Product Costing in Data Pipelines

Li Peng[1,2], Jie You[2], Feng Shi[3], Yihang Wu[1], Jianxin Wang[3], Weiran Liu[2], Xinyu Peng[1,2], Junhua Wang[2], Lei Zhang[2], Lin Qu[2], Jinfei Liu[1]
[1]Zhejiang University [2]Alibaba Group [3]Central South University
[1]{jerry.pl, steven.pengxy}@alibaba-inc.com, {yhwu_is, jinfeiliu}@zju.edu.cn
[2]{youjie.yj, weiran.lwr, junhua.wjh, zongchao.zl, xide.ql}@alibaba-inc.com
[3]{fengshi, jxwang}@csu.edu.cn

ABSTRACT

In data-intensive enterprises, data pipelines are essential for transforming raw data into valuable data products. To provide a sound basis for data pricing and cost optimization, enterprises usually perform *product costing*, which apportions the total cost of all resources (*e.g.*, CPU, Memory) consumed within pipelines to data products. However, product costing in data pipelines faces significant challenges not encountered in traditional physical pipelines, including the non-rivalrous nature of data that complicates the costing formulation, the prevalence of cyclic dependencies that distort the accuracy of results, and the massive scale of pipelines that leads to computational intractability.

In this paper, for the first time, we formalize the problem of product costing in data pipelines as a *manufacturing cost of data products (MCP) problem* to overcome the non-rivalrous nature of data. To solve MCP, we propose CostApp, a hybrid algorithm that achieves both exactness and scalability, even for cyclic and massive-scale pipelines. CostApp performs a dimensionality reduction by isolating cyclic dependencies via a feedback arc set (FAS). It aggregates costs onto a compact set of key vertices using a scalable iterative approach, transforming the intractable computational problem into a reduced linear system that can be solved exactly and efficiently by a basic matrix-based method. Experiments on both real-world data from Alibaba and synthetic data show that CostApp scales to process a production pipeline with over 28 million entities within one hour while preserving exactness, supporting pipelines up to $10^3\times$ larger than the direct matrix-based baseline. We open-source both the Alibaba pipeline dataset and the implementation to facilitate future research.

PVLDB Reference Format:
Li Peng[1,2], Jie You[2], Feng Shi[3], Yihang Wu[1], Jianxin Wang[3], Weiran Liu[2], Xinyu Peng[1,2], Junhua Wang[2], Lei Zhang[2], Lin Qu[2], Jinfei Liu[1]. Product Costing in Data Pipelines. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jerryp1/CostApp/.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

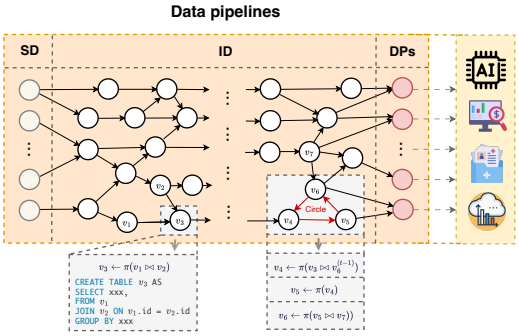


Figure 1: An illustration of data pipelines, where π denotes a join operation. $v_i^{(t-1)}$ represents the version of data entity v_i from the previous day, while entities without superscripts denote the current day's version.

1 INTRODUCTION

Over the past decades, the pervasive adoption of data-driven decision-making has established data as a critical asset [2, 23]. With both data volume and demand rising sharply, deriving value from data increasingly depends on the ability to process and refine it at scale. To transform large amounts of raw data into valuable data products, large data-intensive enterprises (*e.g.*, Google, Meta, and Alibaba) have built extensive data pipelines [3, 4, 20, 43]. These pipelines ingest diverse data sources and coordinate complex production dependencies to create a wide array of valuable data products that support business-critical scenarios, from operational decision-making to machine learning applications [29, 43, 45, 54, 55].

Data pipelines in data-intensive enterprises are characterized by *consolidation* and *periodicity*, presenting requirements for costing. Data is *non-rivalrous in consumption* [22, 37, 53]—it can be consumed repeatedly without depletion. To leverage this, enterprises integrate processing from numerous business domains into a *consolidated* digital supply chain, allowing frequently accessed data to be shared across multiple downstream products to maximize reuse value. Consolidated pipelines weave thousands of lineages together, creating a complex, many-to-many dependency graph. As shown in Fig. 1, source data (SD) is progressively refined through transformations into intermediate data (ID) and, finally, into valuable data products (DPs) ready for use across various businesses. These consolidated pipelines consume massive resources (*e.g.*, , CPU, Memory), presenting requirements for cost accounting. Furthermore, these pipelines operate with workload *periodicity*, typically executing recurrently (*e.g.*, daily extract-transform-load (ETL) jobs) to keep data up-to-date. This continuous and cumulative resource consumption means that even modest unit costs aggregate into substantial financial

expenditures over time. For example, at Alibaba, data pipelines processing terabytes of daily data incur resource costs equivalent to millions of dollars annually. Consequently, the interplay of complex consolidated dependencies and long-term periodic consumption makes cost visibility a necessity for operating data pipelines.

While resource consumption is distributed across all data entities (SD, ID, DP), business value is delivered exclusively through the final DPs (via monetization or usage). Thus, DPs constitute the natural unit for financial accountability. We term this *product costing in data pipelines*—apportioning the total costs of all resources consumed within pipelines to final DPs. This costing provides the necessary basis for data pricing [14, 37] and pipeline optimization [29, 54], yet it remains largely unexplored in existing literature.

Table 1: Comparison of physical and data pipelines.

	Physical pipelines	Data pipelines	Unique challenges
Consumption feature of outputs	rivalrous	non-rivalrous	consumption-based costing not applicable, complicates cost modeling
Type of dependencies	acyclic	cyclic	result accuracy of propagation-based costing solutions is distorted
Scale	small	large	exact costing solutions may be computationally intractable.

Challenges. However, product costing in data pipelines presents unique challenges not encountered in physical pipelines, as summarized in Tab. 1.

First, data's inherent *non-rivalrous nature in consumption* [22, 37, 53] fundamentally challenges cost modeling. In physical manufacturing pipelines, intermediate outputs are typically rivalrous physical goods. The costs embedded in a unit of a physical good are fixed (typically as its marginal cost) and fully transferred downstream upon its consumption. For example, when a car part is used in assembling a vehicle, its cost is fully attributed to that vehicle. Thus, consumption-based costing in physical pipeline naturally upholds the principle of *completeness* [1], meaning *all* consumed resources are fully accounted for within the costing *without* duplication. However, in data pipelines, data is not depleted when used, resulting in (nearly) zero marginal cost for each additional consumption [37]. Applying traditional costing creates a dilemma: allocating based on marginal cost [40] leads to severe under-costing, while charging the full cost to every consumer results in massive over-costing, both of which violate completeness. This difficulty challenges the formulation of a cost model that guarantees the completeness of costing.

