Phoebe: A Learning-based Checkpoint Optimizer

Yiwen Zhu
Microsoft
yiwzh@microsoft.com

Matteo Interlandi
Microsoft
mainterl@microsoft.com

Abhishek Roy
Microsoft
abro@microsoft.com

Krishnadhan Das
Microsoft
krisdas@microsoft.com

Hiren Patel
Microsoft
hirenp@microsoft.com

Malay Bag
Facebook
malayb@gmail.com

Hitesh Sharma
Google
hitesh@outlook.com

Alekh Jindal
Microsoft
aljindal@microsoft.com

ABSTRACT

Easy-to-use programming interfaces paired with cloud-scale processing engines have enabled big data system users to author arbitrarily complex analytical jobs over massive volumes of data. However, as the complexity and scale of analytical jobs increase, they encounter a number of unforeseen problems, hotspots with large intermediate data on temporary storage, longer job recovery time after failures, and worse query optimizer estimates being examples of issues that we are facing at Microsoft.

To address these issues, we propose Phoebe, an efficient learning-based checkpoint optimizer. Given a set of constraints and an objective function at compile-time, Phoebe is able to determine the decomposition of job plans, and the optimal set of checkpoints to preserve their outputs to durable global storage. Phoebe consists of three machine learning predictors and one optimization module. For each stage of a job, Phoebe makes accurate predictions for: (1) the execution time, (2) the output size, and (3) the start/end time taking into account the inter-stage dependencies. Using these predictions, we formulate checkpoint optimization as an integer programming problem and propose a scalable heuristic algorithm that meets the latency requirement of the production environment.

We demonstrate the effectiveness of Phoebe in production workloads, and show that we can free the temporary storage on hotspots by more than 70% and restart failed jobs 68% faster on average with minimum performance impact. Phoebe also illustrates that adding multiple sets of checkpoints is not cost-efficient, which dramatically reduces the complexity of the optimization.

PVLDB Reference Format:
Yiwen Zhu, Matteo Interlandi, Abhishek Roy, Krishnadhan Das, Hiren Patel, Malay Bag, Hitesh Sharma, and Alekh Jindal. Phoebe: A Learning-based Checkpoint Optimizer. PVLDB, 14(11): 2505 - 2518, 2021.
doi:10.14778/3476249.3476298

1 INTRODUCTION

Big data platforms have democratized scalable data processing over the last decade, giving developers the freedom of writing complex programs (also referred to as *jobs*) without worrying about scaling them [3, 9, 15, 42, 44, 49, 50, 61]. However, this flexibility has also led developers into building very large analytical programs that can put the underlying platform under stress. For instance, the scale and flexibility of *Cosmos* [9, 13, 45], a big data analytics platform at Microsoft, empower developers to author large jobs composed of pipelines of SQL-like query statements which are compiled into query plans composed of up to thousands of *stages* (executable units composed by one or more operators) running over hundreds of thousands of processing units scheduled by YARN [53]. Figure 1 shows how the Cosmos workloads in one of the clusters have evolved over the past two years: we see that the total number of *tasks* per job (each corresponding to one process executed in one container) has grown by 34% (shown in blue), while the volume of input data has grown by 80% (shown in orange). Large analytical jobs lead to several operational problems.

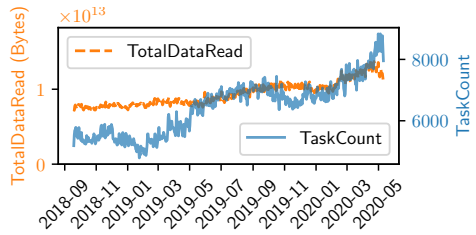

Figure 1: Cosmos job size

1. Large jobs result in machine hotspots that run out of local storage space due to temporary data. Big data systems typically persist intermediate outputs on local SSDs until the end of the query. However, large query plans end up consuming a substantial amount of temporary storage. Figure 2 (left) shows, for one Cosmos cluster, the cumulative distribution of available local SSD storage that is used for storing temporary data. We can see that for different Stock Keeping Units (SKUs), 15 – 50% of the machines run out of local storage on SSDs. This results in not only expensive spilling to HDDs and hence processing slowdown, but also an increase in incidents reporting job failures due to SSD outages. Currently, to avoid the SSD shortage, we need to either cap the number of containers running on each machine (thus wasting expensive CPU and memory resources) [1], or scale the CPU and memory together with the temporary storage in the newer SKUs.

2. Large jobs are prone to longer re-starting time in case of failures. Figure 2 (right) shows the failure rates of jobs with increasing runtimes. We observe that even though a majority of the

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476298

[1] Alternate solution could be to make YARN scheduler aware of the SSD utilization. However, additional parameters are not only harder to tune cluster-wide but also increase the scheduling overhead, which could translate into high costs at scale.

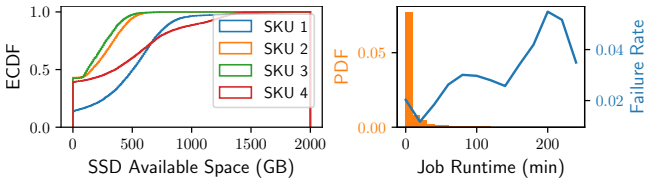

Figure 2: The Empirical Cumulative Density Function (ECDF) for SSD usage (left) and job failure rate with respect to job runtime and the probability density function (PDF) of job runtime distribution (right).

jobs finish within 20 minutes, the failure rates in larger jobs could range as high as 5%. Network/communication failures, changes in cluster conditions, transient system behavior, and user errors are some of the common reasons for job failures in Cosmos. Even though Cosmos already provides a lineage-based mechanism [60] for coping with task failures, improving resiliency to job failures and reducing recovery time are very important, especially as jobs and workloads scale.

3. Large jobs end up having worse query optimizer estimates. Errors in cardinality propagate exponentially [36, 39], and hence complex jobs are more likely to produce poor query plans. A recent trend [31, 54] suggests to re-optimize plans adaptively during job execution, but collecting statistics on-the-fly and on distributed intermediate results is highly not trivial and requires a major overhaul of the runtime system.

Checkpointing and its challenges. The above problems can be solved by "decomposing" large jobs into smaller ones that are separated by persistent (e.g., with 3-way replication) checkpoints on durable global storage. This will allow, for instance, to (1) free up intermediate data on hotspots even before the job completes; (2) fast restart failed jobs from the previous state; (3) collect statistics on the checkpoints and re-optimize large jobs into smaller ones with better estimates; and (4) reduce intermediate data in local storage to avoid wasting resources in newer SKUs. Prior checkpointing approaches include gathering statistics at execution time to dynamically select when to checkpoint [14, 46, 57], however, they require additional dynamic components that are not easy to implement reliably in large production systems such as Cosmos. Alternatively, compile-time approaches use estimates to propose optimal checkpoints during query optimization [10, 52, 58]. However, this requires accurate cost estimates, which is challenging since query optimizer estimates are often off by several orders of magnitudes [16, 28, 47], and even learned approaches are good for only relative plan comparisons [34, 35, 40] while still being significantly off in absolute values. Furthermore, all previous checkpointing approaches considered relatively small tree-shaped plans, whereas modern big data systems like SCOPE easily have complex Direct Acyclic Graphs (DAGs) with thousands of operators [24]. Not to mention, we need a generalized framework that can make checkpointing decisions on these DAGs for different scenarios with different objectives and constraints.

Introducing Phoebe. In this paper, we present Phoebe, a learning-based checkpointing optimizer for determining, at the compile-time, the decomposition (or the "cuts") of large job graphs in big data workloads. Phoebe builds upon the state-of-the-art CLEO [47] cost models and *fine-tunes* its operator-level predictions with historical statistics from past executions. This is possible since production big data workloads are often recurrent, e.g., > 70% in Cosmos [25]. Furthermore, checkpointing decisions only require cost predictions at stage boundaries (i.e., a set of operators that process a partition of data on a given node), which are precise since stages get executed with physical boundaries, thus making the fine-tuning approach highly effective. Phoebe applies a similar fine-tuning when predicting the time to live (TTL) for the output of each stage, which is needed to estimate how long the data lives on temporary storage. Phoebe uses a job runtime simulator and then fine-tunes its estimates with historical TTL of stage outputs.

Apart from cost models, Phoebe also introduces a scalable heuristics based checkpointing algorithm, that (1) can scale to millions of jobs in Cosmos workload; (2) it is two orders of magnitude faster than the optimal Integer Programming (IP) approach; and (3) yet strikingly close to the optimal in meeting the objective value. Finally, to the best of our knowledge, Phoebe is the first checkpointing framework that supports multiple objectives and constraints, and hence could be used in several different scenarios.

To summarize, we make the following key contributions:

- We present Phoebe, a learning-based system that uses past workloads for making checkpoint decisions over future queries in big data workloads. (Section 3)
- We describe accurate stage-wise cost predictors for stage output size, stage runtime, and stage output TTL by fine-tuning over historical statistics seen in the past. (Section 4)
- We introduce a scalable checkpoint optimization algorithm for large query DAGs and global constraints over hundreds of thousands of jobs, that can support several different checkpointing scenarios. (Section 5)
- We evaluate Phoebe over large production Cosmos workloads, and show how the various components contribute towards picking good checkpoints as well as the involved trade-offs. Our results show that Phoebe can free more than 70% of the local storage on hotspots, and reduce the recovery time for failed jobs by 64% on average, while increasing job latency by less than 3%. (Section 6)

Below we first provide a background on Cosmos and SCOPE before presenting each of our contributions.

