Dependency-Driven Analytics: a Compass for Uncharted Data Oceans

Ruslan Mavlyutov
University of Fribourg
ruslan.mavlyutov@unifr.ch

Carlo Curino
Microsoft
ccurino@microsoft.com

Boris Asipov
Microsoft
basipov@microsoft.com

Philippe Cudre-Mauroux
University of Fribourg
pcm@unifr.ch

ABSTRACT

In this paper, we predict the rise of *Dependency-Driven Analytics (DDA)*, a new class of data analytics designed to cope with growing volumes of unstructured data. DDA drastically reduces the cognitive burden of data analysis by systematically leveraging a compact *dependency graph* derived from the raw data. The computational cost associated with the analysis is also reduced substantially, as the graph acts as an index for commonly accessed data items. We built a system supporting DDA using off-the-shelf Big Data and graph DB technologies, and deployed it in production at Microsoft to support the analysis of the exhaust of our Big Data infrastructure producing petabytes of system logs daily. The dependency graph in this setting captures lineage information among jobs and files and is used to guide the analysis of telemetry data. We qualitatively discuss the improvement over the brute-force analytics our users used to perform by considering a series of practical applications, including: job auditing and compliance, automated SLO extraction of recurring tasks, and global job ranking. We conclude by discussing the shortcomings of our current implementation and by presenting some of the open research challenges for Dependency-Driven Analytics that we plan to tackle next.

1. INTRODUCTION

Large companies operate increasingly complex infrastructures to collect, store, and analyze vast amounts of data. At Microsoft, our infrastructure consists of a number of very large clusters (up to 50k nodes each) serving thousands of data scientists, running hundreds of thousands of jobs daily, and accessing billions of files. The exhaust of this infrastructure consists of *petabytes of system logs daily*. These logs faithfully capture all relevant aspects of our infrastructure, applications, and data life-cycle. However, their sheer size and loose structure make them very challenging and expensive to access and analyze.

To address this problem, we built a system—currently in production at Microsoft—that automatically extracts from the logs a compact, higher-level graph representation capturing entities (e.g., jobs, files, machines) and their dependencies (e.g., job reads file), and provide this to our users as a *compass* to navigate this ocean of raw data. The result is a declarative, directed and informed access to the logs that saves users hours of development time, reduces the computational cost of common analyses by orders of magnitude, and enables interactive exploration of the logs—see Figure 1.

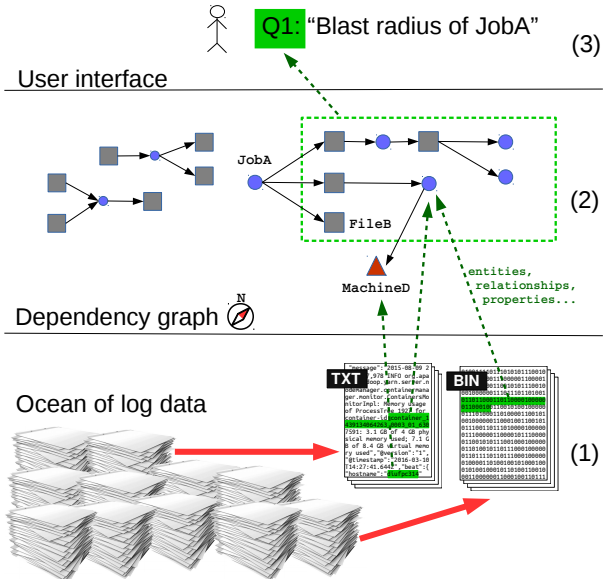

Figure 1: Dependency-Driven Analytics: a log-analytics example. (1) Petabytes of daily logs; (2) Dependency Graph that represents a lightweight "skeleton" of the logs used for navigation; (3) User-level queries return bytes of aggregated data.

A key tenet of our paper is that the log-analysis scenario presented above is just one example of a broadening class of data analytics we call *Dependency-Driven Analytics (DDA)*. In DDA, raw data are (automatically) preprocessed to extract a semantically rich and compact graph structure that captures the key entities in the data as well as their relationships—often dependencies among data items, hence our choice of name. We expect DDA to emerge in settings with *massive volumes of loosely structured data, produced by uncoordinated parties*. In this context, we argue that two costs become prohibitive: 1) cognitive costs for users to understand the many quirks and local semantics of the data and to parse/analyze such raw data, and 2) computational costs when scanning/filtering/joining the data in their raw form.

8th Biennial Conference on Innovative Data Systems Research (CIDR '17) January 8-11, 2017 , Chaminade, California, USA.

Anecdotally, we observed these issues in production settings at Microsoft for the log-analytics example introduced above; Users and cluster operators alike were forced to write complex processing scripts—and invest substantial computational power—to extract usable insight from these raw textual logs. These efforts became unattainable in the long run, as the software components producing the logs are owned by multiple teams and keep evolving independently. Users often gave up or employed rough, indirect measurements to estimate quantities that were precisely reported in the logs. Section 2 discusses DDA as a pattern for data analytics, and presents key application scenarios.

For those reasons, we embarked in building a system to support DDA named Guider. Guider is built from off-the-shelf Big Data and graph DB tools to support the access to very large log data in a more effective and systematic way. Guider is in production today and powers several key use cases at Microsoft, both in production and research stages (the architecture and use cases for Guider are discussed in Section 3).