Second, the workload periodicity of data pipelines leads to a time-versioned nature of data entities, inevitably introducing *cyclic dependencies* in the lineage graph. For instance, a daily user profile update often requires joining new behavioral data with its own previous snapshot, structurally forming a self-dependency or cycle in the lineage graph. Beyond self-loops, complex multi-hop cycles also arise, such as the loop formed by vertices v_4, v_5, and v_6 in Fig. 1, where v_4 relies on $v_6^{(t-1)}$, the version of v_6 from the previous day ($v_4 \leftarrow \pi(v_3 \bowtie v_6^{(t-1)})$). Such cycles are pervasive in production pipelines. They cause basic cost allocation methods based on topological propagation (see §3.3.1) to fail, as costs circulate indefinitely within loops rather than being fully attributed to final data products. Without proper handling, this leads to significant calculation errors that may mislead pricing strategies or undermine cost optimization efforts.

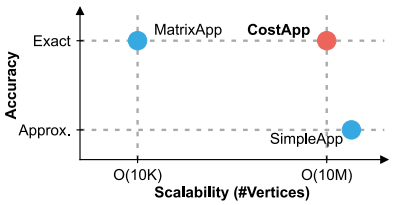


Figure 2: Approaches for product costing in data pipelines.

Third, the consolidation of data pipelines results in a *massive scale* of data entities. While traditional physical supply pipelines typically manage a limited number of entities (*e.g.*, thousands of intermediate units), a consolidated data pipeline at large enterprises often comprises tens of millions of data entities with complex dependencies *e.g.*, 28 million vertices in Alibaba. At this scale, exact costing solutions may become computationally intractable. Specifically, direct algebraic solutions (see §3.3.2) typically require matrix inversion with $O(|V|^3)$ time complexity. For graphs with millions of vertices, such cubic complexity is prohibitive. Therefore, a practical cost calculation method must be inherently scalable and designed to operate efficiently in a distributed environment.

Contributions. To address these challenges, this paper makes the following contributions:

First, to resolve the dilemma caused by non-rivalrous consumption of data, we formalize the problem of product costing as the calculating manufacturing cost of data products (MCP) problem. Unlike the consumption-based approach, we adopt a ratio-based cost model governed by the cost conservation property (Definition 1 in §3). This formulation guarantees completeness of product costing, ensuring precise cost apportionment to final data products without the under-costing or duplication pitfalls of traditional approaches. To resolve MCP problem, we propose two basic solutions: SimpleApp, a distributed, propagation-based algorithm that scales to large pipelines but only yields approximate results, and MatrixApp, which provides an exact solution via matrix inversion but is limited to small pipelines. These basic solutions highlight the fundamental trade-off between exactness and scalability, serving as building blocks for our hybrid approach.

Second, we propose CostApp, a hybrid algorithm that achieves both exactness and scalability by strategically integrating our basic methods. The core idea is to decouple the computational complexity by isolating cyclic dependencies. Specifically, CostApp first identifies a feedback arc set (FAS) and strategically inserts auxiliary *sink* and *source* vertices on these arcs to isolate these cycles. It then employs the scalable SimpleApp on the resulting acyclic graph to efficiently aggregate costs onto these auxiliary sink vertices and the final data products. This aggregation transforms the problem into a reduced linear system while preserving the final manufacturing cost of data products, which is then solved exactly using MatrixApp. This effectively performs a *dimensionality reduction*, shifting the bottleneck from inverting a massive $|V| \times |V|$ matrix (where $|V|$ is millions) to inverting a tiny matrix sized proportional

to the feedback arcs (often $< 0.1\%$ of $|V|$). This approach allows us to achieve exact results with high scalability simultaneously. A comparison that highlights its strengths is outlined in Fig. 2.

Third, we implement CostApp and conduct comprehensive experiments on both real-world data from Alibaba and synthetic data. Results demonstrate that CostApp successfully reduces a computationally prohibitive problem to a solvable one, achieving both result exactness and high scalability that outperform baselines. It successfully processes a production pipeline graph with over 28 million entities in an hour and has been successfully deployed in Alibaba's production environments.

To the best of our knowledge, we are the first to formalize the problem of product costing in data pipelines and to propose a scalable, exact solution. Our contributions are summarized as follows:

- We formalize the data manufacturing cost problem in data pipelines and propose two basic solutions for this problem: one exact for small pipelines and one approximate for large ones.
- We propose CostApp, a novel hybrid algorithm that provides exact and scalable solution for MCP.
- We validate CostApp's superiority through extensive experiments on both real-world and synthetic data. We open-source our real-world data pipeline dataset and implementation to facilitate future research.

2 PRELIMINARY

In this section, we introduce the background knowledge of data pipelines and Pregel-based distributed computation.

2.1 Data Pipelines

In this paper, we focus on *consolidated* data pipelines that consolidate numerous data transformation tasks (*e.g.*, SQL queries, MapReduce jobs) across diverse business domains for centralized management. Such pipelines are widely adopted by large data-intensive enterprises [3, 4, 43]. Their primary objective is to generate high-quality DPs through a sequence of data transformation operations. Typically, these jobs execute in batch mode: reading source data from storage, performing transformations (*e.g.*, cleaning, joining, aggregation, and feature extraction), and persisting the results back to storage. Formally, we denote a processing operation as o, which consumes a set of input data entities to produce a single output data entity. For instance, regarding entity v_3 in Fig. 1, o corresponds to a SQL join-projection operation $v_3 \leftarrow \pi(v_1 \bowtie v_2)$. Within these pipelines, three types of data entities are involved:

- Source data (SD). The starting point of data flow. Typically, they represent raw data collected from users or migrated from other data stores.
- Intermediate data (ID). The entities that are the output of the computation process but are not yet ready to be used as a product. They are materialized to enable data reuse and reduce redundant computation.
- Data products (DPs). The final data entities that are ready to be used as products and will not be processed further.

Model data pipelines as a weighted directed cyclic graph (DCG). To model the evolution of data in a structured and traceable manner, we represent the data pipelines using a data lineage [11, 16, 19, 20], a graph-based model that captures how data progresses from its origin to its final form. Specifically, the pipelines are modeled as an edge-weighted directed graph $G = (V, A, w)$, where

- V is a vertex set that denotes the set of data entities, such as SD, ID and DPs. The vertices with out-degree 0 (called *leaves*) represent DPs. The ones with in-degree 0 (called *roots*) represent SD, and the ones with both in-degree and out-degree at least 1 (called *internal vertices*) represent ID.
- A is a directed arc set that represents the transformation flows that produce new data entities from existing ones.
- $w : A \rightarrow [0, 1]$ is a function that assigns a static cost allocation ratio to each arc, as detailed in §3.

Crucially, G is a *cyclic* graph. As discussed in §1, the time-versioned nature of data processing often introduces dependencies on prior temporal snapshots of entities, structurally forming loops. This necessitates the use of a DCG abstraction rather than a DAG. While data pipeline graphs evolve over time, our analysis focuses on a static snapshot of the graph at a specific time for product costing, as the costing workload requires a fixed graph structure.

Other commonly used notations are summarized in Table 2.

Table 2: Notation table.

Notations	Description
$L(G), R(G)$ and $I(G)$	leaves, roots, and internal vertices
$U_G^1(v)$ (resp., $D_G^1(v)$)	set of all 1-hop upstream (resp., 1-hop downstream) vertices of v
$d_G^+(v)$ (resp., $d_G^-(v)$)	in-degree (resp., out-degree) of v

2.2 Pregel-based Distributed Graph Computation

Our solutions adopt the vertex-centric, distributed Pregel model [32] to support scalable graph-based solutions. The Pregel-based paradigm is well-suited for scalable graph-based computation because it decomposes computation into independent vertex actors that maintain local state (*e.g.*, accumulated cost) and exchange information only with neighbors. Our iterative MC calculation algorithm SimpleApp (see §3.3.1) and the algorithm FindFAS (see §4.2.1) for finding an FAS are both implemented using this Pregel-style abstraction to ensure scalability.

