

Dependency-Driven Analytics: a Compass for Uncharted Data Oceans

Ruslan Mavlyutov^{m,f}, Carlo Curino^m, Boris Asipov^m, Phil Cudre-Mauroux^f
^m Microsoft, ^f University of Fribourg

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 performed 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIDR 2017

© 2016 ACM. ISBN 123-4567-24-567/08/06.

DOI: 10.475/123_4

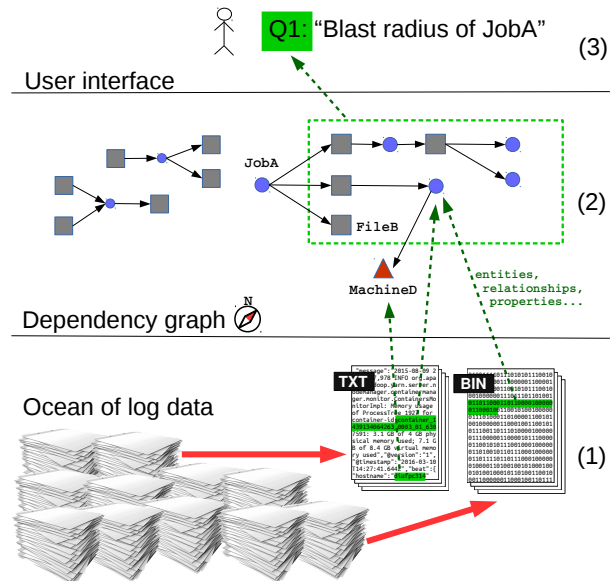


Figure 1: Dependency-Driven Analytics: a log-analytics example. (1) Petabytes of daily logs; (2) Dependency graph that represents a lightweight “skeleton” of the logs used for navigation; (3) User-level queries return bytes of aggregated data.

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 analysis by orders of magnitude, and enables interactive exploration of the logs—see Figure 1.

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 parse/analyze such raw data, and 2) computational costs when scanning/filtering/joining the data in their raw form.

Anecdotaly, 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 be-

came 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 these 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 is 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 Dependency-Driven Analytics in more details 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 [12], however, is that in DDA we do not aim at extracting all 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 the data format, parsing and 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. The log analytics scenario is the one we target with our production system. It is discussed in greater detail and serves as a running example for the rest of the paper.

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 application 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 invest substantial computational power to extract insight from raw textual logs. The resulting “cooked” data are surfaced in purpose-built monitoring tools, dashboards, and alert systems. We make this dire situation more concrete with an example, visually depicted in Figure 2 and summarized below:

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 2). 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 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 [13]) in a matter of seconds. Next, we introduce a more navigational and challenging DDA use case:

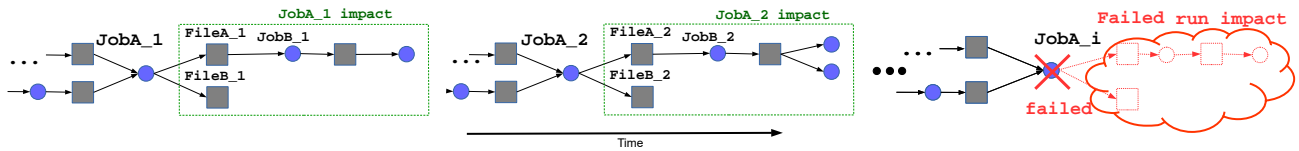


Figure 2: Impact of a failed run for recurring job JobA—the “blast radius” of Example Q1.

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 an error in 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 task are potentially generated by a multitude of systems (Frontend 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 in 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 the $JobA_i$ 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 very targeted.

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 [12]) can be leveraged in this context.

