state portions (indexes to the storage layer) they are responsible for. This enables dynamic adaptation, with addition and removal of workers for load balancing and elasticity. State portions are replicated at multiple workers for fault tolerance.

B.1.5 Graph Stores TAO. TAO [20] is a data store that Facebook developed to manage its social graph, which contains billions of entities (e.g., people and locations) and relations between them (e.g., friendship). TAO offers a simple data model where entities and relations have a type and may contain data in the form of key-value pairs. It provides a restricted API for job definition, to create, delete, and modify entities and relations, and to query relations for a given entity. With respect to other graph data stores, it does not support queries that search for sub-graphs that satisfy specific constraints (path queries or sub-graph pattern queries). TAO is designed to optimize the latency of read jobs, as it needs to handle a large number of simultaneous user-specific queries. To do so, TAO implements shared state in two layers: a persistent storage layer based on a relational database (MySQL) and a key-value in-memory cache based on memcache. TAO is designed for wide-area deployments. Within a single data center, both the persistent layer and the cache are partitioned. The cache is also replicated using a single-leader approach: clients always interact with follower cache servers, which reply to read operations in the case of cache hit and propagate read operations in the case of cache miss as well as write operations to the leader cache server, which is responsible for interacting with the storage layer and for propagating changes. Across data centers, the storage layer is fully replicated with a single-leader approach: reads are served using the data center local cache or storage layer, so they do not incur latency, whereas all write operations are propagated to the leader data center for the storage layer. Data is replicated between the storage layer and the cache as well as across data centers asynchronously, which provides weak consistency and durability. Dynamic reconfiguration is possible in the case of failures: in the case in which a leader cache or storage server fails, replicas automatically elect a new leader.

Unicorn. Unicorn [35] is an indexing system used at Facebook to search its social graph. The shared state of Unicorn consists of inverted indexes that enable retrieving graph entities (vertices) based on their relations (edges) and the data associated with them. For instance, in a social graph, one could use an index on relations to retrieve all people (vertices) that are in a friend relation with a given person, or a string prefix index to retrieve all people with a name that starts with a given prefix. Unicorn is optimized for read-only queries (jobs), in the form of index lookups and set operations (union, intersection, difference) on lookup results. Jobs are evaluated by exploiting a hierarchical organization of workers into three layers: (i) index servers store the shared state (the indexes) partitioned by results, meaning that any index server will store a subset of the results for each index lookup; (ii) a rack aggregator per rack is responsible for merging the (partial) query results coming from individual index servers; and (iii) a top aggregator is responsible for merging the (partial) query results coming from each rack. Interestingly, Unicorn jobs can dynamically start new tasks: this feature is implemented as an `apply` function that performs new lookups starting from the results obtained in previous ones. This feature can be used to implement iterative computations—for instance, to find the friends of friends of a given person, Unicorn can first retrieve the direct friends and then lookup for all their friends, using result set of the first lookup

(direct friends) as a parameter for the second lookup. Unicorn indexes are kept up-to-date with the content of the social graph using a periodic procedure that runs on an external computation engine. Unicorn tolerates failures through replication. However, it is possible for some shared state portions to remain temporarily unavailable: this may result in incomplete results when searching (if some index serves do not reply), which is acceptable in its specific application domain.

B.2 NewSQL Systems

B.2.1 Key-Value Stores Deuteronomy. Deuteronomy [59] is a data store designed for wide-area deployments that decouples storage functionalities from transactional execution of jobs. In Deuteronomy, the driver program runs client-side and submits jobs (queries) to a transaction component. The component ensures group atomicity and isolation for all read and write operations performed on the shared state, using a locking protocol. The actual storage is implemented in a separate layer. Deuteronomy supports any distributed storage component that offers read and write operations for individual elements (e.g., a key-value store), and is oblivious of the actual location of the workers implementing the storage component, which may be geographically distributed. Deuteronomy provides fault tolerance through logging and replication, and enables dynamic reconfiguration by independently scaling both the transactional and the storage component.

FoundationDB. FoundationDB [100] is a transactional key-value store, which aims to offer the core building blocks (hence the name) to build scalable distributed DMSs with heterogeneous requirements. The use of key-value abstractions provides flexibility in the data model, on top of which developers can build various types of abstractions. For instance, each element may encode a relation indexed by its primary key. Jobs consist of a group of read and write requests issued by a driver program that runs client-side. Like Deuteronomy, FoundationDB is organized into layers, each of them offering one of the core functionalities of a transactional DMS and each implemented within a different set of workers, thus enabling independent scaling. A storage layer persists data and serves read and write requests. A log layer manages a WAL. A transaction system handles isolation and atomicity for multiple read and write requests. Transactional semantics is enforced by assigning timestamps to operations and by checking for conflicts between concurrent transactions after they have been executed: in the case of conflicts, the transaction aborts and the driver program is notified. Fault tolerance is implemented by replicating both the log layer (synchronously) and the storage layer (asynchronously). Replication of the storage layer is also used to serve read requests in parallel. Moving data between workers that implement the storage layer and the log layer is also used when adding or removing workers for scalability (dynamic reconfiguration).

Solar. Like Deuteronomy and FoundationDB, Solar [101] is a transactional key-value store that decouples storage of shared state from transaction processing. The transaction layer uses a timestamp-based optimistic concurrency control and stores a WAL in main memory. The storage layer persists checkpoints of the shared state. Together, they form an LSM tree. The key distinguishing feature of Solar is that the transaction layer is implemented as a centralized service, replicated for fault tolerance of the WAL.

B.2.2 Structured and Relational Stores Time-Based protocols Spanner. Spanner [34] is a semi-relational database: shared state is organized into tables, where each table has an ordered set of one or more primary key columns and defines a mapping from these key columns to non key columns. Spanner provides transactional semantics and replication with strong consistency for cluster and wide-area deployments. At its core, Spanner uses standard database techniques: two-phase locking for isolation, two-phase commit for atomicity, and synchronous replication of jobs results using Paxos state machine replication. Jobs are globally ordered using timestamps, and workers store multiple versions of each state element (multi-version concurrency control). This

way, read-only jobs can access a consistent snapshot of the shared state and do not conflict with read-write jobs. A consistent snapshot preserves causality, meaning that if a job reads a version of a state element (cause) and subsequently updates another state element (effect), the snapshot cannot contain the effect without the cause. Traditional databases ensure causality by acquiring read and write locks to prevent concurrent accesses, but this could be too expensive in a distributed environment. The key distinguishing idea of Spanner is to serve a consistent snapshot to read-only jobs without locking. To do so, it uses an abstraction called *TrueTime*, which returns real (wall-clock) time within a known precision bound using a combination of GPS and atomic clocks. Tasks of read-write jobs first obtain the locks for all the data portions they access: a coordinator for the job assigns that job with a timestamp at the end of its time uncertainty range, then it waits until this timestamp is passed for all workers in the system, releases the locks, and writes the results (commits). The waiting time ensures that jobs with later timestamps read all writes of jobs with earlier timestamps without explicit locking. Using TrueTime, Spanner also supports consistent reconfiguration—for instance, to change the database schema or to move data for load balancing. More recently, Spanner has been extended with support for distributed SQL query execution [14].

CockroachDB. CockroachDB [90] is a relational database for wide-area deployments. It shares many similarities with Spanner and integrates storage and processing capabilities within each node. As a storage layer, it relies on RocksDB, a disk-based key-value store that organizes data in LSM trees. It replicates data across nodes, ensuring strong consistency through Raft consensus. On top of this, it partitions data with transactional semantics: it uses an isolation mechanism based on hybrid physical and logical clocks (similar to Spanner) but integrates it with an optimistic protocol that, in the case of conflicts, attempts to modify the timestamp of a job to a valid one rather than re-executing the entire job. CockroachDB compiles SQL queries into a plan of tasks that can be either fully executed on a single worker or in a distributed dataflow fashion. Interestingly, CockroachDB also enables users to configure data placement across data centers. For instance, a table can be partitioned across a *Region* column to ensure that all data about one region is stored within a single data center. This may improve access time from local client and enforce privacy regulations.

Deterministic Execution Calvin. Calvin [92] is a job scheduling and replication layer to provide transactional semantics and replication consistency on top of non-transactional distributed data stores such as Dynamo, Cassandra, and MongoDB. It is currently implemented within the Fauna database.[9] The core Calvin layer is actually agnostic with respect to the specific data and query model. In fact, Fauna supports document and graph-based models in addition to the relational model. Calvin builds on the assumption that jobs are deterministic. Its core idea is to avoids as much as possible expensive coordination *during* job execution by defining a global order for tasks *before* the actual execution. In Calvin, workers are organized in regions: each region contains a single copy of the entire shared state, and each worker in a region is fully replicated in every other region. All workers that contain a replica of the same state portion in different regions are referred to as a replication group. Invocations from clients are organized in batches: all workers in a replication group receive a copy of the batch, and they coordinate to agree on a global order of execution. Under the assumption of deterministic jobs, this approach ensures consistent state across all replicas in a replication group. Replicas are used both to improve access for read-only jobs, which can be executed on any replica, and for fault tolerance, as in the case of a failure remaining replicas can continue to operate without interruption. Invocations are stored in a durable log, and individual workers can be resumed from a state snapshot by reapplying all invocations that occurred after that snapshot. Calvin supports different replication protocols with different tradeoffs between job

[9] https://fauna.com.

response time and complexity in fail over. Deterministic jobs executed in the same order lead to the same results in all (non-failing) replicas. Specifically, they either commit or abort in all (non-failing) replicas. Accordingly, under the assumption that at least one replica for each shared state portion does not fail, Calvin can provide atomicity without expensive protocols such as the classic two-phase commit: if some tasks may abort due to violation of integrity constrains, they simply inform other tasks of the same transaction with a single (one-phase) communication. Global execution order is also used for isolation: Calvin exploits a locking mechanism where tasks acquire locks on their shared state portion in the agreed order. This requires knowing up front the exact state portions accessed within each job: when they cannot be statically determined (e.g., due to state-dependent control flow), Calvin runs reconnaissance jobs that perform all read accesses to determine the state portions of interest. However, during the actual execution, shared state may have changed, and the real jobs may deviate from reconnaissance jobs and try to access different portions, in which case they are deterministically restarted. Interestingly, Calvin provides the same strong semantics both for cluster and for wide-area deployments. The initial coordination increases the latency to schedule a batch of jobs, affecting the response time of individual jobs in wide-area deployments, but the batching mechanisms can preserve throughput.

