Meta-Dataflows: Efficient Exploratory Dataflow Jobs

Raul Castro Fernandez
MIT
raulcf@csail.mit.edu

William Culhane
Imperial College London
w.culhane@imperial.ac.uk

Pijika Watcharapichat
Imperial College London
pw610@imperial.ac.uk

Matthias Weidlich
Humboldt-Universität zu Berlin
matthias.weidlich@hu-berlin.de

Victoria Lopez Morales
Imperial College London
v.lopez-morales@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

ABSTRACT

Distributed dataflow systems such as Apache Spark and Apache Flink are used to derive new insights from large datasets. While they efficiently execute *concrete* data processing workflows, expressed as dataflow graphs, they lack generic support for *exploratory workflows*: if a user is uncertain about the correct processing pipeline, e.g. in terms of data cleaning strategy or choice of model parameters, they must repeatedly submit modified jobs to the system. This, however, misses out on optimisation opportunities for exploratory workflows, both in terms of scheduling and memory allocation.

We describe *meta-dataflows* (MDFs), a new model to effectively express exploratory workflows and efficiently execute them on compute clusters. With MDFs, users specify a *family* of dataflows using two primitives: (a) an `explore` operator automatically considers choices in a dataflow; and (b) a `choose` operator assesses the result quality of explored dataflow branches and selects a subset of the results. We propose optimisations to execute MDFs: a system can (i) avoid redundant computation when exploring branches by reusing intermediate results and discarding results from underperforming branches; and (ii) consider future data access patterns in the MDF when allocating cluster memory. Our evaluation shows that MDFs improve the runtime of exploratory workflows by up to 90% compared to sequential execution.

ACM Reference Format:
Raul Castro Fernandez, William Culhane, Pijika Watcharapichat, Matthias Weidlich, Victoria Lopez Morales, and Peter Pietzuch. 2018. Meta-Dataflows: Efficient Exploratory Dataflow Jobs. In *Proceedings of SIGMOD (SIGMOD'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3183760

1 INTRODUCTION

Analysts use increasingly sophisticated algorithmic techniques to derive value from data. Today, data processing pipelines routinely include complex cleaning strategies, apply advanced data mining algorithms, and train large machine learning (ML) models. To do this at scale, distributed dataflow systems such as Hadoop [4], Spark [40], Flink [3], SEEP [7] and TensorFlow [1] express data processing pipelines as *dataflow graphs* to be executed in parallel on clusters.

In many data processing pipelines users must configure data processing pipelines manually, with little support by distributed dataflow systems. Users choose which algorithms to use and how to tune configuration parameters: an outlier detection algorithm may rely on a threshold to classify input values as outliers; machine learning jobs may include hyper-parameters, such as the initial model weights [34] or the learning rate of a classifier [42]; data cleaning pipelines may use different error detection algorithms.

Consider a data profiling workflow to learn the underlying distribution of a terabyte-sized dataset using a kernel density estimator (KDE) [26]. In Spark, a user would write a dataflow graph to implement the technique and, after careful consideration, choose a handful of parameter configurations for the kernel function (e.g. Gaussian or Top-Hat) and the bandwidth (e.g. $\{0.1, 0.5, 2\}$). They would then submit *multiple* jobs to the cluster, one per configuration. We refer to a family of such related dataflow jobs as an *exploratory workflow*. Finally, the users would compare the results, and identify the configuration that yields the best result, e.g. in terms of mean integrated squared error (MISE). Even in this simple example, the manual parameter exploration requires the independent execution of many dataflow jobs that constitute the exploratory workflow.

Executing exploratory workflows as independent dataflow jobs is inefficient for multiple reasons: (i) each submitted job is executed to completion by the distributed dataflow system, even if a given configuration is inferior compared to others, because its result quality is only assessed after job completion. A better approach would be to terminate underperforming jobs early, and instead use the freed cluster resources for other, more promising configurations; and (ii) the jobs that constitute an exploratory workflow typically have substantial overlap in their intermediate results. Intermediate datasets should be reused across jobs instead of being recomputed.

We observe that current distributed dataflow systems pass up on optimisation opportunities by executing exploratory workflows as independent sequences of jobs. Taking all the jobs of an exploratory workflow into account allows the system to employ more effective scheduling and resource allocations policies.

We describe **meta-dataflows (MDFs)**, a new approach for effectively expressing exploratory workflows and executing them in a distributed dataflow system. An MDF specifies a complete *family* of traditional dataflow graphs and executes them more efficiently because the system can avoid redundant computation through its scheduling policy and perform better memory management for intermediate datasets. In more detail, we make three contributions:

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3183760

(1) MDF model. We introduce a novel *meta-dataflow model* that defines an exploratory workflow as a single job by extending a classical dataflow model with two new dataflow primitives: (a) an `explore` operator permits the exploration of different options in the dataflow graph in the form of separate *branches*, each representing a particular algorithmic or parameter configuration; and (b) a `choose` operator assesses the explored branches, discarding results that yield low utility. It assesses the utility of a branch result by an *evaluator* function and uses a *selection* function to pick a subset of the results for further computation. This requires an on-the-fly modification of the topology of the executed dataflow graph, which is not supported by most existing distributed dataflow systems.

(2) Branch-aware scheduling (BAS). We describe an efficient scheduling algorithm for MDFs, which saves cluster resources and shortens job completion times. The high-level strategy of the *branch-aware scheduling* algorithm is to traverse the MDF breadth-first, but to execute the parallel branches of an `explore` depth-first to obtain the intermediate results required for a `choose` decision. As a result, `choose` operators are executed as early as possible and in an incremental fashion. The dataflow system can discard unnecessary intermediate datasets quickly and, in some cases, avoid the execution of branches altogether. BAS also increases the cases in which datasets remain in memory and can be reused.

(3) Anticipatory memory management (AMM). We propose an *anticipatory memory management* policy to evict intermediate datasets from memory that minimises future reads from disk. AMM considers the data access patterns of branches stipulated in the MDF, the sizes of datasets, and the cost of loading data from disk. AMM addresses pressure on cluster memory created by intermediate datasets generated by branches that compete for memory with each other when `explore` operators have large fan-outs.

We implement MDFs in the SEEP distributed dataflow system [7] and report performance benefits for different exploratory workflows: a deep learning job for training a multi-layer neural network, a data profiling job for distribution estimation, and an analysis job for time series data. Our experiments show the MDF reduces runtime of the deep learning job by 60% compared to separate jobs; for the time series analysis job the MDF yields a runtime improvement of up to 90% by ignoring underperforming branches.

The rest of the paper is organised as follows: §2 gives background on distributed dataflow systems and motivates the need for exploratory workflows; §3 describes our model for meta-dataflows and how it can be applied to exploratory jobs; §4 explains the branch-aware scheduling algorithm and the anticipatory memory management policy; §5 gives implementation details for MDFs as part of distributed dataflow systems; the paper finishes with evaluation results (§6), related work (§7), and conclusions (§8).

2 EXPLORATORY WORKFLOWS

Next we describe a model for distributed dataflow systems (§2.1). We then introduce exploratory workflows (§2.2) and explain how they are not well supported by existing approaches (§2.3). Based on this, we derive a set of requirements for their efficient execution (§2.4).

2.1 Distributed dataflow systems

Modern distributed data processing systems, e.g. Spark [40] and Flink [3], express jobs as *dataflow graphs*, and execute them with data parallelism on a cluster of machines. Dataflow graphs are connected directed graph, $G = (V, E)$, where vertices V are data processing *operators*. Edges, $E \subseteq V \times V$, are data dependencies between them. We formally define our dataflow model in App. A.

Distributed dataflow systems execute dataflow graphs on a set of cluster nodes $\mathcal{N}$. Each node, $n \in \mathcal{N}$, has finite available memory, denoted by $mem(n) \in \mathbb{N}_0$, and unbounded disk storage. Datasets can be partitioned, with partitions stored on different nodes of the cluster. A partition can be stored in main memory or on disk.

To execute a dataflow graph as a *job*, many instances of operators must run on nodes, working on different data partitions in parallel. Systems such as Spark [40] and Flink [3] follow Dryad's execution model [20] and use a scheduler hosted at a *master* node. The scheduler breaks down a job into compute *tasks*, which are pairs of operators and a data partitions over which the operators are applied. Tasks are executed by *worker* nodes. *Stages* group sets of operators whose execution can be pipelined by the system.

Before a worker node executes a task, the respective data partition must be transferred to the worker and loaded into memory. If the worker has insufficient memory available, the system makes an *eviction decision* regarding which dataset to store on disk. Existing systems [40] typically employ a *least-recently used* (LRU) policy [2], i.e. they evict the dataset that has not been used for the longest.

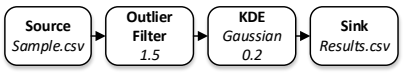
2.2 Exploratory workflows

In data processing pipelines, users must choose appropriate algorithms and parameters for individual steps, from data preparation, such as cleaning and schema matching, to model training when learning a classifier. While in some cases these choices are simple, i.e. the problem is well understood or the user draws from previous experience, in other cases the decision involves an *exploratory* process. We illustrate this process with the following scenario:

*Example 2.1 (***Dataflow for kernel-density estimation***).* We consider an example from the domain of sensor-based management of oil and gas fields [18]. Here, a user wants to detect malfunctioning oil well components based on readings from pressure and flow rate sensors, among others. To identify malfunction in the measurements, a model of regular well operation is created first.

Common data processing pipelines to obtain such a model first remove outliers from the raw sensor data. For example, a basic outlier filter would remove values beyond x-times the standard deviation. The next step is to estimate the distribution of sensor measurements that reflects regular operation, e.g. using *kernel density estimation* (KDE) [41]. KDE yields an estimator $g(x)$ for the unknown function that governs sensor measurements: $g(x) = 1/nh \sum_{i=1}^{n} K(x - x_i/h)$ where K is a *kernel* function (e.g. Gaussian or Top-Hat), and h is a smoothing parameter called *bandwidth*.

To execute the above data processing pipeline, a user could express it as the dataflow graph shown in Fig. 1. A source operator reads the input data. A second operator removes outliers beyond $o = 1.5\times$ of the standard deviation. Then an operator executes the



(a) Job shown as dataflow graph

```
1 val src = readFromFile("sample.csv")
2 val filtered = exploreOutlier.filter(src, 1.5)
3 val result = KDE.estimate(filtered, "gaussian", 0.2)
4 writeToFile("results.csv", result)
```

(b) Job expressed in Scala

Fig. 1: KDE job for cleaning and profiling sensor data

KDE algorithm using a specific kernel function (K = Gaussian) and bandwidth (h = 0.2).

This dataflow graph involves a set of *explorables*, i.e. vertices in for which there exists a choice of algorithm or parameter setting. A user would want to compare the results obtained by different settings for outlier detection, and different choices of kernel functions (e.g. Gaussian, Top-Hat, linear, or cosine) and bandwidth values.

We refer to the process of exploring the choices introduced by explorables in a dataflow graph as an *exploratory workflow*. In such a workflow, a user alters the dataflow graph in terms of its vertices, e.g. changing the operator functions. Using the model from §2.1, an exploratory workflow executes as a set of related dataflow graphs.

When the execution time of the exploratory workflow permits direct interactions, a motivated user can quickly narrow down promising parameter choices by trial-and-error. However, a process requiring users to engage in hours of submitting jobs, assessing their results, and selecting the best job at the end is overly cumbersome.

2.3 Support for exploratory workflows

Work on support for exploratory workflows focused on three areas:

Domain-specific parameter exploration. For jobs in domains with many explorables, such as machine learning, customised operators may support automated tuning of parameters [27, 30]. Existing support, however, is limited to certain domain-specific hyperparameters, such as the learning rate. Exporation automatically searches for the best parameter setting using evaluation criteria bespoke to machine learning algorithms such as the classification error. We lack generic exploratory workflows across domains.