2 BACKGROUND

Cosmos is the state-of-the-art big data analytics platform at Microsoft. It consists of hundreds of thousands of machines executing hundreds of thousands of jobs per day [42, 45]. Cosmos users submit their analytical jobs using SCOPE [9, 62], a SQL-like data flow dialect. SCOPE jobs are compiled into a Direct Acyclic Graph (DAG) of stages which in turn are executed in parallel by a YARN-based scheduler [13]. Figure 3 shows the execution plan for a 7-stage SCOPE job, with each rectangle representing a stage. Within a stage, there can be multiple operators, such as `Extract`, `Filter`, etc., chained together. Each stage is packed into a *task* that runs in parallel on different data partitions on different machines. During the execution, users can monitor the progress of each stage (green means finished, blue means waiting, and white means not started). Based on the execution dependency, we call the dependee an *upstream stage* and the dependent a *downstream*

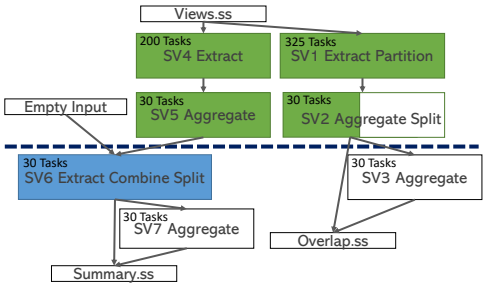

Figure 3: A job execution graph in SCOPE and one potential checkpoint decision (horizontal line).

stage. For instance, stage `SV2_Aggregate_Split` in Figure 3 is an upstream stage of `SV3_Aggregate` and a downstream stage of `SV1_Extract_Partition`. An upstream stage usually (but not always) finishes before its downstream stage. When a stage finishes, its output will be saved to the local SSDs of each server, which we refer to as *Temp Data Storage* in the rest of the paper. Cosmos emits telemetry data recording not only the detailed execution plan for every single job, but also the schedule for every stage, its execution time and output/input size, type of operators involved, etc.

Given the motivation to create persistent checkpoints, we want to decompose a job execution graph by selecting a set of stages for checkpointing, and redirecting their outputs to global HDD storage with 3 replicas. We refer to these selected stages as *checkpoint stages. We want to select the checkpoint stages carefully such that the objective* (e.g., minimizing temp storage load) *could be met, while constraining the global storage costs.* Finding the optimal checkpoint stages is similar to decomposing the execution plan and finding a cut in the graph. The dashed black line in Figure 3 illustrates an example cut in the execution graph for checkpoint selection. When we select Stages `SV_2` and `SV_5` as the checkpoint stages, we need to save the their outputs to global persistent stores. The space needed for the global store is proportional to the sum of the output sizes of Stages `SV_2` and `SV_5`.

Objectives such as minimizing the overall temp data storage of all jobs, or minimizing the recovery/restart time for a failed job, depend on the time a job lives t_u after each stage u, while the global storage constraints depend on the output size o_u of each stage. Furthermore, t_u, is a function of the runtime, r_u, of all stages in the execution graph, i.e., $t_u = f(r_1, r_2, ...r_k)$. Accurate estimations of o_u, r_u, and t_u are therefore crucial for a good checkpoint optimization. In the following sections, we first present an overview of Phoebe, then we will describe the three ML models used for stage-wise costs (o_u and r_u) and time-to-live predictions (t_u), respectively.

3 PHOEBE OVERVIEW

In this section we give an overview of Phoebe and highlight the design choices we have made. As discussed in the previous section, to determine the optimal cut(s) of an execution graph, it is important to estimate the output size, the runtime, and the time-to-live for each stage. Unfortunately, the estimates the query optimizer in big data systems are off by orders of magnitude [24], due to (1) large query execution DAGs where the errors propagate exponentially [36, 43]; (2) prevalent use of custom user-defined functions that are hard to analyze [56]; (3) recent works have exploited workload patterns to learn models for improving the cardinality estimates [16, 28, 56], but still these learned estimates are not accurate enough in absolute values; and (4) the presence of both structured and unstructured input data [51]. *Problem: state-of-the-art cardinality estimation approaches are not good enough for predicting actual output sizes. Design choice: Phoebe augments state-of-the-art learned cardinalities (i.e., CLEO [47]) by focusing on recurring jobs and exploiting historical statistics to instance-optimize the cardinality predictors.*

Previous work on estimating cardinalities focus on improving the query optimizer estimates at the operator level. For checkpointing, however, we need to: (1) estimate the costs at the stage-level, each consisting of multiple operators executing on a task in the same container; (2) operators within a stage could be pipelined in different ways when scheduled on distributed tasks, which makes it non-trivial to combine individual operator costs into stage costs; (3) stage outputs are persistent for the full duration of the job, therefore to estimate the storage costs we need to take into account this temporal dimension. *Problem: cardinality estimates at the operator level need to be aggregated at the stage level and augmented with a time dimension in order to properly model the storage cost. Design choice: Phoebe generates stage-level estimates starting from the operator-level one, and adds a predictor for the time-to-live of each stage.*

SCOPE-like big data engines have query plans that are DAGs of operators, not trees. Furthermore, Scope plans are complex: in our production workloads we have plans easily reaching thousands of operators. Prior works (e.g., MCSN [28], DeepDB [19], NEO [34], NeuroCard [59], TBCNN [37], and [35, 40]) suggest to use DNNs to "learn" the encoding of relatively simple query structures and mapped each operator to neural unit(s). *Problem: Mapping Scope complex plans into deep neural networks results in severe gradient explosion or vanishing problems [18]. Design choice: Phoebe captures the complex structure of big data query execution DAGs using a schedule simulator.* Therefore, in this work, instead of a full black-box approach, we combine the existing work of cardinality estimation with an explainable simulation process, which is a judicious mixture of domain knowledge and principled data-science that leads to optimal results tailored to our complex production workloads.

The checkpoint optimizer (Section 5) uses the above estimates to make the checkpoint decisions. *Problem: Production checkpointing applications may have different objectives while the traditional checkpointing frameworks are rigid. Design choice: Phoebe checkpointing algorithm is based on a "graph cut" algorithm that is adaptive to different objectives and constraints based on the specific application.*

Figure 4 shows the Phoebe architecture that is integrated with an already deployed workload optimization platform, namely Peregrine [22]. Phoebe consists of the following three modules:

(1) The stage cost models take as input the aggregated features at the stage level and uses machine learning methods to predict the duration of each stage, which is measured by the *average* execution time for *all* the tasks of the corresponding stage. Likewise, we also learn models to predict the output size of each stage, i.e., the size of the output of the last operator in the stage.

(2) The time-to-live (TTL) estimator consists of two steps. First, a job runtime simulator infers the start/end times for each stage based on the job execution graph. We assume a stage can only start once all its upstream stages have finished. TTL can be calculated

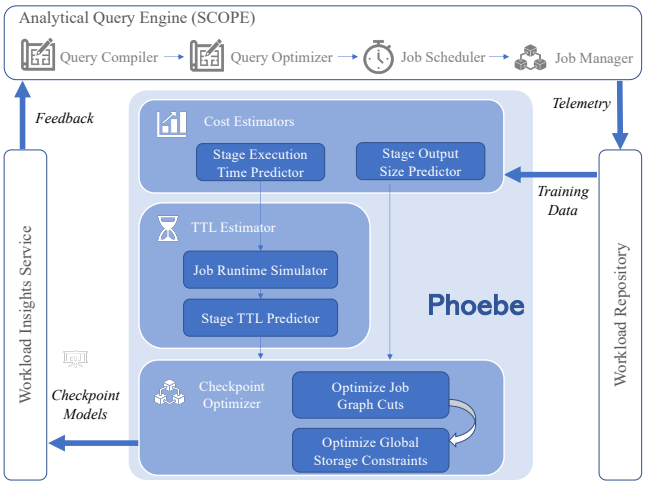

Figure 4: Phoebe architecture and its integration within the workload optimizer in SCOPE.

as the time interval between the estimated stage end time to the estimated job end time. Similar to model stacking, we use a meta-learning model to further improve the TTL prediction from the simulator, i.e., we take the estimated TTL and the estimated time from start (TFS, defined as the time interval from the job start time to the start time of a stage) from the simulator, and use another machine learning model to generate the final TTL prediction.

(3) The checkpoint optimizer uses as input the previous two modules, namely the estimated TTL and the output size of each stage, and selects the optimal set of global checkpoints given a particular objective function. To reduce the computation time for large workload sizes and to apply the storage constraints dynamically at runtime, we introduce a two-phase approach to find the graph cuts and apply the storage constraint separately.

For a new job, the SCOPE compiler makes a call to the Workload Insight Service [22], determines the checkpoint stages, modifies the query plan for materialization (similar as in CloudViews [23]), and sends the information to the job manager, which takes care of checkpointing those stages to the global store. The telemetry data from the query engine is collected into a workload repository and later used by Phoebe to re-train the models.

4 STAGE-LEVEL PREDICTORS

In this section, we discuss the stage-level predictors, namely the predictors for execution time and output size (Section 4.1) and the predictor for TTL (Section 4.2).