Our vision for DDA goes well beyond the initial capabilities of our system, that was custom built for the log analysis instance of DDA. We expect the process of extracting the dependency graph from the raw data to be eventually fully automated, leveraging techniques akin to entity extraction and deduplication. The graph itself is potentially very large and calls for improved scale-out graph and DB technologies—in Guider we are forced to use multiple narrow projections of the original graph to support different applications scenarios. Novel language support is also required to effectively access the graph, relational and unstructured data in a unified way. We present a research agenda in support of DDA in Section 4.

We begin by introducing the notion of Dependency-Driven Analytics in more detail below.

2. DEPENDENCY-DRIVEN ANALYTICS

Dependency-Driven Analytics (DDA) is a new pattern in data analytics in which massive volumes of largely unstructured data are accessed through a compact and semantically-rich overlay structure: the *dependency graph*. The dependency graph fulfills two roles: it serves as 1) a conceptual map of the domain being modeled as well as 2) an index for some of the underlying raw data.

Conceptual Map The dependency graph is a structure overlaid on the raw data, which takes the form of a highly specialized graph and acts as a cognitive guide for the analytics process. Both its nodes and edges are typed. Nodes represent the main *entities* (e.g., users, jobs, tasks, machines or files) that are captured by the raw data, while edges represent their relationships or *dependencies* (e.g., a job reads a file, or a task executes on a machine). Instead of sifting through massive volumes of shapeless data, the user of a DDA system can navigate this conceptual map, and quickly correlate data items which are very far apart in the data but directly or indirectly connected through the dependency graph. Our experience indicates that this *drastically* lowers the cognitive cost of accessing the unstructured data. The dependency graph can be seen as a knowledge base or ontology derived from the raw data; One major difference from standard Knowledge Base Construction [15], however, is that in DDA we do not aim at extracting all pieces of information from the logs, but rather we aim at building a sparse skeleton of key pieces of information that users leverage to traverse and access the raw data.

Index The second key role of the dependency graph is to index key portions of the logs and allow for highly efficient access patterns for DDA, avoiding intractably expensive scans and joins (recall we deal with potentially petabytes of raw data, created by several independent pieces of software). For some of the queries, the graph itself will be sufficient (i.e., the dependency graph is a covering index for some queries), allowing for interactive querying.

We postpone to Section 3 the description of how the graph is ingested, stored, and queried. In the remainder of this section, we introduce three application scenarios that we believe are a good match for DDA.

2.1 Application Scenarios

DDA provides a pattern of analytics that is most suitable under the following conditions: 1) analyses are mostly localized, 2) large-scale data, 3) mostly unstructured data, and 4) data produced by loosely coordinating parties. The combination of these properties makes standard Big Data solutions largely ineffective. The cognitive/development cost of understanding all data formats, of parsing and finally analyzing the data is prohibitive. Similarly, the cost of performing scans/joins/aggregates is excessive with respect to the relatively localized type of analysis. We present three application scenarios for DDA below, which all exhibit the properties described above.

2.1.1 Infrastructure Logs Analysis

The many systems that make up the Microsoft Big Data infrastructure produce a massive volume of system-level logs (petabytes daily), which capture every aspect of our infrastructure, data, and applications lifecycle. These systems are owned and operated by multiple teams and keep evolving independently. Users and cluster operators alike are forced to write complex processing scripts, and to invest substantial computational power, to extract insight from those raw textual logs. The resulting "cooked" data are surfaced in purpose-built monitoring tools, dashboards, and alert systems.

This log analytics scenario is the one we target with our production system and it is discussed in greater detail in Section 3.

2.1.2 Enterprise Search

A large fraction of the information created and processed by large companies is unstructured. Often, the same entities (e.g., customers, transactions or employees) are found in dozens of different enterprise systems ranging from email or document servers to calendars, instant messaging systems and internal social networks. Traditional enterprise search solutions build an inverted index from all sources and let end users query the resulting system through keywords. Such solutions are however incapable of reconstructing the context in which the entities operate, for example to link transactions to their customers, employees forming a team, or track the approval of a spreadsheet. A DDA solution would precisely answer those needs, by modeling key entities in the enterprise and by reconstructing their interdependencies through a dependency graph for further analysis. As extracting from this type of documents is likely more challenging than from more regular infrastructure logs, natural language text analysis solutions (such as [15]) could be leveraged in this context.

2.1.3 Internet of Things (IoT)

Embedded and mobile devices are increasingly connected, forming webs of physical things[1]. Large Webs of things can interconnect thousands of heterogeneous devices at the application layer. Each device, however, maintains its own logs (browse logs, commits, errors, etc.) Building analytics on top of such systems requires to identify, and then match entities across devices. One example is building a graph of users across devices for advertis-

[1]http://webofthings.org/

ing purposes[2]. Further examples include following a single event across multiple sensors, and debugging or auditing complex webs of things. All cases directly match the DDA process sketched above, where a dependency graph has to be modeled, and then instantiated by collecting and analyzing massive amounts of heterogeneous and unstructured data.

