Pipemizer: An Optimizer for Analytics Data Pipelines

Sunny Gakhar, Joyce Cahoon, Wangchao Le, Xiangnan Li, Kaushik Ravichandran, Hiren Patel, Marc Friedman, Brandon Haynes, Shi Qiao, Alekh Jindal, Jyoti Leeka[*]
Microsoft, Keebo
pipemizer@microsoft.com

ABSTRACT

We demonstrate *Pipemizer*, an optimizer and recommender aimed at improving the performance of queries or jobs in pipelines. These job pipelines are ubiquitous in modern data analytics due to jobs reading output files written by other jobs. Given that more than 650k jobs run on Microsoft's SCOPE job service per day and about 70% have inter-job dependencies, identifying optimization opportunities across query jobs is of considerable interest to both cluster operators and users. *Pipemizer* addresses this need by providing recommendations to users, allowing users to understand their system, and facilitating automated application of recommendations. *Pipemizer* introduces novel optimizations that include holistic pipeline-aware statistics generation, inter-job operator push-up, and job split & merge. This demonstration showcases optimizations and recommendations generated by *Pipemizer*, enabling users to understand and optimize job pipelines.

PVLDB Reference Format:
Sunny Gakhar, Joyce Cahoon, Wangchao Le, Xiangnan Li, Kaushik Ravichandran, Hiren Patel, Marc Friedman, Brandon Haynes, Shi Qiao, Alekh Jindal, Jyoti Leeka. Pipemizer: An Optimizer for Analytics Data Pipelines. PVLDB, 15(12): 3710 - 3713, 2022.
doi:10.14778/3554821.3554881

1 INTRODUCTION

Modern data analytics is often expressed as *data pipelines*, where multiple queries are interconnected by their outputs and inputs to execute critical business functions [5]. A wide range of tools have emerged in recent years to create and manage these data pipelines, including Airflow [1], Dagster [6], Azure Data Factory(ADF) [4], AWS Data Pipeline [3], and Google Dataflow [7]. These tools help users identify data pipelines and run them reliably in the cloud. Given how interconnected workloads have become [5], it is important to holistically optimize their performance and costs. Cloud providers run complex analytics pipelines comprising hundreds of thousands of jobs processing petabytes of data daily. The majority of these workloads are made up of interdependent recurring queries that form a data pipeline. Fig.1 illustrates one such production data pipeline consisting of thousands of queries from the Asimov production cluster, built on top of the Cosmos [16] platform at Microsoft. Asimov pipelines analyze telemetry from millions of

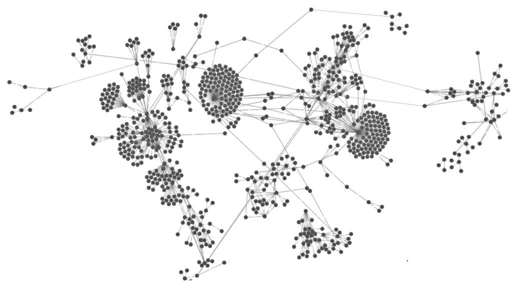


Figure 1: An Asimov production data pipeline. Each vertex is a distinct query; edges show inter-query dependencies.

Windows devices to derive business intelligence. Organizing analytics queries as pipelines helps the Asimov team track the status of devices, test new features, investigate bugs, and push out patches and new functionality swiftly [2]. In the figure, a node corresponds to a recurring query and an edge to a data dependency between two queries. Our analysis found that 73% queries produce data files (referenced as *streams* in Cosmos) that are consumed by one or more consumer queries, and 78% queries consume one or more streams produced by producer jobs.