Explicit Partitioning and Replication Strategies VoltDB. VoltDB [89] is an in-memory relational database developed from the HStore research project [88]. In VoltDB, clients register stored procedures, which are driver programs written in Java and executed system-side. They include multiple jobs, are compiled on registration, and are executed on invocation. Jobs can also write data to sinks. The key idea of VoltDB is to let users control database partitioning and replication, so they can optimize most frequently executed jobs. In particular, VoltDB preserves the same (transactional) execution semantics as centralized databases while minimizing the overhead of concurrency control. By default, all tasks that derive from a single driver program represent a transaction and are guaranteed to execute with group atomicity and isolation. In the worst case, this is achieved through blocking coordination (two-phase commit for atomicity and timestamp-based concurrency control for isolation). However, VoltDB avoids coordination for specific types of jobs by exploiting user-provided data about data partitioning and replication. Users can specify that a relational table is partitioned based on the value of a column. For instance, a `Customer` table may be partitioned by `region`, meaning that all customers that belong to the same region (have the same value for the attribute `region`) are stored in the same shared state portion. Jobs that only refer to a given region can then be executed by the single worker responsible for that state portion, sequentially, without incurring expensive concurrency control overhead. Every table that is not partitioned is replicated in every worker, which optimizes read access from any worker at the cost of replicating state changes. Users need to select the best partitioning and replication schema to improve performance for the most frequent jobs. VoltDB also supports replicating tables (including partitioned ones) for fault tolerance. Replicas are kept up-to-date by propagating and executing tasks to all replicas, under the assumption that tasks are deterministic. VoltDB is designed for cluster deployment as clock synchronization and low latency are necessary to guarantee that timestamp-based concurrency control and replication work well in practice. However, VoltDB also provides a hybrid deployment model where a database is fully replicated at multiple geographical regions. These replicas can be used only as an additional form of fault tolerance (with asynchronous propagation of state) or can serve local clients. In this case, consistency across regions is not guaranteed, and conflicts get resolved using pre-defined automatic rules. VoltDB supports manual reconfiguration of both partitioning and workers but requires stopping and restarting the system.

Primary-Based Protocols Aurora. Aurora [95] is a relational database offered as a service by Amazon. Aurora builds on two key design choices: (i) decouple the storage layer from the query pro-

cessing layer and (ii) store the log of changes (WAL) in the storage layer instead of the actual shared state. Shared state is materialized only to improve read performance, and materialization can be performed asynchronously without increasing write latency. The storage layer—that is, the write log—is replicated both to improve read performance and for fault tolerance. To ensure consistency, read and write operations use a quorum-based approach. The processing layer accepts jobs from clients in the form of SQL queries, which are always executed within a single worker. Specifically, to guarantee isolation, Aurora assumes that a single worker is responsible for processing all read-write jobs at any given point in time. Read-only jobs can be executed on any worker, which can read a consistent snapshot of the state without conflicting with concurrent writes. In Aurora, the storage and processing layers can scale independently: the processing layer is stateless, whereas the storage layer only needs to replicate the log.

Socrates. Socrates [11] is a relational database offered as a service in the Azure cloud platform (under the name *SQL DB Hyperscale*).[10] Like Aurora, Socrates decomposes the functionality of a DMS and implements them as independent services. Its design goals include quick recovery from failure and fast reconfiguration. To do so, it relies on four layers, each implemented as a service that can scale out when needed. First, compute nodes handle jobs, including protocols for group atomicity and isolation. There is one primary compute node that processes read-write jobs and an arbitrary number of secondary nodes that handle read-only jobs and may become primary in the case of failure. Compute nodes cache shared state pages in main memory and on SSD. Second, a log service logs write requests with low latency. Third, a storage service periodically applies writes from the log to store the shared state durably. Fourth, a backup service stores copies of the storage layer for fault tolerance.

B.2.3 Objects Stores Tango. Tango [16] is a service for storing metadata. Application code (the driver program) executes client-side and reads and accesses a shared state consisting of Tango objects. Clients store their view of Tango objects locally in-memory, and this view is kept up-to-date with respect to a distributed (partitioned) and durable (replicated) totally ordered log of updates. Tango objects can contain references to other Tango objects, thus enabling the definition of complex linked data structures such as trees or graphs. Although the log is physically partitioned across multiple computers, all operations are globally ordered through sequence numbers, which are obtained through a centralized sequencer service: the authors demonstrate that this service does not become a bottleneck for the system when serving hundreds of thousands of requests per second. Clients check if views are up-to-date before performing updates, thus ensuring totally ordered, linearizable updates. Tango also guarantees group atomicity and isolation using the log for optimistic concurrency control.

B.2.4 Graph Stores A1. A1 [22] is an in-memory database that resembles TAO and Trinity in terms of data model (typed graphs) and jobs (changes to graph entities and graph pattern matching queries). The distinguishing characteristic of A1 is that it builds on a distributed shared memory abstraction that uses RMDA implemented within network interface cards [41]. A1 stores all the elements of the graph in a key-value store. Jobs may traverse the graph and read and modify its associated data during execution. A1 provides strong consistency, atomicity, and isolation using timestamp-based concurrency control. Fault tolerance is implemented using synchronous replication in memory and asynchronous replication to disk. In the case of failures, users can decide whether to recover the last available state or only state that is guaranteed to be transactionally consistent.

[10] https://docs.microsoft.com/en-us/azure/azure-sql/database/.

C DATA PROCESSING SYSTEMS

This section details individual Data Processing Systems (DPSs). Tables 20, 21, 22, 23, 24, 25, 26, 27 summarize the characteristics of these systems with respect to the classification criteria presented in Section 2.

C.1 Task-Level Deployment

MapReduce. MapReduce [38] is a distributed processing model and system developed at Google in the early 2000s, and later implemented in open source projects such as Apache Hadoop.[11] Its programming and execution models represent a paradigm shift in distributed data processing that influenced virtually all DPSs discussed in this article: developers are forced to write jobs as a sequence of functional transformations, avoiding by design the complexity and cost associated with state management. In the specific case of MapReduce, jobs are constrained to only two processing steps: (i) *map* transforms each input element into a set of key-value pairs, and (ii) *reduce* aggregates all values associated with a given key. Both functions are data parallel: developers specify the map function for a single input element and the reduce function for a single key, and the system automatically applies them in parallel. The data bus is implemented using a distributed filesystem. The system schedules map tasks as close as possible to the physical location of their input in the filesystem. It then automatically redistributes intermediate results by key before scheduling the subsequent reduce tasks. Fault detection is implemented using a leader-worker approach. Tasks that did not complete due to a failure are simply rescheduled. The same approach is used for tasks that take long to complete (stragglers): they are scheduled multiple times if some workers are available, to increase the probability of successful completion. Dynamic scheduling at the granularity of tasks simplifies implementation of dynamic reconfiguration mechanisms to promote elasticity. For instance, Hadoop supports scheduling policies based on user-defined quality of service requirements, such as expected termination time: the scheduler tunes the use of resources (possibly shared with other applications) to meet the expectation while minimizing the use of resources. Several works build on top of MapReduce to offer a declarative language similar to SQL to express analytical queries that are automatically translated into MapReduce jobs [2, 23].

Table 20. Data Processing Systems: Functional Model

	Driver Exec	Driver Exec Time	Invoc of Jobs	Sources	Sinks	State	Deployment
Dataflow Task Deployment Systems							
MapReduce	sys	reg	sync	passive	yes	no	cluster
Dryad	sys	reg	unknown	passive	yes	no	cluster
HaLoop	sys	reg	unknown	passive	yes	no	cluster
CIEL	sys	reg	async	passive	yes	no	cluster
Spark	configurable	reg	sync+async	passive	yes	no	cluster
Spark Streaming	configurable	reg	async	both	yes	yes	cluster
Dataflow Job Deployment Systems							
MillWheel	sys	reg	async	both	yes	yes	cluster
Flink	configurable	reg	async	both	yes	yes	cluster
Storm	sys	reg	async	both	yes	yes	cluster
Kafka Streams	client	reg	async	both	yes	yes	cluster
Samza / Liquid	client	reg	async	both	yes	yes	cluster
Timely dataflow	client	reg	sync	both	yes	yes	cluster
Graph Processing Systems							
Pregel	sys	reg	unknown	passive	yes	yes	cluster
GraphLab	sys	reg	unknown	passive	yes	yes	cluster
PowerGraph	sys	reg	unknown	passive	yes	yes	cluster
Arabesque	sys	reg	unknown	passive	yes	yes	cluster
G-Miner	sys	reg	unknown	passive	yes	yes	cluster

[11] https://hadoop.apache.org.