Workflow orchestration. A common approach is for users to create their own *orchestration* scripts that coordinate the execution of an exploratory workflow. Such scripts must compare the quality of executed dataflow jobs and, based on this, make decisions about the next job to submit for execution. Generic cluster workflow systems [11, 21, 33] assist with such orchestration tasks, but the lack of integration with a dataflow model has severe drawbacks: the scheduling logic must be expressed explicitly, which is error-prone and incurs programming effort that could be avoided by suitable abstractions for exploratory workflows; at the same time, optimisation opportunities among related dataflow jobs are neglected, as job execution is independent of any orchestration script.

Dataset materialisation. To avoid re-computing datasets used multiple times, datasets may be *materialised*. Dataflow systems employing lazy evaluation of processing pipelines thus provide means to define explicit materialisation points. In Spark [40], cache and persist primitives enforce the materialisation of a resilient distributed dataset (RDD). Alternatively, materialisation can be handled outside of the dataflow system by a generic caching layer [15, 23].

Beyond the obvious drawback of forcing users to decide on the location of materialisation points and their types (memory, disk, or a combination of both) in advance, the main limitation is that dataset materialisation is not integrated with the scheduling of dataflow tasks. Materialised datasets may still be evicted repeatedly from memory (under an LRU policy)—preventing this requires a different scheduling strategy. Fundamentally, existing materialisation and caching policies assume independent jobs and fail to exploit the high overlap in exploratory workflows.

2.4 Requirements

When users prepare exploratory workflows with many explorables, the number of dataflow jobs to be executed explodes. To reduce overall completion time, we identify the following requirements:

(R1) Avoidance of unnecessary computation: *Computation must not be performed for underperforming or superfluous choices of explorables.* In an exploratory workflow, unnecessary computation may occur for two reasons: (R1a) the quality of intermediate results in a dataflow may already indicate a bad choice for an explorable; and (R1b) particular choices of an explorable may become irrelevant in the light of earlier results obtained for another explorable.

(R2) Reuse of intermediate results: *Intermediate results must be reused by different jobs.* When considering different choices for multiple explorables, many intermediate results can be reused. For example, a data profiling task that is common to all jobs in an exploratory workflow should only be executed once.

(R3) Early discarding of datasets: *Intermediate datasets must be discarded as soon as they are no longer needed.* Distributed dataflow systems maintain datasets in memory for efficient processing. To reduce memory pressure during the execution of an exploratory workflow, datasets that are not needed for further processing should therefore be discarded as soon as possible.

(R4) Workflow-aware memory management: *The management of cluster memory must consider data access patterns in exploratory workflows.* Exploratory workflows access intermediate datasets in a predictable manner. By taking access patterns into account when deciding which datasets to maintain in memory and which to evict to disk, a system can reduce the access cost to intermediate datasets.

The above requirements therefore call for the tighter integration of the dataflow jobs in an exploratory workflow, together with bespoke scheduling and memory management techniques. Next we introduce a new dataflow model that achieves this goal.

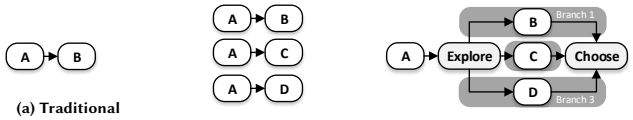
3 META-DATAFLOWS

We now describe *meta-dataflows* (MDFs), a new abstraction for exploratory workflows.

3.1 Meta-dataflow model

The main idea behind a meta-dataflow is to integrate a *family* of related dataflow graphs with different settings for explorables into a single dataflow graph. This enables the execution of exploratory workflows by submitting a single dataflow job.

The meta-dataflow model extends the common dataflow model from §2.1 with support for dataflow actions at the *meta*-level. As



(a) Traditional dataflow (b) Set of traditional dataflows (c) Meta-dataflow with three branches

Fig. 2: Exploratory workflows and meta-dataflows

shown in Fig. 2, choices for explorables in an MDF are represented through `explore` operators, which then converge through `choose` operators. A path between an `explore` and a `choose` operator, referred to as a *branch*, represents one setting for an explorable. Intuitively, an `explore` represents the beginning of a branch, whereas a `choose` controls which intermediate datasets from branches should be used for further processing. Operating on dataflows at the *meta*-level is inspired by previous work that has focused on adding control flow primitives to a dataflow abstraction [24, 37].

Each explorable results in an `explore` operator in the MDF. The operators succeeding the `explore` model the choices for an algorithm or parameter setting; operators preceding the `explore` and succeeding a `choose` represent computation that is independent of the explorables. The MDF model thus addresses requirement (R2) from §2.4: intermediate results are only generated once and reused, instead of being generated multiple times in separate jobs.

A `choose` operator assesses the quality of the results produced by each of the branches. It does this based on some predefined evaluation measure and selects a subset of datasets for further processing. This way, MDFs address the requirement to avoid unnecessary computation due to underperforming branches (R1a), which is similar in spirit to top-k plans in workflows [29].

The result quality of a branch can often be assessed independently. Hence, a `choose` operator can execute *incrementally* as soon as at least one of its branches has completed. It then evaluates and potentially discards a dataset, even before other branches have executed. This addresses the requirement to discard datasets early (R3).

Incremental execution of a `choose` operator may also indicate that some branches no longer need to be executed. For example, some choices of an explorable may lead to worse result quality compared to already executed branches. If so, the MDF can avoid this type of unnecessary computation (R1b). Crucially, this requires support for dynamic changes to the dataflow—a feature unavailable in mainstream distributed dataflow systems.

MDF definition. More formally, we define a meta-dataflow as:

*Definition 3.1 (***MDF***).* A meta-dataflow (MDF) is a dataflow graph, $G = (V, E)$, where $V_< \subseteq V$ is a set of `explore` operators, and $V_> \subseteq V$ is a set of `choose` operators, such that (i) for all $v \in V_<$, it holds that $|\bullet v| = 1$ and $|v \bullet| > 1$; and (ii) for all $v \in V_>$, it holds that $|\bullet v| > 1$ and $|v \bullet| = 1$.[1] A path $\pi(v, v')$ between operators $v, v' \in V$ is called *branch* if $v \in V_<$ and $v' \in V_>$.

The MDF model allows for hierarchical nesting of `explore` and `choose` operators. A branch may contain further branches that are defined through sequences of such operators. Execution semantics of an MDF extends the standard execution semantics of dataflow graphs. An operator $v \in V \setminus (V_< \cup V_>)$ can be executed if all preceding operators $v' \in \bullet v$ have been executed. When executing v, its operator function f_v is applied to the input datasets (see App. A).

Input datasets for an `explore` operator must be processed by each branch. Hence, the execution semantics of `explore` is defined as:

*Definition 3.2 (***Explore semantics***).* Let $G = (V, E)$ be an MDF. The semantics of an `explore` operator $v \in V_<$, $\bullet v = \{v'\}$ is defined such that (i) its operator function $f_v : \mathcal{D} \to \mathcal{D}^o$ with $o = |v \bullet|$ $f_v(d) \mapsto d^o$, i.e. `explore` simply forwards the datasets; (ii) v can be executed if v' has executed.

A `choose` operator selects among the datasets generated by its branches. Intuitively, the semantics of `choose` is given by (i) an *evaluator* function that calculates a score for the result dataset of a branch; and (ii) a *selection* function that picks the datasets of a subset of branches based on their scores, discarding the rest.

*Definition 3.3 (***Choose semantics***).* Let $G = (V, E)$ be an MDF. The semantics of a `choose` operator $v \in V_>$ is defined by its operator function $f_v : \mathcal{D}^i \to \mathcal{D}$ with $i = |\bullet v|$, which is $f_v(d_1, \ldots, d_i) \mapsto \rho_v((d_1, \phi_v(d_1)), \ldots, (d_i, \phi_v(d_i)))$ where $\phi_v : \mathcal{D} \to \mathbb{R}$ is an *evaluator* function that calculates a score per branch; and $\rho_v : (\mathcal{D} \times \mathbb{R})^i \to \mathcal{D}$ is a *selection* function that picks datasets from branches based on scores and concatenates them for further processing.

MDFs support different types of evaluator and selection functions. An evaluator function may compute a score over the values of a result dataset or its metadata. For example, $\phi_v(d) \mapsto |d|$ calculates a score based on the dataset size, e.g. to detect erroneous intermediate results due to too aggressive filtering. A typical selection function is *top-k*, which picks the datasets from k branches with the highest scores ($\oplus$ denotes concatenation of datasets, see App. A),

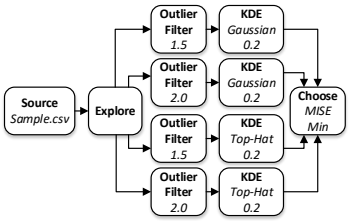
$$\rho_v : ((d_1, r_1), \ldots, (d_i, r_i)) \mapsto d' \text{ where } d' = \bigoplus_{1 \le j \le i \,\wedge\, |\{I \in \{1,\ldots,i\} | r_I \ge r_j\}| \le k} d_j.$$

Other common functions are *min* or *max*, and predicates that check that the evaluation scores are above or below a threshold (*threshold*) or fall within an interval (*interval*). Selection may also refer to the first-k scores that satisfy thresholds (*k-threshold*) or intervals (*k-interval*), or the most frequent value (*mode*).

*Example 3.4 (***MDF for KDE application***).* A user wants to explore the impact of different outlier thresholds (e.g. 2.5 instead of 1.5) and kernel functions (e.g. a Top-Hat kernel instead of Gaussian kernel) in the KDE job from Fig. 1. Figs. 3a and 3b show an MDF with four branches between the `explore` and `choose` operators. The `choose` uses an evaluator function ϕ that calculates the mean integrated squared error (MISE) as a score; the selection function ρ is defined as the minimum, i.e. only the dataset for the branch with the lowest MISE is returned as the result of the MDF.

MDF optimisations. Tab. 1 shows different optimisations possible during execution for combinations of evaluator and selection functions with particular properties. An evaluator function may be *convex* or *monotone* over the choices of an explorable. For example, exploring the parameter range of the simple outlier filter from Ex. 3.4 yields a monotone function. Selection functions are often *associative* and sometimes also *non-exhaustive*, i.e. a subset of results may be selected without insight into the remaining results.

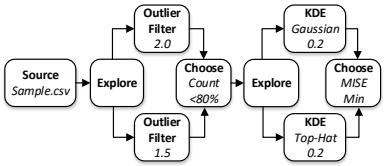
[1] $\bullet v$ and $v \bullet$ refers to the pre- and post-sets of operator v, respectively; see App. A.



(a) MDF exploring outlier thresholds and kernel functions

```
val src = readFromFile("sample.csv")
val result =
  EXPLORE(t=seq(1.5, 2), k=seq("gaussian", "top-hat"), {
    val filtered = Outlier.filter(src, t)
    val estimated = KDE.estimate(filtered, k, 0.2)
  }).CHOOSE(mise(estimated), min)
writeToFile("results.csv", result)
```

(b) Above MDF in Scala syntax



(c) MDF that chooses datasets at intermediate stage

Fig. 3: Sample MDFs for KDE job

Exploratory workflows can generally be expected to rely on a set of common evaluator and selection functions, for which the above properties are known. For example, as detailed above, a common evaluator determines the score based on the size of a dataset, which is a monotone function. Tab. 1 lists the properties of widely used selection functions. For domain-specific functions, however, the respective properties need to be provided by a user.

The above properties can then be exploited to execute MDFs more efficiently. Due to incremental evaluation of a choose operator, if the selection function is *associative*, datasets created by a branch are discarded as soon as it becomes clear that they are not processed further. Associative selection functions, however, also enable the detection of superfluous branches, which are not executed.