Next, we present a practical implementation of DDA for infrastructure log analysis.

3. Guider: A PRACTICAL TAKE ON DDA

Historically, Guider was built to support only auditing and compliance applications (see Section 3.2.3). Therefore, it employed a dependency graph focused on tracking provenance information. Provenance is tracked at different levels of granularity (pipeline/-datasets, job/file, and task/column), allowing for different levels of inspection. The graph also captured all ingress/egress operations from outside the cluster. Over time, we realized that focusing only on provenance was restrictive, and we started to track telemetry information for every entity that appeared in the dependency graph. For example, the total CPU hours consumed by JobA over time, or the timestamp when FileB is read by JobC. This enabled a much broader set of analyses, which we discuss in the second part of Section 3.2, and provided the seed for our DDA vision. At the time of writing, the deployed version of Guider sits somewhere in between its original tightly focused mission and the more ambitious DDA vision.

3.1 Architecture overview

With reference to Figure 2, Guider's architecture is organized as follows:

Dependency Definition: an offline modeling step akin to schema design for relational databases or ontology definition for the Semantic Web, where key entities and their dependencies are manually or automatically defined. Each entity/dependency is associated with a set of extraction rules, instructing the DDA runtime on how to derive identifiers, relations, and attributes from the raw data. This step can be challenging for free-form textual data [15], but is typically easier for unstructured data produced by automated systems (e.g., IoT or infrastructure logs), whose highly regular structure is easier to learn from and more conducive of rule-based extraction. In our current implementation of Guider this step is completely manual, however in Section 4.2 we discuss ways to automate this process.

As an example of extraction rules, we show how the total CPU hours consumed by a job can be derived from logs. The components of our BigData infrastructure track job execution at each worker node in our clusters, collecting telemetry at the granularity of each task/process that belong to a job. Specifically, infrastructure daemons running on each node monitor and log to local disk all state transitions of a job's task, and timestamps them precisely. The two main transitions we care about are the ones marking the start and end times of a task, logged respectively in a `ProcStarted` and `ProcEnded` files. From this, we can then derive the duration of each task, which can be then aggregated to compute the total processing time consumed by the job.

Since this information is potentially scattered over 100k individual files (2 files for each of the over 50k nodes that make up our larger clusters), these logs are continuously uploaded to our DFS, where they can be processed further. The rule takes the form of a SQL-like script [5] shown below. This script collects and parses the logs. The textual logs are transformed into a tabular format by means of an `EventExtractor` C# UDF, which behaves like a very robust parser (automatically correcting or skipping oddly formed lines). The tabular representations are then joined, and aggregated into a <JobId, procHours> summary table that is stored as an intermediate file `processingHours.csv`.

```
extStart = EXTRACT *
  FROM "ProcStarted_%Y%m%d.log"
  USING EventExtractor("ProcStarted");

startData = SELECT ProcessGuid AS ProcessId,
    CurrentTimeStamp.Value AS StartTime,
    JobGuid AS JobId
  FROM   extStart
  WHERE  ProcessGuid != null AND
    JobGuid != null AND
    CurrentTimeStamp.HasValue;

...

procH = SELECT SUM((End - Start).TotalMs)
    /1000/3600 AS procHours,
    endData.JobId,
    EndTime.Date AS Date
  FROM  startData INNER JOIN endData ON
    startData.ProcessId == endData.ProcessId AND
    startData.JobId == endData.JobId
  GROUP BY JobId, Date;

OUTPUT (SELECT JobId, procHours FROM procH)
  TO "processingHours.csv";
```

Dependency Graph Extraction: this is a runtime phase in which the rules described above are applied to the raw data. This is akin to an ETL step for a relational data warehouse, but differs as the dependency graph is a sparse structure, which only acts as a guide to access the raw data. In Guider today, we solve this through the notion of a *data scanner*, a batch process running daily. Each scanner processes the logs of a different system component and collects a different portion of the dependency graph (e.g., we have a job telemetry scanner analyzing solely job-level telemetry from the application logs, or a file-access scanner, consuming filesystem logs and collecting all read/write events for each file).

The script shown above is one of the scripts used by the job telemetry scanner. At our scale, the "small" input logs from the example above consists of multiple TBs of daily data. An extended version of the above script runs in 6-7 min by employing over 3k parallel tasks. The computation spans hundreds or thousands of machines, as the underlying infrastructure optimizes task location to make data accesses node or rack-local.

This batch-oriented solution, while very scalable, prevents us from achieving real-time analysis as the logs are being produced. While micro-batching style techniques [20] could be leveraged to improve the time-to-insight, we are currently working on a substantially more efficient approach, specialized for log collection and aggregation.

Dependency Graph Storage: given the extracted nodes and edges, we need to effectively and compactly store the resulting graph. The key intuition in that context is that dependency graphs are highly structured compared to standard graphs. Nodes and Edges are typed and therefore can be stored and indexed very efficiently.