Execution proceeds in synchronized iterations called *supersteps*. In each superstep, a user-defined compute() runs in parallel on every active vertex: it processes incoming messages, updates local state (*e.g.*, adding received cost), and emits messages for delivery in the next superstep. A vertex may call voteToHalt() to become inactive and is reactivated if it later receives a message. The job completes when all vertices are inactive and no messages remain in transit.

3 PROBLEM FORMULATION AND BASIC SOLUTIONS

In this section, we formalize the data cost model and the target problem we aim to solve. Then, we present two basic solutions that partially solve the problem.

3.1 Data Cost in Data Pipelines

First, we formalize the cost of manufacturing a data entity v within data pipelines as a structured accumulation of resource expenditures across dependencies.

Direct cost (DC). As stated in §1, the manufacturing of data entities in pipelines is an ongoing process involving periodic recomputation and storage. Consequently, it incurs direct resource fees of CPU usage, memory usage, disk I/O, and storage usage [30, 42, 49, 53]. We define the *direct cost*, denoted as $C_{dc}(v)$, as the total quantified monetary cost incurred *solely* by producing and maintaining v:

$$C_{dc}(v) = C_{CPU}(v) + C_{Mem}(v) + C_{IO}(v) + C_{Sto}(v) \tag{1}$$

where $C_{CPU}, C_{Mem}, C_{IO}(v), C_{Sto}(v)$ denote the monetary cost of quantified resources usage. This represents the fundamental cost injected into the system. The total cost injected into the pipelines is thus $C_{total} = \sum_{v \in V(G)} C_{dc}(v)$.

It is worth noting that the specific composition of $C_{dc}(v)$ is *orthogonal* to our target product costing problem and solutions. Our model is agnostic to the underlying billing schemes and can be flexibly adapted to diverse real-world billing policies [49].

Manufacturing cost (MC). The *manufacturing cost*, denoted as $C_{mc}(v)$, is defined as the total cost of *all* resources consumed in manufacturing entity v across its entire production lineage. It comprises the entity's own direct cost and the inherited costs from its upstream inputs. Formally, for an entity v with upstream inputs $U_G^1(v)$, the manufacturing cost is defined recursively as:

$$C_{mc}(v) = C_{dc}(v) + \sum_{u \in U_G^1(v)} w(u,v) \cdot C_{mc}(u) \tag{2}$$

Here, $w(u,v)$ is the *cost allocation ratio*, representing the fraction of u's manufacturing cost allocated to v. The second term, $\sum_{u \in U_G^1(v)} w(u,v) \cdot C_{mc}(u)$, represents the *input cost* $C_{ic}(v)$, capturing the cumulative value inherited from direct upstream dependencies.

Cost allocation ratio. Unlike traditional consumption-based costing, our ratio-based strategy is consumption-independent, accommodating the non-rivalrous nature of data. Crucially, to ensure that the total cost of v is fully and precisely distributed across its lineage without duplication (*completeness*, see Theorem 2), we enforce the following conservation property.

Definition 1 (Cost Conservation). *A data pipeline graph $G = (V, A, w)$ satisfies the cost conservation property if the sum of cost allocation ratios for any non-leaf node $u \in V(G) \setminus L(G)$ equals unity:*

$$\sum_{v \in D_G^1(u)} w(u,v) = 1$$

This ratio definition is flexible and can be adapted to various allocation strategies, including equal-distribution, usage-based [42, 49], or value-based proxies [7, 21, 48]. For instance, under a usage-based strategy, the weight can be derived from the normalized data volume read by downstream vertices [42]: $w(u,v) = \frac{\text{read\_volume}(v \leftarrow u)}{\sum_{k \in D_G^1(u)} \text{read\_volume}(k \leftarrow u)}$ where $\text{read\_volume}(v \leftarrow u)$ denotes the volume of data read from u to produce v.

Example 1. Consider v_1, v_2, and v_3 in Fig. 1. Let $w(v_1, v_3) = 1$, $w(v_2, v_3) = 0.6$, $C_{mc}(v_1) = 3, C_{mc}(v_2) = 5, C_{dc}(v_3) = 2$, then we have $C_{ic}(v_3) = 1 \times 3 + 0.6 \times 5 = 6$. Thus, $C_{mc}(v_3) = 6 + 2 = 8$.

3.2 Product Costing in Data Pipelines as an MCP Problem

Based on the definition of MC, we demonstrate that the problem of product costing can be reduced to solving an MCP problem.

MC of DPs completely recovers the total cost of the pipelines. The primary goal of product costing is to apportion the total direct cost of the pipelines to the final DPs, with *completeness* guarantee that ensures *all* costs are accounted for *without* duplication. Our crucial observation is that this goal can be achieved by calculating the manufacturing cost, $C_{mc}(p)$, for each DP, as it inherently accumulates all inherited upstream expenditures with its direct cost. We state this property formally below.

Theorem 2 (Completeness of Product Costing). *Given a data pipeline graph $G = (V, A, w)$ that satisfy the cost conservation property, the total manufacturing cost of data products (leaf nodes) is equal to the total direct cost injected into the system. Formally:*

$$\sum_{p \in L(G)} C_{mc}(p) = \sum_{v \in V(G)} C_{dc}(v)$$

The proof is demonstrated in Appendix A[1]. Theorem 2 shifts our goal to accurately computing the MC of all DPs, which we term the *calculating the MC of DPs (MCP) problem.*

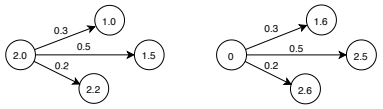
Definition 2 (The MCP Problem). *Given an edge-weighted DCG $G = (V, A, w)$, a DC vector $\mathbf{d}$ where $\mathbf{d}_i = C_{dc}(v_i)$ for all $v_i \in V$, the MCP problem is to compute the MC vector $\mathbf{c}^L \in \mathbb{R}^{|L(G)|}$ such that each $\mathbf{c}_i^L$ represents the MC of a leaf node (DP) $p_i \in L(G)$.*

Desired goals. We define the desired goals for an algorithm that solves MCP problem as follows.

- **Exactness**. The algorithm must compute the MC $C_{mc}(p)$ for each data product $p \in L(G)$ exactly as defined by the cost model (Eq. 2).
- **Scalability** The algorithm must scale to large data pipelines (*e.g.*, millions of vertices) and complete the computation within a practical time window (*e.g.*, for daily billing cycles).

The necessity of these goals stems from both economic and technical imperatives. First, the exactness of the MCP solution servers as the foundation for the completeness of product costing. It is fundamental to financial accountability, particularly for data pricing and pipeline optimization. Given that cost-based pricing is prevalent in practice [10, 33, 49] and resource optimization is critical, any inaccuracy risks distorting pricing strategies, eroding revenue, and misallocating budgets. Second, scalability is essential for handling enterprise-level data pipelines, which often comprise millions of entities and require timely cost reporting.

[1]The complete appendix can be found in the full version of our paper, available at https://github.com/jerryp1/CostApp/blob/main/CostApp.pdf



(a) Before CA. **(b) After CA.**

Figure 3: An example of a CA operation.