With *monotonic* and *convex* evaluator functions, it is possible to reason that datasets from not-yet-executed branches are inferior to those already obtained. In this case, remaining branches are not executed, and execution continues with the downstream operators.

In some cases it is possible to skip remaining branches regardless of evaluator behaviour. When scores of different branches are never compared directly and it is only necessary to select k sufficiently good results, the decision to discard not-yet-executed branches is independent of properties (monotonicity or convexity) of the score computation. For example, if the goal in Ex. 3.4 is to find k estimators for which the MISE is below a threshold, not-yet-executed branches become superfluous once k such estimators are found.

3.2 Patterns for MDFs

Next we discuss common patterns in MDF exploratory workflows.

Exploration scopes. In many dataflow jobs, explorables have a scope, i.e. they concern only a subset of the steps of a data processing pipeline. This scope is encoded in the structure of an MDF: it is opened by an explore and closed by a choose. Incorporating an explicit end of the scope of an explorable is an important pattern for MDFs, because it enables more efficient execution. The earlier a scope is closed by a choose operator, the sooner can underperforming branches be terminated. Consider the following example:

Table 1: Optimisations for different choose operator functions

Evaluator properties	Selection properties	Discard branches incrementally	Discard superfluous branches
monotone	associative	✓	✓
convex	associative	✓	✓
none	associative & non-exhaustive	✓	✓
none	associative	✓	
Properties of common selection functions:			
associative	top-k, min/max, threshold, interval, k-threshold, k-interval		
non-exhaustive	k-threshold, k-interval		
none	mode		

*Example 3.5 (***Scoped MDF for KDE job***).* Fig. 3c shows a variant of the KDE MDF with a limited scope for the exploration of the outlier removal configuration. It avoids superfluous computation when overly aggressive outlier removal discarded too much data: an initial choose operator selects only datasets for which the outlier detection removes less than 20% of the input data. The outlier filter is executed once per explorable configuration.

Evaluation of iterative computation. Dataflow jobs that perform a fixpoint computation require support for iteration: for example, in a dataflow graph for solving a classification problem, a user may want to try different features to model the problem, i.e. different sets of features become the explorables. An MDF must execute a fixpoint computation that iterates over the training data until convergence, doing this once for each explorable.

In a naive version, each branch in the MDF would run until completion before making a decision of its quality with a choose operator, i.e. each fixpoint operation must finish before the model can be evaluated. To avoid full execution of branches, however, a choose operator is incorporated in the iteration itself. It then terminates the branch early if, e.g. the computation is not converging.

Cross validation of ML models. Cross validation [13] is a widely used statistical procedure to assess prediction models: it splits input data into training and test data. Multiple successive rounds of training and validation are performed with different splits of the data to reduce variability of the end result. This can be expressed as an MDF as follows: an explore operator splits the input data, a trainer trains the ML model, and a choose operator selects the highest quality result. The trainer and choose operators execute multiple rounds of validation, and then assessing model quality.

4 SCHEDULING & MEMORY MANAGEMENT

Below, we first present a model for the execution of MDFs (§4.1) and then propose a *branch-aware scheduling* algorithm for MDFs, which schedules choose operators early to allow for the termination of underperforming branches and increases the reuse of intermediate datasets (§4.2). Finally, we describe an *anticipatory memory management* (AMM) policy, which takes dataset access patterns of MDF branches into account when making eviction decisions (§4.3).

4.1 MDF execution model

To support MDFs, a distributed dataflow system must *schedule* their execution, including the `explore` and `choose` operators. Next we define the notion of a schedule for an MDF and derive requirements for an efficient MDF scheduling algorithm.

Execution of an MDF happens in stages that group operators, see App. A ($\bullet T$ and $T\bullet$ denote pre- and post-sets of stage T, respectively). A *schedule* defines an order of stages:

Definition 4.1 (MDF Schedule). A schedule for an MDF, $G = (V, E)$, is a sequence $\langle T_1, \ldots, T_k \rangle$ of stages, such that (i) it is complete, i.e. there is a stage T_i, $1 \leq i \leq k$, for each sink $v \in V$, $v\bullet = \emptyset$, of the MDF; and (ii) it respects data dependencies, i.e. for each stage T_i, $1 \leq i \leq k$ with $\bullet T_i \neq \emptyset$, required input datasets have been produced already, i.e. $\bullet T_j \subseteq \bigcup_{1 \leq j \leq i} T_j$.

The execution of an MDF schedule $\langle T_1, \ldots, T_k \rangle$ yields a sequence $S = \langle s_0, \ldots, s_k \rangle$ of valid states (see App. A). A state $s_i = (D_i, \delta_i, \mu_i)$ represents the situation after execution of stage T_i, in terms of the available datasets (D_i), their partition sizes at nodes (δ_i), and their storage locations (μ_i). We assume that a schedule is feasible: partitions $d \in D_{i-1}$ needed as input for stage T_i are maintained in memory; for all nodes $n \in \mathcal{N}$, it holds that $d \in \mu_{i-1}(n)$.

Each state transition realised by the execution of a stage has a *cost*. Since the time needed to perform the computation is fixed and does not depend on the scheduling order, this cost is given by the sizes of dataset partitions that need to be loaded into memory at all nodes. An MDF *scheduler* tries to create a schedule with minimal cost. As the cost depends on the states visited during execution, it can only be assessed in retrospect: in a given state, a scheduler cannot reliably estimate which dataset partitions need to be spilled to disk (when cluster memory is exhausted) and which datasets may still be needed (because some branches of the MDF may not execute due to `choose` operators) in some future state. Scheduling of MDFs therefore differs from scheduling of dataflows built of traditional relational operators [9, 16, 38].

A natural direction is to try to leverage cost-based query optimisation techniques from relational DBMS, which can find a good execution plan based on performance statistics of operators and their data access costs, even dynamically [9]. These techniques, however, are tailored to relational operators. In line with other dataflow models, MDFs do not make assumptions on the used operators but consider them to be black-box. This makes it hard to decide a-priori which branches to skip. While some prior work has focused on the optimisation of individual black-box operators [10, 28, 35], we are concerned with the optimisation of entire exploratory workflows. A promising line of work [17] proposes to retrieve statistics for cost-based optimisation in the context of ETL dataflows with black-box operators, which we could leverage in MDF.

Given the absence of information on operator costs and the sizes of future dataset partitions, MDF scheduling must be conducted in an online manner and follow *stage scheduling*: upon the completion of a stage, the next stage to be executed must be determined. When cluster nodes exhaust their available memory, intermediate datasets are spilled to disk. In that case, *memory management* becomes relevant, as it determines which dataset partitions to evict to disk. Next we show how both problems can be solved for the MDF model.

4.2 Branch-aware scheduling

Existing dataflow systems use a breadth-first search (BFS) strategy to schedule stages. With BFS the initial stages execute to completion before the next stages are scheduled. When scheduling the `explore` stage of an MDF, the branches representing all explorables would thus be scheduled from their initial stages until reaching the corresponding `choose` operators. This has two disadvantages: (1) it is memory-intensive because it produces intermediate datasets for each explorable, making the memory usage grow linearly with the number of explorables; (2) all branches must be executed until completion before a `choose` operator makes a decision. The latter is a consequence of the data-parallel processing in distributed dataflow systems [3, 40], which execute one stage at a time. In cases in which a `choose` can select a branch early, there is a lost opportunity to save resources by avoiding unnecessary computation.

We instead aim to schedule individual branches until completion before scheduling others, giving `choose` operators opportunity for early evaluation. Our *branch-aware scheduling* (BAS) uses depth-first traversal between an `explore` operator and its corresponding `choose`. Each `choose` operator is split into two functions: the evaluator function is executed by worker nodes and applied directly to the result datasets of each branch; the selection function is executed during the scheduling decision by the master node.

With BAS, all cluster memory is dedicated to the execution of a single branch at a time. As `choose` operators evaluate the quality of branches as soon as they finish, it is possible to terminate the remaining branches early if the current branch passes the `choose` evaluation criterion. If the current branch does not satisfy the criterion, its allocated memory can be freed immediately.

The order in which BAS executes branches affects efficiency. To realise such optimisations, BAS respects *scheduling hints* on the order in which explorables (and thus branches) are considered as well as dependencies between them. Scheduling hints may be derived from properties of `choose` operators (see Tab. 1), stem from domain knowledge, or originate from models dynamically learned during the execution of an MDF: (i) scheduling hints may define priorities of the different choices of an explorable. For example, having an evaluator function that is convex over the choices of an explorable permits the scheduler to quickly identifying branches to select via binary search; (ii) scheduling hints may induce dependencies among the choices of different explorables. Random strategies work well in hyper-parameter search for ML models [5], and depth-first traversal by BAS may be relaxed in case of nested `explore` operators: changing choices of outer explorables before all inner explorables have been considered would prevent some optimisation opportunities, such as selecting branches with high quality results more quickly; (iii) scheduling hints may also be stateful and take intermediate results into account. Model-based optimisation of hyper-parameter search [19] relies on regression models to describe dependencies between explorables and the quality of results. Maintaining such models during MDF execution enables dynamic prioritisation of branches. Such techniques are orthogonal to the optimisations discussed earlier (Tab. 1): datasets will still be discarded incrementally, and superfluous branches be pruned.

Algorithm 1: Branch-aware scheduling for MDF $G = (V, E)$

```
1  T_exec ← ∅ ;                                          // Stages executed so far
2  T_open ← {T ∈ T_G | •T = ∅} ;                         // Stages ready for execution, in general
3  T_cand ← T_open ;                                     // Stages that shall be executed next
4  while T_cand ≠ ∅ do
5      T_next ← hinted_scheduling(T_cand) ;              // Stage to be executed next
       // Check if next stage is a single choose operator
6      if T_next = {v_next} ∧ v_next ∈ V_> then
7          run_workers(φ_{v_next}) ;                     // Execute choose evaluator function
8          run_master(ρ_{v_next}) ;                      // Execute choose selection function
           // Record stage as executed, if all predecessors executed
9          if •T_next ⊆ T_exec then T_exec ← T_exec ∪ {T_next} ;
10     else
11         run_workers(T_next) ;                         // Execute stage
12         T_exec ← T_exec ∪ {T_next} ;                  // Record stage as executed
       // Record current stages that are ready for execution
13     T_open ← T_open ∪ T_cand ∖ {T_next}
       // Determine new stages that are ready for execution
14     T_cand ← {T ∈ T_next• | •T ⊆ T_exec}
       // If no new stages became ready, resort to old ready stages
15     if T_cand = ∅ then T_cand ← T_open ;
```

Algorithm. We formalise the BAS algorithm in Alg. 1. Given an MDF, $G = \langle V, E \rangle$, BAS schedules stages (i.e. comprising operators with narrow dependencies, see App. A) for execution in a step-wise fashion. It maintains three sets: stages that have been executed (T_{exec}), stages that are ready for execution (T_{open}), and stages that are candidates for being executed next (T_{cand}).

Initially, T_{open} and T_{cand} contain the stages with the source vertices of the dataflow graph. As long as the set of candidate stages is non-empty, stages are scheduled for execution (lines 4–15). A candidate stage is chosen, based on scheduling hints or randomly (line 5). If it contains a choose operator (choose operators are assigned to separate stages), its evaluator function is executed by the workers (line 7) and the selection function by the master (line 8). The stage of the choose is recorded as executed only if all its predecessors have executed, i.e. all branches that lead to the choose (line 9). All other stages are scheduled for execution on the workers (line 11) and recorded as executed (line 12).