In our current implementation, the dependency graph is partitioned by node and edge type, as well as by time (each day makes up a separate partition). The primary storage is a DFS [5]. In order to enable interactive access, we designed several projections

[2]See the 2016 CIKM Cross-Device Challenge: https://competitions.codalab.org/competitions/11171

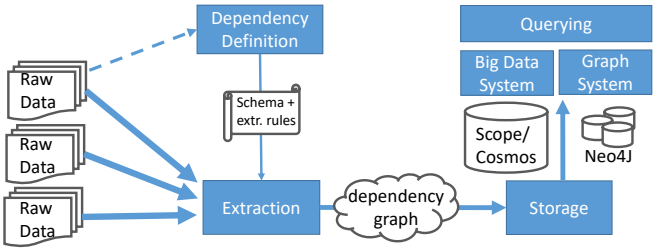

Figure 2: Guider's architecture.

of the graph that fulfill specific application needs. For example, the telemetry information we derive with the above rule is stored in multiple files (one per day), and it is kept separate from other job-level metadata. This allows us to perform cheap file-level filtering of the provenance graph contents, and load 1 day worth of telemetry information in a single-node Neo4J [16] instance.

Beside basic partitioning, we also perform various aggregations on the graph, such as leveraging the recurrent nature of the jobs to produce a summarized provenance-only graph that captures recurring jobs (and files) in a compact, non-redundant way. This allows us to load up to 30 days of provenance-only information in a single-node instance of Neo4J and power the auditing/compliance scenario of Section 3.2.3.

When full-detail (e.g., task-level) and long time-horizons are required (e.g., for the SLO extraction use case), we resort to slower but massively scalable Big Data infrastructures accessing this graph representation directly from the DFS—as we discuss next.

Dependency Graph Querying: the querying patterns to support DDA queries are unique, as they involve both graph traversals, graph algorithms, as well as relational and unstructured data processing. This is the next logical step in the progression from relational to Big Data querying. In Big Data querying, the focused has shifted from relational-only queries to the support of unstructured data and arbitrary UDFs and UDAs. With DDA, we introduce the need to *also* handle graph data. Our current solution is quite limited in that regard, as it allows to run pure graph queries on one Neo4J instance only, or complex analytics using a full-fledged, scan-oriented, Big Data solution (running over the DFS version of the data or the raw logs). We considered several other graph technologies, including TitanDB, Spark GraphFrames, Tinkerpop/Tinkergraph, but at the current state of stability/development they were not sufficiently capable for our setting. This experience indicates that industry is in desperate need for a fully capable, scale-out graph solution that can handle complex, large, and structured graphs, as well as a unified surface language for graph, relational and unstructured data handling.

The above limitations forced us to expose different APIs for different tasks:

- Most users simply access the data through our system front-end UI, which provides an interactive browsing and search experience, as shown in the partially-anonymized screenshot of Figure 3. The figure shows a user analyzing the upstream and downstream dependencies of a file of interest (non-descriptively named *Output.ss*.
- Highly interactive graph queries are supported through graph query languages (Cypher/Gremlin) and are executed on a fraction of the overall data, which is loaded in the main memory of a Neo4J single-instance server. Through a currently somewhat inelegant process, we can support arbitrary partitions to be loaded to Neo4J.
- Full-graph OLAP-style analysis of the graph need to be coded in a SQL-like scripting language, and operate directly on a DFS-resident copy of the data.

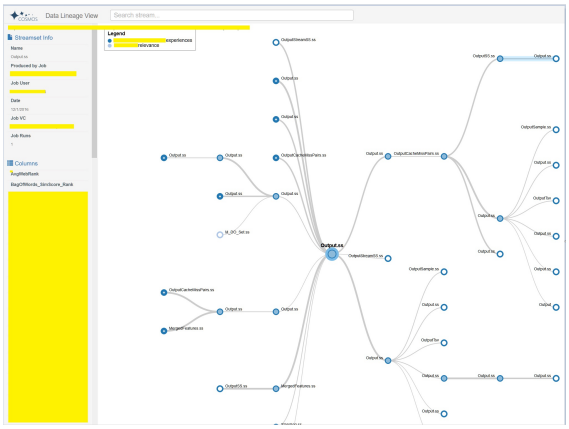

Figure 3: Guider's interactive browsing UI.

Guider is currently in use at Microsoft to support a combination of production and research efforts that require obtaining detailed insight from the data. We report on a few of those next, starting from the auditing/compliance use case that initially motivated building Guider

3.2 Applications

We introduced our prototype to several product and research teams inside Microsoft, which proposed and helped us implement a surprisingly high number of applications for Guider We briefly describe some of these applications below.

3.2.1 Global Job Ranking

Estimating the importance of the various jobs running on the cluster is often essential, for example when selecting the order in which jobs should run, or to cope with capacity impairments (if only a fraction of jobs can run, how do we choose?). There are many ways to define the importance of a job, one being the job contribution to future cluster operations. The number of jobs consuming the job output or their derivatives is a first approximation of that metric. We implemented a Global Job Ranking tool at Microsoft using that metric and leveraging Guider to traverse the dependency among jobs. This is equivalent to performing query Q1 (shown below) for all jobs. This operation is daunting when performed on raw logs, but becomes a rather standard graph traversal and aggregation when leveraging the dependency graph.