Table 21. Data Processing Systems: Jobs Definition

	Jobs Def API	Exec Plan Def	Task Comm	Exec Plan Struct	Iter	Dyn Creat	Nature of Jobs	State Man	Data Par API	Placem -Aware API
Dataflow Task Deployment Systems										
MapReduce	lib	expl	impl	dataflow	no	no	one-shot	absent	yes	no
Dryad	lib	expl	impl	dataflow	no	no	one-shot	absent	yes	no
HaLoop	lib	expl	impl	cycl dataflow	yes	no	one-shot	absent	yes	no
CIEL	lib	expl	impl	dyn dataflow	no	yes	one-shot	absent	yes	no
Spark	lib (+DSLs)	expl (+impl)	impl	dataflow	no	no	one-shot	absent	yes	no
Spark Streaming	lib (+DSLs)	expl (+impl)	impl	dataflow	no	no	cont	impl	yes	no
Dataflow Job Deployment Systems										
MillWheel	lib	expl	impl	dataflow	no	no	cont	impl	yes	no
Flink	lib (+DSLs)	expl (+impl)	impl	dataflow	limited	no	one-shot^{B} cont^{S}	impl	yes	no
Heron	lib	expl	impl	dataflow	no	no	cont	impl	yes	no
Kafka Streams	lib (+DSL)	expl (+impl)	impl	dataflow	no	no	one-shot^{B} cont^{S}	impl	yes	no
Samza / Liquid	lib	expl	impl	dataflow	no	no	one-shot^{B} cont^{S}	impl	yes	no
Timely dataflow	lib	expl	impl	cycl. dataflow	yes	no	cont	impl	yes	no
Graph Processing Systems										
Pregel	lib	expl	expl	graph	yes	no	cont	expl	yes	yes
GraphLab	lib	expl	expl	graph	yes	no	cont	expl	yes	yes
PowerGraph	lib	expl	expl	graph	yes	no	cont	expl	yes	yes
Arabesque	lib	expl	expl	graph	yes	no	cont	expl	yes	yes
G-Miner	lib	expl	expl	graph	yes	yes	cont	expl	yes	yes

^{B}, in batch processing; ^{S}, in stream processing.

Table 22. Data Processing Systems: Jobs Compilation and Execution

	Jobs Comp Time	Use Resources Info (Comp)	Granul of Depl	Depl Time	Use Resources Info (Depl)	Manag of Res
Dataflow Task Deployment Systems						
MapReduce	reg	dynamic	task	task activ	dynamic	shared
Dryad	reg	dynamic	task	task activ	dynamic	sys-only
HaLoop	reg	dynamic	task	task activ	dynamic	shared
CIEL	reg	dynamic	task	task activ	dynamic	sys-only
Spark	reg	dynamic	task	task activ	dynamic	shared
Spark Streaming	reg	dynamic	task	task activ	dynamic	shared
Dataflow Job Deployment Systems						
MillWheel	reg	static	job	job compil	static	sys-only
Flink	reg	static	job	job compil	static	sys-only
Heron	reg	static	job	job compil	static	sys-only
Kafka Streams	reg	static	job	job compil	static	sys-only
Samza / Liquid	reg	static	job	job compil	dynamic	shared
Timely dataflow	reg	static	job	job compil	static	sys-only
Graph Processing Systems						
Pregel	reg	static	job	job compil	static	shared
GraphLab	reg	static	task	task activ	static	sys-only
PowerGraph	reg	static	task	task activ	static	sys-only
Arabesque	reg	static	job	job compil	static	sys-only
G-Miner	reg	static	task	task activ	static	sys-only

Table 23. Data Processing Systems: Data Management

	Elem Struc	Temp Elem	Bus Conn	Bus Impl	Bus Persist	Bus Partition	Bus Repl	Bus Inter
Dataflow Task Deployment Systems								
MapReduce	general	no	mediated	distr fs	persist	yes	no	hybrid
Dryad	general	no	direct or mediated	configurable	persist or ephem	yes	no	push or hybrid
HaLoop	general	no	mediated	distr fs	persist	yes	no	hybrid
CIEL	general	no	direct or mediated	configurable	persist or ephem	yes	no	push or hybrid
Spark	general+spec	no	mediated	distr fs (+cache)	persist	yes	no	hybrid
Spark Streaming	general+spec	yes	mediated	distr fs (+cache)	persist	yes	no	hybrid
Dataflow Job Deployment Systems								
MillWheel	general	yes	direct	RPC	ephem	yes	no	push
Flink	general+spec	yes	direct	net chan	ephem	yes	no	push
Storm	general	yes	direct	net chan	ephem	yes	no	push
Kafka Streams	general+spec	yes	mediated	kafka	persist	yes	yes	hybrid
Samza / Liquid	general	yes	mediated	kafka	persist	yes	yes	hybrid
Timely dataflow	general	yes	direct	net chan	ephem	yes	no	push
Graph Processing Systems								
Pregel	typed graph	no	direct	net chan	ephem	yes	no	push
GraphLab	typed graph	no	direct	mem	persist	yes	no	hybrid
PowerGraph	typed graph	no	direct	mem	persist	yes	no	hybrid
Arabesque	typed graph	no	direct	net chan	ephem	yes	no	push
G-Miner	typed graph	no	direct	net chan	ephem	yes	no	pull

Table 24. Data Processing Systems: State Management

	Elem Struct	Stor Medium	Stor Struct	Task St	Shared St	St Part	Repl	Repl Consist	Repl Prot	Update Propag
Dataflow Task Deployment Systems										
MapReduce	n.a.	n.a.	n.a.	no	no	n.a.	n.a.	n.a.	n.a.	n.a.
Dryad	n.a.	n.a.	n.a.	no	no	n.a.	n.a.	n.a.	n.a.	n.a.
HaLoop	n.a.	n.a.	n.a.	no	no	n.a.	n.a.	n.a.	n.a.	n.a.
CIEL	n.a.	n.a.	n.a.	no	no	n.a.	n.a.	n.a.	n.a.	n.a.
Spark	n.a.	n.a.	n.a.	no	no	n.a.	n.a.	n.a.	n.a.	n.a.
Spark Streaming	n.a.	n.a.	n.a.	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Dataflow Job Deployment Systems										
MillWheel	n.a.	n.a.	n.a.	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Flink	n.a.	n.a.	n.a.	no^{B} / yes^{S}	no	n.a.	n.a.	n.a.	n.a.	n.a.
Storm	n.a.	n.a.	n.a.	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Kafka Streams	n.a.	n.a.	n.a.	no^{B} / yes^{S}	no	n.a.	n.a.	n.a.	n.a.	n.a.
Samza / Liquid	n.a.	n.a.	n.a.	no^{B} / yes^{S}	no	n.a.	n.a.	n.a.	n.a.	n.a.
Timely dataflow	n.a.	n.a.	n.a.	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Graph Processing Systems										
Pregel	vertex	mem	user-def	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
GraphLab	vertex/edge	mem	user-def	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
PowerGraph	vertex/edge	mem	user-def	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Arabesque	sub-graph	mem	user-def	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
G-Miner	sub-graph	mem	user-def	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.

^{B}, in batch processing; ^{S}, in stream processing.

Dryad. Several systems developed in parallel and after MapReduce inherit and extend the core concepts in its programming and execution models. Dryad [51] generalizes the programming model, representing jobs as arbitrary acyclic dataflow graphs. Like MapReduce, it uses a leader-worker approach, where a leader schedules individual tasks (operators in the dataflow graph), but it enables different types of channels (data bus in our model), including shared memory on the same machine, TCP channels across machines, or distributed filesystems. The leader is also responsible for fault detection. Jobs are assumed to be deterministic, and in the case of failure, the failing task is re-executed: in the case of ephemeral channels, also upstream tasks in the dataflow graph are re-executed to re-create the input for the failing task.

HaLoop. Systems like MapReduce and Dryad are constrained to acyclic job plans and cannot natively support iterative algorithms. HaLoop [21] addresses this limitation with a modified version of MapReduce that (i) integrates iterative MapReduce jobs as first class programming con-

Table 25. Data Processing Systems: Group Atomicity, Group Isolation, Delivery, and Order

	Aborts	Protocol	Assumptions	Level	Impl	Assumptions	Delivery	Nature of Ts	Order
Dataflow Task Deployment Systems									
MapReduce	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
Dryad	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
HaLoop	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
CIEL	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
Spark	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
Spark Streaming	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no or ingest	always
Dataflow Job Deployment Systems									
MillWheel	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	ingest	always
Flink	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no or ingest	always
Storm	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	most / least / exact	no	n.a.
Kafka Streams	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	least	no or ingest	eventually
Samza / Liquid	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	least	no or ingest	eventually
Timely dataflow	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	event	always
Graph Processing Systems									
Pregel	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
GraphLab	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
PowerGraph	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
Arabesque	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
G-Miner	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.

Table 26. Data Processing Systems: Fault Tolerance

	Detection	Scope	Comput Recov	State Recov	Guarantees for State	Assumptions
Dataflow Task Deployment Systems						
MapReduce	lead-work	comput	task	n.a.	n.a.	REPLAY
Dryad	lead-work	comput	task	n.a.	n.a.	REPLAY
HaLoop	lead-work	comput	task	n.a.	n.a.	REPLAY
CIEL	lead-work	comput	task	n.a.	n.a.	REPLAY
Spark	lead-work	comput	task	n.a.	n.a.	REPLAY
Spark Streaming	lead-work	comput+task st	task	log+checkp	valid or same	REPLAY
Dataflow Job Deployment Systems						
MillWheel	lead-work	comput+task st	job	log+checkp	same	REPLAY
Flink	lead-work	comput+task st	job	log+checkp	valid or same	REPLAY
Storm	lead-work	comput+task st	task	ack+checkp	none or valid or same	REPLAY
Kafka Streams	lead-work	comput+task st	job	log+checkp	valid or same	REPLAY
Samza / Liquid	lead-work	comput+task st	job	log+checkp	valid or same	REPLAY
Timely dataflow	lead-work	comput+task st	job	log+checkp	same	REPLAY
Graph Processing Systems						
Pregel	lead-work	comput+task st	job	checkp	same	n.a.
GraphLab	lead-work	comput+task st	job	checkp	valid	n.a.
PowerGraph	lead-work	comput+task st	job	checkp	valid or same	n.a.
Arabesque	lead-work	comput+task st	job	checkp	valid	n.a.
G-Miner	lead-work	comput+task st	tas	checkp	valid	n.a.