Next, the algorithm updates the sets of stages that are ready for execution (T_{open}) and that are candidates for the next execution (T_{cand}). The latter is set to all succeeding stages of the one executed last if their respective preceding stages have executed (line 14). The current branch is executed until completion. Stages of other branches are only considered if the stages succeeding the one executed last are not yet ready for execution. In this case, the set of ready stages (T_{open}) is taken as the new set of candidates (line 15).

*Example 4.2 (***Branch-aware scheduling***).* Fig. 4 shows the schedule obtained by BAS for the first part of the KDE MDF from Fig. 3c: (i) the source operator loads the input data, which yields a dataset d_0; (ii) tasks are scheduled for the stage that includes the outlier filter with a threshold of 2.0, resulting in dataset d_1, and the subsequent evaluation, which yields result score r_1; (iii) the selection function is evaluated by the master and leads to the eviction of dataset d_1; (iv) the same is done for the second branch of the MDF, producing a dataset d_2 with a score r_2; and (v) the selection function is evaluated again, and the dataset is kept for further processing.

Regardless of the branch selection order, for MDF execution, BAS is superior to traditional BFS-based scheduling. As detailed

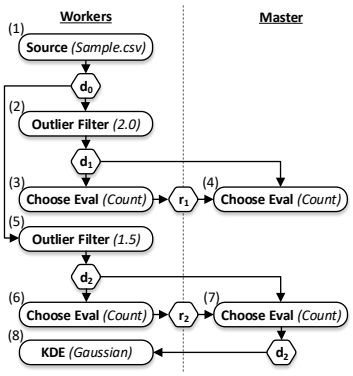


Fig. 4: Example of branch-aware scheduling for KDE MDF

above, the cost of a schedule is determined by the sizes of dataset partitions that need to be loaded by all workers. In the general case, BAS reduces the number of datasets to be stored, which lowers the cost of a schedule: when using depth-first traversal between explore and choose operators instead of BFS, at most as many, and typically significantly fewer datasets are stored.

Let $s = (D, \delta, \mu)$ be an execution state of MDF $G = (V, E)$ (see App. A), and $V_T \subseteq V$ the set of executed operators. For a dataset $d \in D$, we denote by $con(d) \subseteq V$ the consuming operators in the MDF. Then, D_s^c is the subset of datasets in state s that are still needed to complete execution, i.e. $D_s^c = \{d \in D \mid (con(d) \setminus V_T) \neq \emptyset\}$.

Theorem 4.3. *Let $s_A = (D_A, \delta_A, \mu_A)$ and $s_F = (D_F, \delta_F, \mu_F)$ be two states during the execution of an MDF with BAS or BFS, respectively. Then, it holds that if the same datasets have been produced to reach both states, the number of datasets still to be stored with BAS is at most the number obtained with BFS:*

$$D_A = D_F \Rightarrow |D_{s_A}^c| \leq |D_{s_F}^c|.$$

A discussion and proof of this result can be found in App. B.

4.3 Anticipatory memory management

When datasets exceed the available memory of cluster nodes, they must be evicted and spilled to disk. A *memory management policy* decides which datasets to evict, and affects execution performance of future stages that access evicted datasets. Existing systems such as Spark employ a *least-recently-used* (LRU) policy: the system maintains information about the last access of each dataset, e.g. through timestamps; upon exhausting memory, the datasets that were used the longest time ago are spilled to disk.

MDFs typically have operators with large fan-outs: an explore operator has a high out-degree when datasets used as input are passed to a large number of explored branches. Existing systems do not account for such graphs in their memory management policies because, in traditional dataflow graphs, a single dataset is rarely used as input to many operators. For MDFs, an LRU policy makes inefficient eviction decisions: the large fan-out of explore operators means that a recently unused dataset may still be required as input for future operators and should remain in memory. This inefficiency of an LRU policy aggravates at more deeply nested branches.

We observe that the structure of the MDF provides information on how datasets will be accessed during execution: datasets produced by stages upstream of an explore operator may be accessed

Algorithm 2: Anticipatory memory management for MDFs

input : $G = (V, E)$, an MDF,
$n \in \mathcal{N}$, a cluster node for which memory is exhausted,
(D, δ, μ), the current state of MDF execution,
$V_T \subseteq V$, the set of operators of executed stages so far,
α, the ratio of reading/writing data to disk/memory.
output: $d^* \in D$, dataset of which the partition at n is evicted.

// Compute how often a dataset can still be used as input of operators
for $d \in D$ **do**
  $v \leftarrow pro(d)$;
  $acc(d) \leftarrow |v \bullet \setminus V_T|$;
// Compute preference for all known datasets currently maintained in memory at node n
for $d \in \mu(n)$ **do** $pre(d) = acc(d) \cdot \delta(n, d) \cdot \alpha$;
// Select dataset with lowest preference value for eviction
$d^* \leftarrow \arg\min_{d \in \mu(n)} pre(d)$;
return d^*;

multiple times, once for each branch originating from the explore operator; other datasets may be subject to pruning of branches by a choose operator and hence not accessed at all beyond the choose.

Knowledge of the data access patterns alone is not sufficient to make effective memory management decisions, as the cost of loading a dataset partition into memory depends on its size. Hence, the effectiveness of a given eviction decision also depends on the, potentially unknown, sizes of future intermediate dataset partitions.

In the absence of precise information about future dataset sizes, we *order* datasets by the preference to be kept in memory based on (i) how often a dataset will be accessed and (ii) the cost of loading the dataset from disk. When memory is exhausted at a cluster node, the dataset with the lowest preference value is evicted.

Algorithm. We describe the algorithm for *anticipatory memory management* (AMM) in Alg. 2. It is invoked when, during the execution of an MDF, $G = \langle V, E \rangle$, a cluster node $n \in \mathcal{N}$ exhausts its memory. We assume that (D, δ, μ) is the current execution state after the stages with the operators in V_T have already been executed. The AMM algorithm also takes as input a hardware-specific ratio of the cost of reading/writing data from/to disk and memory, respectively. It returns a dataset partition at node n to be evicted.

The algorithm first calculates how often each dataset known in the current state, $d \in D$, is still used as input of operators according to G. For each dataset d, it determines the operator that produced it, $pro(d)$. The successors of this operator in G, which have not yet been executed as part of a stage, may still need to access dataset d in the future. The number of future accesses is denoted by $acc(d)$.

Next, AMM assigns a preference $pre(d)$ to each dataset d maintained in memory at node n. This preference represents the relative importance of keeping each dataset in memory. It is computed based on the number of future accesses $acc(d)$, the size of the partition of dataset d at node n, and a disk/memory cost ratio α.

The ratio α is computed for a specific cluster hardware ahead of time. If w_d, w_m, r_d, and r_m are the times to write a fixed amount of data to disk and to memory, and to read it from disk and from memory, respectively, the ratio α is defined as: $\alpha = w_d\ r_m / w_m\ r_d$. Finally, the AMM algorithm returns the dataset with the lowest preference, and the partition of this dataset at node n is evicted.

5 IMPLEMENTATION

In this section, we describe the implementation of MDF as part of the SEEP distributed dataflow systems [7]. Its master/worker architecture is similar to other modern dataflow systems such as Spark [40] and Flink [3]. The master node runs a *scheduler* that instructs worker nodes which stages to execute from the dataflow graph. Each worker has a *memory allocator* that manages memory and makes eviction decisions. We explain how we adapted SEEP's scheduler and memory allocator to support MDF, and finish with a discussion of fault-tolerance and straggler mitigation.

Scheduler. The SEEP scheduler must identify explore and choose operators in the MDF. When it finds an explore, it triggers the branch-aware scheduling, until it detects a choose. For that, it filters out the stages in the scheduling queue to retain only those of the same branch. The other stages are moved to a *pending branch queue*, which is accessed by the choose function, as explained next.

The choose operator must pick active branches according to the selection function, and assign the output of these branches to tasks of the following stage. Crucially, after a choose operator executes, the scheduler may decide to change the dataflow dynamically, e.g. by pruning a branch. Dynamically changing the dataflow is supported in SEEP by executing the choose operator in the master—with the control at the master, the schedule is rewritten based on the outcome of choose. After a decision is made, execution continues by scheduling the remaining stages from the *pending branch queue*, or the remainder of the job if no more branches are available.

Memory allocator. Each worker has a memory allocator that can load datasets into memory or spill them to disk. To support the AMM policy, we change two components: (i) the master must implement a policy and a mechanism, which exchanges information about memory management with workers; and (ii) each worker must enforce the eviction strategy, as dictated by the master. During scheduling, the workers notify the master about available datasets and memory. The master uses this information to execute the AMM policy, and produces a list of datasets ordered by their preference to remain in memory. This list is sent to workers with each scheduling decision, which evict datasets based on the preference values.

Fault-tolerance and stragglers. Two common problems of executing dataflow jobs on clusters are (i) *node failures*, which must be handled by a fault-tolerance mechanism, and (ii) *stragglers*, which are underperforming workers that increase the job completion time.

SEEP uses a checkpoint-based fault tolerance mechanism; qother mechanisms, such as those based on recomputation [40], can also be applied. If execution during branch exploration fails, recovery is the same as that for any operator failure: the master maintains the results of the evaluation function at choose operators, which are small in size. Thus the result can be recovered from the master rather than executing entire branches to recompute.

Mitigating straggling workers when executing MDFs does not require changes to the dataflow system and can leverage existing mechanisms. A potential new source of stragglers, however, can be the execution of the selection functions of choose operators at the master. We have not found this to be an issue in practice though: on a cluster of 10 machines with a low-end master node, we can execute 2 million choose invocations per second when collecting results. If the choose execution becomes a bottleneck, an alternative design would execute selection functions at worker nodes, and then relinquish control to the master.

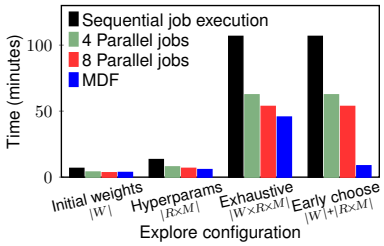


Fig. 5: Deep learning job

Fig. 6: Data profiling job

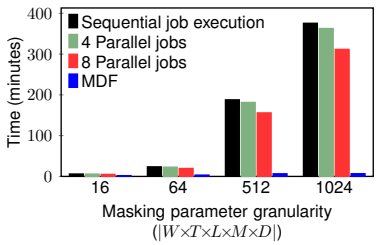


Fig. 7: Time series analysis job

6 EVALUATION

We experimentally evaluate the performance benefit of MDFs in three common domains for exploratory workflows (§6.1). We then investigate the scalability of MDFs (§6.2), the impact of MDF topology on performance (§6.3), and the behaviour of MDFs with different CPU and memory resources (§6.4).

Experimental set-up. We have added MDF support, as described in §5, to SEEP, an open-source distributed dataflow system [7, 8]. The architecture of SEEP is representative of that of other systems such as Spark [40] or Flink [3], and its scheduler is similar to Dryad's [20]. §6.1 compares SEEP's performance to that of Spark.

We experiment on a private cluster with 1 master node and 12 worker nodes. Each node has a quad-core Intel Xeon E3-1220 CPU, 16 GB of RAM, and a 1 Gbps Ethernet connection. We use Ubuntu Linux 14.04.3 with the Linux kernel 3.1 and OpenJDK JVM 8.

For all experiments, we consider the average results over 3 runs and include the minimum and maximum as error bars. Given the small variance between runs, the error bars are often not visible.

Exploratory workflows. We use the following workflows, for which the MDFs can be found in App. C:

(1) Deep learning job. Deep learning networks (DNNs) use large volumes of data to produce accurate machine learning models for image classification, speech recognition, and text understanding. Exploratory workflows for training DNN models can include explorables for hyper-parameters, with the goal of finding the best configuration that results in the highest classification accuracy.