2.1.3 Internet of Things (IoT)

Embedded and mobile devices are increasingly connected, form-

ing webs of physical things¹. 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 advertising purposes². Further example include following a single event across multiple sensors, 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 3.2.1). Therefore, the dependency graph used focused on tracking provenance information. Provenance was 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 on provenance only was restrictive, and we started to track telemetry information for every entity that appeared in the dependency graph. For example, the amount of resources consumed by JobA over time, or the time at which FileB was read by JobC. This enabled a much broader set of analysis (see second part of Section 3.2), and provided the insight behind our DDA vision.

3.1 Architecture overview

With reference to Figure 3, 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 extraction rules, instructing the DDA runtime on how to find them in the raw data. This step can be challenging for free-form textual data [12], 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 manual, and the extraction rules take the form of SQL-like scripts [4] that process relevant portions of the logs, running on our internal Big Data infrastructure.

Dependency Graph Extraction: this is a runtime phase in which the rules defined above are applied to the raw data. This is akin to an ETL step for a relational data warehouse, but differs as the

¹<http://webofthings.org/>

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

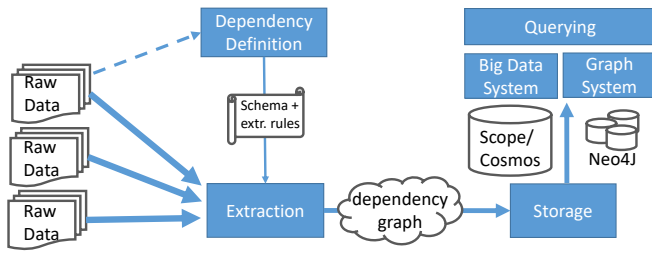


Figure 3: Guider’s architecture.

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 a given log). This scripts run on our existing Big Data infrastructure. This batch-oriented solution prevents us from achieving real-time analysis as the logs are being produced. Separate ongoing effort at Microsoft are addressing these concerns³.

Dependency Graph Storage: given the extracted nodes and edges, we need to effectively and compactly store the resulting graph. The key intuition 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 distributed file system (DFS) [4]. In order to enable interactive access, we designed several projections of the graph that fulfill specific application needs. For example, we leverage the recurrent nature of the graph 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 in a single-node instance of Neo4J [13] and power the auditing/compliance scenario of Section 3.2.1. Similarly, we can load 1 day worth of job-level telemetry data, or 1 hour of task-level data. When full-detail and long time-horizons are required (e.g., for the SLO extraction use case), we resort to slower but scalable Big Data infrastructure accessing the graph from the DFS.

Dependency Graph Querying: finally, 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 step in the progression from relational to Big Data querying. In Big Data querying, the focus shifted in supporting unstructured data and UDFs, and with DDA we introduce the need to *also* handle graph data. Our current solution is quite limited in this regard, as it allows to run only pure graph queries on the Neo4J instance, 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 indicate that industry is in desperate need for a fully capable, scale-out graph solution, that can handle complex, large, and structured graphs.

Guiders 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 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 regulation prevent data to flow across geographic boundaries, like European Union regulations prohibitin 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 [2].

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.

3.2.2 Morpheus

Morpheus [8] 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 expectation 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 jobs instances into groups of periodic jobs (Job Templates)—see Figure 2. 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 outputs are consumed by other jobs or egress operations). Having access to rich information about the jobs such as their validity intervals, usual job running times and resource utilization profiles enables Morpheus to automate this complex process.

³Citation will appear at CR time.

3.2.3 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 of Section 2 for all jobs. As we discussed, this operation is daunting when performed on raw logs, but becomes a rather standard graph traversal and aggregation when leveraging the dependency graph.

3.2.4 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 cut that fits 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 thus 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 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, and Q4 calculate the ratio of jobs that are recurrent vs ad-hoc. These two queries are routinely calculated and displayed as part of a monitoring dashboard. They were chosen as they are easily expressed both on the dependency graph and over the raw logs.

We compare three variants for each query: 1) a baseline accessing the raw logs⁴ using a SQL-Like Big Data runtime [4], 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 [13]. 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.