REPLAY, sources are replayable.

Table 27. Data Processing Systems: Dynamic Reconfiguration

	Goal	Automated	State Migration	Task Migration	Add / Del Slots	Restart
Dataflow Task Deployment Systems						
MapReduce	elast	yes	n.a.	n.a.	yes	no
Dryad	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
HaLoop	elast	yes	n.a.	n.a.	n.a.	no
CIEL	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Spark	elast	yes	n.a.	n.a.	yes	no
Spark Streaming	elast	yes	yes	yes	yes	no
Dataflow Job Deployment Systems						
MillWheel	load balan	yes	yes	yes	yes	no
Flink	elast	yes	yes	yes	yes	yes
Storm	elast	no	yes	yes	yes	yes
Kafka Streams	elast	no	yes	yes	yes	no
Samza / Liquid	elast	no	yes	yes	yes	no
Timely dataflow	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Graph Processing Systems						
Pregel	load balan	yes	yes	yes	no	no
GraphLab	load balan	yes	yes	yes	no	no
PowerGraph	load balan	yes	yes	yes	no	no
Arabesque	load balan	yes	yes	yes	no	no
G-Miner	load balan	yes	yes	yes	no	no

cepts, (ii) optimizes task scheduling by co-locating tasks that reuse the same data across iterations, (iii) caches and indexes loop-invariant data to optimize access across iterations, and (iv) caches and indexes results across iterations to optimize the evaluation of fixed point conditions for termination.

CIEL. CIEL [72] extends the dataflow programming model of Dryad by allowing tasks to dynamically create other tasks. This enables defining the job plan dynamically based on the results of data computation, which can be used to implement iterative algorithms. The programming model ensures that tasks cannot have cyclic dependencies, thus avoiding deadlocks in the execution. The execution model and the fault tolerance mechanism remain identical to Dryad, the only exception being that the list of tasks that the leader schedules and tracks can dynamically change during the execution, thus allowing the execution flow to be defined at runtime based on the actual content of input data.

Spark. Spark [99] inherits the dataflow programming model of Dryad and supports iterative execution and data caching like HaLoop. First, Spark jobs are split in sequences of operations that do not alter data partitioning, called *stages*. Multiple tasks are scheduled for each stage, to implement data parallelism. Second, the driver program can run either client-side or system-dice, and can dynamically spawn new jobs based on the results collected from previous jobs. This enables data-dependent control flow under the assumption that control flow conditions are evaluated in the driver program, and is used to implement iterative algorithms. Third, as in HaLoop, intermediate results that are reused by the same or different jobs (as in the case of iterative computations) can be cached in main memory to improve efficiency. Spark provides domain-specific libraries and languages for structured (relational) data [12], graphs [46], and machine learning computations [69], which often include ad hoc job optimizers. Spark also inherits the fault tolerance model of MapReduce: as tasks are stateless, failing tasks are simply re-executed starting from their input data; if the input data is not available anymore (e.g., in the case of intermediate results not persisted on any other node), all tasks necessary to reconstruct the input are also re-executed. Task-level deployment also enables runtime reconfiguration to provide elasticity.

Spark Streaming. Spark Streaming [98] implements streaming computations on top of Spark by splitting the input stream into small batches and by running the same jobs for each batch. Spark Streaming implements task state using native Spark features: the state of a task after a given invocation is implicitly stored as a special data item that the task receives as input in the subsequent invocation. This also enables reusing the fault tolerance mechanism of Spark to persist or recompute state as any other data element. Another benefit of using the Spark system is simple integration of static and streaming input data. The main drawback of the approach is latency: since input data needs to be accumulated in batches before processing, Spark Streaming can only provide latency in the range of seconds.

C.2 Job-Level Deployment

MillWheel. MillWheel [7] is a framework to build general-purpose and large-scale stream processing systems. Jobs consist of data-parallel tasks expressed using an imperative language. As part of their computation, tasks can use the MillWheel API to access (task local) state, for logical time information, and to produce data for downstream tasks. MillWheel implements the communication between tasks (the data bus) using point-to-point remote procedure calls, and uses an external storage service to persist task state and metadata about global progress (watermarks). Upstream tasks are acknowledged when downstream tasks complete, and the storage service is kept consistent with atomic updates of state and watermarks. In the case of failure, individual tasks can rely on the external storage to restore a consistent view of their state, and they can discard duplicate

invocations from upstream tasks in the case some acknowledgements are lost. The same approach is also used for dynamic load distribution and balancing.

Flink. Flink [24] is a unified execution engine for batch and stream processing. In terms of a programming model, it strongly resembles Spark, with a core API to explicitly define job plans and domain-specific libraries for structural (relational) data, graph processing, and machine learning. One notable difference involves iterative computations: Flink supports them with native operators (within jobs) rather than controlling them from the driver program. Flink adopts job-level deployment and implements the data bus using direct TCP channels. This brings several implications. First, tasks are always active and compete for workers resources. For each processing step in the execution plan, Flink instantiates one (data-parallel) task for each CPU core available on workers. In practice, each CPU core receives one task for each step and scheduling is delegated to the operating system that hosts the worker. Second, in the case of stream processing, tasks can continuously exchange data, which traverses the graph of computation leading to a pipelined execution. As there is no need to accumulate input data in batches, this processing model may reduce latency. Third, as tasks cannot be deployed independently, fault tolerance requires restarting an entire job. In the case of stream processing jobs, Flink takes periodic snapshots of task states: in the case of a failure, it restores the last snapshot and replays all input data that was not part of the snapshot (assuming it remains available). Fourth, dynamic reconfiguration builds on the same mechanism: when the number of available slots changes, the system needs to restart and resume jobs from a recent snapshot.

Storm. Storm [94] and its successor Heron [56] are stream processing systems developed at Twitter. They offer lower-level programming API than previously discussed dataflow systems, where developers fully implement the logic of each processing step. At the time of writing, both Storm and Heron have experimental higher-level APIs that mimic the functional approach of Spark and Flink. Storm and Heron adopt job-level deployment and implement the data bus as direct network channels between workers. They implement fault tolerance by acknowledging every message. If a message is not acknowledged within a given timeout, the sender replays it, which leads to at least once delivery (messages may be duplicated). The same applies in the case of state: state is checkpointed but may be modified more than once in the case of duplication. Dynamic reconfiguration is possible but required redeploying and restarting the entire job.

Kafka Streams. Kafka [55] is a distributed communication platform designed to scale in terms of clients, data volume, and production rate. Kafka offers logical communication channels named *topics*: producers append immutable data to topics and consumers read data from topics. Topics are persistent, which decouples data production and consumption times. Topics may be partitioned to improve scalability: multiple consumers may read in parallel from different partitions, possibly hosted on different physical nodes. Topics may also be (semi-synchronously) replicated for fault tolerance. With respect to our model in Section 2, Kafka represents the implementation of a persistent data bus that external clients can use to exchange immutable data. Kafka Streams [18] implements batch and stream processing functionalities on top of Kafka. Its programming model is similar to that of Spark and Flink, with a core functional API and a higher-level DSL for relational data processing (KSQL). As in Flink, all tasks for a job are instantiated and scheduled when the job starts and continuously communicate in a pipelined fashion using Kafka as the data bus. Each channel in the job logical plan is implemented as a Kafka topic, and data parallelism exploits topic partitioning, allowing multiple tasks to simultaneously read from different partitions. Interestingly, Kafka Streams does not offer resource management functionalities but runs the driver program and the tasks within clients: each job definition is associated with a unique identifier, and clients can offer resources for a job—that is, become workers—by referring to the job identifier. The

policy for allocating tasks to slots is similar to that of Flink: each slot represents a physical CPU core and receives one task for each computation step in the logical plan. Task state for streaming jobs is stored on Kafka, following the same idea of persisting state as a special element in the data bus that we already found in Spark Streaming. Fault detection relies on Kafka ability to detect when consumers disconnect. For fault recovery, Kafka Streams adopts a two-phase commit protocol to ensure that upon activation a task consumes its input, updates its state, and produces results for downstream tasks atomically. In the case of failure, a task can resume from the input elements that were not successfully processed, providing exactly once delivery of individual elements. We still classify the system as offering at least once delivery, because, in the case of timestamped elements, it does not implement any mechanism to process them in timestamp order but retracts and updates its results upon receiving elements out of order (resulting in visible changes at the sinks). Storing both data and task state on Kafka allows for dynamic reconfiguration that involves addition and removal of clients (workers) at runtime. The same approach of Kafka Streams is used at LinkedIn in the Samza system [74], which is the core for the platform to integrate data from multiple sources and offer a unique view to the back-end system, updated incrementally as new data becomes available.

Timely Dataflow. Timely dataflow [71] is a unified programming model for batch and stream processing, which is lower level and more general than the dataflow model of systems such as Flink. In timely dataflow, jobs are expressed as a graph of (data-parallel) operators and data elements carry a logical timestamp that tracks global progress. Management of timestamps is explicit, and developers control how operators handle and propagate them. This enables implementing various execution strategies. For instance, developers may choose to complete a given computation step before letting the subsequent one start (mimicking a batch processing strategy as implemented in MapReduce or Spark), or they may allow overlapping of steps (as it happens in Storm or Flink). The flexibility of the model allows for complex workflows, including streaming computations with nested iterations, which are hard or even impossible to express in other systems. Timely dataflow is currently implemented as a Rust library:[12] as in Kafka Streams, developers write a program that defines the graph of computation using the library API, and run multiple instances of the program, each of them representing a worker in our model. At runtime, the program instantiates the concrete tasks, which communicate with each other either using shared memory (within one worker) or TCP channels (across workers). Timely dataflow provides API to checkpoint task state and to restore the last checkpoint for a job. Dynamic reconfiguration is currently not supported.