Example Q1: "job failure blast radius". *Consider a user/operator who wants to quantify the impact of a failed run of a recurring production job $JobA$ on downstream jobs—where impact is measured as the sum of CPU-hours of the affected jobs, and "downstream jobs" are jobs that directly or transitively depend on $JobA$'s output (see Figure 4). This user has to: access massive amounts of logs, finding the entries for each historical run $JobA_i$ of job $JobA$, parse the job-execution logs, collect a list of output files $out(JobA_i)$, search for all job instances J_i such that $input(J_i) \cap output(JobA_i) \neq \emptyset$, repeat recursively until no more*

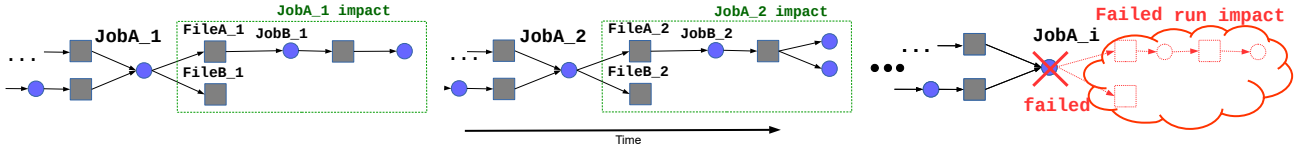

Figure 4: Impact of a failed run for recurring job JobA—the *"blast radius"* of Example Q1.

jobs are found, deduplicate this job list, join it with the corresponding telemetry logs, sum all CPU-hours used by the jobs downstream of each daily run $JobA_i$, and finally average across runs.

Answering query Q1 requires a detailed understanding of the log format and its many quirks (e.g., inconsistent field naming, different system components reporting time in UTC or local time-zones, etc.). This translates in hundreds of lines of complex analytics scripts, which need to be run on a Big Data infrastructure burning tens or hundreds of CPU-hours to scan and dissect potentially many TBs of textual logs.

On the other hand, given a dependency graph capturing job/file lineage as well as basic telemetry information, the user can answer Q1 with minimal effort. Using a language akin to Gremlin as our graph query language, query Q1 can be expressed as:

```
graph.traversal().V()
  .has("JobTemplateName","JobA_*")
  .local(emit().repeat(out()))
  .times(100).hasLabel("job")
  .dedup().values("TotalCPUTime").sum())
  .mean()
```

This can be written in minutes, without any knowledge of the log format, and can be run on a single-node, in-memory, graph database (e.g., Neo4J) in a matter of seconds. Next, we introduce a more navigational and challenging DDA use case:

3.2.2 Debugging

Example Q2: "Debugging a recurring job" *Consider the task of debugging the failure of instance $JobA_i$ of the recurring job $JobA$ from Example Q1. Isolating the root cause of the failure (e.g., a misbehaving UDF in an upstream job) is non-trivial, as this may be local to $JobA_i$ or due to a malformed input file produced by a buggy upstream job. The job's owner starts by analyzing the logs of $JobA_i$. If nothing looks suspicious, the user will turn to analyzing the job's input. This means looking for log lines listing the input files for the job, and separately inspecting their metadata. If an input looks unusual (e.g., smaller than normal), the user can scan the logs again looking for jobs that wrote to this file (before $JobA_i$ attempted read). The user then inspects those logs. This alternation of navigation and inspection continues until evidence of the root cause is located. Note that the logs explored in this manner are potentially generated by a multitude of systems (Front-end nodes, Job Runtime, Cluster Scheduler, DFS, UDFs, etc.)*

Performing the above task over raw logs is painstaking, as key information such as job input/output are buried in the middle of massive amounts of unstructured text logs—most of which are irrelevant to the task at hand. On the other hand, given a dependency graph, the navigational portion of Q2 becomes trivial—a basic graph traversal similar to the one from query Q1. Moreover, since the job being investigated is recurrent, the job's owner can perform simple graph analytics in search of abnormalities (e.g., in the size of the inputs to $JobA$). It is important to point out that the overall process *still requires direct access to some of the raw logs*, to determine the root cause of $JobA_i$'s failure. This is quintessential of DDA, where the dependency graph acts as a guide to access raw data. The development and computational cost is however reduced by orders of magnitude, as the user access of raw logs is targeted.

3.2.3 Auditing and Compliance

Data within large organizations are often stored in multiple infrastructures and across several geographic regions subjected to different regulations. Current storage and access technologies guarantee data integrity and protect against unauthorized access, but cannot enforce legal regulations or business rules. A number of important regulations prevent data to flow across geographic regions, like European Union regulations prohibiting the copy of "personal data" beyond jurisdictional boundaries. There are also laws and business rules for data at rest, for example requirements to delete personal data identifying users access to online services after a given period of time (e.g., 18 months), or business agreements restricting 3rd-party data access to certain scenarios [3].

Current data stores are unable to identify policy violations in that context (e.g., based on flexible business rules). At Microsoft, we have deployed a monitoring system leveraging DDA for that purpose. The resulting Compliance Monitoring system is rule-based. Compliance managers derive compliance rules from business requirements and regulations, and enter them using a Web interface. "User IP addresses must not be retained over 180 days" or "Personally identifiable information must not be combined with anonymous identifiers" are typical examples of such rules.