⁴To be precise the logs accessed are already partially preprocessed, as computing on the truly raw logs would require an additional 1 or

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

Our only goal with this preliminary evaluation is to highlight with a couple of simple examples the large potential of DDA and systems like Guider in drastically reducing development 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 are new and some will look familiar. In the latter case, this discussion serves as further evidence of their industrial relevance and potential for impact.

4.1 Dependency Definition and Extraction

In Guider we solved dependency definition and extraction by manually defining extraction rules from each of the types of logs we ingest, and running them as batch computation on a Big Data framework. This lacks two desirable properties: 1) automatic dependency definition, and 2) continuous / real-time dependency extraction.

The reason for automation is obvious: even for a reasonably simple instance of DDA such as log analysis, the process of defining 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 [9], standardizing around 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 the a-posteriori ingestion of logs.

Parsing and entity recognition is reasonably easy for logs thanks to the repetitive, streamlined and abundant nature of this data. In other settings such as Enterprise Search (see Section 2), this requires more advanced techniques such as [12]. Entity Linking or co-reference resolution is challenging regardless of the data source format. As a simple example, in a Hadoop-based DDA prototype (parallel to the Guider effort) we had to resolve to temporal analytics to (probabilistically) match Yarn jobs to the HDFS files they produce. Developing dedicated matching techniques and (un)supervised models in this context is a major research challenge.

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

2 orders of magnitude IO and computational cost.

to ingest logs at such a massive scale, but stream-oriented entity extraction and linking at this scale remains an open problem.

4.2 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 constraints⁵ (as part of the dependency definition process), and leverages them during query optimization.

5. RELATED WORK

Provenance and versioning systems Provenance management for databases and scientific workflows has been extensively studied [14, 5, 7, 16, 15]. 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 of proposed in literature. File-level and object-store provenance tracking was explored in [10] and [11] respectively.

Our effort is, in spirit, closest to the Goods effort from Google [9]. 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 [3] focuses on facilitating collaboration, by providing Git style versioning for databases. This relates to our effort, but assumes 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 [12] will be instrumental to DDA in settings such as Enterprise Search.

Graph technologies In building Guider, we tested several graph databases: TitanDB, Graphframes [6], Tinkerpop, Neo4j [13]. 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 till each of our application 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 language and sys-

⁵Note that the constraints in this example enable push-down predicates *across* tables, enabling sophisticated query rewrites.

tems that support mixing graph, relational and unstructured data. Two particularly interesting efforts are [6], introducing support for graph processing in Spark and [1] exploring theoretical and system 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 make 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 for DDA.

7. REFERENCES

- [1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. In *SIGMOD*, 2016.
- [2] K. Bellare, C. Curino, A. Machanavajhala, P. Mika, M. Rahrkar, and A. Sane. WOO: A scalable and multi-tenant platform for continuous knowledge base synthesis. *PVLDB*, 2013.
- [3] A. Bhardwaj, S. Bhattacharjee, 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, 2008.
- [8] S. A. J. et al. Morpheus: Towards automatic slo for enterprise clusters. *To appear: OSDI*, 2016.
- [9] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google’s datasets. In *SIGMOD*.
- [10] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer. Provenance-aware storage systems. In *ATC*, 2006.
- [11] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer. Provenance for the cloud. In *FAST*, volume 10, pages 15–14, 2010.
- [12] J. Shin, S. Wu, F. Wang, C. De Sa, C. Zhang, and C. Ré. Incremental knowledge base construction using deepdive. *PVLDB*, 2015.
- [13] A. Vukotic, N. Watt, T. Abedrabbo, D. Fox, and J. Partner. *Neo4j in Action*. Manning, 2015.
- [14] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.
- [15] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 2013.
- [16] E. Wu, S. Madden, and M. Stonebraker. Subzero: A fine-grained lineage system for scientific databases. In *ICDE*, 2013.