3.3 Basic Solutions

We present two basic solutions for MCP problem, SimpleApp and MatrixApp. While neither fully satisfies both exactness and scalability goals simultaneously, they serve as fundamental building blocks for our proposed hybrid approach (see §4).

3.3.1 The SimpleApp Algorithm. A straightforward idea to solve MCP problem is to iteratively propagate costs from upstream vertices to downstream ones along the lineage until they fully accumulate at the leaves. Based on this strategy, we introduce SimpleApp, a scalable, vertex-centric algorithm designed to simulate cost propagation along the lineage.

At the heart of SimpleApp is an iterative cost propagation mechanism. It is based on a stateful cost variable $C(v)$ and a cost apportion operation CA that transfers cost states. $C(v)$ represents the cost currently held by v, and initialized with its direct cost *i.e.*, $C(v) \leftarrow C_{dc}(v)$. In each iteration, a cost apportion operation CA is applied to non-leaf vertices to propagate their costs downstream. CA(v) is a two-step state update:

(1) **Distribute Cost:** For each direct downstream neighbor $v' \in D_G^1(v)$, its cost is updated by adding a proportional share of v's cost: $C(v') \leftarrow C(v') + w(v, v') \cdot C(v)$.
(2) **Reset Cost:** The cost at v is set to zero, signifying that its contribution has been fully passed on: $C(v) \leftarrow 0$.

An example of a CA operation is shown in Fig. 3.

Apportion cost via CA Operations based on Pregel. Building upon CA, we introduce SimpleApp that repeatedly applies the CA to all non-leaf vertices to solve MCP problem. We adopt a Pregel-based implementation [32] since CA is *vertex centric* and naturally fits distributed graph processing frameworks, enabling a scalable solution.

Algorithm 1: SimpleApp (Pregel-style pseudocode)

```
Input: Graph G = (V, A, w), stateful cost variable C(v)
Output: C^out(v)
1 Each vertex v ∈ V sets C^out(v) ← C(v) ;          // Initialization
2 while any non-leaf vertex v is active do
3     foreach active vertex v do in parallel
          /* CA operation in Pregel style                    */
4         compute(msgs)
5             update C^out(v) ← C^out(v) + Σ_{m_i ∈ msgs} m_i;
6             sends w(v, v') · C^out(v) to each v' ∈ D_G^1(v);
7             updates C^out(v) ← 0 ;
8             voteToHalt();
              // Reactivated upon receiving a message
9 return C^out(v)
```

The algorithm SimpleApp is shown in Alg. 1. It takes as input the graph $G = (V, A, w)$, the stateful cost variable $C(v)$. The output is the final cost state $C^{out}(v)$ for all vertices $v \in V$. The algorithm proceeds in supersteps.

Initially, each vertex v initiates a new state $C^{out}(v)$ (line 1). In each superstep, every active non-leaf vertex performs CA operation (line 3). Each vertex collects all incoming costs from the messages received from upstream neighbors and accumulates these into its cost state $C^{out}(v)$ (line 5). SimpleApp then pushes its current cost to its immediate downstream neighbors according to the edge weights (line 6), and resets its cost to zero (line 7), indicating that its cost is "flowed-away". The process iterates until all non-leaf vertices become inactive. Finally, the algorithm returns the final cost state $C^{out}(v)$ for all $v \in V$ (line 10). The final cost state of DPs $\mathbf{c}^L$ can be extracted from $C^{out}(v)$ as $\mathbf{c}_i^L \leftarrow C^{out}(p_i)$ for $p \in L(G)$.

SimpleApp is scalable to large graphs due to its Pregel-based distributed parallel computing paradigm [32].

Limitations. For acyclic graphs (DAGs), SimpleApp is guaranteed to terminate and yield exact results (Theorem 3). However, in graphs with cycles (DCGs), costs circulate indefinitely, preventing the algorithm from terminating naturally. While forcing termination via a superstep threshold (see Appendix B) ensures convergence, it inevitably results in approximate solutions, failing the exactness goal.

Theorem 3. *For any DAG $G = (V, A, w)$ and $C(v)$ initialized as $C_{dc}(v)$, SimpleApp(G,C) terminates and correctly computes the MC vector $\mathbf{c}^L$ where $\mathbf{c}_i^L = C_{mc}(p_i)$ for all leaves $p_i \in L(G)$ (solve MCP problem) in at most P supersteps, where P is the length of the longest path in G.*

The proof is demonstrated in Appendix C.

3.3.2 The MatrixApp Algorithm. To address the cyclic dependencies where SimpleApp fails, we formulate the problem as a system of linear equations.

Linear System Formulation. As defined in Eq. 2, the manufacturing costs in a cyclic graph are mutually dependent. For each vertex v, its cost is the sum of its direct cost and the apportioned costs from its inputs:

$$C_{mc}(v) - \sum_{u \in U_G^1(v)} w(u, v) \cdot C_{mc}(u) = C_{dc}(v) \tag{3}$$

Considering all vertices $v \in V(G)$ indexed from 1 to n, this forms a global system of $|V(G)|$ linear equations. Let $\mathbf{c}$ be the vector of manufacturing costs $[C_{mc}(v_1), \ldots, C_{mc}(v_n)]^T$, $\mathbf{d}$ be the vector of direct costs $[C_{dc}(v_1), \ldots, C_{dc}(v_n)]^T$, $\mathbf{W}$ is a *cost allocation matrix* where $\mathbf{W}[i, j] = w(v_i, v_j)$ if there is an arc from v_i to v_j and 0 otherwise, and $\mathbf{I}$ is an identity matrix. The system can be expressed in matrix form (see Appendix D for detailed derivation) as:

$$\mathbf{c} = \mathbf{d} + \mathbf{W}^\mathsf{T}\mathbf{c} \tag{4}$$

Solving this system for $\mathbf{c}$ gives a closed-form solution to MCP problem, as shown in Alg. 2.

The invertibility of $(\mathbf{I} - \mathbf{W}^\mathsf{T})$ is guaranteed by the spectral structure of $\mathbf{W}$: all entries are non-negative, each row sum is at most 1, and at least one row sum is strictly less than 1 (the leaves). A formal proof is given in Appendix E.

Algorithm 2: MatrixApp

Input: Cost allocation matrix $\mathbf{W}$, DC vector $\mathbf{d}$
Output: MC vector of DPs $\mathbf{c}^L$

1 Compute $\mathbf{c} \leftarrow (\mathbf{I} - \mathbf{W}^\top)^{-1}\mathbf{d}$;
2 Extract the sub-vector $\mathbf{c}^L$ from $\mathbf{c}$;
3 **return** $\mathbf{c}^L$

Limitations. While MatrixApp guarantees exactness by definition, it is computationally prohibitive for large-scale pipelines. Matrix inversion generally incurs cubic time complexity $O(|V|^3)$ and quadratic space complexity $O(|V|^2)$. Even with optimizations for sparse matrices, this approach becomes computationally and memory-prohibitive for pipelines with millions of entities, failing the scalability goal.

Our proposed SimpleApp and MatrixApp for the MCP problme highlight a fundamental trade-off between exactness and scalability, motivating our hybrid solution in the next section.