We create an MDF that covers three stages: data pre-processing, DNN model training, and model validation. In the training stage, the MDF explores: (i) eight weight initialisation strategies based on either Gaussian or uniform distributions (W); (ii) four learning rates (R=$\{0.0001, 0.001, 0.005, 0.01\}$); (iii) four momentum values (M=$\{0.25, 0.5, 0.75, 0.9\}$). With all possible values of initialisation strategies and hyper-parameters, the number of paths to explore becomes $|W \times R \times M| = 128$. The MDF trains the model using the CIFAR-10 dataset that contains RGB image data commonly used for benchmarking ML jobs [22]. After an epoch of training, the classification accuracy is measured using validation images.

(2) Time series analysis job. A common task in time series analysis is to determine which data points, or groups thereof, are of interest for further analysis. A typical processing pipeline has three stages: (i) *masking* data points in the series based on the value ranges within a sliding window; (ii) *marking* discrete events that indicate drastic changes in the series; and (iii) *detecting* sequences of discrete events, each indicating a change of a particular magnitude. We use a real-world dataset of a million sensor measurements from oil wells [18] and create an MDF with five explorables: masking uses (i) different window lengths (W=$\{2, \ldots, 9\}$); (ii) thresholds for the permitted data difference in the window (T=$\{1.0001, \ldots, 1.5\}$); marking relies on (iii) different window lengths (L=$\{2, \ldots, 10\}$); (iv) value differences (M=$\{0.1, \ldots, 10.0\}$); and (v) event durations (D=$\{2k, \ldots, 20k\}$). We consider different granularities of these parameters, and explore combinations of them at each granularity, yielding between 16 and 1024 branches. The obtained intermediate result is then evaluated in terms of the aggressiveness of masking: the MDF evaluates the number of resulting data points, and the ratio of masked data points should not exceed a threshold.

(3) Data profiling job. We use the kernel-density estimation (KDE) job from §2.2 to create an MDF. It processes a synthetic dataset with 100 million normally distributed random values. The MDF has multiple explorables: (i) the data pre-processing method between normalisation and standardisation (N); (ii) the kernel function that is used in the estimation of the distribution (K=$\{$biweight, triweight, $\ldots\}$) and (iii) the kernel bandwidth parameter (B=$\{0.1, 0.2, 0.3\}$). To evaluate the effectiveness of branches, the MDF uses hold-out samples (1% of the dataset) and computes the log likelihood of the probability density function values of the hold-out samples.

(4) Synthetic job. Finally, we create a synthetic MDF that offers control over the branch structure, and computational cost. The MDF processes string/integer pairs, and uses two nested `explores`, B_1 and B_2. Each `explore` performs an algebraic operation on each branch, updating the integer value in tuples accordingly. The algebraic operation is performed a configurable number of times per data item, which permits us to tune the processing cost.

6.1 How do MDFs affect completion time?

We study the job completion times of MDFs against three baseline approaches: (i) sequential executes separate jobs for each explorable setting in sequence. Each job utilises the full cluster, and once it is finished, the next job is scheduled; (ii) 4-parallel submits four jobs in parallel to the cluster until all the jobs have run; and (iii) 8-parallel executes eight parallel jobs. The deployment uses 8 worker nodes, and the parallel deployments share the memory of workers equally.

Fig. 5 shows the completion time for the deep learning job with different explorables. The first set of bars shows the time to explore just the *initial weights* W; the second set of bars reflects the exploration of all combinations of *hyper-parameters*, $R \times M$. In the first configuration, differences between all approaches are small, and completion times are short; in the second one, parallel execution (4-parallel and 8-parallel) leads to slight speed-ups compared to sequential execution. MDF offers further improvement as it preprocesses the dataset only once and reuses it across explored paths.

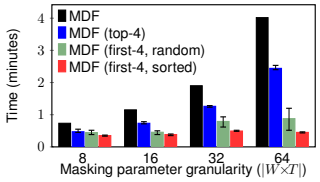


Fig. 8: Impact of optimisations

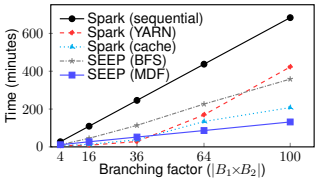


Fig. 9: Completion times of MDF

The final two sets of bars consider both the initial weights and the hyper-parameters, thus exploring the full set of combinations. The first option, *exhaustive* exploration, considers all combinations of weights with all combinations of hyper-parameters, thus exploring a number of options equal to the cardinality of the cross product of each set, $|W \times R \times M|$. For the deep learning job, however, it is possible to explore the initial weights first, then choose the best result as the starting point for the exploration of the hyper-parameters within a single job. The last set of bars therefore considers this *early choose* approach, which reduces the explored paths to $|W| + |R \times M|$.

Under *exhaustive* exploration, MDF reduces completion time by 60% compared to sequential. Parallel execution fully utilises the cluster, thus achieving better performance. MDF offers further improvement—reducing completion times by 28% and 15% compared to 4-parallel and 8-parallel, respectively—because it does not repeatedly pre-process the training data across explored paths.

MDF with *early choose* exploration reduces completion time by 85% compared to 8-parallel, the best-performing baseline approach. Here the MDF can avoid training the model with poorly chosen initial weights W, thus only exploring promising combinations of the other parameters $R \times M$. Compared to exhaustive exploration, this requires fewer intermediate datasets to be kept in memory.

Fig. 6 shows the completion times for the data profiling job. MDF consistently finishes fastest. It reduces completion time by 70%, on average, when exploring KDE configurations compared to sequential because it reuses the output of data pre-processing operators for subsequent exploration of the kernel functions and bandwidths. The benefit of MDF depends on the input size: the normalisation process is inexpensive, but it requires a linear scan over the entire dataset and therefore has increased cost as the dataset size grows. With MDF, this input data is read only once.

Among the baselines, the completion time of parallel execution is shorter than that of sequential, and 8-parallel is faster than 4-parallel. In this job, the computation and I/O operations can be overlapped among the parallel jobs. A higher degree of parallelism, however, increases memory pressure, which limits performance.

Fig. 7 shows the results of the time series analysis job. The completion time of sequential grows linearly with the number of explored combinations of $W \times T \times L \times M \times D$, as expected. With parallel execution, the completion time increases more slowly due to better cluster utilisation. All three baselines are significantly slower than MDF, though. With MDF, the choose after the masking operator selects only a subset of intermediate results for further processing (event marking and sequence detection), thereby terminating underperforming branches. This reduces completion time between 60% and 98% over parallel and sequential execution, respectively.

In Fig. 8, we investigate the effect of the optimisations from §3.1 and the scheduling hints from §4.2. We use the time series analysis job and vary the choose function. The MDF bars are the same as those in Fig. 7, exploring all branches to completion. By selecting only the top-4 results at the choose operator, as shown by the MDF (top-4) bar, the system reduces completion time by 34%–39%, discarding datasets incrementally. When the MDF can select any 4 results that meet a threshold (rather than the top-4), this effect becomes more pronounced—in this experiment, half of the results meet the threshold. As the selection function is now not only associative but also non-exhaustive, significant savings ensue.

As shown by the MDF (first-4, random) bar, executing branches in random order (as suggested by random search in hyper-parameter optimisation [5]), leads to a large variability in completion times depending on the order (we show the minimum, average and maximum of 12 runs). Yet, the maximum is always less than that of MDF (top-4), showing an 85% improvement in the best case.

The scheduling hints can be exploited to optimise the search further. If the evaluator is monotonic, domain-specific hints on the order in which the values of the explorable are considered reduce the completion time: as shown by the MDF (first-4, sorted) bar, the system consistently executes faster with these scheduling hints.

Finally, we compare MDF against Spark [40] using the synthetic job. We compare (i) Spark (sequential), the equivalent sequential jobs executed by Spark; (ii) Spark (YARN), parallel jobs executed by Spark together with YARN [36]; (iii) Spark (cache), a single Spark job in which we explicitly designate intermediate datasets for reuse using Spark `cache()` statements. Since Spark does not use AMM for eviction, we empirically determine which datasets to retain—when instructing Spark to cache all datasets, execution is slower than without caching; (iv) SEEP (BFS), a single job in SEEP that traverses the dataflow in a breadth-first (BFS) manner; and (v) SEEP (MDF). Given the simple nature of computation in the synthetic job, the implementations in SEEP and Spark are the same.

Fig. 9 shows the completion time as the branching factors $|B_1|$ and $|B_2|$ vary. We use the same branching factor in the inner and outer explores, i.e. $|B_1|=|B_2|$. The Spark (sequential) deployment is consistently the worst, as it neither removes redundant computation nor parallelises the job. Spark (YARN) improves performance, especially when there are few branches that benefit from parallelism. The improvement plateaus for larger number of branches as the redundant computation increases with each job.

SEEP (MDF) outperforms Spark (YARN) by 69% and Spark (cache) by 37% when executing 100 branches. It outperforms Spark (YARN) because of the reuse of intermediate results. It outperforms Spark (cache) because of the better memory management introduced by AMM. The performance of SEEP (BFS) is worse than SEEP (MDF) and Spark (cache) because it does not use BAS. In summary, the combination of BAS and AMM, as realised in MDF, achieves the best performance for exploratory workflows, even when compared with a judiciously designed dataflow in Spark.

6.2 How scalable are MDFs?

We evaluate scalability with respect to the number of workers and the input data. We break down the completion time results in terms of the impact of traditional (LRU) and optimised (AMM) memory management, both with and without incremental evaluation of the choose operator (incremental; see §3.1). Both the incremental choose evaluation and AMM are meant to decrease completion

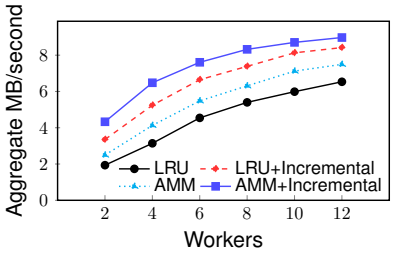


Fig. 10: Completion times (workers)

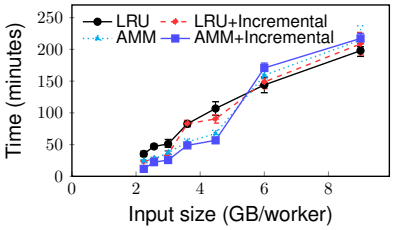


Fig. 11: Completion times (data)

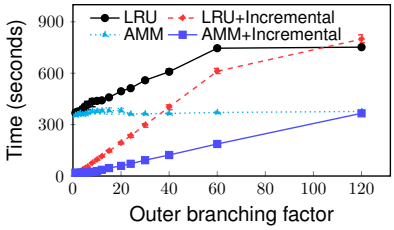


Fig. 12: Completion times (topology)

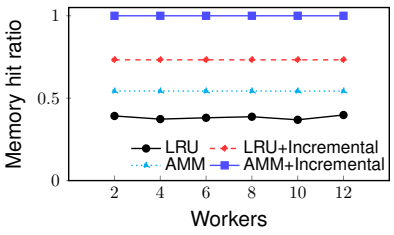


Fig. 13: Memory hit ratio (workers)

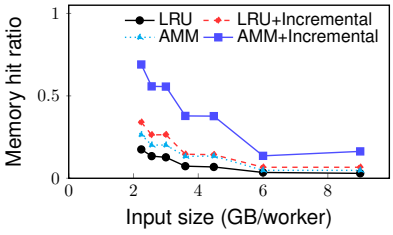


Fig. 14: Memory hit ratio (data)

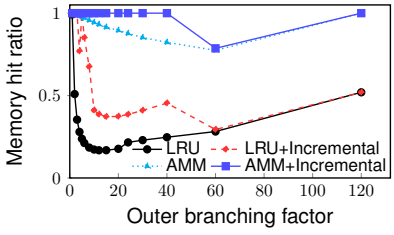


Fig. 15: Memory hit ratio (topology)

time by accessing more data in memory instead of disk, albeit via different mechanisms. To measure their effect, we also report the *memory hit ratio*, which is the fraction of data accesses that read data residing in memory.