C.3 Graph Processing

Pregel. Pregel [66] is a programming and execution model for computations on large-scale graph data structures. Pregel jobs are iterative: developers provide a single function that encodes the behavior of each vertex v at each iteration. The function takes in input the current (local) state of v and the set of messages produced for v during the previous iteration; it outputs the new state of v and a set of messages to be delivered to connected vertices, which will be evaluated during the next iteration. The job terminates when vertices do not produce any message at a given iteration. Vertices are partitioned across workers, and each task is responsible for a given partition. Jobs are continuous, as tasks are activated multiple times (once for each iteration) and store the vertex state across activations (in their task state). Tasks only communicate by exchanging data (messages between vertices) over the data bus, which is implemented as direct channels. One worker acts as a leader and is responsible for coordinating the iterations within the job and for detecting possible

[12] https://github.com/TimelyDataflow.

failures of other workers. Workers persist their state (task state and input messages) at each iteration: in the case of a failure, the computation restarts from the last completed iteration. Several systems inherit and improve the original Pregel model in various ways: we discuss some key variants through the systems that introduced them. The interested reader can find more details and systems in the survey by McCune et al. [68].

GraphLab. GraphLab [62] abandons the synchronous model of Pregel, where all vertices execute an iteration before any can move to the subsequent one. GraphLab schedules the execution of tasks that update vertices. During execution, tasks can read the value of neighboring vertices and edges (rather than receiving update messages, as in Pregel), and can update the value of outgoing edges. We model this style of communication as a pull-based persistent data bus.[13] Tasks are scheduled and execute asynchronously, without barriers between iterations. This paradigm is suitable for machine learning and data mining computations that do not require synchronous execution for correctness and can benefit from asynchronous executions for performance. GraphLab still supports some form of synchronization between tasks. For instance, users can grant exclusive or non-exclusive access to neighboring edges and vertices. GraphLab implements this synchronization constraints either with a locking protocol or with scheduling policies that prevent execution of potentially conflicting tasks.

PowerGraph. PowerGraph [45] observes that vertex-centric execution may lead to unbalanced work in the (frequent) scenario of skewed graphs. It proposes a solution that splits each iteration into four steps: (i) *gather* collects data from adjacent vertices and edges, (ii) *sum* combines the collected data, (iii) *apply* updates the state of the local vertex, and (iv) *scatter* distributes data to adjacent edges for the next iteration. These steps can be distributed across all workers and executed in a MapReduce fashion. PowerGraph tasks can be executed synchronously, as in Pregel, or asynchronously, as in GraphLab, depending on the specific problem at hand.

Sub-Graph Centric Systems. Graph mining problems typically require retrieving sub-graphs with given characteristics. A class of systems designed to tackle these problems uses a sub-graph centric approach. We model these systems by considering the input graph as a static data source and by storing the state of each sub-graph in task state. Arabesque [91] explores the graph in synchronous rounds: it starts with candidate sub-graphs consisting of a single vertex and at each round expands the exploration by adding one neighboring vertex or edge to a candidate. G-Miner [30] spawns a new task for each candidate sub-graph, allowing tasks to proceed asynchronously. When scheduled for execution, a task can update its (task) state. G-Miner supports dynamic load balancing with task stealing.

D OTHER SYSTEMS

This section details data-intensive systems that do not clearly fall within the categories of Data Management Systems and Data Processing Systems. Tables 28, 29, 30, 31, 32, 33, 34, 35 summarize the characteristics of these systems with respect to the classification criteria in Section 2.

D.1 Computations on DMSs

Percolator. Percolator [76] builds on top of BigTable and is used to automatically and incrementally maintain views when BigTable gets updated. For instance, it is used within Google to incrementally maintain the indexes of its search engine as Web pages and links change. Percolator enables

[13] Another way to model this communication paradigm is by saying that tasks have access to a global shared state representing vertices and edges. However, this would not capture the strong connection between tasks and the vertices they update.

Table 28. Other Systems: Functional Model

	Driver Exec	Driver Exec Time	Invoc of Jobs	Sources	Sinks	State	Deployment
Computations on Data Management Systems							
Percolator	sys	start	sync	active	yes	yes	cluster
F1	client	reg	sync+async	both	yes	yes	cluster + wide
Trinity	client	reg	unknown	no	no	yes	cluster
New Programming Models							
SDG	sys	reg	sync	active	yes	yes	cluster
TensorFlow	client	reg	sync	passive	yes	yes	cluster
Tangram	sys	reg	sync	passive	yes	no	cluster
ReactDB	sys	start	sync+async	no	no	yes	cluster (not impl)
Hybrid Systems							
S-Store	sys	reg	sync+async	active	yes	yes	cluster (not impl)
SnappyData	sys	reg	sync+async	both	yes	yes	cluster
StreamDB	sys	start	async	active	yes	yes	cluster
TSpoon	configurable	reg	sync+async	active	yes	yes	cluster
Hologres	client	reg	unknown	both	yes	yes	cluster

Table 29. Other Systems: Jobs Definition

	Jobs Def API	Exec Plan Def	Task Comm	Exec Plan Struct	Iter	Dyn Creat	Nature of Jobs	State Man	Data Par API	Placem -Aware API
Computations on Data Management Systems										
Percolator	imperative +BigTable jobs	impl	impl	task (put / get)	no	no	one-shot	expl	no	no
F1	SQL	impl	impl	datafl+coord	no	no	one-shot	expl	no	no
Trinity	crud+lib	impl+expl	impl + expl	task (put / get) +graph	yes	no	one-shot	expl	yes	no
New Programming Models										
SDG	imperative	impl	impl	stateful datafl	yes	no	cont	expl	yes	no
TensorFlow	lib	impl	impl	stateful datafl	yes	yes	one-shot	expl	yes	yes
Tangram	lib	expl	impl	stateful datafl	yes	no	one-shot	expl	yes	no
ReactDB	lib	impl+expl	expl	workfl	yes	no	one-shot	expl	no	no
Hybrid Systems										
S-Store	lib+SQL	impl+expl	impl	workfl+datafl	yes	no	one-shot^{B} cont^{S}	expl+impl	yes	no
SnappyData	lib+SQL	impl+expl	impl	workfl+datafl	no	no	one-shot^{B} cont^{S}	expl	yes	yes
StreamDB	lib	expl	impl	datafl	no	no	cont	expl	yes	yes
TSpoon	lib	expl	impl	datafl	no	no	cont	impl	yes	no
Hologres	declarative DSL	impl	impl	workfl	unknown	yes	one-shot	expl	no	no

^{B}, in batch processing; ^{S}, in stream processing.

Table 30. Other Systems: Jobs Compilation and Execution

	Jobs Comp Time	Use Resources Info (Comp)	Granul of Depl	Depl Time	Use Resources Info (Depl)	Manag of Res
Computations on Data Management Systems						
Percolator	reg	static	job	job compil	static	shared
F1	exec	static	job	job compil	static	sys-only
Trinity	exec	static	job	job compil	static	sys-only
New Programming Models						
SDG	exec	static	job	job compil	static	sys-only
TensorFlow	exec	static	task	task activ	static	sys-only
Tangram	exec	dynamic	task	task activ	dynamic	shared
ReactDB	reg	static	job	job compil	static	sys-only
Hybrid Systems						
S-Store	exec	static	job	job compil	static	sys-only
SnappyData	exec	static	task	task activ	dynamic	sys-only
StreamDB	reg	static	job	job compil	static	sys-only
TSpoon	exec	static	job	job compil	static	sys-only
Hologres	exec	static	task	task activ	dynamic	sys-only

developers to register driver programs within the system, which are invoked when a given BigTable column changes. Each driver program is executed on a single process server-side, and can start multiple jobs that read and modify BigTable columns. All these jobs are executed as a group ensuring group atomicity through two-phase commit and group isolation (snapshot isolation) through timestamps. Percolator relies on an external service to obtain valid timestamps to interact with BigTable and saves metadata about running transactions on additional BigTable

Table 31. Other Systems: Data Management

	Elem Struc	Temp Elem	Bus Conn	Bus Impl	Bus Persist	Bus Partition	Bus Repl	Bus Inter
Computations on Data Management Systems								
Percolator	wide-column	yes	direct	net chan + RPC	ephem	yes	no	pull
F1	relational	no	direct	net chan	ephem	yes	no	push
Trinity	typed graph	no	direct	net chan	ephem	yes	no	push
New Programming Models								
SDG	general	no	direct	net chan	ephem	yes	no	push
TensorFlow	tensors	no	direct	net chan	ephem	yes	no	push
Tangram	general	no	mediated	distr fs	persist	yes	yes	hybrid
ReactDB	relational	no	direct	net chan	ephem	yes	no	push
Hybrid Systems								
S-Store	relational	yes	direct +mediated^S	net chan +DB^S	ephem +persist^S	yes	yes+no	push or hybrid
SnappyData	relational	no	mediated	distr fs (+cache)	persist	yes	yes	hybrid
StreamDB	relational	no	direct	net chan	ephem	yes	no	push
TSpoon	general	yes	direct	net chan	ephem	yes	no	push
Hologres	structural	no	direct	net chan	ephem	yes	no	pull

^{S}, in stream processing.