4.1 Execution Time & Output Size Estimators

4.1.1 Input Features. Cosmos implements a state-of-the-art query optimizer and learned cost models, CLEO [47]. CLEO generates a collection of cost models, one for each common sub-graph in the plans. Each sub-graph corresponds to the same root physical operator (e.g., `Filter`) and all upstream operators (e.g., `Scan`). The rationale is that cloud workloads are quite diverse in nature, and historically the one-fits-all models have failed to improve the estimated costs. CLEO [47] has proved to have 2 to 3 orders of magnitude more accurate than existing approaches. In this work, we leverage CLEO and extend it in three ways.

Table 1: Cost model features

Feature Group	Feature Name	Feature Description
Query Optimizer Features	*Estimated Cost, Estimated Input Cardinality, Estimated Exclusive Cost, Estimated Cardinality* for the last operator of the stage	Numeric features from the optimizer's internal information
Historic Statistics	the *Exclusive Time* and the *Output Size* for the job template and operator combination	The historic average of the statistics
Normalized File Path/Job Name	`Norm Job Name`, `Norm Input Name`	Text features

(1) We use CLEO operator-level features as input to generate stage-level estimates. Stage-level estimates do not correspond to any sub-graphs, but they are estimates of all the operators combined into a stage by the SCOPE optimizer. These input features are directly accessible from the SCOPE optimizer.

(2) We use historical data coming from the previous occurrences of the recurrent job to instance-optimize the predictors. In Cosmos, even with a large number of recurrent jobs, the parameters, inputs and execution plan can vary significantly over time. Therefore, it is important to not only capture repetitive patterns but also leverage the specific context of each of the stages in the workload.

(3) Finally, the input file paths and the job names often preserve information of file type, or locations and can be used as text features. For instance, a log file with file names including "`log`" usually consists of raw text in string format, which makes it more time-consuming to process compared to input with an ending of `*.ss` (structured steam [51], a SCOPE internal file format).

In summary, we constructed three groups of features as shown in Table 1. As we will see in Section 6.1, it is the combination of these input feature sets that yield the best prediction accuracy.

4.1.2 Model Implementation. NimbusML [12] is an open-source python package for ML.NET [1]. We tried different ML learners from NimbusML (e.g., linear regression, ensemble regression, etc.) and found that the LightGBM learner [27] is the best in terms of prediction accuracy for our use case. We developed two groups of models, each of them using only the non-textual features (i.e., query optimizer features and historic statistics): (1) **General model**, i.e., one model for all stages; and (2) **Stage-type specific model**. In fact, we observe that stages can be divided by their *type* based on the operators involved. Similar to CLEO [47] where the model is sub-graph specific, the stage-type corresponds to a unique set of (usually one or two) operators forming the stage, e.g., an `Extract_Split` stage has a `Process` operator followed by a `Split` operator. In the production workload used in this paper, we observed 33 stage types. We therefore train stage-type specific models, each with more homogeneous data. The stage-type specific models capture the heterogeneity of runtime variation across different combinations of operators. Given that we only select recurrent jobs for the checkpoint mechanism, it is desirable for the cost model to "overfitted" the selected recurrent jobs.

To leverage text features such as `Norm Input Name` and the `Norm Job Name` where simple One Hot Encoding [6] is not possible, we trained a customized word embedding using a language model [5] and integrated it with another DNN model with 2 hidden layers to predict the final targets as a benchmark. We host the end-to-end model training process on Azure ML for better experiment tracking and model archiving [11].

4.2 Time-to-Live Estimator

The time-to-live (TTL) estimator predicts the average lifetime of the intermediate output of each stage, defined as the time interval from the average end time of *all* tasks in the corresponding stage to the end time of the job. This is different from the estimation at the sub-tree/sub-query level because the TTL can be impacted by operators not included in the sub-tree which determine the job end time. Instead of training a DNN model to capture the complex dependency structure between stages (as in [34, 35, 40]), we introduce a simple schedule simulator to mimic the job execution process in Cosmos. The TTL estimator consists of two steps. First, a job runtime simulator takes as input (1) the stage execution time (i.e., the average task latency) estimated by the stage execution time predictor from the previous section; and (2) the execution graph to simulate the job execution process. Second, we develop another machine learning model to further improve the TTL prediction based on the simulator output. In the following sections, we discuss each step in more detail.

4.2.1 Job Runtime Simulator. The job runtime simulator estimates the start and end time of each stage based on the predictions of the stage execution time and the dependency relationship in the execution graph. To simplify the modeling, we assume strict stage boundaries, i.e., each stage can only start after all of its upstream stages have finished. A topological sorting [55] algorithm sorts all stages in a linear order based on the execution graph[2], such that an upstream stage executes before a downstream stage. The schedule simulator uses the linear ordering of the stages to estimate the stages' start and end times. For each stage, the simulator calculates its start time based on the maximum end time of all its upstream stages, and estimates its end time based on the estimated stage execution time from the stage execution time predictor.

Algorithm 1 shows the detailed schedule simulator process. Based on the linear ordering from the topological sorting algorithm, we schedule stages sequentially from the front of the ordered stack (from position 0). The TTL can be calculated as the time interval between the stage end time and the job end time.

4.2.2 Fine-tuning. While the simulator assumes a strict stage boundary and captures the dependency between stages, it doesn't simulate the pipelined operation. In Cosmos, for some stage types, tasks can start before all the tasks of upstream stages finish. Strict stage boundary assumption is helpful for computation efficiency; however, it potentially results in overestimating the TTL. Therefore, we create an ML model to systematically adjust for this bias by stage-type. We observed that some of the stage types usually have longer or shorter TTL, such as `Extract`, that always starts before all the other stages thus with longer TTL or an `Aggregate` stage that

[2]This is very similar to how the SCOPE job manager schedules tasks in Cosmos.

Algorithm 1: `ScheduleSimulator`

Input : execution graph G, ordered stack R, estimated execution time T
Output : start time for stages $D[s]$
end time for stages $P[s]$
Initialize : $D[s]$ =Null, $\forall s \in R$
$P[s]$ =Null, $\forall s \in R$

foreach *stage* $s \in R$ **do**
  MaxUpstreamEndTime = 0
  if s.UpstreamStages!= *Null* **then**
    foreach upstream $\in s$.UpstreamStages **do**
      MaxUpstreamEndTime =
      max {MaxUpstreamEndTime, P[upstream]}
  $D[s]$ = MaxUpstreamEndTime
  $P[s] = D[s] + T[s]$
return D, P

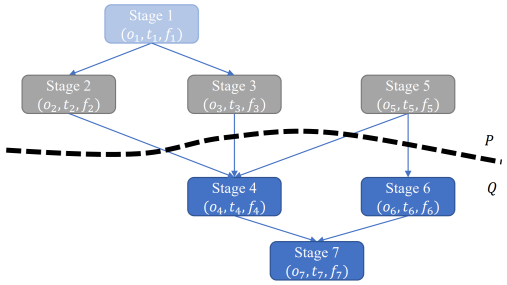

Figure 5: Graph cut in the integer programming.

tends to be placed towards the end of the job thus has shorter TTL. Therefore, we develop machine learning models per stage-type to have different adjustment mechanisms to achieve better accuracy. The input feature for the stacking model includes the estimated TTL from the simulator as well as the time from start (TFS), which is defined as the time interval between the start time of the job to the start time of the corresponding stage. Those two values define the "position" of this stage throughout the execution of the job.

5 CHECKPOINT OPTIMIZER

We now describe the checkpoint optimizer. Similar to Flint [46], we only consider a set of "frontiers" of the program's lineage graph. The checkpointing problem can then be naturally mapped to finding a cut in the execution graph. We categorize stages in a job execution graph into three groups (with respect to each cut):

(1) Group I: the checkpoint stages, i.e., stages that need to persist their outputs to the global storage;
(2) Group II: stages that have finished executing before the checkpoint stages; and
(3) Group III: stages that will execute after the checkpoint stages.

Phoebe's checkpointing optimizer is extensible: it can be tuned using different objective functions based on different checkpointing applications, as described earlier in Section 1. In particular, we discuss two of the applications in this section, namely freeing up temp data storage on hotspots and quickly restarting failed jobs. Below, we first show an IP formulation of the single-cut checkpoint problem. Then we show how it can be extended for multiple cuts.

Table 2: Notation

variable	description
u	stage index
S	set of stages
E	set of edges
o_u	output size of stage u
t_u	time-to-live of stage u
G	auxiliary variable, the total global storage usage
T	auxiliary variable, the total saving for temp data storage
α	cost factor of using global storage
z_u	decision variable representing if stage u is before the cut
d_{uv}	decision variable representing if edge (u, v) is on the cut
g_u	auxiliary variable representing if u is a checkpoint stage
C	set of cuts
c	cut index
$z_u^{(c)}$	decision variable representing if u is before the cut c
$d_{uv}^{(c)}$	decision variable telling if edge (u, v) is on the cut c

Finally, to solve the IP efficiently, we split our formulation into two sub-problems: (1) a heuristic approach to obtain efficiently the optimal solution without considering the global storage cost and with one cut; and (2) enforcing the global storage constraints.