Efficient analytics over data pipelines presents several challenges. First, *identifying* data pipelines is non-trivial and typically achieved by tedious, manual collaboration across large teams. Second, once identified, it is difficult for developers to *optimize* the performance and cost of data pipelines because they lack a holistic, global view of the entire pipeline. This is because producers and consumers lack clear contracts that define job boundaries and the whereabouts of jobs in the data pipeline. For example, consumer jobs do not typically come with metadata about the underlying input data properties. Similarly, producer jobs do not consider how their outputs are consumed by subsequent jobs. As a result, there is a critical need to develop an automated approach to optimize these complex data pipelines. Finally, optimizing data pipelines is computationally intractable due to the large number of jobs that have functional dependencies in time.

Optimizing data pipelines is radically different than optimizing individual queries. First, query engines are not aware of data pipelines. Current workflow tools that orchestrate data dependencies are completely siloed from the query engine. Second, prior work on multi-query optimization treats query workloads either as a set of queries [10] or, at best, as a sequence of queries [8], and focuses on optimizing the cumulative execution cost of all queries. Furthermore, these solutions have not been optimized to operate at the scale necessary for a service like Cosmos, which runs more than 650k jobs per cluster per day.

[*]Work done while all authors were at Microsoft. Shi Qiao and Alekh Jindal are currently at Keebo.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554881

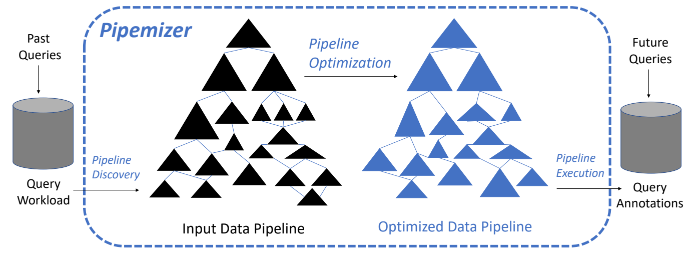


Figure 2: Pipeline Optimization.

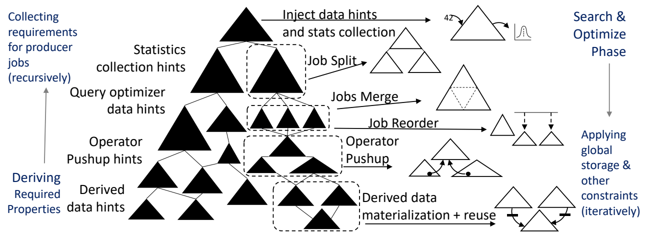

Collecting requirements for producer jobs (recursively)
Deriving Required Properties
Statistics collection hints
Query optimizer data hints
Operator Pushup hints
Derived data hints
Inject data hints and stats collection
Job Split
Jobs Merge
Job Reorder
Operator Pushup
Derived data materialization + reuse
Search & Optimize Phase
Applying global storage & other constraints (iteratively)

Figure 3: Architecture of Pipeline Optimizer.

In this paper, we present a demonstration of *Pipemizer*, a *pipe*line opti*mizer* that introduces a novel architecture consisting of pipeline discovery, optimization, and execution stages (section 3) to address the aforementioned challenges. The *Pipemizer* demonstration will allow attendees to identify, optimize and visualize data pipelines. Users will be able to enable and disable various Pipemizer optimizations (section 4) and view the performance advantages afforded by *Pipemizer*. In summary, we make following contributions:

- We demonstrate that *Pipemizer* is a novel pipeline optimization framework which optimizes query plans within a data pipeline.
- *Pipemizer* identifies data pipelines by mining producer-consumer relationship between jobs from telemetry. It introduces novel optimizations for efficiently executing this graph (section 3).
- SCOPE runs hundreds of thousands of jobs daily. This explodes the optimization complexity due to presence of very large number of job dependencies which together result in a much larger overall DAG. Thus rendering state-of-the-art multi-query optimization techniques impractical [17, 18]. *Pipemizer* introduces novel techniques for reducing this complexity and is the first framework tested at scale and in production at Microsoft.
- *Pipemizer* introduces an explainable approach that visually represents data pipelines and gives actionable recommendations.

2 *PIPEMIZER* OVERVIEW

We next describe the *Pipemizer* architecture which consists of three stages: Pipeline Discovery, Optimization, and Execution.