Table 32. Other Systems: State Management

	Elem Struct	Stor Medium	Stor Struct	Task St	Shared St	St Part	Repl	Repl Consist	Repl Prot	Update Propag
Computations on Data Management Systems										
Percolator	wide-col	hybrid	LSM trees	no	yes	yes	backup	n.a.	n.a.	n.a.
F1	relational	service	Spanner	no	yes	yes	yes	strong	leader	op
Trinity	typed graph	hybrid	map	no	yes	yes	yes	weak	leader	unknown
New Programming Models										
SDG	general	mem	user-def	no	yes	yes	no	n.a.	n.a.	n.a.
TensorFlow	general	mem	user-def	no	yes	yes	no	n.a.	n.a.	n.a.
Tangram	general	mem	key-val	no	yes	yes	no	n.a.	n.a.	n.a.
ReactDB	relational	mem	unknown	no	yes	yes	no	n.a.	n.a.	n.a.
Hybrid Systems										
S-Store	relational	mem	unknown	yes^S	yes	yes	n.a.	n.a.	n.a.	n.a.
SnappyData	relational	mem	key-val	yes^S	yes	yes	backup	n.a.	n.a.	n.a.
StreamDB	relational	mem	unknown	no	yes	yes	yes	strong	no readable st	unknown
TSpoon	n.a.	n.a.	n.a.	yes	no	n.a.	n.a.	n.a.	n.a.	n.a.
Hologres	structural	service	LSM tree	no	yes	yes	no	n.a.	n.a.	n.a.

^{S}, in stream processing.

Table 33. Other Systems: Group Atomicity, Group Isolation, Delivery, and Order

	Aborts	Protocol	Assumptions	Level	Impl	Assumptions	Delivery	Nature of Ts	Order
Computations on data management systems									
Percolator	sys+job	blocking	n.a.	blocking	ts	none	exact	event	n.a.
F1	sys+job	blocking	none	blocking	lock+ts	none	exact	no	n.a.
Trinity	n.a.	n.a.	n.a.	blocking	SEQ	1P	most	no	n.a.
New programming models									
SDG	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
TensorFlow	n.a.	n.a.	n.a.	config	barrier	n.a.	exact	no	n.a.
Tangram	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	exact	no	n.a.
ReactDB	sys+job	blocking	none	blocking	ts	none	exact	no	n.a.
Hybrid systems									
S-Store	sys+job	blocking	DC	blocking	ts	DC	exact	ingest	always
SnappyData	sys+job	blocking	none	blocking	lock+ts	none	exact	no or ingest	always
StreamDB	job	coord free	dataflow	coord free	ts	dataflow	exact	no	n.a.
TSpoon	sys+job (task)	blocking	none	conf	conf	none	exact	no or ingest	always
Hologres	sys+job	blocking	none	n.a	n.a.	n.a.	exact	no	n.a.

DC, jobs are deterministic; SEQ, jobs are executed sequentially, with no interleaving; 1P, jobs access a single state portion.

columns. Changes performed during the execution of a driver program may trigger the execution of other driver programs. However, these executions are independent, and atomicity and isolation are not guaranteed across them.

F1. F1 [83] builds a relational database on top of the storage, replication, and transactional features of Spanner. F1 inherits all features of Spanner and adds distributed SQL query evaluation, support for external data sources and sinks, and optimistic transactions. F1 converts SQL queries into a plan

Table 34. Other Systems: Fault Tolerance

	Detection	Scope	Comput Recov	State Recov	Guarantees for State	Assumptions
Computations on Data Management Systems						
Percolator	lead-work	shared st	n.a.	log+repl	same	STOR
F1	lead-work	shared st	n.a.	log+checkp+repl	same	STOR
Trinity	p2p	comput+shared st	job	checkp+repl	none	none
New Programming Models						
SDG	lead-work	comput+task st	job	log+checkp	valid	REPLAY
TensorFlow	n.a.	shared st	n.a.	checkp	valid	STOR
Tangram	lead-work	comput+shared st	task	checkp	valid	REPLAY
ReactDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Hybrid systems						
S-Store	lead-work	comput+shared st	job	log+checkp	valid or same	REPLAY
SnappyData	p2p	comput+task st+shared st	task	checkp+repl	valid or same	REPLAY
StreamDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
TSpoon	lead-work	comput+task st	job	log+checkp	valid or same	REPLAY
Hologres	lead-work	shared st	unknown	log	same	STOR

STOR, storage layer is durable; REPLAY, sources are replayable.

Table 35. Other Systems: Dynamic Reconfiguration

	Goal	Automated	State Migration	Task Migration	Add /Del Slots	Restart
Computations on Data Management Systems						
Percolator	load balan	yes	yes	n.a.	yes	no
F1	change schema+load balan	yes	yes	no	yes	no
Trinity	avail	yes	yes	yes	unknown	no
New Programming Models						
SDG	elast	yes	yes	yes	yes	no
TensorFlow	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Tangram	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
ReactDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
Hybrid Systems						
S-Store	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
SnappyData	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
StreamDB	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
TSpoon	elast	no	yes	yes	yes	yes
Hologres	load balan+elast	yes	yes	yes	yes	no

that can be either fully executed on a single coordinator worker or include dataflow sub-plans that are executed on multiple workers and managed by the coordinator. This execution mode mimics dataflow DPSs. To optimize read-intensive, analytical jobs, F1 introduces optimistic transactions. They are split into two phases: the first one reads all data needed for processing, and the second one attempts to write results. The read phase does not block any other transaction, and so it can be arbitrary long (as in the case of complex data analytics). The subsequent write phase will complete only if no conflicting updates from other transactions occurred during the read phase.

Trinity. Trinity [80] is a graph data store developed at Microsoft with similar characteristics as TAO in terms of data model (typed graphs), storage model (in-memory key-value store backed up in a shared distributed filesystem), and guarantees (weak consistency without transactions). The main distinguishing feature of Trinity is the ability to perform more complex computations on graphs, including those that require traversing multiple hops of the graph (e.g., graph pattern matching) and iterative analytical jobs (e.g., vertex-centric computations, as introduced by Pregel [66]). To support these computations, Trinity lets users define different communication protocols that govern data exchange over the data bus during job execution. For instance, data may be buffered and aggregated at sender side or at receiver side. For fault tolerance, Trinity stores the association between shared state portions and workers on the distributed filesystem, and updates it in the case of failure. It also uses checkpoints within long-lasting iterative computations to resume them in the case of failure.

D.2 New Programming Models

D.2.1 Stateful Dataflow SDG. SDG [43] is a programming model that extracts a dataflow graph of computation from imperative code (Java programs). In SDG, developers write driver programs

that include mutable state and methods to access and modify it. Code annotations are used to specify state access patterns within methods. The program executor (a client-side compiler) analyzes the program to extract state elements and task elements, representing shared state and data-parallel tasks in our model. If possible, state elements are partitioned across workers. Similarly, task elements are converted into multiple concrete tasks, each accessing one single state element from the shared state portion of the worker it is deployed on. For instance, consider a program including a matrix of numbers and two methods to update a value in the matrix and to return the sum of a row. The matrix would be converted into a state element partitioned by row (since both methods can work on individual rows). Each method would be converted into a data-parallel task, with one instance of the task per matrix partition. State elements that cannot be partitioned are replicated in each worker, and the programming model supports user-defined functions to merge changes applied to different replicas. In terms of execution, SDGs are similar to stream processing systems such as Storm or Flink: jobs are continuous, and tasks communicate through direct TCP channels. SDGs rely on periodic snapshots and re-execution for fault tolerance: the same mechanism is adopted to dynamically scale in and out individual task elements depending on the input load they receive [26].

TensorFlow. TensorFlow [1] is a system for large-scale machine learning that extends the dataflow model with explicit shared mutable state. Jobs represent machine learning models and include operations (data transformations) and variables (shared mutable state elements representing the parameters of the machine learning model). Frequently, jobs are iterative and update variables at each iteration. The specific application scenario does not require strong consistency guarantees for accessing shared state, so tasks are allowed to execute and read/write variables asynchronously. If needed, TensorFlow permits some form of barrier synchronization—for instance, to guarantee that all tasks perform an iteration step using a given value of variables before they get updated with the results of that step. TensorFlow tasks can be executed on heterogeneous devices (e.g., hardware accelerators), and users can express explicit placement constraints. Since version 2, part of the dataflow plan may be defined at runtime, meaning that tasks can dynamically define and spawn downstream tasks based on the input data. Variables can be periodically checkpointed to durable storage for fault tolerance. In the case of failure, workers can be restarted and they restore the latest checkpointing available, with no further consistency guarantees. TensorFlow has been conceived from the very beginning as a distributed platform, but many other libraries for machine learning, initially designed for a single machine, inherited its stateful dataflow execution model: the most prominent and most widely adopted example is PyTorch.[14]

Tangram. Tangram [50] is a data processing framework that extends the dataflow model with explicit shared mutable state. It implements task-based deployment but allows tasks to access and update an in-memory key-value store as part of their execution, which enables optimizing algorithms that benefit from fine-grained updates of intermediate states of computations (e.g., iterative algorithms or graph processing algorithms). By analyzing the execution plan, Tangram can understand which parts of the computation depend on mutable state and which parts do not, and optimizes fault tolerance for the job at hand. Immutable data is recomputed using the same (lineage) approach of MapReduce and Spark, thus re-executing only tasks that are necessary to rebuild the data. Mutable state is periodically checkpointed. In general, Tangram does not provide group atomicity or isolation for state: simultaneous accesses to the same state portions from multiple tasks may be executed in any order.

[14] https://pytorch.org.