5.1 Integer Programming

Table 2 summarizes the notation. Let us consider the case of adding one cut in the execution graph. For stage u, assume its output size $o_u(\geq 0)$ and time-to-live (TTL) $t_u(\geq 0)$ are known (see Figure 5). Let $z_u,\ \forall u \in S$ be a set of binary decision variables representing if stage u is executed before the graph cut, i.e., on the P-side of the cut (e.g., $z_u = 1$ indicates that stage u is before the cut). Let $d_{uv},\ \forall (u, v) \in E$ be another set of decision variables representing if edge (u, v) is on the cut, where E denotes the set of edges in this execution plan. For instance, in Figure 5, d_{24}, d_{34}, d_{54} and $d_{56} = 1$. The total number of decision variables is $|S| + |E|$, and the number of possible combinations is $2^{|S|+|E|}$ as they are all binary.

We can now use the two sets of binary decision variables to decide the grouping of a given stage u:

- Group I: $z_u = 1$ and $d_{uv} = 0,\ \forall (u, v) \in E$,
- Group II: $z_u = 1$ and $\exists d_{uv} = 1,\ \forall (u, v) \in E$, and
- Group III: $z_u = 0$.

The space needed for global storage is proportional to the sum of the total output sizes for stages in Group II. The checkpoint optimization problem can be formulated as an IP as follows:

$$\max_{\substack{z_u, \forall u \in S \\ d_{uv}, \forall (u,v) \in E}} \ \text{Obj}(z_u, d_{uv}) \tag{1}$$

$$\text{s.t. } G = \sum_{u \in S} o_u g_u, \tag{2}$$

$$g_u \geq d_{uv},\ \forall (u, v) \in E,\ \forall u, \tag{3}$$

$$d_{uv} - z_u + z_v \geq 0,\ \forall (u, v) \in E, \tag{4}$$

$$d_{uv} \in \{0, 1\},\ \forall (u, v) \in E, \tag{5}$$

$$z_u \in \{0, 1\},\ \forall u \in S. \tag{6}$$

Equation (1) can be replaced by different objective functions depending on the application. We will discuss two possible applications in the following sections. Constraint (2) calculates the total global storage needed from stages in Group II. It can be estimated by examining the output size on the upstream side of the edges on the cut. In Constraint (3), $g_u = 1$ for stages that have any edges on the cut ($d_{uv} = 1$). Note that, this requires that the objective function minimizes G and g_u. Constraint (4) ensures that if stage u is on the side before the cut (the side of P) and v is after the cut, $d_{uv} = 1$.

Multiple Cuts. Let $z_u^{(c)}$ denote the binary decision variable indicating whether stage u is before the cut $c \in C = \{0, 1, \cdots, K\}$, $K \geq 1$. Let $d_{uv}^{(c)}$ denote if edge (u, v) is on the cut c. The optimization problem for multi-cuts can be formulated as follows:

$$\max_{\substack{z_u^{(c)}, \forall u \in S,\ c \in C \\ d_{uv}^{(c)}, \forall (u,v) \in E,\ c \in C}} \ \text{Obj}\left(z_u^{(c)}, d_{uv}^{(c)}\right) \tag{7}$$

$$\text{s.t. } G = \sum_{u \in S} o_u g_u, \tag{8}$$

$$g_u \geq d_{uv}^{(c)},\ \forall (u, v) \in E,\ \forall c \in \{1, \cdots, K\},\ \forall u, \tag{9}$$

$$z_u^{(c-1)} \leq z_u^{(c)},\ \forall c \in \{1, \cdots, K\} \tag{10}$$

$$d_{uv}^{(c)} - z_u^{(c)} + z_v^{(c)} \geq 0,\ \forall (u, v) \in E,\ \forall c \in C, \tag{11}$$

$$\sum_{c \in C} d_{uv}^{(c)} \leq 1\ \forall (u, v) \in E \tag{12}$$

$$d_{uv}^{(c)} \in \{0, 1\},\ \forall (u, v) \in E,\ \forall c \in C, \tag{13}$$

$$z_u^{(c)} \in \{0, 1\},\ \forall u \in S,\ \forall c \in C. \tag{14}$$

The total number of decision variables is $(K+1) \cdot (|S|+|E|)$, and the number of possible combinations is $2^{(K+1)\cdot(|S|+|E|)}$. Constraint (9) is similar to (3) and requires that the objective function aims to minimize G. Constraint (10) ensures that the cut $c-1$ comes "before" the cut c, i.e., all the stages before the cut $c-1$ will also be before the cut c. Constraint (11) is similar to the single-cut formulation, and Constraint (12) ensures that the cuts do not overlap.

5.2 Freeing Temp Data Storage in Hotspots

With checkpointing, temp data for stages before the cut can be cleared when all the checkpoint stages finish as opposed to the end of the job. And this time difference is equal to the shortest TTL among those stages (the last one to finish). For saving temp data storage on hotspots, we maximize the saving calculated as the product of the shortest TTL and the sum of the total output sizes for stages in Group I and II, $\sum_{v \in S: z_v = 1} o_v \min_{u: z_u = 1} t_u$. Considering the cost factor of using global storage to be α, the IP formulation is as follows:

$$\max_{\substack{z_u, \forall u \in S \\ d_{uv}, \forall (u,v) \in E}} \ T - \alpha G \tag{15}$$

$$\text{s.t. } T = \sum_{u \in S} o_u w_u, \tag{16}$$

$$w_u \leq t + M(1 - z_u),\ \forall u \in S, \tag{17}$$

$$w_u \leq M z_u,\ \forall u \in S, \tag{18}$$

$$t \leq t_u + M(1 - z_u),\ \forall u \in S, \tag{19}$$

$$(2) - (6),$$

where, T calculates the total saving for temp data storage and M is a sufficiently large number. Given Constraints (17) and (18), $w_u = t$ if $z_u = 1$ else 0 in the maximization problem. Therefore, $w_u = 0$ for stages after the cut with $z_u = 0$. Similarly, Constraint (19) calculates the minimum TTL for stages before the cut in this maximization

problem. $t \le t_u$ when $z_u = 1$ and $t \le M$ when $z_u = 0$, which can be ignored. This set of constraints ensure that in the maximization problem, t is equal to the minimum of $t_u \ \forall u$ where $z_u = 1$.

For the cases with multiple cuts, the formulation is as follows:

$$\max_{z_u^{(c)}, \forall u \in S,\ c \in C} \quad T - \alpha G \tag{20}$$

$$\text{s.t. } T = \sum_{u \in S} o_u \sum_{c \in C} w_u^{(c)}, \tag{21}$$

$$\Delta z_u^{(0)} = z_u^{(0)}, \tag{22}$$

$$\Delta z_u^{(c)} = z_u^{(c)} - z_u^{(c-1)}, \ \forall c \in \{1, \cdots, K\}, \tag{23}$$

$$w_u^{(c)} \le t^{(c)} + M(1 - \Delta z_u^{(c)}), \ \forall u \in S, \forall c \in C, \tag{24}$$

$$w_u^{(c)} \le M \Delta z_u^{(c)}, \ \forall u \in S, \ \forall c \in C, \tag{25}$$

$$t^{(c)} \le t_u + M(1 - z_u^{(c)}), \ \forall u \in S, \ \forall c \in C, \tag{26}$$

$$(8) - (14).$$

Constraints (22) and (23) introduce $\Delta z_u^{(0)}$, indicating if stage u is before the cut 0, and $\Delta z_u^{(c)} \ \forall c \in \{1, \cdots, K\}$ indicating if stage u is between the cuts $c-1$ and c. Given Constraints (24) and (25), $w_u^{(c)} = t^{(c)}$ for stages with $\Delta z_u^{(c)} = 1$, else 0 in this maximization problem. $t^{(c)}$ in Constraint (26) calculates the minimum TTL for stages before the cut c. $t^{(c)} \le t_u$ when $z_u^{(c)} = 1$, and $t^{(c)} \le M$ when $z_u^{(c)} = 0$. In the maximization problem, $t^{(c)}$ is equal to the minimum of $t_u \ \forall u$ where $z_u^{(c)} = 1$. Combining Constraints (24), (25) and (26), for each stage, we calculate the corresponding temp data saving based on the minimum TTL for stages in the same group who are between the same pair of cuts. Since all constraints in the formulation are linear, the IP can be solved by existing solvers.

Solving the above IP over large execution graphs can take long, and given that we need to solve the IP for every job, it can quickly become operationally expensive. Therefore, we propose a heuristic-based solution to solve the IP formulation from the previous section at interactive speed, i.e., it can be run during job execution. The key idea is to maximize the objective and apply the global storage constraint separately, i.e., ignore the cost of using global storage when determining the cuts and consider adding only one cut. As a result, the IP formulation reduces to have only one set of decision variables, $z_u \ \forall u \in S$.

$$\texttt{(OptCheck1)} \quad \max_{z_u, \forall u \in S} T \tag{27}$$
$$\text{s.t. } (6),\ (16) - (19).$$

The above reduction has the following interesting property.