Number of worker nodes. We observe the effect of the number of worker nodes on the completion time for the synthetic job. We vary the workers from 2 to 12. The input data per worker is constant, so the aggregate input size increases as workers are added.

Fig. 10 shows the rate at which the input data is processed. AMM with incremental is the best performing followed by LRU with incremental, which indicates that the incremental `choose` yields the biggest benefits in terms of completion time. When not using incremental evaluation, AMM still performs better than LRU alone.

The scaling behaviour for AMM and incremental remains unchanged as we increase the number of workers, which means that they do not negatively affect the scalability of the system. Across the approaches, we observe sublinear scaling due to the overhead of the extra workers in our current implementation. Fig. 13 shows that, since the input size per worker is constant, the memory hit ratio is not affected by the number of workers.

Dataset size. Next we investigate how MDFs scale with increasing input dataset sizes. We vary the dataset size from 2 GB to 9 GB per worker. Each node has 10 GB of available memory.

Fig. 11 shows the completion times; Fig. 14 shows the memory hit ratios. For the memory hit ratio, we see different behaviour based on dataset size: initially the memory-hit ratio decreases up to 6 GB of data, and then remains constant for the rest of the experiment. During the first phase, completion times increase more than linearly with size because a growing amount of data is accessed from disk; once most are disk based, the time increases linearly. We also observe that the constant overhead of AMM is higher than that of LRU. This is more than offset by its reduction in disk accesses.

These results show how the memory hit ratio affects completion times. With AMM and incremental, MDF achieves better memory hit ratios, which leads to a reduction of the completion time.

6.3 What is the impact of MDF topology?

Next we investigate how the branching factor of an MDF influences the completion time and the memory hit ratio. We use the synthetic job with its two nested `explores`, and adjust the branching factors.

Incremental `choose` evaluation and AMM are different mechanisms to improve memory hit ratios to reduce MDF job completion time: AMM by controls eviction when necessary, and incremental greedily consumes data. Since AMM is affected by data access frequency, and incremental is affected by the availability of data to choose greedily, we create an experiment to control both of these with a fixed total number of branches in the MDF. We use 120 branches, which is a highly composite number: it allows many different branching factors for the inner and outer `explore` operators such that $|B_1 \times B_2|$=120.

Fig. 12 and Fig. 15 show the completion times and memory hit ratios, respectively, as the outer branching factor, B_1, increases. Incremental `choose` evaluation significantly reduces completion times, especially when the outer branching factor is low and the inner branching factor is high. This is because datasets are discarded by the inner `choose` earlier in the job, freeing up memory for other data. Incremental is less effective when the outer branching factor is high because more datasets must be kept until the `choose` operator of the outer `explore` later in the job.

AMM also outperforms LRU, particularly for high outer branching factors. Here some datasets are reused more often, increasing the utility of keeping them in memory rather than evicting them in favour of less-used datasets. By accounting for the increased utility, AMM shows stable behaviour across all branching factors.

In summary, incremental and AMM provide complementary improvements. incremental is beneficial when datasets can be processed greedily and discarded early; AMM helps when datasets must be evicted and not all datasets have the same access frequency.

6.4 How does resource usage affect MDFs?

Finally, we study how CPU usage and memory availability affect completion times of MDF jobs. We use the synthetic job with

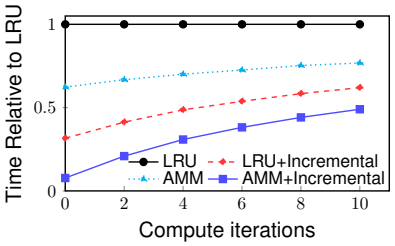


Fig. 16: Relative completion time (processing cost)

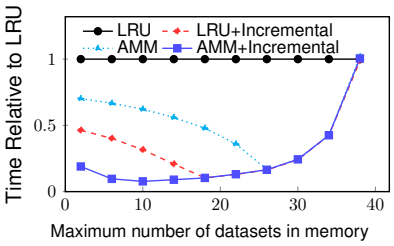


Fig. 17: Relative completion time (available memory)

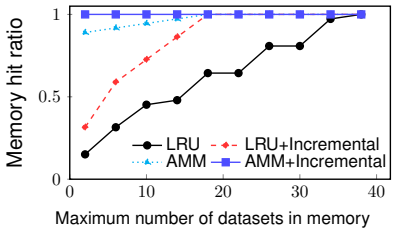


Fig. 18: Memory hit ratio (available memory)

$|B_1|=|B_2|=5$. This provides non-trivial branching at both `explore` levels while a reasonable fraction of all data fits in memory.

CPU. We observe the impact of CPU usage by changing the processing cost of branches in the synthetic job. Fig. 16 shows the relative completion time (normalised against the LRU baseline) when increasing the processing cost. As expected, AMM with incremental performs the best, followed by LRU with incremental. Using AMM alone results in the lowest benefit.

The relative improvement of AMM with incremental over LRU decreases with the processing cost: as the processing becomes more costly, the job becomes compute-bound, which means that the reduction in I/O operations due to AMM has less impact. Similarly, any reduction in completion time due to the incremental `choose` evaluation becomes offset by the more costly branch computations.

Memory. Next we increase the available memory at each worker while maintaining the same input data size for the synthetic job. Fig. 17 shows the completion time (again normalised against LRU), and Fig. 18 shows the memory hit ratio. When little data fits in memory, completion time with AMM and incremental is significantly better than with LRU because of fewer disk accesses. As the amount of available memory increases, all approaches experience better memory hit ratios, thus reducing completion time. This effect is strongest for LRU, which makes the least effective eviction decisions, and thus benefits the most from more available memory.

As more data fits into memory, the relative benefit of AMM with incremental reduces: as the memory hit ratios of all approaches reach 1 (Fig. 18), completion times also converge. LRU, however, requires more available memory to reach a high memory hit ratio, showing that AMM with incremental uses memory more efficiently.

7 RELATED WORK

Scientific workflow management. Pegasus [12], Azkaban [33], Luigi [32], and Oozie [21] orchestrate the execution of multiple jobs, but do not handle data sharing or optimised memory allocation.

Work on *provenance* of scientific workflows [11] focuses on how data and results of jobs can be shared, with a focus on cataloguing and providing access to datasets. MDFs are orthogonal to this work.

Data sharing in jobs. Tachyon [23] uses a storage layer to cache recently accessed in-memory data, following an LRU policy. Nectar [15] manages the storage and caching of datasets. It can share intermediate datasets generated by sub-computations of jobs.

In contrast, MDFs do not only target dataset caching but also schedule computation efficiently for exploratory workflows. MDFs are implemented in a dataflow system and do not require a caching layer. Sharing opportunities for intermediate datasets are made explicit in MDFs through the `explore` and `choose` operators, which permits sophisticated memory management policies.

Automated parameter exploration. Exploratory workflows are prevalent in machine learning algorithms due to the many hyper-parameters they need to configure. Spark ML [30] and Keystone ML [31] permit the declaration of hyper-parameters to be explored through ML-specific API functions when training models.

These approaches make assumptions about the nature of the dataflow job, which allows domain-specific optimisations such as grid or random search for hyper-parameters [5]. Instead, MDFs target arbitrary dataflow graphs and can execute complex dataflows with multiple phases of exploration and reuse of intermediate results.

DryadOpt [6] is a library implemented on top of DryadLINQ [39], performing exhaustive search of the solution space for optimisation algorithms implemented as dataflow graphs. It breaks the original problem into subproblems using a branch-and-bound approach, forming a search tree that can be executed in parallel. DryadOpt targets optimisation problems only, and its pruning strategy would not be applicable in other domains. MDFs require users to manually specify selection strategies, making them more generally usable.

Dynamic query optimisation. MDF modifies a dataflow on-the-fly, which is necessary to skip branches after a `choose` operator. Although most dataflow systems do not support dynamic dataflow topologies, the ideas have appeared in query optimisers for relational DBMS before. StarBurst [16] introduces a *choose-plan* operator [9, 14] that permits the execution of *dynamic query evaluation plans* in which the best query plan is only decided at runtime. In addition, support for dynamically controlling workflows appears in the context of optimising business processes [37] and scientific workflows [25]. In this paper, we have shown how dynamic changes to the dataflow topology along with judicious memory management can speed up modern exploratory workflows used by analysts.

8 CONCLUSIONS

We presented *meta-dataflows* (MDFs), a new dataflow model for exploratory workflows. The idea behind MDFs is to capture the expertise of users who want to explore a dataset with a range of related dataflow jobs with different algorithms or parameters. By specifying such exploratory workflows as a single integrated MDF, we demonstrated performance gains due to discarding unnecessary computation and utilising cluster memory better.

Acknowledgements. This research was partially supported by BP plc, a Google research faculty award, and the German Research Foundation (DFG) under grant agreement 246594964.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, et al. 2016. TensorFlow: A System for Large-scale Machine Learning. *USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2016).

[2] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. 1971. Principles of Optimal Page Replacement. *Journal of the ACM (JACM)* (1971).

[3] Alexander Alexandrov, Rico Bergmann, et al. 2014. The Stratosphere Platform for Big Data Analytics. *Conference on Very Large Data Bases (VLDB)* (2014).

[4] Apache. 2017. Hadoop. http://hadoop.apache.org/. (2017).

[5] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research* (2012).

[6] Mihai Budiu, Daniel Delling, et al. 2011. DryadOpt: Branch-and-Bound on Distributed Data-Parallel Execution Engines. *IEEE International Parallel & Distributed Processing Symposium (IPDPS)* (2011).

[7] Raul Castro Fernandez, Matteo Migliavacca, et al. 2013. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. *SIGMOD* (2013).

[8] Raul Castro Fernandez, Matteo Migliavacca, et al. 2014. Making State Explicit for Imperative Big Data Processing. *USENIX Annual Technical Conference (ATC)* (2014).

[9] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. *SIGMOD* (1994).

[10] Andrew Crotty, Alex Galakatos, et al. 2015. An Architecture for Compiling UDF-centric Workflows. *VLDB* (2015).

[11] Susan B. Davidson and Juliana Freire. 2008. Provenance and Scientific Workflows: Challenges and Opportunities. *ACM International Conference Management of Data (SIGMOD)* (2008).

[12] Ewa Deelman, Karan Vahi, et al. 2015. Pegasus, A Workflow Management System for Science Automation. *Future Generation Computer Systems* (2015).

[13] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2001. *The elements of statistical learning*. Vol. 1. Springer series in statistics Springer, Berlin.

[14] G. Graefe and K. Ward. 1989. Dynamic Query Evaluation Plans. *SIGMOD* (1989).

[15] Pradeep Kumar Gunda, Lenin Ravindranath, et al. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. (2010).

[16] L. M. Haas, W. Chang, et al. 1990. Starburst Mid-Flight: As the Dust Clears. *TKDE* (1990).

[17] Ramanujam Halasipuram, Prasad M Deshpande, et al. 2014. Determining Essential Statistics for Cost Based Optimization of an ETL Workflow. (2014).

[18] Matthew Hill, Murray Campbell, et al. 2008. Event Detection in Sensor Networks for Modern Oil Fields. *DEBS* (2008).

[19] Frank Hutter, Holger H. Hoos, et al. 2011. Sequential Model-Based Optimization for General Algorithm Configuration. *LION* (2011).

[20] Michael Isard, Mihai Budiu, et al. 2007. Dryad: Distributed Data-parallel Programs From Sequential Building Blocks. *EuroSys* (2007).