Pipeline Discovery. Pipelines are discovered in *Pipemizer* using telemetry logs from Airflow, ADF, etc. In Cosmos we discover pipelines by analyzing past query workloads in SCOPE using the Peregrine framework [14, 15]. Pipemizer first collects query execution plans and runtime metrics. Then it analyzes telemetry to identify data pipelines by finding recurring producer and consumer jobs, i.e., queries executed at regular intervals with same scripts. Finally, *Pipemizer* identifies streams emitted by Producer Jobs that are used as input to Consumer Jobs. This information is used to form an edge in the producer-consumer graph. The graph represents *Input Data Pipelines*, as shown in Fig. 2. Visualizing this graph helps teams in Microsoft identify and understand their data pipelines.

In this stage, *Pipemizer* also collects interesting properties such as job vitals, runtime statistics, and query access patterns of jobs.

Pipeline Optimization. *Pipemizer* takes the resulting producer-consumer graph as input. It then uses the graph to identify consumer job requirements that, when satisfied by producer jobs, lead to a reduction in resource utilization and processing time. *Pipemizer* achieves this by attempting to find the most efficient way (see section 3) to execute the graph while satisfying consumer job requirements. The output of this stage is an *Optimized Data Pipeline* shown in Fig. 2.

Pipeline Integration in Existing Engine. *Pipemizer* injects its optimizations into existing query execution engines through two channels. In one channel, the optimizer for the consumer job can request and consume targeted statistics from a producer job to correct for various derived properties. In the other, pipeline optimization recommendations are served as annotations to the optimizer for a producer job. These recommendations can be applied automatically in SCOPE using Peregrine [14, 15].

3 PIPELINE OPTIMIZATION

Pipemizer uses a two-phase, iterative approach for optimization: *derive* and *apply*. In the derive phase, *Pipemizer* collects, from each data pipeline, consumer jobs requirements (e.g., operator push-up, statistics) in a bottom-up manner. In the apply phase, *Pipemizer* applies its optimizations and generates recommendations, which are surfaced to users. Scalable Optimization: These phases are scaled by mapping them to massively parallel SCOPE engine.

3.1 Derive

The derive phase takes as input the producer-consumer graph from the Pipeline Discovery stage described in section 2. First, in a bottom-up manner, it identifies the consumer job requirements that producer jobs must satisfy. For example, the output of producer jobs must be sorted on P.x, statistics are required on P.y and P.yz, output of producer job must project away columns P.a and P.b, output of producer job must satisfy filter predicates in consumer jobs, etc. Since requirements of different consumer jobs can be conflicting (i.e., one consumer job may need a producer job's output to be sorted on P.x, while another requires the output to be sorted on P.y), *Pipemizer* chooses the requirement that optimizes the overall pipeline as described next in Apply phase.

Example requirements collected in derive phase are highlighted on the left side of Fig. 3 as triangles: • stats • layout • indexes • operator pushup • failure probability • scheduling/IO overhead • reuse opportunity (described later in seconds 3.3, 3.4, 3.5).

3.2 Apply

Pipemizer systematically combines the requirements collected in the *Derive* stage from each producer job along the following dimensions: sorting columns, partitioning columns, statistics collection columns, filter predicates, and projection push-up columns. Along each dimension, *Pipemizer* chooses consumer requirements that optimize the entire pipeline to be pushed to producer jobs. For

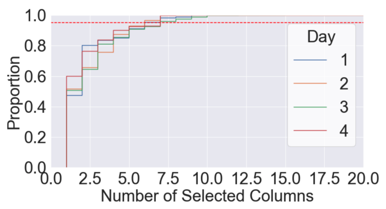


Figure 4: Number of columns requested per data stream in analysis of over 1M+ daily streams in Cosmos in last quarter.

example, for projection OPERATOR PUSH-UP it selects an intersecting set of columns that satisfies all consumer jobs when pushed up to producer jobs. For PHYSICAL DESIGN, *Pipemizer* also recommends partitioning and sorting the output of producer jobs, thus eliminating the need for multiple consumer jobs to re-partition/re-sort data while satisfying storage and compute constraints.