4 THE IMPROVED ALGORITHM: COSTAPP

In this section, we introduce CostApp, a novel hybrid algorithm that strategically combines our two basic methods. It delivers an exact, scalable solution for the MCP problem on cyclic graphs.

4.1 Overview of CostApp

The high-level idea of CostApp is decoupling and reduction: it transforms an intractable, large-scale problem into an equivalent but easily solvable, compact one. The algorithm CostApp is illustrated in Alg. 3, with a running example shown in Fig. 4. CostApp operates in two main phases:

Phase 1: Acyclic Cost Aggregation. This phase uses scalable, distributed graph algorithms to propagate all direct costs throughout the acyclic portions of the pipelines and consolidate them onto a small, well-defined set of terminal vertices. The process begins by identifying the cyclic dependencies using our scalable FindFAS algorithm (line 1) to obtain an FAS. Instead of simply removing these arcs, we isolate the cycles using GraphEdit (line 2), which replaces each feedback arc with a special path through auxiliary *sink* and *source* vertices. They effectively isolate the cyclic dependency and render the majority of the graph acyclic, enabling us to use the highly scalable SimpleApp algorithm to efficiently propagate all direct costs (line 5). Upon completion, all costs are precisely aggregated onto a small set of terminal vertices *i.e.*, the original data products and the newly created sink nodes. This sets the stage for an exact solution.

Phase 2: Cyclic Cost Resolution. With all costs consolidated on the terminal vertices, this phase constructs and solves a compact linear system to resolve the remaining cyclic dependencies. First, we extract the aggregated costs from Phase 1 to form a new, reduced direct cost vector $\mathbf{d}_{\text{term}}$ (line 6). We then compute the reduced cost allocation matrix, $\mathbf{W}_{\text{term}}$, using GetMatrix (line 7), which captures the cost-flow relationships *only among* these terminal vertices. This transforms the original large-scale problem into a compact and equivalent linear system, which we solve exactly and efficiently using MatrixApp* (a variant of MatrixApp, see §4.3.3) (line 8). This final step yields the true manufacturing costs for all DPs, accurately accounting for the complex cyclic flows isolated in Phase 1.

Algorithm 3: CostApp

Input: Graph $G = (V, A, w)$, initial cost state $C(v) = C_{dc}(v)$ for all $v \in V$
Output: The MC vector of DPs $\mathbf{c}^L$

/* Phase 1: Acyclic Cost Aggregation */
1 $A_{\text{FAS}} \leftarrow \text{FindFAS}(G)$; // (1.a) Identifing cycles
2 $(G_1, V_{sk}, V_{sr}) \leftarrow \text{GraphEdit}(G, A_{\text{FAS}})$; // (1.b) Isolating cycles
3 Let G_{DAG} be the subgraph of G_1 without sink-to-source arcs;
4 Let V_{term} be the set of terminal vertices, comprising the leaves of G_{DAG}, *i.e.*, $L(G) \cup V_{sk}$;
5 $C_1 \leftarrow \text{SimpleApp}(G_{DAG}, C)$; // (1.c) Aggregating costs
/* Phase 2: Cyclic Cost Resolution */
6 $\mathbf{d}_{\text{term}} \leftarrow \text{ExtractCosts}(C_1, V_{\text{term}})$;
7 $\mathbf{W}_{\text{term}} \leftarrow \text{GetMatrix}(G_{DAG}, V_{sk}, L(G))$; // (2.a) Getting the cost allocation matrix
8 $\mathbf{c}^L \leftarrow \text{MatrixApp}^*(\mathbf{W}_{\text{term}}, \mathbf{d}_{\text{term}})$; // (2.b) Solving a compact system for final costs
9 **return** $\mathbf{c}^L$

Key design. The key design behind CostApp is the injection of auxiliary sink and source vertices on the feedback arcs. To overcome the *cyclic* and *massive scale* challenges (stated in §1), CostApp introduces sink vertices that act as *collectors* to intercept all costs that would flow over feedback arcs. This enables us to simultaneously *isolate cycles* and *simplify the cost flow dependencies* in Phase 1. This sets up the stage for building a reduced linear system. To model the reduced system, source vertices are introduced to serve as *re-injection points*. It allows us to capture how the aggregated cost at sink vertices would re-enter the original graph, based on which we can efficiently solve the reduced system in Phase 2.

By strategically injecting auxiliary vertices and combining SimpleApp and MatrixApp, CostApp shifts the computational bottleneck from inverting a massive $|V| \times |V|$ matrix to inverting a tiny one (often less than 0.1% of $|V|$ in size). Our design is simple yet effective, achieving a dramatic performance gain while preserving the exactness of results. We now detail each phase.

4.2 Phase 1: Acyclic Cost Aggregation

The goal of phase 1 is to aggregate all direct costs $C_{dc}(v)$ from millions of internal vertices onto a small set of terminal vertices, while preserving the manufacturing cost of every DP. To achieve this, directly using SimpleApp on the original graph to push cost would fail, as costs would circulate indefinitely in cycles. Our solution, therefore, involves three steps: (1) identifying cycles via finding an FAS, (2) isolating cycles via graph editing, and (3) aggregating costs on the resulting acyclic graph.

4.2.1 1. Identifying Cycles via Finding an FAS. To make cost aggregation possible, we must first break all cycles while preserving the majority of the original graph structure. A standard way in graph theory is to find a FAS, *i.e.*, a subset of arcs whose removal makes the graph acyclic. While finding a minimum FAS is an NP-hard problem [15, 17, 24], but for our purpose *any* valid FAS suffices, as long as it can be computed efficiently on large graphs.