The core of the system is a Big Data job operating on the dependency graph and taking the list of compliance rules as an input. The system returns a list of violations, which are used to generate notifications to compliance managers. Prior to the introduction of the DDA-based Compliance Monitoring system, all auditing was manual, and substantially more costly. Once deployed, the automated DDA-based Compliance Monitoring system detects compliance violations on a continuous basis, providing improved coverage and latency.

3.2.4 Morpheus

Morpheus [9] is a research system running on top of the resource management infrastructure of our clusters. It enforces jobs deadlines (decreasing deadline violations by an order of magnitude), while retaining a high cluster utilization. Morpheus uses Guider to access the dependency graph as its main inference tool. The graph is used to infer user expectations as explicit SLOs inferred from historical data. In the process, the system makes heavy use of telemetry information (job resource consumption, job temporal access pattern, etc.). The data stored in the graph allow to group job instances into groups of periodic jobs (Job Templates)—see Figure 4. For every periodic job, we query Guider to obtain the earliest possible submission time (earliest time when all input data are available) as well as the latest completion time (latest time before any of the job's output is consumed by other jobs or egress operations). Having access to rich information about the jobs such as their validity intervals, usual job running times or resource utilization profiles enables Morpheus to automate this complex process.

Table 1: DDA vs raw log baseline

Q3: "Average Count of inputs per Job"			
	Raw	Graph on DFS	Neo4J
LoC	50	25	7
Run-time	58min	11min	4.9sec
CPU-time	563h	25h	<40sec
IOs	18.6TB	61GB	<24GB (RAM)

Q4: "Ratio of recurring vs Ad-hoc Jobs"			
	Raw	Graph on DFS	Neo4J
LoC	22	21	3
Run-time	8min	24min	10.9sec
CPU-time	41h	14h	<90sec
IOs	966GB	638MB	<24GB

3.2.5 Datacenter Migration

The Big Data clusters we operate at Microsoft are multi-tenant and often serve hundreds to thousands of customers from several business units. As their computational demands grow and saturate the clusters (after all possible capacity expansion), some tenants must be relocated to new clusters. The difficulty of that task comes from the fact that the business units share a lot of data and often have complex interdependencies at the job level (jobs of one business units often consume the job outputs of other business units). We recently leveraged Guider to find an optimal solution to that problem. By querying Guider we constructed a specialized projection of the graph at the tenant level, with edges storing the amount reads/writes shared by two tenants. At this point, determining which tenants are easier to migrate becomes a matter of performing a partitioning of the graph (i.e., searching for a graph cuts that fit in the target clusters, and minimizing the cost of the edges being cut). In practice, this is further complicated by the presence of business restrictions. We solved this using an Integer Linear Programming formulation that captures all business restrictions as well as cluster capacity limitations and cross-tenant data sharing. This provided invaluable input to the migration team, and helped minimizing the cost of the migration.

3.3 Preliminary Evaluation

As a demonstration of the DDA concept and the Guider system, we showcase two queries (Q3 and Q4) performing simple aggregates on jobs and files over one month of data. Q3 computes the average count of inputs for each recurring job, while Q4 calculate the ratio of jobs that are recurrent vs ad-hoc. These two queries are routinely evaluated and displayed as part of a monitoring dashboard. They were chosen as they are easily expressed both on the dependency graph and over raw logs.

We compare three variants for each query: 1) a baseline accessing the raw logs[3] using a SQL-Like Big Data runtime [5], 2) the same runtime operating on the DFS-resident copy of the dependency graph, and 3) a Cypher query running on a single-instance Neo4J graph DB [16]. Recall that in Neo4J we load only a projection of the data sufficient to answer the two queries. In all cases, the IOs reported are after all possible partition pruning.

The results are presented in Table 1. Even for these simple queries, the differences are substantial. Comparing Q3 and Q4 on a DDA system vs operating directly on the raw logs, we observe: 1) almost an order of magnitude less code, 2) up to 3 orders of magnitude less IOs, 3) up to 5 orders of magnitude less cpu-time, and 4) up to 3 orders of magnitude shorter run-times.

Our only goal with this preliminary evaluation is to highlight through a couple of simple examples the large potential of DDA and of systems like Guider in drastically reducing development and computational costs.

[3]To be precise the logs accessed are already partially preprocessed, as computing on the truly raw logs would require an additional 1 or 2 orders of magnitude IO and computational costs.

4. RESEARCH AGENDA

Guider is a first practical step towards building a DDA system. We describe below fundamental research challenges to achieve the full DDA vision. Some of the challenges we discuss below are new, while some will look familiar. In the latter case, our discussion serves as further evidence of their industrial relevance and potential for impact.

4.1 Graph Storage and Querying

Creating, storing and manipulating the dependency graph at scale is a major technical issue; our current solution leverages two different back-ends, and lacks an integrated language surface. This is far from optimal. There is a clear need for more efficient and scalable graph data management solutions, which could also elegantly integrate with mechanisms for non-graph data manipulation.