D.2.2 Relational Actors ReactDB. ReactDB [79] extends the actor-based programming model with database concepts such as relational tables, declarative queries, and transactional execution semantics. ReactDB builds on the abstraction of reactors, which are logical actors that embed state in the form of relational tables. Each reactor can query its internal state using a declarative language (SQL) or can explicitly invoke other reactors. Invocations across reactors are asynchronous and retain transactional semantics: clients invoke a root reactor and all invocations it makes belong to the same root-level transaction, and are atomic and isolated. The core idea of ReactDB is that developers control data partitioning across reactors and distributed execution: on one extreme, they can place all state in a single reactor and mimic a centralized database; on the other extreme, they can assign a single table (or a single partition of a table) to each reactor, which maximizes distributed execution. With this model, the execution plan is partly implicit (within a single reactor) and partly explicit (calls between reactors). ReactDB is currently a research prototype. As such, it lacks a distributed implementation, replication, fault tolerance, and dynamic reconfiguration mechanisms.

D.3 Hybrid Systems

S-Store. S-Store [27] integrates stream processing capabilities within a transactional database system. It builds on H-Store [88], the research prototype that later evolved into VoltDB [89], and adopts the same approach to implement transactional guarantees with limited overhead. It extends H-Store by enabling stream processing jobs, represented as a dataflow graph of tasks that may access local task state as part of their processing. S-Store exploits the database state to implement the shared state (visible to all tasks), the task state (visible only to individual tasks of stream processing jobs), and the data bus (that stream processing tasks use to exchange data streams). Input data (for streaming jobs) and transaction invocations (for data management jobs) are handled by the same engine, which schedules task execution ensuring that dataflow order is preserved for stream processing jobs. S-Store supports two fault tolerance mechanisms, one ensuring same state recovery through a command log and a periodic snapshot, and one ensuring valid state by replaying streaming data. Although the S-Store prototype is not distributed, we included it in the survey for its original integration of data management and data stream processing, and because its core concepts can easily lead to a distributed implementation.

SnappyData. SnappyData [70] aims to unify data processing abstractions (both for static and for streaming data) with mutable shared state. To do so, it builds on Spark and Spark Streaming as job execution engines but extends them to enable writing to a distributed key-value store. Users write jobs as declarative SQL queries. SnappyData analyzes jobs and classifies them as lightweight (transactional) or heavy (analytical): in the first case, it directly interacts with the underlying key-value store, whereas in the latter case, it compiles them into an Spark dataflow execution plan. Based on the application at hand, users can decide how to store shared state (e.g., in row or in column format) and how to partition and replicate it, and how to associate shared state portions to workers, to maximize co-location of state elements that are frequently accessed together. Interestingly, SnappyData also supports probabilistic data and query models that may sacrifice precision to reduce latency. SnappyData supports group atomicity and group isolation (up to the repeatable-read isolation model) using two-phase commit and multi-version concurrency control. Fault detection is performed in a distributed manner, to avoid single points of failures. Fault recovery is based on replication of the key-value store, which is also used to persist the checkpoints of the data used by Spark during long-running one-shot and continuous jobs.

StreamDB. StreamDB [31] integrates shared state and transactional semantics within a distributed stream processing system. From stream processing systems, StreamDB inherits a dataflow

execution plan with job-level deployment. As in DMSs, tasks have access to a shared state, which represents relational tables that can be replicated and partitioned horizontally and/or vertically across workers. Each worker is responsible for reading and updating its portion of the shared state. Input requests represent invocations of jobs: they are timestamped when received by the system and each worker executes tasks from multiple jobs in timestamp order, thus ensuring that jobs are executed in a sequential order without using explicit locks (group isolation). The results of a job are provided to sinks that subscribed to them. StreamDB is a research prototype and currently requires developers to explicitly define how to partition the shared state and to write the dataflow execution plan for all the jobs. It lacks fault tolerance and reconfiguration mechanisms.

TSpoon. Like StreamDB, TSpoon [5] also integrates data management capabilities within a distributed stream processing system. Unlike StreamDB, it does not provide a shared state. Instead, it builds on the programming and execution models of Flink and enriches them with (i) the possibility to read (query) task state on demand and (ii) transactional guarantees in the access to task state. TSpoon considers each input data element as a notification of some change occurred in the environment in which the system operates. Developers can identify portions of the dataflow graph (denoted as transactional sub-graphs) that need to be read and modified in a consistent way, meaning that each change should be reflected in all task states or none (group atomicity) and the effects of changes should not overlap in unexpected ways (group isolation). TSpoon implements atomicity and isolation by decorating the dataflow graph with additional operators that act as transaction managers. It supports atomicity under the assumption that jobs abort either due to a system failure or due to some inconsistency during the update to the state of individual tasks. It supports different levels of isolation (from read committed to serializable) with different tradeoffs between guarantees and runtime overhead. It supports different isolation protocols, both based on locks and on timestamps.

Hologres. Hologres [53] is a system developed at Alibaba to integrate analytical (long-running) and interactive (lightweight) jobs. The system is designed to support high volume data ingestion from external sources, continuously compute derived information, store it into a shared state, and make it available to external sinks. Hologres uses a modular approach, where the storage layer is decoupled from the processing layer and delegated to external services (e.g., a distributed filesystem). It adopts a leader-worker approach: the shared state is partitioned across workers, and each worker stores a log of updates for the partition it is responsible for and an in-memory store that is periodically flushed on the durable storage service. Hologres supports a structured data model, where data is organized into tables that can be stored row-wise, column-wise, or both depending on the access pattern, element by element rather than for range scan or aggregation. A distinctive feature of the system is its scheduling mechanism. Each job is decomposed into tasks, and jobs execution is orchestrated by a coordinator, which assigns tasks to workers based on their current load and their priority. For instance, analytical tasks may be assigned a lower priority to guarantee low response time for interactive queries. Hologres supports group atomicity using two-phase commit, but it does not support group isolation. Replication of the shared state is not currently implemented. Fault tolerance relies on logging and checkpointing and assumes that the storage layer is durable. Dynamic reconfiguration is a design concern, and includes migration of shared state portions but also reconfiguration of execution slots to enforce load balancing or user-defined specification of priorities.

E RELATED SURVEYS AND STUDIES

This section presents related surveys and studies that complement our work, putting it in a broader context.

The book by Kleppmann [54] presents the main design and implementation strategies to build data-intensive applications. It is complementary to our work, as it discusses key design concepts in greater detail, but it does not provide a unifying model nor a taxonomy of systems.

The work on highly available transactions by Bailis et al. [15] provides a conceptual framework that unifies various guarantees associated with data management in distributed systems. It influenced our discussion of task grouping and replication management.

Various works by Stonebraker and colleagues [85–88] guided our classification of DMSs and the terminology we adopted for the NoSQL and NewSQL classes. In this area, Davoudian et al. [36] present a survey of NoSQL stores, which expands some of the concepts presented in this article, particularly those related to data models and storage structures. Other works explore approaches dedicated to specific data models, such as time-series management [52] and graph processing [68], or strategies to adapt to multiple data models [63].

In the domain of DPSs, the dataflow model has received much attention, with work focusing on state management [93], handling of iterations [44], parallelization and elasticity strategies [77], and optimizations for stream processing [48].

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of OSDI 2016*.

[2] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. 2009. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. *Proceedings of the VLDB Endowment* 2, 1 (2009), 922–933.

[3] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, et al. 2020. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3181–3194.

[4] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of ICDE 2000*. IEEE, Los Alamitos, CA.

[5] L. Affetti, A. Margara, and G. Cugola. 2020. TSpoon: Transactions on a stream processor. *Journal of Parallel and Distributed Computing* 140 (2020), 65–79.

[6] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. 2015. Challenges to adopting stronger consistency at scale. In *Proceedings of HotOS 2015*.

[7] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. 2013. MillWheel: Fault-tolerant stream processing at Internet scale. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1033–1044.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, et al. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.

[9] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, et al. 2014. AsterixDB: A scalable, open source BDMS. *Proceedings of the VLDB Endowment* 7, 14 (2014).

[10] J. C. Anderson, J. Lehnardt, and N. Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media.

[11] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, et al. 2019. Socrates: The new SQL server in the cloud. In *Proceedings of SIGMOD 2019*. ACM, New York, NY.

[12] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, et al. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of SIGMOD 2015*. ACM, New York, NY.

[13] J. Arulraj and A. Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of SIGMOD 2017*. ACM, New York, NY.

[14] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, et al. 2017. Spanner: Becoming a SQL system. In *Proceedings of SIGMOD 2017*. ACM, New York, NY.

[15] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. 2013. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment* 7, 3 (2013), 181–192.

[16] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. 2013. Tango: Distributed data structures over a shared log. In *Proceedings of SOSP 2013*. ACM, New York, NY.

[17] L. Baresi, A. Leva, and G. Quattrocchi. 2021. Fine-grained dynamic resource allocation for big-data applications. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1668–1682.

[18] B. Bejeck. 2018. *Kafka Streams in Action: Real-Time Apps and Microservices with the Kafka Streams API*. Manning.