Recommendations generated by *Pipemizer* shown on right side of Fig. 3 are: • statistics generation • physical design • operator push-up • job split & merge • job reorder (described later in sections 3.3, 3.4, 3.5).

The rest of this section describes three key core sets of pipeline optimizations that we have implemented: (1) Pipeline-aware Statistics, (2) Operator Push-Up, and (3) Job Split, Merge & Job Reordering.

3.3 Pipeline-aware Statistics

A significant number of Cosmos' data streams are greater than 1 TB with hundreds of columns, complicating efforts in collecting statistics and profiling each stream. We assessed means to reduce the cycles required for statistics collection and observed that not all dimensions of a stream are necessary for pipeline query optimization. We thus apply a data-driven approach to streamline the generation of statistics for SCOPE.

We achieve this by implementing a column provenance feature that, at compile time, tracks what and how each input column is consumed as data propagates and evolves in a job's operator tree. This usage history is then shared with future producers in the pipeline. Current insights indicate that building statistics on the top five most accessed columns leads to decent coverage in our production pipeline as highlighted in Fig. 4. For our online pipeline optimization strategy, a producer job and its respective consumer jobs use a distributed cache and a protocol to log and exchange knowledge of interesting statistics. Conflicts may arise due to consumer jobs' interest in different portions of the same input schema, so we implemented an offline ranking algorithm that is executed periodically to rank the input columns by importance. The algorithm ranks columns based on frequency and compute spent on processing related portion of the data. Statistics generated include data distribution, heavy hitters and sketches.

3.4 Operator Push-Up

In Cosmos, we observe significant overlap across consumer jobs, i.e. part of query plan is duplicated across multiple consumer jobs, thus generating redundant cost. We focus on pushing common subexpressions to producer jobs in order to optimize resource consumption. We call this operation Operator Push-Up. An example push-up operation is highlighted below in which Q1 produces a stream q1 that is consumed by jobs Q2 and Q3. These consumer jobs apply a highly selective filter predicate on col3. Pushing this predicate to Q1 saves on storage and compute costs.

```
Q1: x = SELECT f(c1) AS c3 FROM t1; OUTPUT s1 TO "q1";
Q2: y = SELECT * FROM "q1" WHERE c3 > 10;
Q3: z = SELECT f1(c3) AS c4 FROM "q1" WHERE c3 > 10;
```

Pipemizer identifies common subexpressions and recommends their push-up to producer jobs. We show physical design and projection push-up operations in detail in this demo, described in section 4; however, the optimization is also applicable to other operators.

3.5 Job Implementation & Reordering

Cosmos users have the freedom to write both large and small jobs as shown in Table 1, thus straining the underlying system. We assess the heterogeneity among job runtimes on one of Cosmos' most utilized clusters. The distribution is bimodal, revealing that jobs can be bucketed into "large" (≥50 compute hours) jobs, versus "small" jobs (<50 compute hours). Large jobs are candidates for job split while small jobs are candidates for job merge.

3.5.1 Job Split and Job Merge. *Pipemizer* aims to merge small jobs to avoid the strain on global storage due to three-way replication in Cosmos. We achieve these merges using classical graph partitioning techniques that minimize the number of edges between groups of small jobs [13]. Large jobs, on the other hand, either fail needing longer restart times, or unnecessarily delay future jobs that depend on its output. We rely on Phoebe [20] to address the first issue by check-pointing intermediate stages of a job to global storage. We leave the latter challenge for future work.

Table 1: Job runtimes on a Cosmos Cluster over a week.