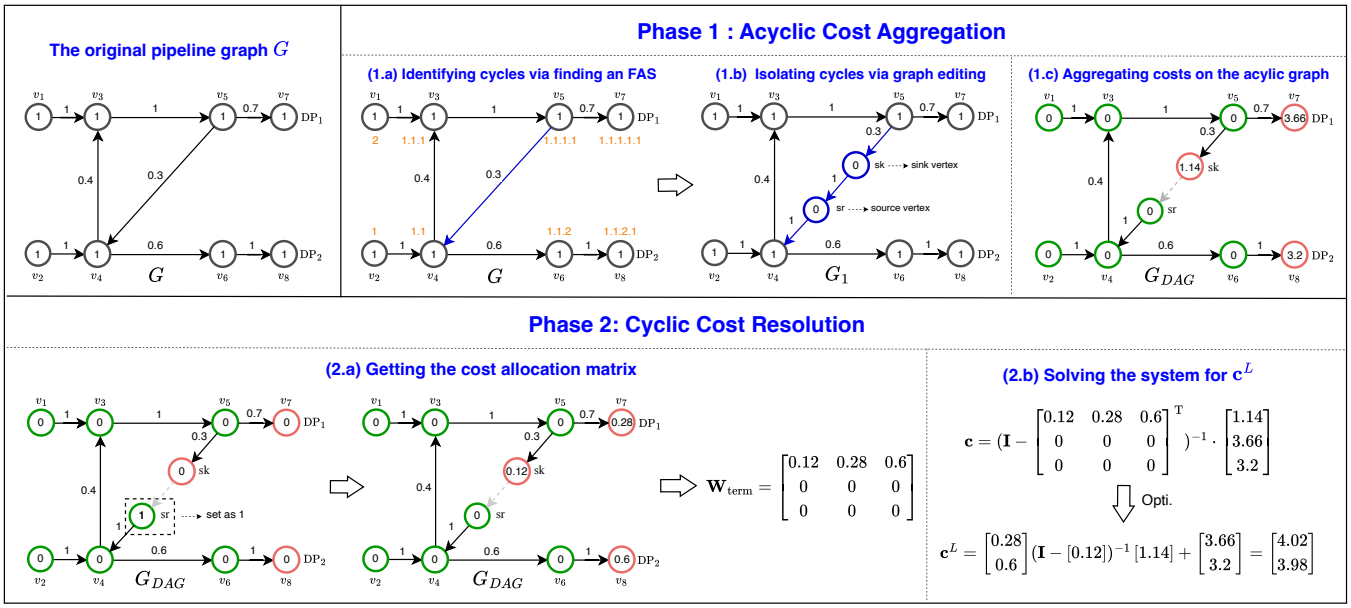


Figure 4: An illustrative workflow of CostApp. The initial cost state of each vertex is set to 1 for better illustration. (1.a) Ordering labels (orange) from Label identifies the feedback arc (blue). (1.b) New sink/source vertices (blue) are inserted to isolate the cycle. (1.c) SimpleApp aggregates costs onto terminal vertices (red), leaving the remaining vertices (green) with zero cost. (2.a) Unit costs are injected at source vertices to derive $\mathbf{W}_{\text{term}}$. (2.b) MatrixApp solves the reduced linear system to yield exact MCs.

We therefore design a scalable Pregel-style algorithm FindFAS with the following high-level strategy. First, we establish a linear ordering over all vertices by assigning each vertex a hierarchical label. Given this ordering, any arc $\langle u, v\rangle$ that goes *backward* (*i.e.*, the label of u is larger than that of v) is classified as a feedback arc. The set of all such arcs forms a valid FAS.

Our strategy is to first establish a linear ordering of all vertices in the graph G. Given such an ordering, any arc $\langle u, v\rangle$ where u appears after v is considered a *backward* or inconsistent arc. The set of all such inconsistent arcs constitutes a valid FAS. Based on this strategy, we devise a distributed labeling algorithm to generate this ordering, followed by a simple check to identify the inconsistent arcs.

Distributed labeling for vertex ordering. To construct the ordering, we use a synchronous labeling algorithm Label. Each vertex v maintains a label $Label(v)$, represented as a dot-separated sequence of positive integers (e.g., '1.2.1'). The labels induce a lexicographic order $>_{lo}$ (Definition 3).

Definition 3 (Lexicographical Label Ordering). *Given two distinct labels $Label_A = a_1.a_2.\ldots.a_p$ and $Label_B = b_1.b_2.\ldots.b_q$, we say $Label_A$ is lexicographically greater than $Label_B$, denoted $Label_A >_{lo} Label_B$, if either:*

(1) $Label_A$ is a prefix of $Label_B$ (i.e., $p < q$ and $a_i = b_i$ for all $1 \le i \le q$).
(2) There exists an index $k \le \min(p, q)$ such that $a_i = b_i$ for all $i < k$ and $a_k < b_k$.

The algorithm Label is designed based on Pregel to support large graphs. It starts by assigning distinct integer labels to all roots (vertices with in-degree 0). In each superstep, every active vertex v:

(1) collects labels from its upstream neighbors,
(2) adopts the lexicographically largest incoming label if it is greater than $Label(v)$, and
(3) propagates an extended version of its new label (by appending a suffix) to its downstream neighbors.

This process repeats until no vertex updates its label. The full Pregel-style pseudocode is given in Appendix F. With each vertex labeled, we can classify arcs and identify an FAS.

Theorem 4. *Label terminates and assigns a stable label to every vertex in at most $L_p + 1$ supersteps, where L_p is the length of the longest simple path in G.*

The proof is similar to the proof of Theorem 3 and we omit it here for brevity.

Finding an FAS. Once Label terminates, every arc (u, v) falls into exactly one of the following types:

(1) direct forward arc: $Label(u)$ is a prefix of $Label(v)$ (*e.g.*, $Label(v_4) = 1.1$, $Label(v_6) = 1.1.2$ in Fig. 4);
(2) cross forward arc: $Label(u) >_{lo} Label(v)$ but $Label(u)$ is not a prefix of $Label(v)$;
(3) feedback arc: $Label(v)$ is a prefix of $Label(u)$ (*e.g.*, $Label(v_5) = 1.1.1.1$, $Label(v_4) = 1.1$ in Fig. 4).
(4) cross backward arc: $Label(v) >_{lo} Label(u)$ but $Label(v)$ is not a prefix of $Label(u)$ (*e.g.*, $Label(v_1) = 2$, $Label(v_3) = 1.1.1$ in Fig. 4).

The set of all arcs satisfying condition (3) constitutes FAS *i.e.*, A_{FAS}. Intuitively, after removing A_{FAS}, every remaining edge follows the label order, so any directed path must follow a strictly increasing sequence of labels and thus cannot form a cycle.

Lemma 1. *The set $A_{FAS} = \{\langle u, v\rangle \in A(G) \mid Label(v) >_{lo} Label(u)\}$ is a valid feedback arc set for G.*

The proof is in Appendix G.

Therefore, the overall FindFAS procedure first runs Label and then, in a single superstep, lets each vertex inspect its incoming arcs and mark those with $Label(u) >_{lo} Label(v)$ as feedback arcs. Its superstep complexity is therefore $O(L_p)$, dominated by Label.

An illustration of FindFAS is shown in Fig. 4 (1.a), each vertex is assigned a hierarchical label (highlighted in orange) by calling Label (*e.g.*, , v_4 gets '1.1'). Then, arcs that point backward according to these labels are identified as feedback arcs (highlighted in blue, *e.g.*, , (v_5, v_4) where $Label(v_4) >_{lo} Label(v_5)$).

4.2.2 Isolating Cycles via Graph Editing. To isolate cyclic dependencies, simply deleting the FAS arcs would distort the true dependencies and lead to incorrect cost calculations. Instead, our goal is to *isolate* the cyclic dependencies while preserving the cost-flow behavior seen from the original vertices. and at the same time create explicit *entry* and *exit* points that allow us to aggregate and later resolve the cyclic costs.

To this end, we introduce a graph transformation GraphEdit. For each feedback arc $(u, v) \in A_{\text{FAS}}$, GraphEdit performs the following local rewrite on G:

- Remove the original arc (u, v).
- Create two new auxiliary vertices: a *sink* vertex v_{sk} and a *source* vertex v_{sr}, and add them to V_{sk} and V_{sr}, respectively.
- Insert three arcs forming the path $u \to v_{sk} \to v_{sr} \to v$: (u, v_{sk}) with weight $w(u, v)$, and (v_{sk}, v_{sr}), (v_{sr}, v) both with weight 1.

An illustration of this transformation is shown in Fig. 4(1.b). The feedback arc (v_5, v_4) is replaced by the path $v_5 \to v_{sk} \to v_{sr} \to v_4$, where v_{sk} and v_{sr} are newly created sink and source vertices, respectively. Intuitively, v_{sk} acts as a *collector*: it intercepts all cost that would have flowed from u into the cycle through (u, v), allowing Phase 1 to aggregate such *cycle-bound* cost onto a small set of sink vertices. Symmetrically, v_{sr} serves as a *re-injection point*: in Phase 2, it enables us to model how the aggregated cost at v_{sk} would have re-entered the original downstream part of the graph, via a reduced cost allocation matrix $\mathbf{W}_{\text{term}}$. A formulation of GraphEdit is illustrated in Appendix H.