[21] Mohammad Islam, Angelo K. Huang, et al. 2012. Oozie: Towards a Scalable Workflow Management System for Hadoop. *SWEET@SIGMOD* (2012).

[22] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning Multiple Layers of Features From Tiny Images. (2009).

[23] Haoyuan Li, Ali Ghodsi, et al. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. *ACM Symposium on Cloud Comp. (SoCC)* (2014).

[24] Derek G. Murray, Malte Schwarzkopf, et al. 2011. CIEL: A Universal Execution Engine for Distributed Data-flow Computing. *NSDI* (2011).

[25] Eduardo Ogasawara, Jonas Dias, et al. 2011. An algebraic approach for data-centric scientific workflows. *VLDB* (2011).

[26] Emanuel Parzen. 1962. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics* (1962).

[27] Fabian Pedregosa, Gaël Varoquaux, et al. 2011. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* (2011).

[28] Astrid Rheinländer, Ulf Leser, et al. 2017. Optimization of Complex Dataflows with User-Defined Functions. *ACM Surveys* (2017).

[29] A. Simitsis, K. Wilkinson, et al. 2013. HFMS: Managing the lifecycle and complexity of hybrid analytic data flows. *ICDE* (2013).

[30] Apache Spark. 2017. ML Pipelines. http://spark.apache.org/docs/latest/ml-guide.html. (2017).

[31] Evan Sparks, Shivaram Venkataraman, et al. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. *ICDE* (2017).

[32] Spotify. 2017. Luigi. https://github.com/spotify/luigi. (2017).

[33] Roshan Sumbaly, Jay Kreps, et al. 2013. The Big Data Ecosystem at LinkedIn. *ACM International Conference Management of Data (SIGMOD)* (2013).

[34] Ilya Sutskever, James Martens, et al. 2013. On the Importance of Initialization and Momentum in Deep Learning. *ICML* (2013).

[35] Kostas Tzoumas, Johann-Christoph Freytag, et al. 2013. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. *ICDE* (2013).

[36] Vinod Kumar Vavilapalli, Arun C. Murthy, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. *ACM SoCC* (2013).

[37] Marko Vrhovnik, Holger Schwarz, et al. 2007. An Approach to Optimize Data Processing in Business Processes. *VLDB* (2007).

[38] Sai Wu, Feng Li, et al. 2011. Query optimization for massively parallel data processing. *SOCC* (2011).

[39] Yuan Yu, Michael Isard, et al. 2008. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. *OSDI* (2008).

[40] Matei Zaharia, Mosharaf Chowdhury, et al. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012).

[41] Mohammed J. Zaki and Wagner Meira Jr. 2014. *Data Mining and Analysis: Fundamental Concepts and Algorithms*. Cambridge University Press.

[42] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. *Arxiv preprint arXiv:1212.5701* (2012).

A DISTRIBUTED DATAFLOW MODEL

Dataflow graph. A dataflow graph is a connected directed graph, $G = (V, E)$, where the vertices V are data processing *operators*, and the edges, $E \subseteq V \times V$, are data dependencies between them. We denote the pre- and post-sets of a vertex $v \in V$ as $\bullet v = \{v' \mid (v', v) \in E\}$ and $v\bullet = \{v' \mid (v, v') \in E\}$, respectively. In a dataflow graph, an operator v with $\bullet v = \varnothing$ is called a *source*, and an operator v with $v\bullet = \varnothing$ is a *sink*. Two operators v and v' are connected by a *path*, denoted by $\pi(v, v')$, if there exist edges $e_1, \ldots, e_n \in E$ with $e_i = (v_i, v_{i+1})$, $v_1 = v$, and $v_{n+1} = v'$.

Most existing systems execute dataflow graphs without cycles, i.e. the graph is a directed acyclic graph (DAG). To support iterative computation, cycles in the dataflow graph are either unrolled [40] or encapsulated by special iteration operators [3]. In this paper, we assume, without loss of generality, that dataflow graphs are acyclic, and iterations are unrolled.

Data model. We model the processed data in terms of *finite datasets* of a domain $\mathcal{D}$ without imposing assumptions on the structure of data (e.g. relational tuples or key/value pairs). We assume that datasets $d, d' \in \mathcal{D}$ can be concatenated, denoted by $d \oplus d'$. The semantics of operators in the dataflow graph is defined in terms of a function over datasets: for each operator $v \in V$ in $G = (V, E)$, there is an operator function $f_v : \mathcal{D}^i \to \mathcal{D}^o$ where $i = |\bullet v|$ and $o = |v \bullet|$ are the in- and out-degrees of the operator, respectively.

Execution model. *Stages* group sets of operators. Intuitively, a stage comprises operators for which execution at a worker can be pipelined. The respective dependencies are derived from a dataflow graph, $G = (V, E)$, whose edges E may specify *narrow* or *wide dependencies* [40]: there is a narrow dependency between operators $v \in V$ and $v' \in v\bullet$, denoted by $v \rightarrowtail v'$, if each partition produced by f_v is used in at most one partition over which $f_{v'}$ is evaluated (e.g. *map* and *filter* functions); there is a wide dependency if partitions produced by f_v are used in more than one partition when evaluating $f_{v'}$ (e.g. a *group-by* function). For dataflow graph $G = (V, E)$, a stage is a set of operators, $T = \{v_1, \ldots, v_n\} \subseteq V$, that have only narrow dependencies, $v_i \rightarrowtail v_{i+1}$, $1 \leq i < n$. We denote the set of stages of G that are maximal, i.e. all operators not contained in a stage T have a wide dependency with at least one operator in T, as $T_G \subseteq 2^V$.

A possible *execution order* of stages is induced by a topological sort of the vertices in G, ensuring that data dependencies between operators are satisfied. We lift the notions of pre- and post-sets from the vertices of a dataflow graph to stages: $\bullet T$ and $T\bullet$ denote the sets of stages that must be executed before and after stage T, respectively.

Based on the above notions, the execution of an MDF can be described in terms of *states*. A state (D, δ, μ) is characterised by a set of datasets, $D \subseteq \mathcal{D}$, and two functions: $\delta : \mathcal{N} \times \mathcal{D} \to \mathbb{N}_0$ assigns the size of a partition to a node and dataset; and $\mu : \mathcal{N} \to \mathcal{D}$ assigns partitions that are kept in memory to nodes. A state needs to be *valid*, i.e. the total size of partitions kept in memory at a node must not exceed its memory limit: for all $n \in \mathcal{N}$, it holds that $\sum_{d \in \mu(n)} \delta(n, d) \leq mem(n)$.

B DEPTH- VS. BREADTH-FIRST SCHEDULE

We now show that a breadth-first traversal by the scheduler would require the distributed dataflow system to store at least as many, and often significantly more, datasets after each stage has completed as depth-first traversal, even in the worst case. Fewer maintained datasets should correlate to fewer evictions to disk, which is why we use depth-first traversal for BAS.

B.1 Collapsed MDFs

We define a new structure, called a *collapsed* MDF, to analyse how many datasets must be maintained after stages of a certain depth complete when `choose` stages select a single dataset, which is the minimal MDF graph which is necessary to find the number of datasets at a given set of stages.

In order to generate a collapsed MDF we choose a set of *stages of interest*. These are all stages which are at the same point of execution in sibling branches. For instance, they may be all the children of the root, or all the grandchildren. We then construct the collapsed MDF which determines how many datasets are in the system when those stages are run, which means we can ignore any datasets which have already been discarded because they will no longer be used and datasets which have not yet been created. In order to analyse all stages in an MDF if may be necessary to create multiple collapsed MDFs, but we show that our conclusions hold for all of them.

When a group of stages are collapsed, they are replaced by a single stage with the same number of inputs as the first stage in the group and the same number of outputs as the last stage in the group. To create a collapses MDF, we collapse groups of stages with the following rules:

- (1) Any `explore`/`choose` structures that have encountered a `choose` before the stages of interest are collapsed into single stages. All intermediate datasets for the branches within the structure are fully discarded after the `choose`, and thus will no longer be maintained when encountering the stages of interest.
- (2) Any `explore`/`choose` structures that `explore` after the stages of interest are collapsed into single stages because they do not create datasets that affect the counts at the stages of interest. The result of these first two rules is that all `explore` stages in the collapsed MDF will appear before the stages of interest, and all `choose` stages will appear after.
- (3) `Choose` stages and children of `explore` stages are the only types of stages which change the number of datasets in the system. Any other stage discards its input as it creates its output, meaning the total number of datasets remains the same in those stages. Thus we can analyse subsequent stages as a single unit as long as we do not collapse an `explore` stage with its child or a `choose` stage with its parent. Any strings of stages which match this can be collapsed. This includes any stages which were collapsed in the first two rules.

Fig. 19 and Fig. 20 give an example of collapsing an MDF. Fig. 19 shows an MDF. The blue dotted box shows stages of interest which are at the same depth to be analysed by a collapsed MDF. Following the rules, any groups of stages in red boxes can be collapsed into single stages for analysing the number of datasets at the stages of interest, which results in the collapsed MDF in Fig. 20.

Fig. 20 further compares the two scheduling strategies. The scheduling order of a stage and the number of datasets which must be maintained after that stage executes is shown for both breadth- and depth-first traversal for every stage. It is clear in this example that for no stage does depth-first traversal maintain more datasets than breadth-first traversal. However, even in this small example, there is a stage where breadth-first must maintain 8 datasets to the 4 required by depth-first—a 2× difference. Both strategies hit their peak required number of datasets at the same stage, which is 9 for breadth-first, and 5 for depth-first.

B.2 Number of datasets maintained

We now determine how many datasets must be maintained by the system after each stage completes. Each stage outputs a single dataset, which is read as input only by its children. Once all stages which read a dataset have completed, it is no longer necessary to maintain that dataset. Thus the number of datasets to be maintained is the number of stages that have executed minus the stages whose children have all executed.

We make two simplifying assumptions: (i) every `explore` has the same degree of explorables, defined by a global breadth variable $B \geq 2$; (ii) sibling branches are symmetric, i.e. any nested `explore` operators are nested in all sibling branches.

If these assumptions do not hold we can analyse the offending parts of the collapsed MDF piecemeal. Note that the collapsed MDF is a structure of nested collapsed MDFs. We can set the stages of interest as the stages just before the `explore-choose` structure which breaks the assumption, and the analysis for those stages will be correct. We can then analyse the pieces from the offending pieces separately.

In the cases of depth-first search, analysing each piece, then adding it to the number of datasets maintained at the stage of interest which is its parent gives a correct accounting of the number of datasets maintained within the piece. In a breadth-first approach each stage must add not only the branches at the stage of interest which is the parent of the piece, but *also* the number of branches at the same depth in sibling which run before the piece in question.

Thus it is sufficient to prove that the number of datasets maintained for depth-first traversal is at least as many as the number maintained by breadth-first at the stages of interest which are the parents to each piece. Then it will also hold in each of the pieces.

Stages within the collapsed MDF are identified by two variables: a local depth variable d indicates the nesting level; the source and sink are at d=0. A local breadth variable b indicates in which order a stage is executed relative to the other stages within the same depth. For each depth, the stages are numbered from 1 to B^d.