Compute Time (in hours)	#Jobs	Percentage of Jobs
≤ 1	765 K	13%
≤ 50	3352 K	58%
≤ 100	3901 K	68%
All	5729 K	100%

3.5.2 Job Reordering, Data Materialization, and Reuse. We schedule producer-consumer jobs to get maximum subexpression reuse. Polaris [9] is able to maximize sub-expression re-use for concurrently running queries. We leave the design of the scheduler, which maximizes compute reuse, for future work.

4 DEMONSTRATION

This demonstration is done using SCOPE preloaded with Asimov inspired producer-consumer graph on TPC-DS. Recommendations are generated by *Pipemizer*. The audience can apply the recommendations on queries. We encourage the audience to modify the queries and explore different execution scenarios, viz.: Pipeline Identification and Visualization; Physical Design, columnar push-up and statistics generation optimizations, as shown in Fig. 5.

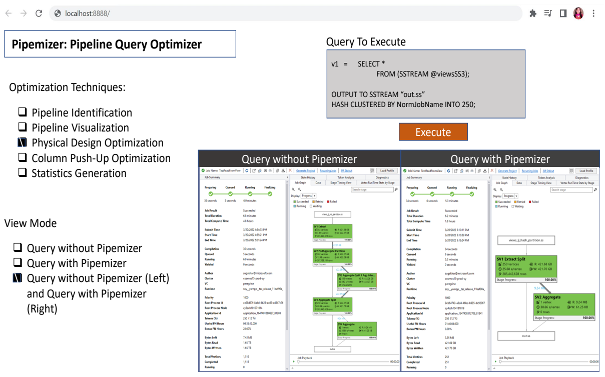
4.1 Scenario 1: Visualizing pipelines

The audience can input a set of queries and view the producer-consumer graph generated, similar to Fig. 1.

4.2 Scenario 2: Physical Design Push-Up

Many SCOPE users do not specify partitioning/sorting of output streams, causing consumer jobs with same physical design requirements to re-partition or re-sort streams multiple times, wasting compute and storage. In Asimov, more than half of the intermediate outputs are re-partitioned or re-sorted on the same attributes in consumer jobs, leading to high operational cost. With *Pipemizer*, partition/sorting can be Pushed Up from consumers to producers.

In our demonstration, the audience is encouraged to write their own queries, view recommendations generated by *Pipemizer*, apply generated recommendations, view performance of queries before and after applying recommendations. Fig. 5 shows the user interface, with an example job in SCOPE before and after applying physical design recommendation.



localhost:8888/

Pipemizer: Pipeline Query Optimizer

Query To Execute

```
v1 =    SELECT *
              FROM (SSTREAM @viewsSS3);

OUTPUT TO SSTREAM "out.ss"
HASH CLUSTERED BY NormJobName INTO 250;
```

Optimization Techniques:

- ☐ Pipeline Identification
- ☐ Pipeline Visualization
- ☑ Physical Design Optimization
- ☐ Column Push-Up Optimization
- ☐ Statistics Generation

View Mode

- ☐ Query without Pipemizer
- ☐ Query with Pipemizer
- ☑ Query without Pipemizer (Left) and Query with Pipemizer (Right)

Figure 5: Demo User Interface

4.3 Scenario 3: Projection Push-Up

4.3.1 Projection Push-Up: Consumer jobs often use only a subset of columns from input streams generated by producer jobs. Since these streams are saved in global storage in Cosmos, we project out unused columns from producer jobs to save on storage and compute costs. In our production clusters, we found that more than 25% of recurring streams have more than 10 unused columns. Thus eliminating unused columns saves storage and compute cost.

In this demonstration the audience will be encouraged to write their own queries, see the projection push-up recommendations generated by *Pipemizer*, observe reduction in storage and compute.

4.4 Scenario 4: Pipeline-aware Statistics

Fig. 6 is one example of a job whose total compute time improves by >10% when statistics are available on the join column, *JobID*. Since statistics collected on this column in its producer job suggest a lack of skew, the optimizer knows to inactivate the inappropriate skew-join data hint the user injected, thus improving the subsequent query plan and the job's runtime. In the demo, we show the audience how our ranking algorithm identifies the most important columns to collect statistics on and how the subsequent statistics collected results in a better query plan.