Proposition 5.1. *For any model primitives t_u and o_u, there exists optimal solutions z^* of problem* `OptCheck1` *such that there exists $v \in S$ such that $z_u^* = 1, \forall u \in S : t_u \ge t_v$ and $z_u^* = 0, \forall u \in S : t_u < t_v$.*

Proof. We prove by contradiction, suppose there exists an optimal solution z' such that there exists $u, u' \in S$, such that $z'_u = 0, z'_{u'} = 1$ and $t_u > t_{u'}$. Constructing a new solution z^* as follows:

$$z_u^* = 1, \tag{28}$$

$$z_v^* = z'_v, \forall v \ne u. \tag{29}$$

$$\sum_{v \in S : z_v^* = 1} o_v \min_{u : z_u^* = 1} t_u = \sum_{v \in S : z_v^* = 1} o_v \min_{u : z'_u = 1} t_u \ge \sum_{v \in S : z'_v = 1} o_v \min_{u : z'_u = 1} t_u. \tag{30}$$

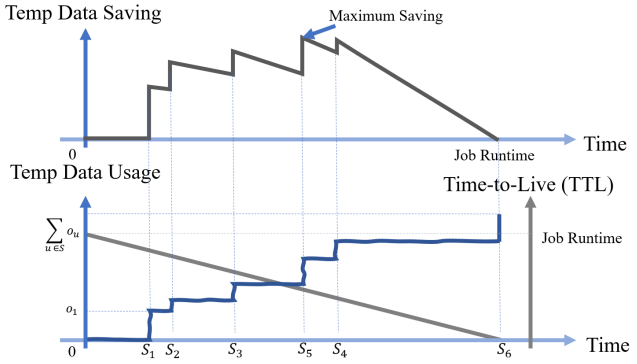

Figure 6: Potential temp data saving as a function of time.

This indicates that for any optimal solutions z' that does not satisfy the conditions stated in the proposition, we can always find another one z^* with at least the same objective value. □

Therefore, **we can enumerate the different combinations for z_u for all the stages in an efficient way**. Specifically, we start from the first stage (ordered by TTL) and set its z_u to 1 and the others to 0. Then, we incrementally add more stages to the side before the cut, set their z_u to 1, and evaluate the corresponding objective value for each new solution. This is equivalent to finding the optimal timestamp before which all stages have $z_u = 1$ and the total number of combinations is $|S|$. We consider the global storage cost separately and discuss that in Section 5.4.

This enumeration process is illustrated in Figure 6. Considering the execution process of a query job, after each stage finishes, the corresponding output will be saved to the temp data storage. Therefore, the total size of the temp data storage being used is increasing monotonically in time with more stages finishing. On the other hand, assuming that at the end time of stage s, we decide to clear the temp data storage for all the previous stages, the resulting saving for the temp data storage is equal to the TTL for this stage multiplied by the current temp storage usage size. The global storage space needed for Group III stages can be inferred based on the current execution status for each stage.

5.3 Restarting Failed Jobs

Some approaches [14, 43, 46] assume a distribution of the inter-arrival time for failures/interrupts given the mean time between failures/interrupts (MTBF/MTBI) as parameters. Others [10, 52, 58] assume a fixed probability of failure for an operator running on a worker node. In Cosmos, the execution time of tasks (30-40 seconds on average) has much less variation compared to the total job runtime (ranging from seconds to days). Therefore we find both the two assumptions above to hold.

Let δ denote the average failure rate for a task, it can be approximated by the average task runtime and MTBF, given that the average runtime is much shorter than MTBF:

$$\delta \approx \mathbb{E}(\text{Task Runtime}) / MTBF. \tag{31}$$

For Cosmos, $\delta \ll 0.05$, meaning that MTBF is in the order of hours. The probability of failure for stage u with v_u tasks is equal to the probability of having no tasks to fail, which follows a binomial

distribution:

$$1 - (1-\delta)^{v_u} \approx \delta v_u. \tag{32}$$

Let $\bar{t_u}$ denote the time from start (TFS) for stage u, indicating the time interval from the job start time to the stage start time. The IP formulation for maximizing the recovery time saving with a single cut and not consider the global storage is as follows:

$$\text{(OptCheck2)} \quad \max_{z_u, \forall u \in S} P_F \bar{T} \tag{33}$$

$$\text{s.t. } \bar{T} \leq \bar{t_u}(1 - z_u) + M z_u, \tag{34}$$

$$P_F = \prod_{u \in S} (1 - \delta v_u)^{z_u} \left(1 - \prod_{u' \in S} (1 - \delta v_{u'})^{1 - z_{u'}}\right), \tag{35}$$

$$z_u \in \{0, 1\}, \ \forall u \in S, \tag{36}$$

where Constraint (34) calculates the minimum TFS for all stages in Group III, i.e., $z_u = 1$. Constraint (35) calculates the probability of failure in one of the stages on the side after the cut but not before. One can prove the optimal solution of the problem `OptCheck2` has the similar property as in `OptCheck1`. Therefore, the same heuristic solution can be applied to incrementally add stages to the side before the cut and search for the optimal checkpoint time. To maximize the expected time saving for the checkpoint mechanism considering the probabilities of stage failures, the trade-off is between a higher failure probability and a larger saving for the restarting time. A figure similar to Figure 6 can be generated to show the changes of the probability of failing after a stage and the corresponding recovery time saving estimated based on the time from start (TFS) for the corresponding checkpoint stages, as a function of time.

5.4 Capacity Constraints on Global Storage

The global storage is a separate storage system where the data is 3x replicated and, for operational reasons, it is cleaned regularly (e.g., every 7 days). At SCOPE scale, this translates in a capacity in the order of a few PBs per day. In the previous sections we did not consider the cost of using the global storage. However, we can incorporate global storage constraints when selecting the *jobs* for checkpointing. Given the hundreds of thousands of jobs running every day, we can only checkpoint a fraction of them due to the checkpoint costs and overheads involved. Therefore, based on the objective value and the space needed for global storage for each job, we can be more selective about which job to checkpoint and achieve a high cost-benefit ratio. For instance, as shown in Figure 2, long-running jobs are more likely to fail due to the large number of tasks. Thus checkpointing can be enabled only for long-running jobs.

Consider a time period T (e.g., 1 day) and let the checkpoint budget for global storage be W. Let w_i denote the global storage space needed and π_i denote the ratio between the objective value and the global storage needed for job i. The problem of applying global storage constraint can be seen as an online stochastic knapsack problem [33]. The problem is challenging and it has been proved that there is no online algorithm that achieves any non-trivial competitive ratio [33]. In Phoebe, we propose a simpler threshold-based algorithm that takes into account the arrival rate of jobs and the distributions for the weights and value-to-weight ratio based on the model estimation. The intuition behind this approach is that we want to select the items that are most "cost-effective" and, ideally, an optimal policy will select the items with high values of π. With higher job arrival rates, the probability of selecting each job is further reduced. Thus, if an item with estimated weight w_i arrives with estimated value-to-weight ratio of π_i at time t, when the remaining resource capacity $n(t)$ is larger or equal to its weight w_i and its value-to-weight ratio is larger than the predefined threshold, π^*, the item is accepted, otherwise the item is rejected. That is,

$$D\big(w_i, \pi_i, n(t)\big) = \begin{cases} 1 \text{ if } \pi_i \geq \pi^* \text{ and } n(t) \geq w_i, \\ 0 \text{ otherwise,} \end{cases} \tag{37}$$

where, $D\big(w_i, \pi_i, n(t)\big)$ is the binary decision variable to accept/reject an item i. This policy ensures that in the set of accepted items, their average estimated value-to-weight ratios are larger than π^*. Assuming that the two distributions of weight and value are independent, we can define:

$$\pi^* = \Phi_\pi(1 - p), \text{and} \tag{38}$$

$$p = \frac{W}{\lambda T \mathbb{E}_w(w)}, \tag{39}$$

where $\Phi_\pi(\pi)$ is the cumulative distribution function for π, and p denotes the ratio between the total resource capacity, i.e., the total global available storage space, and the expected total weights for all the arriving items. The denominator calculates the expected total weights by taking the product of the arrival rate λ, T, according to Little's Law [32], and the expected weight for the items, $\mathbb{E}_w(w)$, assuming that the weight distributions for all items are i.i.d. Without capacity constraints, one can show that this threshold of π^* results in the expected total weights for the selected items to be equal to the total capacity.

5.5 Discussion

We now discuss the trade-offs with our checkpoint optimizer. The heuristic method for selecting the checkpoint stages is simple and fast. It does not require solvers for the optimization formulation, however it involves sorting stages as a pre-processing step and uses brute-force search for identifying the optimal set of checkpoints. Since we enumerate over all feasible solutions, considering the global storage constraint at the same time would be expensive as for each possible cut, the global data size needs to be computed by examining the full execution graph. Therefore, we incorporate the global storage constraint in a separate step, which dramatically reduces the computation complexity. The heuristic method, however, is not as flexible as the holistic integer programming solution in terms of adding additional constraints or considering multiple cuts. However, practically, it is more desirable to create single checkpoints in more jobs than multiple checkpoints in a given job.

The same formulation as in `OptCheck` and `OptCheck2` can be generalized to optimize the checkpoint decisions jointly across multiple jobs, e.g. jobs in the queue. However, the computation complexity increases significantly. In Phoebe, we consider a two-step approach to (1) choose the most profitable jobs for the checkpoint using an online stochastic knapsack framework as shown in Section 5.4; and (2) pick the optimal checkpoints for the selected jobs.

Finally, Phoebe's architecture is flexible: modules can be replaced or removed depending on the performance requirement. For instance, the ML predictor for TTL can be removed as the optimizer can use the estimated TTL from the job runtime simulator directly. In the extreme case, the optimizer can use some basic "prediction" for the stage costs, e.g., assuming a constant output size and runtime for each stage and simply make decisions based on the query

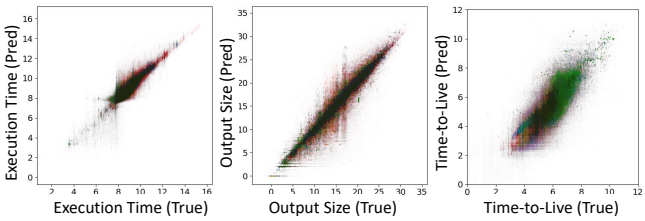

Figure 7: Accuracy for LightGBM models for predicting stage execution time (left), stage output size (middle), and time-to-live (right).