[19] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of ICDE 2011*. IEEE, Los Alamitos, CA.
[20] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, et al. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of ATC 2013*.
[21] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. 2010. HaLoop: Efficient iterative data processing on large clusters. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 285–296.
[22] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of SIGMOD 2020*. ACM, New York, NY.
[23] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, et al. 2019. Apache Hive: From MapReduce to enterprise-grade big data warehousing. In *Proceedings of SIGMOD 2019*. ACM, New York, NY.
[24] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink™: Stream and batch processing in a single engine. *Data Engineering Bulletin* 38, 4 (2015), 28–38.
[25] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo. 2022. Run-time adaptation of data stream processing systems: The state of the art. *ACM Computing Surveys* 54, 11s (2022), Article 237, 36 pages.
[26] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of SIGMOD 2013*. ACM, New York, NY.
[27] U. Cetintemel, J. Du, T. Kraska, S. Madden, D. Maier, J. Meehan, A. Pavlo, M. Stonebraker, E. Sutherland, and N. Tatbul. 2014. S-Store: A streaming NewSQL system for big velocity applications. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1633–1636.
[28] K. M. Chandy and L. Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems* 3, 1 (1985), 63–75.
[29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2 (2008), Article 4, 26 pages.
[30] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng. 2018. G-Miner: An efficient task-oriented graph mining system. In *Proceedings of EuroSys 2018*. ACM, New York, NY.
[31] H. Chen and M. Migliavacca. 2018. StreamDB: A unified data management system for service-based cloud application. In *Proceedings of SCC 2018*. IEEE, Los Alamitos, CA.
[32] K. Chodorow. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly Media.
[33] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1277–1288.
[34] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems* 31, 3 (2013), Article 8, 22 pages.
[35] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, et al. 2013. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1150–1161.
[36] A. Davoudian, L. Chen, and M. Liu. 2018. A survey on NoSQL stores. *ACM Computing Surveys* 51, 2 (2018), Article 40, 43 pages.
[37] A. Davoudian and M. Liu. 2020. Big data systems: A software engineering perspective. *ACM Computing Surveys* 53, 5 (2020), Article 110, 39 pages.
[38] J. Dean and S. Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* 51, 1 (2008), 107–113.
[39] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of SOSP 2007*. ACM, New York, NY.
[40] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Rethinking stateful stream processing with RDMA. In *Proceedings of SIGMOD 2022*. ACM, New York, NY, 1078–1092.
[41] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. 2014. FaRM: Fast remote memory. In *Proceedings of NSDI 2014*.
[42] J. Fang, Y. Mulder, J. Hidders, J. Lee, and H. P. Hofstee. 2020. In-memory database acceleration on FPGAs: A survey. *VLDP Journal* 29, 1 (2020), 33–59.
[43] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. 2014. Making state explicit for imperative big data processing. In *Proceedings of ATC 2014*.
[44] G. E. Gévay, J. Soto, and V. Markl. 2021. Handling iterations in distributed dataflow systems. *ACM Computing Surveys* 54, 9 (2021), Article 199, 38 pages.
[45] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of OSDI 2012*.

[46] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of OSDI 2014*.
[47] M. Guerriero, D. A. Tamburri, and E. Di Nitto. 2021. StreamGen: Model-driven development of distributed streaming applications. *ACM Transactions on Software Engineering and Methodology* 30, 1 (2021), Article 1, 30 pages.
[48] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys* 46, 4 (2014), Article 46, 34 pages.
[49] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee. 2021. FPGA acceleration for big data analytics: Challenges and opportunities. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 30–47.
[50] Y. Huang, X. Yan, G. Jiang, T. Jin, J. Cheng, A. Xu, Z. Liu, and S. Tu. 2019. Tangram: Bridging immutable and mutable abstractions for distributed data analytics. In *Proceedings of ATC 2019*.
[51] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys 2007*. ACM, New York, NY.
[52] S. K. Jensen, T. B. Pedersen, and C. Thomsen. 2017. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering* 29, 11 (2017), 2581–2600.
[53] X. Jiang, Y. Hu, Y. Xiang, G. Jiang, X. Jin, C. Xia, W. Jiang, et al. 2020. Alibaba Hologres: A cloud-native service for hybrid serving/analytical processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3272–3284.
[54] M. Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.
[55] J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of NetDB 2011*.
[56] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. 2015. Twitter Heron: Stream processing at scale. In *Proceedings of SIGMOD 2015*. ACM, New York, NY.
[57] A. Lakshman and P. Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (2010), 35–40.
[58] R. Lee, M. Zhou, C. Li, S. Hu, J. Teng, D. Li, and X. Zhang. 2021. The art of balance: A RateupDB experience of building a CPU/GPU hybrid database product. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2999–3013.
[59] Justin Levandoski, David Lomet, and Kevin Keliang Zhao. 2011. Deuteronomy: Transaction support for cloud data. In *Proceedings of CIDR 2011*.
[60] J. Lin. 2017. The lambda and the kappa. *IEEE Internet Computing* 21, 5 (2017), 60–66.
[61] G. Liu, L. Chen, and S. Chen. 2021. Zen: A high-throughput log-free OLTP engine for non-volatile main memory. *Proceedings of the VLDB Endowment* 14, 5 (2021), 835–848.
[62] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. 2012. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.
[63] J. Lu and I. Holubová. 2019. Multi-model databases: A new journey to handle the variety of data. *ACM Computing Surveys* 52, 3 (2019), Article 55, 38 pages.
[64] T. Macedo and F. Oliveira. 2011. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. O'Reilly Media.
[65] S. Maiyya, F. Nawab, D. Agrawal, and A. El Abbadi. 2019. Unifying consensus and atomic commitment for effective cloud data management. *Proceedings of the VLDB Endowment* 12, 5 (2019), 611–623.
[66] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of SIGMOD 2010*. ACM, New York, NY.
[67] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *Proceedings of ICDE 2014*. IEEE, Los Alamitos, CA.
[68] R. Ryan McCune, T. Weninger, and G. Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys* 48, 2 (2015), Article 25, 39 pages.
[69] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, et al. 2016. MLlib: Machine learning in apache spark. *Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[70] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. 2017. SnappyData: A unified cluster for streaming, transactions and interactive analytics. In *Proceedings of CIDR 2017*.
[71] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of SOSP 2013*. ACM, New York, NY.
[72] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. 2011. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of NSDI 2011*.
[73] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, et al. 2013. Scaling memcache at Facebook. In *Proceedings of NSDI 2013*.
[74] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. 2017. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1634–1645.
[75] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.

[76] D. Peng and F. Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of OSDI 2010.*
[77] H. Röger and R. Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys* 52, 2 (2019), Article 36, 37 pages.
[78] M. J. Sax, G. Wang, M. Weidlich, and J. C. Freytag. 2018. Streams and tables: Two sides of the same coin. In *Proceedings of BIRTE 2018.* ACM, New York, NY.
[79] V. Shah and M. Antonio Vaz Salles. 2018. Reactors: A case for predictable, virtualized actor database systems. In *Proceedings of SIGMOD 2018.* ACM, New York, NY.
[80] B. Shao, H. Wang, and Y. Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of SIGMOD 2013.* ACM, New York, NY.
[81] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems.* Lecture Notes in Computer Science, Vol. 6976. Springer, 386–400.
[82] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge computing: Vision and challenges. *IEEE Internet of Things Journal* 3, 5 (2016), 637–646.
[83] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, et al. 2013. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1068–1079.
[84] V. Srinivasan, B. Bulkowski, W. L. Chu, S. Sayyaparaju, A. Gooding, R. Iyer, A. Shinde, and T. Lopatic. 2016. Aerospike: Architecture of a real-time operational DBMS. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1389–1400.
[85] M. Stonebraker. 2010. SQL databases v. NoSQL databases. *Communications of the ACM* 53, 4 (2010), 10–11.
[86] M. Stonebraker. 2012. New opportunities for new SQL. *Communications of the ACM* 55, 11 (2012), 10-11.
[87] M. Stonebraker and U. Cetintemel. 2005. "One size fits all": An idea whose time has come and gone. In *Proceedings of ICDE 2005.* IEEE, Los Alamitos, CA.
[88] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. 2007. The end of an architectural era: (It's time for a complete rewrite). In *Proceedings of VLDB 2007.*
[89] M. Stonebraker and A. Weisberg. 2013. The VoltDB main memory DBMS. *IEEE Data Engineering Bulletin* 36, 2 (2013), 21–27.
[90] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, et al. 2020. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of SIGMOD 2020.* ACM, New York, NY.
[91] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of SOSP 2015.* ACM, New York, NY.
[92] A. Thomson, T. Diamond, S. C. Weng, K. Ren, P. Shao, and D. J. Abadi. 2012. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of SIGMOD 2012.* ACM, New York, NY.
[93] Q.-C. To, J. Soto, and V. Markl. 2018. A survey of state management in big data processing systems. *VLDB Journal* 27, 6 (2018), 847–872.
[94] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, et al. 2014. Storm@Twitter. In *Proceedings of SIGMOD 2014.* ACM, New York, NY.
[95] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. 2017. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of SIGMOD 2017.* ACM, New York, NY.
[96] A. Visheratin, A. Struckov, S. Yufa, A. Muratov, D. Nasonov, N. Butakov, Y. Kuznetsov, and M. May. 2020. Peregreen—Modular database for efficient storage of historical time series in cloud environments. In *Proceedings of ATC 2020.*
[97] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. 2010. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys 2010.* ACM, New York, NY.
[98] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of SOSP 2013.* ACM, New York, NY.
[99] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, et al. 2016. Apache Spark: A unified engine for big data processing. *Communications of the ACM* 59, 11 (2016), 56–65.
[100] J. Zhou, M. Xu, A. Shraer, B. Namasivayam, A. Miller, E. Tschannen, S. Atherton, A. J. Beamon, R. Sears, and J. Leach. 2021. FoundationDB: A distributed unbundled transactional key value store. In *Proceedings of SIGMOD 2021.* ACM, New York, NY.
[101] T. Zhu, Z. Zhao, F. Li, W. Qian, A. Zhou, D. Xie, R. Stutsman, H. Li, and H. Hu. 2019. Solar: Toward a shared-everything database on distributed log-structured storage. *ACM Transactions on Storage* 15, 2 (2019), Article 11, 26 pages.
[102] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A fast and cost-efficient storage engine using DRAM, NVMe, and RDMA. In *Proceedings of SIGMOD 2022.* ACM, New York, NY, 685–699.

Received 19 March 2022; revised 3 April 2023; accepted 31 May 2023