One interesting avenue for future work is to leverage the properties of the dependency graph to optimize storage and query processing. For example, the dependency graph in the log analytics use case is a temporal DAG with causal edge semantics, whose nodes and edges are typed, with strict domains and ranges. Stating it more generally, the dependency graph is more "structured" than typical graphs, i.e., it has a clearer schema. This opens up many options for optimization, and to support a blurring of the graph/relational divide. As an example, recall our running example Q1; Without special optimizations, a relational system would require multiple joins between all nodes and edges. The fact that the structure we operate on is a causal, temporal graph allows us to prune all nodes/edges that "precede" the source node $JobA_i$ chronologically. This pruning can be pushed down to the storage of the graph (like we do in our time-partitioned DFS copy of the graph) to dramatically reduce IO. The fact that nodes and edges are typed can also be used for pruning, further reducing the cardinality of scans and joins. We advocate for a system that automatically derives such complex constraints[4] (as part of the dependency definition process), and leverages them during query optimization.

4.2 Automated Dependency Graph Construction

As stated above, our current prototype requires users to manually write log mining scripts, and to update them whenever the log format gets updated. However, we envision to automate this process. The reason for automation is obvious: even for a reasonably simple instance of DDA such as log analysis, the process of defining all relevant entities/dependencies and their extraction rules is costly, and given a fast evolving ecosystem, likely requires continuous maintenance. To orient the reader on how costly this process is, Hadoop alone has 11k log statements, spread over a 1.8M lines codebase, that received 300k lines of code change in 2015 alone. And this is just one of the many interacting systems in the ecosystem. As observed in [10], standardizing shared formats is impractical even within a single company, as too many different teams contribute to the software ecosystem. The remaining alternative is to automate

[4]Note that the constraints in this example enable push-down predicates *across* tables, enabling sophisticated query rewritings.

the a-posteriori ingestion of logs. In the following, we describe our current efforts towards that end.

Automatically constructing the dependency graph from log data requires understanding the semantics of the logs, including all entities appearing in the logs as well as all relationships between them. In rare cases, the text of the logs is self-contained and is sufficient for human readers to understand all the entities and relationships that are mentioned. However, in most cases the logs favor brevity, are difficult to read even for humans, and do not contain extensive information on the way the system operates. Non-experts who do not have a deep understanding of the system architecture often have troubles reconstructing the processes that are described in the logs. In addition, important relationships are often not explicitly stated in the log (in Hadoop, for instance, the fact that an application created a file has to be inferred by joining data from both HDFS and YARN logs). Sometimes also, log data is stored in a tabular form, giving little to no context to infer semantics.

Given those limitations and our own experience, we believe that inferring log semantics from the logs themselves is impractical in general. Instead, we propose an alternative solution that takes advantage of the fact that the source code of the system producing the logs is often available. Having the source code available offers us a number of key benefits:

- The source code is by definition interpretable by machines. Hence, the parsers, compilers or interpreters that were written for a particular programming language can be leveraged for manipulating the source code.
- The source code holds a full description of all the entities appearing in the logs, including what kind of classes of entities are being used and how the entities interact with each other.
- The code describes exactly how the log lines are getting produced and how the entities and relationships are serialized, precluding the need to apply Data Mining or NLP techniques like named-entity recognition for vertical domains [14] or word-embedding to capture such information.

Running the code in debug or tracing mode may give precise information on the entities, on how they interact and on how they are serialized in the log lines. However, such a process is costly and often impossible to automatize taking into account the distributed nature of many systems. Instead, we suggest to use static code analysis below.

Code parsers used in modern IDEs (e.g., javaparser.org) are able to reconstruct class hierarchies (when considering the OOP paradigm), associated objects, as well as invoked methods for each particular instance. That information allows us in turn to associate log lines with particular method invocations generating the log messages.

By combining data on several levels of the function call stack, we automatically build a set of templates for the log messages. Each template is a string with slots for placeholders. Each slot is characterized by a (Java) class name and the property (or method name) that is used to get the placeholder's value. We give some examples below:

Template:

```
<System.currentTimeMillis()>
INFO
<FSNamesystem.class.getName()>:
allowed=<TicketService.Allowed()>
ugi=<UserGroupInformation.ToString()>
(auth:SIMPLE)
ip=<InetAddress.ToString()>
cmd=<"listStatus">
src=<Path.ToString()>
dst=<null>
perm=<null>
proto=<Server.getProtocol()>
```

A log message associated with the template:

```
2016-02-19 23:59:54.458 INFO
FSNamesystem.audit: allowed=true
ugi=hbase (auth:SIMPLE)
ip=/134.21.215.91
cmd=listStatus
src=/hbase/archive
dst=null perm=null
proto=rpc
```

The set of templates that can be extracted in that way can then be used to parse the log dataset. Log parsing produces a stream of values, each associated with particular properties (which in our context correspond to classes in the source code). The next step is to associate them with specific objects (class instances). Our main idea in this context is to identify and rely on discriminative object fields (those that have a unique value for every instance of a class). Usually these fields are primary keys or unique names that can be expressed as UUIDs or strings.

The underlying idea is that every log message, while providing some information about one or several objects, should also clearly identify such objects. As such, we may expect to find in each template several mentions about a given class, with at least one of them being discriminative (to uniquely identify the entity in the log message).

Extraction of discriminative fields can be undertaken in a rule-based manner or via Machine Learning. There are several factors that could hint for discriminative properties:

- The frequency of the field in the templates should be in the same ballpark as the frequency of the class. Typically, the discriminative fields of a class are also those that are appearing most frequently.
- The type of the field. We can expect discriminative fields to be expressed via strings or UUIDs. From our own analyses, the likelihood to have them as numeric values is very low.
- The field name as well as its context in the template often give additional information about its role.