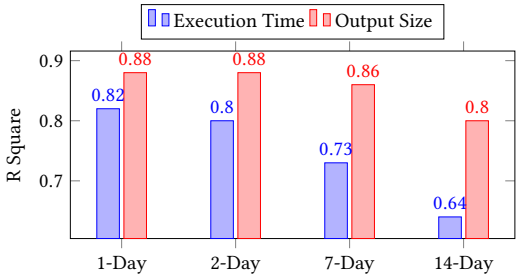

Figure 8: Performance accuracy with testing days further away from the training period.

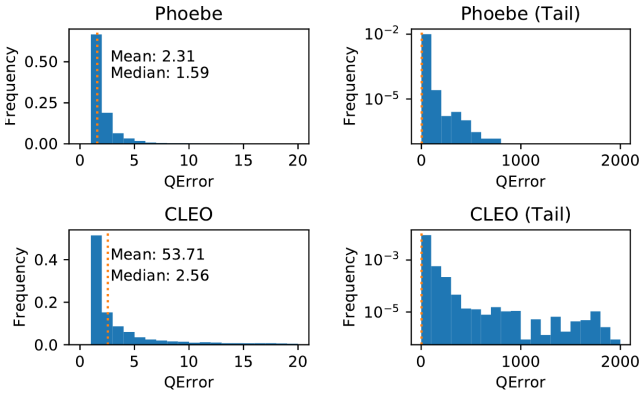

Figure 9: Accuracy for end-to-end job run-time prediction.

execution graph and the outputs of the job runtime simulator (results can be seen in Section 6). In this sense, the cuts are mostly selected based on the count of stages.

6 EXPERIMENTS

In this section, we evaluate Phoebe (1) using back-testing over production Cosmos workloads; and (2) executing a smaller set of jobs for impact on performance. We first evaluate the accuracy of stage-wise cost models (output size and execution time), and then evaluate the effectiveness of checkpoint optimization for two applications, namely, freeing up local storage on hotspots and quickly restarting failed jobs. We also provide anecdotal evidence of how Phoebe can help other checkpoint applications. For all experiments, we used the production workload for SCOPE jobs over different days (with hundreds of thousands of jobs per day).

6.1 Stage-Level Predictors

We evaluated the prediction accuracy for both the LightGBM and DNN models as discussed in Section 4.

LightGBM Models. We developed the general and stage-type specific LightGBM models with 5-days data (with 13.0 million samples) and testing on 1-day data (with 2.9 million samples). Figure 7 shows the prediction accuracy for the stage-type specific models color-coded by the stage types. Since there are 33 stage-types, we have 33 models each for predicting the execution time and the output size. The R^2 values for the LightGBM models are 0.85 and 0.91 for the execution time and output size predictions respectively.

The R^2 for predicting the time-to-live (TTL) is 0.35 (see right of Figure 7), which is not as good due to slightly over-estimation. Future work can focus on improving the accuracy by incorporating more specific rules in the simulator or applying more fine-grained stacking models, such as dependency-type specific models. However, the correlation between the prediction and the true value is relatively high (0.77). Note that, in the optimization module, the absolute values for TTL are not as important as the relative scale. A model that can capture the correct order of the TTL is anyways helpful in improving the checkpoint objective (see Section 6.2).

Based on Permutation Feature Importance (PFI) [7], the top 5 important features for one of the trained cost models are: *Estimated Exclusive Cost* (0.75), *Estimated Cardinality* (0.13), Historic `MergeJoin` Latency (0.10), *Estimated Input Cardinality* (0.06), and Historic `Reduce` Latency (0.06), measured by the reduction of R^2 if shuffling the corresponding entry of the features. We can see that **it is the mixture of estimated cardinality from the query optimizer and the historic information that jointly improves the prediction accuracy.**

To better understand the implication of the models, we ran additional experiments to use the *perfect cardinality estimation* as inputs. The R^2 metric is improved only by 0.04-0.05, which indicates the effectiveness of our approach that automatically adjusts for the biases in the inputs (i.e., cardinality estimates). If we use stage-type as features, the accuracy is not as good: for the prediction of the output sizes, the R^2 is reduced from 0.91 to 0.84; for the execution time instead the R^2 is reduced from 0.85 to 0.72.

Another important measurement of performance is the generalization of the trained model to future days, which determines the frequency needed to retrain the ML models. In production, we need to determine the frequency for retraining and deploying new models to keep up with the changes in data distribution. In Figure 8, we measure the performance of the trained model on testing data that is further away from the time frame where the training data is extracted from (e.g., 1 day after, 2 days after, etc.). We can see that the accuracy reduces gradually as the testing dates move over time.

DNN Models. We developed the general DNN models for both the execution time and output size using the same data. The R^2 values for the DNN models are 0.84 and 0.89 for the execution time, and output size prediction, respectively.

Benchmark of Existing Models. Figure 9 shows the distribution of prediction accuracy for the end-to-end job execution time from Phoebe (top) compared with CLEO [47] (bottom), measured by QError [36] in origin scale (e.g., in seconds). The QError is a commonly

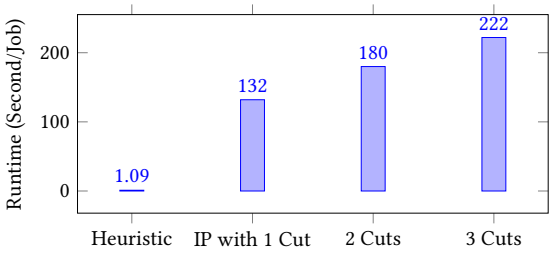

Figure 10: Runtime comparison of the heuristic solution with IP solution having different numbers of cuts.

used metric for measuring the accuracy of cardinality estimates [38]. It is defined by: $\text{QError}(y, \hat{y}) = \max\{y/\hat{y}, \hat{y}/y\}$ where y and $\hat{y}$ denote the actual and predicted values respectively. The QError of CLEO has a long tail (see right of Figure 9), meaning that for a small portion of jobs, the estimation is not good, and those are usually long-running jobs (>66% longer on average than all the jobs). This is consistent with our observation that the cost models have lower accuracy for large complex query plans.

Discussion. **The training time for DNN models is much longer than the LightGBM models.** Each (general) DNN model takes over 40 hours with a standard virtual machine with 6 Cores, 56 GB RAM, and an NVIDIA Tesla K80 GPU. This is partially due to the large size of the training data as well as the introduction of the LSTM layer for the featurization of the text columns. One potential improvement is to replace it with attention layers such as transformers to allow better parallelization with GPU. Compared with the general DNN models, the stage-type specific model based on LightGBM is slightly more accurate. And the training time is much shorter, in the order of minutes. Therefore, in the following sections, we use the prediction results from LightGBM as the input to the optimizer.

6.2 Freeing Temp Data Storage in Hotspots

We now evaluate the checkpoint algorithm for freeing up temp data storage on local SSDs in hotspots using back-testing, i.e., to see how well the algorithm would have done ex-post.

Integer Programming. We implemented the integer programming (formulation for freeing up temp data storage subject to different numbers of cuts as well as the cost for global data storage). We used Google OR-Tools [41] with CBC solver [17] on Azure ML [11]. Indeed, the IP solution is more general since it considers multiple cuts and different costs for the global storage. However, as illustrated in Figure 10, the processing time of using solvers is two orders of magnitude longer than the heuristic solution that looks for the optimal checkpoint time. The IP solutions provide interesting insights:

- **Adding more cuts is not cost-effective.** Figure 11 shows the Pareto frontier for freeing up temp data storage. The x-axis shows the median usage for the global storage normalized by the total original temp data usage across all jobs. The y-axis shows the median temp data saving also normalized by the original temp data usage. As we can see, adding more cuts is only helpful for large jobs (with >14 GB*Hour temp data usage), and not for all jobs. Therefore, we can have different checkpoint strategies for different types of jobs.

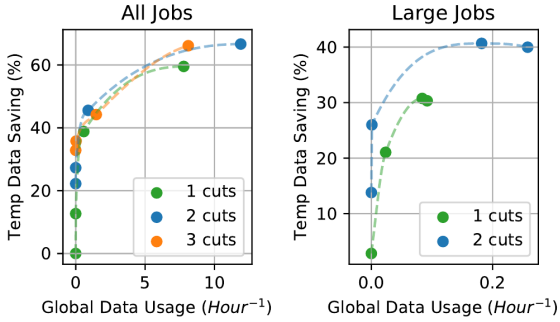

Figure 11: Pareto frontier for multiple cuts.

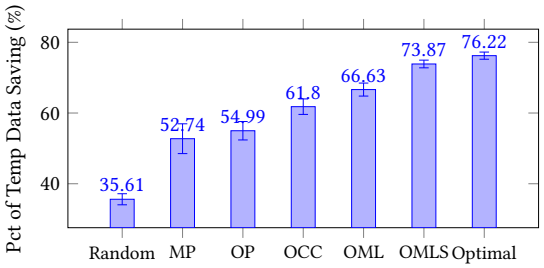

Figure 12: Percentage of temp data storage saving for different checkpointing approaches.