Crucially, this transformation does not change the manufacturing cost of any original vertex. The path $u \to v$ in G is replaced by an equivalent path $u \to v_{sk} \to v_{sr} \to v$ in G_1 that carries exactly the same cost (up to the introduced auxiliary vertices), and all other edges remain untouched. As a result, from the perspective of the original vertices, the total cost transmitted along every path is preserved. This is formalized below.

Theorem 5 (Equivalence of MC of DPs after Graph Editing). *Let G_1 be the graph produced by applying GraphEdit to a graph G with a feedback arc set A_{FAS}. The manufacturing cost $C_{mc}(p)$ for any original data product $p \in L(G)$ is identical in both the original system defined on G and the transformed system defined on G_1.*

The proof is provided in Appendix I.

4.2.3 Aggregating Costs on the Acyclic Graph. After graph editing, we obtain a modified graph G_1 with additional sink/source vertices and arcs. Let A_{SS} denote the set of sink-to-source arcs $\{(v_{sk}, v_{sr})\}$. By construction, removing A_{SS} yields an acyclic subgraph $G_{DAG} = G_1 \setminus A_{SS}$. The leaves of G_{DAG}, which we call *terminal vertices*, are

$$V_{\text{term}} = L(G_{DAG}) = L(G) \cup V_{sk},$$

that is, all original DPs (leaves of G) and all newly created sink vertices.

The acyclic structure of G_{DAG} allows us to apply the scalable SimpleApp algorithm to propagate costs exactly. We run SimpleApp on G_{DAG} (Alg. 3, line 3), initializing each vertex v with its direct cost $C_{dc}(v)$. By Theorem 3, this process terminates in at most P supersteps, where P is the length of the longest path in G_{DAG}, and correctly transfers all costs to the leaves of G_{DAG}, *i.e.*, to V_{term}.

Let C_1 denote the resulting cost state. It satisfies a crucial property:

- For every terminal vertex $t \in V_{\text{term}}$, $C_1(t) > 0$ in general;
- For every non-terminal vertex $v \in V(G_1) \setminus V_{\text{term}}$, $C_1(v) = 0$.

As illustrated in Fig. 4(1.c), the entire initial direct cost of the pipeline has now been losslessly consolidated onto the much smaller terminal set $V_{\text{term}} = \{v_7, v_8, v_{sk}\}$ (highlighted in red). In other words, Phase 1 has used a scalable graph algorithm to transform a large, distributed MCP instance into a compact representation over a small number of terminal vertices, thereby setting up Phase 2 to resolve the remaining cyclic dependencies via a tiny linear system.

4.3 Phase 2: Cyclic Cost Resolution

Phase 1 has effectively consolidated the entire pipeline's direct costs onto the terminal vertices V_{term}. Consequently, MCP problem is reduced to determining how the aggregated costs at the sink vertices V_{sk} are redistributed among themselves and ultimately to the data products $L(G)$. This redistribution is governed by the feedback relationships isolated in the $v_{sk} \to v_{sr}$ paths, which we now resolve as a compact linear system.

4.3.1 Formulating the Reduced Linear System. To formulate the reduced system that only includes V_{term}, our key insight is that we can treat the aggregated costs C_1 on V_{term} as *new* direct costs for this reduced system. We denote this reduced cost vector by $\mathbf{d}_{\text{term}}$, where each entry corresponds to $C_1(v)$ for a terminal vertex v in V_{term}. Consequently, this reduced system follows our original cost definition for their final manufacturing costs, $\mathbf{c}_{\text{term}}$ (Eq. 2):

$$\mathbf{c}_{\text{term}} = \mathbf{d}_{\text{term}} + \mathbf{W}_{\text{term}}^{T}\mathbf{c}_{\text{term}} \tag{5}$$

Here, $\mathbf{c}_{\text{term}}$ is the vector of final manufacturing costs for the terminal vertices, and $\mathbf{W}_{\text{term}}$ is a reduced cost allocation matrix that captures how cost flows *among* these terminal vertices via the isolated cyclic dependencies.

This equation has exactly the same form as the system defined in §3.3.2, and $\mathbf{W}_{\text{term}}$ inherits the same non-negativity and row-sum properties as $\mathbf{W}$. Therefore, $(\mathbf{I} - \mathbf{W}_{\text{term}}^{\mathsf{T}})$ is invertible and the reduced system has a unique solution (proved in Appendix E). It allows us to solve this system using MatrixApp to get $\mathbf{c}_{\text{term}}$ and thus MCP problem be solved exactly.

Since $\mathbf{d}_{\text{term}}$ is already determined by Phase 1 (from C_1 in §4.2.3), the remaining challenge in Phase 2 is to compute the reduced allocation matrix $\mathbf{W}_{\text{term}}$. We describe how to do this efficiently by leveraging source vertices in the next step.

4.3.2 Getting the Cost Allocation Matrix. The remaining task in Phase 2 is to construct the reduced cost allocation matrix $\mathbf{W}_{\text{term}}$. This matrix captures the cost-flow relationships *among* terminal vertices, *i.e.*, an entry $\mathbf{W}_{\text{term}}[i, j]$ represents the fraction of cost that flows from terminal vertex t_i to t_j.

Recall that $V_{\text{term}} = L(G) \cup V_{sk}$ and that every sink $v_{sk} \in V_{sk}$ has a corresponding source vertex $v_{sr} \in V_{sr}$ connected by a sink-to-source arc. Our crucial observation is that for a sink vertex t_i, $\mathbf{W}_{\text{term}}[i, j]$ is equivalent to the cost received by terminal vertex t_j when a *unit cost* is injected at the corresponding source vertex v_{sr} and allowed to traverse the acyclic graph G_{DAG}. This is because the only way for cost to flow from t_i to other terminal vertices in G_1 is through the path $t_i \to v_{sr} \to \cdots \to t_j$. This observation allows us to compute $\mathbf{W}_{\text{term}}$ efficiently using SimpleApp by injecting unit cost to each source vertex.

Based on this idea, we design GetMatrix to obtain $\mathbf{W}_{\text{term}}$ based on Pregel, as shown in Alg. 11. To enable efficient computation within a single Pregel job, we adopt a *map-based* cost propagation approach. Specifically, instead of a scalar cost, we assign a cost state map to each vertex (lines 3) to record {vertex: cost} pair. We inject unit costs at all source vertices (lines 4-6) and propagate them through G_{DAG} using SimpleApp* (line 7). Due to the cost conservation property (Definition 1), the final cost map at any terminal vertex t_i constitutes the i-th column of $\mathbf{W}_{\text{term}}$ (lines 8-11). Thus, all columns of $\mathbf{W}_{\text{term}}$ are obtained in a *single* Pregel job, setting the stage for solving the reduced system in the next step.

Note that this map-based approach relies on a variant of SimpleApp that can aggregate cost maps by summing values of the same keys. We term this variant SimpleApp* and provide its Pregel-style pseudocode in Appendix J. Noting that GetMatrix (line 8 in Alg.3) and SimpleApp (line 6 in Alg. 3) both operate with Pregel on G_{DAG}, they can be merged into a single Pregel workload to avoid duplicated overhead. We denote this merged workload as S&G in the following parts.