The overall approach consists then of three steps:

- Identify templates and their structure;
- For every object class, identify discriminative fields;
- Join data for every object based on discriminative field values.

We are currently experimenting with this process. We still face a number of issues in that context:

- Identifying the life cycle of entities and how they evolve from a given state to another is hard and requires to correctly identify and interconnect the various stages in the log messages;
- Identifying the lifespan of an entity is often challenging, given that both the creation and the destruction of the entity can be missing from the logs;
- Generally speaking, relationships between entities can only be inferred based on the fact that they co-occur in log messages. The message type is often the only descriptor of such relationships, whose semantics are hence very difficult to guess.

To further complicate this task, our next challenge is to achieve all this in near real-time as data items are produced across hundreds of thousands of machines. At Microsoft, we have built solutions to ingest logs at such a massive scale, but stream-oriented entity extraction and linking at this scale remains an open problem.

5. RELATED WORK

Provenance and versioning systems: Provenance management for databases and scientific workflows has been extensively studied [17, 6, 8, 19, 18, 11, 2]. Both theoretical and system efforts have focused on capturing fine-grained provenance for complex queries. In this regard, Guider is less ambitious as it focuses on coarser-grained provenance, and lacks many of the sophistications proposed in the literature. File-level and object-store provenance tracking was explored in [12] and [13] respectively.

Our effort is, in spirit, closest to the Goods effort from Google [10]. Both approaches reconstruct provenance a-posteriori from a multitude of systems and operate at massive scale. We use provenance as a skeleton to analyze telemetry and raw logs, while Goods focuses on metadata indexing and dataset search. DDA extends to a wide range of application scenarios beyond provenance.

Datahub [4] focuses on facilitating collaboration by providing Git-style versioning for databases. This relates to our effort, but assumes that data owners are actively engaging with the datahub ecosystem. Scale and legacy issues makes this impractical in our setting.

Entity extraction and linking is a hard and well studied problem, especially for free form textual data. Efforts such as [15] will be instrumental to DDA in settings such as Enterprise Search.

Graph technologies: In building Guider, we tested several graph databases: TitanDB, Graphframes [7], Tinkerpop, and Neo4j [16]. Neo4j was the most mature and performant system we found, but its lack of support for scale-out forced us to resort to project/reduce the graph sizes until each of our applications could fit in one machine. Support for scale-out graph computing is in large demand. Also, we believe there is a strong industrial need for better languages and systems that support mixing graph, relational and unstructured data. Two particularly interesting efforts in that context are [7], introducing support for graph processing in Spark and [1] exploring theoretical as well as practical aspects of graph/relational query optimization.

6. CONCLUSIONS

We presented a vision for a new pattern in data analytics: Dependency-Driven Analytics (DDA). DDA solutions pivot around the automatic extraction of a dependency graph from large volumes of mostly unstructured data, to guide users in efficiently navigating and querying the data. We made this vision concrete by presenting a production system that implements DDA for a specific use case: log analytics. The benefits observed through this use case and related industry trends inspire us to pursue the broader vision of DDA.

7. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and conference chairs, for their thoughtful comments. We are also indebted to many present and former colleagues: Carrie Culley, Sudheer Dhulipalla, Chris Douglas, Raman Grover, Saikat Guha, Tim Hermann, Virajith Jalaparti Konstantinos Karanasos, Subru Krishnan, Rahul Potharaju, Sriram Rao, Javier Salido, Leena Sheth, Arun Suresh, Kai Zheng. They offered invaluable advises and practical guidance, that helped shape the DDA vision, and its initial implementation.

8. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, 2016.

[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proceedings of the VLDB Endowment*, 5(4):346–357, 2011.

[3] K. Bellare, C. Curino, A. Machanavajihala, P. Mika, M. Rahurkar, and A. Sane. WOO: A scalable and multi-tenant platform for continuous knowledge base synthesis. *PVLDB*, 2013.

[4] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798*, 2014.

[5] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008.

[6] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[7] A. Dave, A. Jindal, L. E. Li, R. Xin, J. Gonzalez, and M. Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2016.

[8] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.

[9] S. A. J. et al. Morpheus: Towards automatic slos for enterprise clusters. *To appear: OSDI*, 2016.

[10] A. Y. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 795–806, 2016.

[11] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. *Proc. VLDB Endow.*, 9(3):216–227, Nov. 2015.

[12] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *ATC*, 2006.

[13] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer. Provenance for the cloud. In *FAST*, volume 10, pages 15–14, 2010.

[14] R. Prokofyev, G. Demartini, and P. Cudré-Mauroux. Effective named entity recognition for idiosyncratic web collections. In *WWW*, 2014.

[15] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 2015.

[16] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner. *Neo4j in Action*. Manning, 2015.

[17] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.

[18] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 2013.

[19] E. Wu, S. Madden, and M. Stonebraker. Subzero: A

fine-grained lineage system for scientific databases. In *ICDE*, 2013.

[20] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Presented as part of the*, 2012.