- **There are jobs with "free" cuts.** Based on the IP formulation that inserts a high cost for global storage, we found jobs with sub-graphs that are independent of each other, resulting in *free* checkpoints, i.e., not requiring any global storage.

Optimal Checkpoint Time. Figure 12 shows the daily average percentage of temp data storage saving for the different approaches based on 6-day's observations with approximately 405,000 jobs in one of the Cosmos clusters. The error bar shows the standard deviation. We consider the following approaches:

(1) *Random*: using a random checkpoint selector that randomly selects stages as the global checkpoints;
(2) *Mid-Point (MP)*: using the estimated stage scheduling from the job runtime simulator, cut the execution graph into two based on the *mid-point* of the total job execution time;
(3) *Optimizer + Estimated Cost (OP)*: using the estimated cost from the query optimizer for the output size/execution time for each stage as the inputs to the job runtime simulator and the proposed optimizer;
(4) *Optimizer + Constant Cost (OCC)*: similar to *Optimizer + Estimated Cost*, assuming a simple constant for the output size and execution time for each stage;
(5) *Optimizer + ML Cost Models (OML)*: the proposed optimizer that uses ML predictions as inputs and uses the estimated TTL from the job runtime simulator;
(6) *Optimizer + ML Cost Models + Stacking Model (OMLS)*: the proposed optimizer using the TTL from the stacking model;
(7) *Optimal*: the optimal off-line checkpoint optimizer based on the knowledge of the true output size and TTL for all stages.

We can see that the random optimizer can only free up 36% of temp data storage on average, measured by the portion of temp data

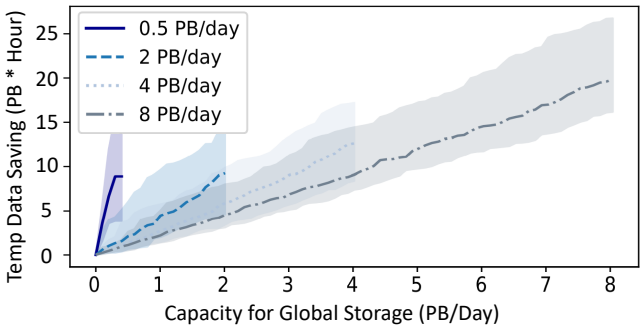

Figure 13: Cumulative temp data storage saving as a function of global storage used.

storage that can be cleared in the unit of Petabytes*Hour (PB*Hour). By further improving on the estimation of the output size, our proposed optimizer without the stacking model can free up to 67% of temp data storage on average, while with the stacking model the percentage further increases to 74%. This is not very far from the optimal off-line optimizer, which frees up to 76%. While we thought that using the cost from the optimizer (Estimated Costs) can improve the performance compared to using the constant cost, the corresponding temp data saving is actually worse, which is due to the large errors in cost estimation from the optimizer. Thus, **using the job runtime simulator and learned stage-wise costs significantly increases the saving in temp data storage on hotspots.**

Figure 13 shows the temp data saving with respect to the cumulative usage of the global data storage under different capacity constraints for the global storage. The 5^{th} and 95^{th} confidence level on the saving is also shown. We can see that with a larger capacity, the total temp data saving increases. However, because we are less selective about the jobs, the average value-to-weight ratio for the selected pool of jobs and the slope for the curve decreases. Therefore, to determine the best size of the capacity limit for the global storage, we should take into account the corresponding costs for the two types of storage and determine the optimal value.

6.3 Restarting Failed Jobs

We now present the experiment results for the heuristic method for maximizing the recovery time when restarting failed jobs. We used 1-day's data with approximately 62,000 jobs, and we evaluate 4 algorithms to select the optimal checkpoints:

(1) *Random*: using a random checkpoint selector that randomly selects stages as the global checkpoints;
(2) *Mid-Point*[3]: cut the execution graph based on the mid-point of the total job execution time based on estimated scheduling;
(3) *Phoebe*: the proposed method that uses the model prediction for the stage scheduling as well as the estimated probability of failures for each stage.

[3]Note that the cost-based approach proposed by [43] for minimizing the potential maximal total cost (execution cost and materialization cost for the dominant execution path) through enumeration is infeasible in Cosmos because of the size of the DAGs and the presence of spool operators [30]. Mid-point is a simple heuristic that extracts the "longest" execution path according to the stage execution schedule, and picks the mid-point timestamp and the corresponding last finished stages as the checkpoints. This strategy maximizes the total cost reduction on the dominant path.

(4) *Optimal*: the optimal off-line checkpoint optimizer based on the knowledge of the actual stage scheduling and failure probabilities for all stages.

Figure 14 shows the distribution of the percentage of recovery time saving for the 4 algorithms at the job level. The average percentage saving for the expected recovery time is 36% for the Random algorithm, 41% for the Mid-Point algorithm, 64% for Phoebe, and 73% for the Optimal algorithm. We can see that by introducing the predictions on stage scheduling and the estimated failure probabilities, the recovery time saving improves. However, more work can be done to further improve the estimation accuracy for the TFS.

6.4 Overheads and Production Test

Phoebe adds the following overheads on top of the SCOPE compiler: (1) metadata and model lookup: 15ms on average; (2) scoring and optimization: 1.09s; and (3) query optimization, which is negligible. As we materialize checkpoints by adding an additional stage that executes in parallel, the overhead for data writing is usually hidden by other parts of the job execution plan. In sum, we are expecting approximately 1s overhead in total compared to several minutes of end-to-end job compilation, which is acceptable.

We deployed Phoebe in the production environment and applied the checkpoint mechanism to over 1000 random analytic jobs (514 hours of total job execution time). The median increase in latency was just 1.8%. If a more constrained overhead is required, we can select simpler predictors as for example suggested in Figure 12. We tested on another 256 large jobs (with >1h job runtime). Figure 15 shows the distributions of percentage impact on latency and IO time. While the IO time for some jobs increased by >20%, the median increase for latency is only 2.6%. Among those long-running jobs, the average percentages of data checkpointed and temp data storage saved are 12.3% and 48.6% respectively.

6.5 Other Checkpoint Applications

We present a couple of anecdotal evidence on how Phoebe can also help other checkpoint applications. For instance, in one of our new SKUs in Cosmos, the local SSDs are not scaled in the same ratio as CPU cores. As a result, accommodating similar volumes of temporary storage would preclude from leverage the full compute capacity. Using Phoebe, we could reduce the temp storage load, and help increase the number of containers per machine by up to 28%. In another application, extremely large SCOPE jobs could run for several hours, making the query plan highly sub-optimal. Splitting one such large production job into smaller ones that have more accurate cost estimation thus better-optimized query plans resulted in its runtime to decrease from 30+ to 20+ hours.

7 RELATED WORK

In this section, we describe the different checkpointing mechanisms and systems. We also discuss the recent machine learning based approaches that predict characteristics like cardinality, operator cost, and resource allocation for query plans.

Runtime Checkpointing. This set of techniques make checkpointing decisions while the job is running. They are dynamic and more suitable for black-box workloads. By optimizing the checkpointing intervals [14], runtime checkpointing is easier to optimize

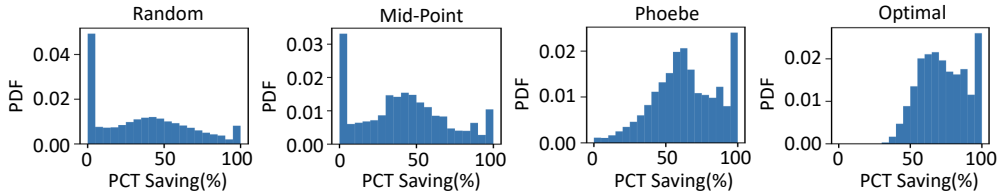

Figure 14: Distribution of expected recovery time saving (at the job level) for failed jobs.

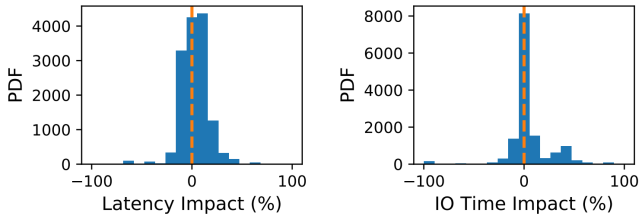

Figure 15: Checkpointing impact on latency and IO time.

and more applicable for transient resource environments. For example, Flint [46] and TR-Spark [57] focused on optimizing the checkpoint intervals and the selection of servers/tasks for replication to minimize the application runtime.

Compile-time Checkpointing. This set of techniques leverage query characteristics and propose optimal checkpoints at the task or operator-level. For example, FTOpt [52] developed a cost-based fault-tolerance optimizer to determine the (1) fault-tolerance strategy and (2) the frequency of checkpoints for each operator in a query plan. FTOpt works with non-blocking query plans only. Osprey [58] proposed the intra-query checkpoint mechanism for a distributed database system, and preserved the intermediate outputs for all sub-queries. But Osprey does not support general workloads with self-joins and nested queries. Similarly, [10] proposed a divide-and-conquer algorithm to solve a similar optimization problem. However, they do not consider temporary data saving costs or enforce any global storage constraints. The authors in [43] describe a cost-based checkpoint approach by enumerating all possible query plans and materialization configurations. Their algorithm relies on the cardinality estimates provided by the optimizer to find the query plans with the shortest dominant path under mid-query failures. However, prior work [56] has shown that big data query optimizers often under / over estimate cardinalities. They are also not applicable for distributed systems as they do not consider estimating costs for stages instead of operators. Production systems frequently encounter large jobs with hundreds or even thousands of stages, where the above algorithms prove to be inefficient.