5 RELATED WORK

Pipeline optimization, especially in the context of Machine Learning pipelines [19], is prevalent. However, existing approaches do not

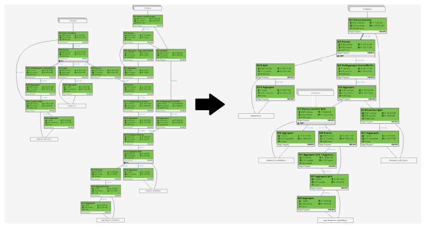


Figure 6: Query plan shown on right results in >10% improved runtime with pipeline-aware statistics.

work for queries expressed in SQL dialects. *Pipemizer* fills this gap and demonstrates solutions for optimizing large scale production database pipelines.

Other systems have looked at data pipelines from the perspective of scheduling [11, 12]. Wing [11] schedules jobs based on its impact on pending jobs. MQO [12] develops a schedule for maximizing concurrent execution of queries containing common subexpressions. Our work differs in two ways: (1) we view scheduling from the perspective of maximizing subexpression reuse by materializing within a storage budget; and (2) we introduce a holistic framework incorporating a suite of techniques for optimizing pipelines.

6 CONCLUSION

We demonstrated how *Pipemizer* leverages query workloads from big data analytical engines such as SCOPE to discover data pipelines, optimize resource consumption, and provide pipeline-aware optimization related annotations back to the query engine.

REFERENCES

[1] [n.d.]. Apache Airflow. https://airflow.apache.org/.
[2] [n.d.]. Asimov Windows Telemetry. https://mywindowshub.com/microsoft-uses-real-time-telemetry-asimov-build-test-update-windows-9/.
[3] [n.d.]. AWS Data Pipeline. https://aws.amazon.com/datapipeline/.
[4] [n.d.]. Azure Data Factory. https://metaflow.org/.
[5] [n.d.]. CIDR 2021 Keynote by Benoit Dageville, Snowflake Co-Founder President of Products. http://cidrdb.org/cidr2021/keynotespeakers.html.
[6] [n.d.]. Dagster. https://dagster.io/.
[7] [n.d.]. Google Dataflow. https://cloud.google.com/dataflow.
[8] Sanjay Agrawal et al. 2006. Automatic physical design tuning: workload as a sequence. In *SIGMOD*.
[9] Josep Aguilar-Saborit et al. 2020. POLARIS: the distributed SQL engine in azure synapse. *PVLDB* (2020).
[10] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-if" index analysis utility. *ACM SIGMOD Record* (1998).
[11] Andrew Chung et al. 2020. Unearthing inter-job dependencies for better cluster scheduling. In *OSDI 20*.
[12] Nilesh N Dalvi et al. 2003. Pipelining in multi-query optimization. *J. Comput. System Sci.* 66, 4 (2003).
[13] Per-Olof Fjällström. 1998. *Algorithms for graph partitioning: A survey.*
[14] Alekh Jindal et al. 2019. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing*. 416–427.
[15] Alekh Jindal et al. 2021. Production Experiences from Computation Reuse at Microsoft.. In *EDBT*. 623–634.
[16] Conor Power et al. 2021. The Cosmos Big Data Platform at Microsoft: Over a Decade of Progress and a Decade to Look Forward. *PVLDB* 14, 12 (2021).
[17] Prasan Roy et al. 2000. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*.
[18] Guoping Wang and Chee-Yong Chan. 2013. Multi-query optimization in mapreduce framework. *PVLDB* (2013).
[19] Doris Xin et al. 2021. Production machine learning pipelines: Empirical analysis and optimization opportunities. In *SIGMOD*.
[20] Yiwen Zhu et al. 2021. Phoebe: a learning-based checkpoint optimizer. *PVLDB* (2021).