For depth-first traversal, we assume no early or incremental `choose`. This considers a worst case scenario because it requires the

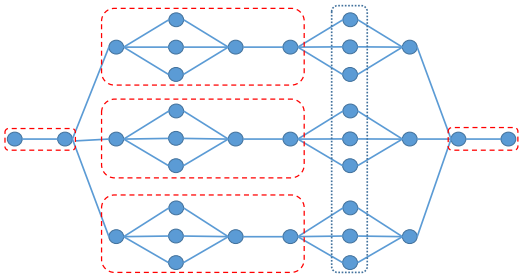


Fig. 19: Red dashed boxes are collapsible for finding the number of datasets maintained after stages within the blue dotted box (see Fig. 20)

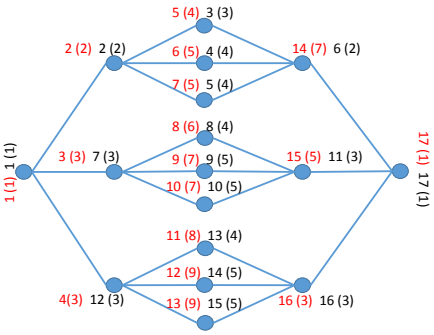


Fig. 20: Schedule order and maintained datasets for breadth-first (red) and depth-first (black) traversal

maintenance of a maximum number of datasets until a choose is ready to complete. For each depth up to d, the system must maintain the results from the choose of any sibling that has completed. It must also maintain the stages for the path along the current branch unless it is exploring a path from the last child at that depth. Thus the number of datasets which the system is required to maintain after a stage denoted by b, d in a depth-first execution of a collapsed MDF is:

$$1 + \sum_{x=1}^{d} \left\lfloor \frac{(b-1) - \left\lfloor \frac{b-1}{B^x} \right\rfloor B^x}{B^{x-1}} \right\rfloor + 1 - \left\lfloor \frac{(b-1) - \left\lfloor \frac{b-1}{B^x} \right\rfloor B}{(1-\frac{1}{B})B^x} \right\rfloor \quad (1)$$

For breadth-first traversal, the system only needs to maintain datasets that are produced by stages at two different depths: (i) those from the immediately previous depth with at least one child at the current depth, which has not been explored; second, those from the current depth, which have been explore. Thus the number of datasets which the system is required to maintain after a stage denoted by b, d in a breadth-first execution of a collapsed MDF is:

$$B^{d-1} - \left\lfloor \frac{b}{B} \right\rfloor + b \quad (2)$$

B.3 Proofs

Explore stages, d=0. In this case, there is no explore, so there is only a single stage to run. Thus there is no difference in the two scheduling strategies.

Explore stages, $b \leq B$ and $d \geq 1$. We again use the $1+(b-1)\sum_{x=1}^{d} d = b+d$ replacement for the first two terms of Eq. 1. Furthermore, when $b=B$ and $x=1$, $\left\lfloor \frac{(b-1)-\left\lfloor \frac{b-1}{B^x} \right\rfloor B}{(1-\frac{1}{B})B^x} \right\rfloor = 1$ within the range $b \leq B$, this makes it equivalent to $\left\lfloor \frac{b}{B} \right\rfloor$, so the entire replacement for Eq. 1 is $b + d - \left\lfloor \frac{b}{B} \right\rfloor$.

Thus we conclude that a breadth-first approach is required to maintain at least as many datasets if the difference between Eq. 2 and this upper bound of Eq. 1 is non-negative:

$$B^{d-1} - d \geq 0 \quad (3)$$

To determine when the lefthand side of this inequality is minimal, we analyse the behaviour and find the global minimum within the constraints $B \geq 2$, $d \geq 1$. We first find the derivatieve of the left side of the inequality with respect to B, which is $(d-1)B^{d-2}$. This is negative for all values of $B \geq 2$, $d \geq 1$. Thus the minimum occurs when B is minimal, i.e. B=2, and the inequality holds for all values if $2^{d-1} - d \geq 0$.

We find the derivative of the lefthand side of this new inequality with respect to d, $2^{d-1}\log(2) - 1$, where log is the natural logarithm. There is a single root at $d = 2\log_4(e) \approx 1.44 < 2$. The value at d=2 is positive, which means the derivative is positive for all values $d \geq 2$. Thus, within our constraints, the inequality is minimal when d is minimal, i.e. d=2, and the inequality holds for all values if $0 \geq 0$, which is true.

Therefore we have shown that a breadth-first traversal is required to maintain at least as many datasets as depth-first after any stage defined by $b \leq B$. This includes all stages at d=1, as those depths contain no stages such that $b > B$ in a collapsed MDF. □

Note that our conclusion is a boundary when the inequality is minimal. If we use other variables for B and d, there may be a much larger difference. For instance, at a stage at $d = 3$ when $B = 10$, Eq. 3 shows a difference of at least 98 datasets must be maintained, and the actual number may be higher if we used Equations 1 and 2 rather than the bounding equations.

Explore stages, $b > B$ and $d \geq 2$. For all values, $\left\lfloor \frac{(b-1)-\left\lfloor \frac{b-1}{B^x} \right\rfloor B^x}{B^{x-1}} \right\rfloor \leq B-1$. If it equals $B-1$, $\left\lfloor \frac{(b-1)-\left\lfloor \frac{b-1}{B^x} \right\rfloor B}{(1-\frac{1}{B})B^x} \right\rfloor = 1$. Thus $\left\lfloor \frac{(b-1)-\left\lfloor \frac{b-1}{B^x} \right\rfloor B^x}{B^{x-1}} \right\rfloor - \left\lfloor \frac{(b-1)-\left\lfloor \frac{b-1}{B^x} \right\rfloor B}{(1-\frac{1}{B})B^x} \right\rfloor \leq B-2$. Given this, Eq. 1 is bounded from above by $1 + \sum_{x=1}^{d} B - 1 = 1 + dB - d$.

We also note that $-\left\lfloor \frac{b}{B} \right\rfloor \geq -\frac{b}{B}$, so Eq. 1 is bounded from below by $B^{d-1} - \frac{b}{B} + b$. Thus a stage is required to maintain at least as many datasets in breadth-first traversal as depth-first if the following holds:

$$B^{d-1} - \frac{b}{B} + b - 1 - dB + d \geq 0 \quad (4)$$

To determine when the lefthand side of this inequality is minimal, we analyse the behaviour and find the global minimum within the constraints $d \geq 2$, $B \geq 2$, and $B + 1 \leq b \leq B$.

First we find the derivative of the lefthand side of the inequality with respect to b, which is $\frac{B-1}{B}$. For $B \geq 2$, this derivative is always positive. Thus the minimum occurs when d is minimal, i.e. $d = B + 1$, for all values of B and d, and the inequality holds for all values if $B^{d-1} - \frac{B+1}{B} + B - dB + d \geq 0$.

Next we find the derivative of the lefthand side of the new inequality with respect to B, which is $\frac{(d-1)(B^d-B^2)+1}{B^2}$. This is positive for all values of $d \geq 2$ and $B \geq 2$. Thus, for all values of d, the minimum occurs when B is minimal, i.e. B=2, and the inequality holds for all values if $2^{d-1} + \frac{1}{2} - d \geq 0$.

Finally we find the derivative of the lefthand side of this new inequality with respect to d. This is $\frac{1}{2}(2^d \log(2) - 2)$ where log is the natural logarithm. This has a single root at $d = 2\log_4(e) \approx 1.44 < 2$. The value at d=2 is positive, which means the derivative is positive for all values $d \geq 2$. Thus, within our constraints, the inequality is minimal when d is minimal, i.e. d=2, and the inequality holds for all values if $\frac{1}{2} \geq 0$, which is true.

Therefore we have shown that a breadth-first traversal is required to maintain at least as many datasets as depth-first after any stage defined by $d \geq 2$, $B \geq 2$, and $B + 1 \leq b \leq B$ in a collapsed MDF. □

Again, this is the minimal difference between the two approaches. If we look at the stage where $d = 3$, $B = 10$, and $b = 10^3$, Eq. 4 shows we should expect breadth-first traversal to need at least 972 datasets more than depth-first.

Choose stages. We note that, for depth-first traversal, the output of all stages between the `explore` and its matching `choose` stage are discarded by the time the `choose` completes, including the output of the `explore`. Additionally, no stages which are not between the `explore` and `choose` run, which means no datasets are created or discarded in any sibling branches. Therefore, the number of datasets necessary to be maintained after a `choose` is the same as the number after its matching `explore`, as determined by Eq. 1.

For breadth-first traversal, however, the number of datasets necessary to be maintained after a `choose` executes may not be the same as its matching `explore`. This is because the depth of the input is greater than the depth of the `explore` operators. The equation for the number of datasets maintained after a `choose` stage matching the `explore` stage denoted by b, d is:

$$B^{d+1} - Bb + b \tag{5}$$

We now find that, for any value of b, the difference between Eq. 5 and Eq. 2 is non-negative. Since a choose stage reads B inputs at a time, it is not necessary to consider values of b which are not multiples of B. Thus we can do not need the floor function for $\left\lfloor \frac{b}{B} \right\rfloor$.

$$B^{d+1} - Bb - B^{d-1} + \frac{b}{B} \geq 0 \tag{6}$$

The derivative of the lefthand side of this inequality with respect to b is $\frac{1}{B} - B$, which is negative for $B \geq 2$. Thus the inequality is minimised when b is maximised, i.e. b=B^d, and the inequality holds for all values if $0 \geq 0$, which is true. Thus each the system must maintain at least as many datasets after a `choose` stage as after its matching `explore` stage.

We already showed that the system must maintain at least as many datasets after `explore` stages in a breadth-first traversal of the collapsed MDF as in a depth-first traversal. We have now shown that a the system maintains the same number of datasets after `choose` stage as its matching `explore` stage in depth-first traversal, and at least as many after `choose` stage as its matching `explore` stage in breadth-first traversal. Thus by the transitive property it follows that the number of datasets maintained after a `choose` stage in breadth-first traversal is at least as many as after the same `choose` stage in depth-first traversal. □

```
val src = readFromFile("cifar_10.dat")
val model =
  EXPLORE(i=seq("Gaussian(0,0.1)", "Gaussian (0,0.05) ", "Uniform(−1,1)")
          r=seq(0.0001, 0.001, 0.005, 0.01),
          m=seq(0.25, 0.5, 0.75, 0.9), {
    val result = DNN.training(src,i,r,m)
  }).CHOOSE(validate(result), top-1)
writeToFile("results.csv", result)
```

Fig. 21: MDF of the Deep learning job

```
val src = readFromFile("time_series.csv")
val masked =
  EXPLORE(w=seq(2,3,4,5,6,7,8,9),
          t=seq(1.0001,1.0005,1.001,1.005,1.01,1.05,1.1,1.5), {
    val masked_res = Data.query(src,masking_query(w,t))
  }).CHOOSE(count(masked_res), threshold(0.8))
val marked  = Data.query(masked,marking_query)
val detected = Data.query(marked,detection_query)
writeToFile("results.csv", detected)
```

Fig. 22: MDF of the Time series analysis job

```
val src = randomStringIntPairs()
val result =
  EXPLORE(w1=seq(10,100,1000,10000), {
    val first_op = Math.op(src,w1)
    val first_res =
      EXPLORE(w2=seq(10,100,1000,10000), {
        val second_res = Math.op(src,w2)
      }).CHOOSE(int_value(second_res), max)
  }).CHOOSE(int_value(first_res), max)
writeToFile("results.csv", result)
```

Fig. 23: MDF of the Synthetic job

C MDF LISTINGS

This section shows the four MDFs used in our experiments in §6.

The Deep learning job covers three steps: data pre-processing, DNN model training, and model validation. The MDF, shown in Fig. 21, explores weight initialisation strategies, learning rates, and momentum values in the training of a model.

The Time series analysis job proceeds in three steps: masking of data points; marking of discrete events; and detecting sequences of discrete events. In our setup, we considered explorables relates to the masking of data points, varying the size of a sliding window and the threshold for removing data with the MDF in Fig. 22.

In the main part of the paper, we used the Data profiling job as a running example. Its MDF code was shown already in Fig. 3b.

Finally, the Synthetic job processes string/integer pairs. In two nested `explore`, algebraic operations update the integer values of tuples, as illustrated in Fig. 23.