Checkpointing in streaming systems. The checkpointing problem has also been studied in the context of stream processing systems. In streaming systems, fault-tolerant operator implementations employ logging [4], replication [29], and scaling out techniques [8] to reduce the cost of failure recovery. In contrast, we employ a restart-after-failure mechanism that minimizes the wasted computation in large-scale analytical programs.

We now review prior work that uses machine learning for cost prediction and query optimization.

Cost Models and Query Optimization [35] proposed a *plan-structured deep neural networks* to incorporate information from individual operator level to the full query plan and predict the job performance. Neo [34] used a similar DAG representation, the tree convolution [37], to predict the total execution latency for a given query plan. By searching over the space of the plans, the model discovered the optimal plan with the minimum expected execution time. [40] introduced the *subquery representation* for each state. The concept is similar to the *internal neural unit* [35], i.e., to concatenate the featurization of a sub-query with a new operation for the representation of a larger sub-query. Unfortunately, DNN solutions are not suitable for larger complex query graphs, that are found in big data systems such as Cosmos, due to the fact that many CNN layers will be stacked, which results in gradient vanishing or explosion. This is the same problem faced by DNN models before residual/highway connections were introduced [18, 20]. Whether a similar approach applied in our case is future work. [26] proposed to featurize query plans using topological features which doesn't capture the detailed dependency between operator and stages, therefore is too general to capture the heterogeneity among complex query plans. Similar to CLEO [47], [2] used prediction of child operators as input to ML models to predict the parent operators' cost. This model also leads to error prorogation that is not suitable for large query plans. ML has also been used to improve the estimation for cardinality [16, 28, 56]. Other works have considered persisting intermediate results that overlap across queries [21, 48] for computation reuse. In contrast, we focus on stage-wise cost models. To the best of our knowledge, Phoebe is the first system to propose cost models at the stage-level in a large distributed system with large, complex production workloads.

8 CONCLUSION

This paper revisits the checkpointing problem in the context of big data workloads, with complex query DAGs. We introduce Phoebe, a learning-based checkpoint optimizer to select the optimal set of stages to checkpoint, subject to different objective functions and storage constraints. Phoebe leverages multiple machine learning models built upon state-of-the-art cost models at the operator level, to accurately predict the cost of each stage, including the execution time, the output size, and the time-to-live. Based on the estimated costs, Phoebe's checkpoint optimizer selects the optimal set of stages that maximize the checkpointing objectives, such as freeing up temp data on local SSDs, or recovery time for failed jobs. We validated Phoebe using production workloads, with results showing that Phoebe is able to free up >70% of temp data storage on hotspots, improve the recovery time of failed jobs by >60% on average, and yet having acceptably low performance impact (less than 3%).

REFERENCES

[1] Zeeshan Ahmed, Saeed Amizadeh, Mikhail Bilenko, Rogan Carr, Wei-Sheng Chin, Yael Dekel, Xavier Dupre, Vadim Eksarevskiy, Senja Filipi, Tom Finley, et al. 2019. Machine Learning at Microsoft with ML. NET. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2448–2458.

[2] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley B Zdonik. 2012. Learning-based query performance modeling and prediction. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 390–401.

[3] Amazon.com, Inc. 2020. *Amazon Athena*. Retrieved July 4, 2020 from hhttps://aws.amazon.com/athena/

[4] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. 2005. Fault-tolerance in the Borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 13–24.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of machine learning research* 3, Feb (2003), 1137–1155.

[6] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. springer.

[7] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[8] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. 725–736.

[9] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[10] Ting Chen and Kenjiro Taura. 2013. A selective checkpointing mechanism for query plans in a parallel database system. In *2013 IEEE International Conference on Big Data*. IEEE, 237–245.

[11] Microsoft Corp. 2020. *Azure Machine Learning*. Retrieved July 4, 2020 from https://azure.microsoft.com/en-us/services/machine-learning/

[12] Microsoft Corp. 2020. *Nimbusml*. Retrieved December 14, 2020 from https://docs.microsoft.com/en-us/NimbusML/overview

[13] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, et al. 2019. Hydra: a federated resource manager for data-center scale analytics. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 177–192.

[14] John T Daly. 2006. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future generation computer systems* 22, 3 (2006), 303–312.

[15] Francesco Diaz and Roberto Freato. 2018. Azure Data Lake Store and Azure Data Lake Analytics. In *Cloud Data Design, Orchestration, and Management Using Microsoft Azure*. Springer, 327–392.

[16] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[17] J Forrest, T Ralphs, S Vigerske, B LouHafer, Kristjansson, jpfasano, EdwinStraver, M Lubin, HG Santos, and M Saltzman. 2020. *coin-or/cbc: Version 2.9.9*. Retrieved July 4, 2020 from https://zenodo.org/badge/latestdoi/30382416

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[19] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[20] Wei-Ning Hsu, Yu Zhang, Ann Lee, and James Glass. 2016. Exploiting depth and highway connections in convolutional recurrent deep neural networks for speech recognition. *cell* 50 (2016), 1.

[21] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[22] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing*. 416–427.

[23] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data*. 191–203.

[24] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. 2020. Microlearner: A fine-grained Learning Optimizer for Big Data Workloads at Microsoft. *Under Submission* (2020).

[25] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. 2016. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 117–134.

[26] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Bertty Contreras-Rojas, Rodrigo Pardo-Meza, Anis Troudi, and Sanjay Chawla. 2020. ML-based cross-platform query optimization. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1489–1500.

[27] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in neural information processing systems*. 3146–3154.

[28] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *9th Biennial Conference on Innovative Data Systems Research*.

[29] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. 2008. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment* 1, 1 (2008), 574–585.

[30] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment* 13, 3 (2019), 348–361.

[31] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. 2018. RIOS: Runtime Integrated Optimizer for Spark. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. 275–287.

[32] John DC Little. 1961. A proof for the queuing formula: L= λ W. *Operations research* 9, 3 (1961), 383–387.

[33] Alberto Marchetti-Spaccamela and Carlo Vercellis. 1995. Stochastic on-line knapsack problems. *Mathematical Programming* 68, 1-3 (1995), 73–104.

[34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1705–1718.

[35] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-structured deep neural network models for query performance prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.

[36] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.

[37] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[38] Parimarjan Negi, Ryan Marcus, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2020. Cost-guided cardinality estimation: Focus where it matters. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 154–157.

[39] Thomas Neumann and Cesar Galindo-Legaria. 2013. Taking the edge off cardinality estimation errors using incremental execution. *Datenbanksysteme für Business, Technologie und Web (BTW) 2019* (2013).

[40] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2018. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*. 1–4.

[41] Laurent Perron and Vincent Furnon. [n.d.]. *OR-Tools*. Google. https://developers.google.com/optimization/

[42] Raghu Ramakrishnan, Baskar Sridharan, John R Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, et al. 2017. Azure data lake store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 51–63.

[43] Abdallah Salama, Carsten Binnig, Tim Kraska, and Erfan Zamanian. 2015. Cost-based fault-tolerance for parallel data processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 285–297.

[44] Aurobindo Sarkar and Amit Shah. 2018. *Learning AWS: Design, build, and deploy responsive applications using AWS Cloud components*. Packt Publishing Ltd.

[45] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, et al. 2019. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-based Platforms. In *Proceedings of the ACM Symposium on Cloud Computing*. 441–452.

[46] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. 2016. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–15.

[47] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 99–113.

[48] Vikram Singh and SK Jain. 2016. A progressive query materialization for interactive data exploration. In *Proceeding of 1st International Workshop Social Data Analytics and Management (SoDAM'2016) Co-located at 44th VLDB*. 1–10.

[49] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[50] Jordan Tigani and Siddartha Naidu. 2014. *Google BigQuery Analytics*. John Wiley & Sons.

[51] Edward John Triou, Fei Xu, Hiren Patel, Jingren Zhou, et al. 2020. Analyzing multiple data streams as a single data object. US Patent 10,565,208.
[52] Prasang Upadhyaya, YongChul Kwon, and Magdalena Balazinska. 2011. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 241–252.
[53] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 5.
[54] Herman van Hövell Wenchen Fan and MaryAnn Xue. 2020. *Adaptive query execution in Apache Spark*. Retrieved Jan 20, 2021 from https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html
[55] Wikipedia. 2020. *Topological Sorting*. Retrieved July 4, 2020 from https://en.wikipedia.org/wiki/Topological_sorting
[56] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.
[57] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. 2016. Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 484–496.
[58] Christopher Yang, Christine Yen, Ceryen Tan, and Samuel R Madden. 2010. Osprey: Implementing MapReduce-style fault tolerance in a shared-nothing distributed database. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 657–668.
[59] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard. *Proceedings of the VLDB Endowment* 14, 1 (Sep 2020), 61–73. https://doi.org/10.14778/3421424.3421432
[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, 15–28.
[61] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.
[62] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. 2012. SCOPE: parallel databases meet MapReduce. *The VLDB Journal* 21, 5 (2012), 611–636.