4.3.3 Solving the Reduced Linear System. With $\mathbf{W}_{\text{term}}$ and $\mathbf{d}_{\text{term}}$ in hand, we can solve MCP problem by invoking MatrixApp to solve the compact linear system defined in §4.3.1. Instead of naively inverting the full matrix $\mathbf{W}_{\text{term}}$, we exploit the topological structure of V_{term} to further reduce the computational cost.

Recall that $V_{\text{term}} = L(G) \cup V_{sk}$, Since data products are leaves in the global graph, they do not propagate costs. Consequently, the reduced allocation matrix $\mathbf{W}_{\text{term}}$ is *block upper triangular*:

$$\mathbf{W}_{\text{term}} = \begin{bmatrix} \mathbf{W}_{sk,sk} & \mathbf{W}_{sk,dp} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}.$$

This structure allows us to solve the system and obtain the final MC of DPs by computing:

$$\mathbf{c}_{dp} = \mathbf{d}_{dp} + \mathbf{W}_{sk,dp}^{T}(\mathbf{I}_{sk} - \mathbf{W}_{sk,sk}^{T})^{-1}\mathbf{d}_{sk}$$

Algorithm 4: GetMatrix

Input: Graph $G_{DAG} = (V, A, w)$, set of sink vertices V_{sk}, data products $L(G)$.
Output: Cost allocation matrix $\mathbf{W}_{\text{term}} \in \mathbb{R}^{k \times k}$

```
/* 1. Setup and Initialization                                          */
1  Let V_term = L(G) ∪ V_sk, k = |V_term|, m = |V_sr|;
2  Initialize W_term as a k × k zero matrix;
3  C(v) ← empty map ∅ for each v ;            // Initiates a cost state map
4  for i ← 1 to m do
5      Let v_sr,i be the i-th source vertex;
6      C(v_sr,i) ← {i : 1} ;                  // Inject a unit cost
   /* 2. SimpleApp* aggregates maps by summing values of the same keys  */
7  C^out ← SimpleApp*(G_DAG, C)
   /* 3. Fill the matrix with the results                                */
8  for i ← 1 to k do
9      Let t_i be the i-th terminal vertex;
10     for j ← 1 to m do
11         W_term[j, i] ← C^out(t_i).getValue(j, 0) ;   // Get value for
             source index i, default to 0 if missing
12 return W_term
```

where $\mathbf{d}_{dp}$ and $\mathbf{d}_{sk}$ are the subvectors of $\mathbf{d}_{\text{term}}$ corresponding to data products and sink vertices, respectively. We term this variant as MatrixApp*.

This optimization further reduces the dominant inversion cost to $O(|V_{sk}|^3)$, followed by a cheaper matrix-vector multiplication. In typical data pipelines, the number of sink arcs (equals the number of feedback arcs) is vastly smaller than the total number of vertices ($|V_{sk}| \ll |V(G)|$). The exactness of CostApp is preserved due to Theorems 5, 3 and the correctness of MatrixApp.

4.4 Efficiency Analysis

The efficiency of CostApp is rooted in its hybrid design, which transforms a computationally intractable problem into a compact one while preserving exactness. By strategically isolating cyclic dependencies, it leverages the strengths of both distributed graph processing and solving linear systems. Tab. 3 summarizes the efficiency and complexity comparison of the algorithms.

Table 3: Complexity comparison.

Category	Algorithm / Module	Scalability	Exactness	Time Complexity
Baselines	MatrixApp	Low	Exact	$O(\|V\|^3)$
	SimpleApp	High	Approximate	$O(S \cdot (\|V\| + \|A\|))$
Ours	**CostApp**	**High**	**Exact**	$\mathbf{O(L_p \cdot (\|V\| + \|A\|) + \|A_{FAS}\|^3 + \|A_{FAS}\| \cdot \|DPs\|)}$
	FindFAS	-	-	$O(L_p \cdot (\|V\| + \|A\|))$
	S&G (SimpleApp + GetMatrix)	-	-	$O(L'_p \cdot (\|V\| + \|A\|))$
	MatrixApp*	-	-	$O(\|A_{FAS}\|^3) + \|A_{FAS}\| \cdot \|DPs\|$

Notation: $|V|$ is the number of vertices, $|A|$ is the number of arcs, $|A_{FAS}|$ is the size of the feedback arc set, $|DPs|$ is the number of DPs, S is the number of supersteps (determined by configured accuracy-runtime trade-off, see §5.2.1), $L_p \approx L'_p$ are the longest simple path length of G, G_{DAG}, respectively.

As shown in Tab. 3, the total complexity of CostApp is the sum of its distributed and centralized components: $O(L_p \cdot (|V| + |A|)) + O(|A_{\text{FAS}}|^3 + |A_{\text{FAS}}| \cdot |DPs|)$. The first term, $O(L_p \cdot (|V| + |A|))$, represents the near-linear complexity of the Pregel-based modules (FindFAS, S&G). These steps handle the bulk of the computation and dominate the overall runtime. The second term, $O(|A_{\text{FAS}}|^3 + |A_{\text{FAS}}| \cdot |DPs|)$, is the complexity of MatrixApp*, where $O(|A_{\text{FAS}}| \cdot |DPs|)$ represents the complesity of matrix multiplication in MatrixApp*.

5 EXPERIMENTS

In this section, we conduct a comprehensive experimental evaluation of our proposed CostApp for MCP problem. Our goal is to answer the following research questions:

- **RQ1: Exactness and Scalability.** How CostApp outperform baselines in terms of exactness and scalability?
- **RQ2: Modular Evaluation.** What are the performance characteristics of CostApp's individual components, and where are the primary computational bottlenecks?
- **RQ3: Real-world Efficiency.** How effective and practical is CostApp when applied to large-scale, real-world data pipelines?

5.1 Experimental Setup

Our distributed tasks, including Pregel-based graph computations and SQL queries, are conducted on the MaxCompute platform [9] that is deployed on production clusters at Alibaba. The matrix related modelus (MatrixApp and MatrixApp*) are performed on a single machine because distributed matrix inversion suffers from challenging deployment [34], prohibitive communication overhead and complex synchronization requirements [13]. Any improved implementation on matrix inversion benefits CostApp as well. Optimizations for matrix inversion are orthogonal to our contribution and thus beyond the scope of this paper. The concrete settings are shown in Table 4. Unless otherwise specified, all tasks are executed based on the setting in Table 4. We denote S&G as the combination of SimpleApp and GetMatrix modules in CostApp (lines 5-6 in Alg. 3). MatrixApp* denotes the linear system solver constituted in CostApp (line 8 in Alg. 3).

Table 4: Execution form of modules of CostApp and the setting of platforms.

Modules	Execution form
FindFAS	Pregel
GraphEdit	SQL
S&G	Pregel
MatrixApp*	Single Machine

(a) Execution form of modules.

Pregel/SQL	|workers|	100
	memory per worker	30GB
	|threads| per worker	1
	|cpu cores| per worker	8
Single Machine	memory	100GB
	|cpu cores|	8

(b) Platform settings.

Baselines. Since our work represents the first effort to perform product costing in data pipelines and solve the unique challenges therein, we compare CostApp against our two basic solutions in §3.3, which are directly applying SimpleApp and MatrixApp on the original graph G.

Datasets. We use both real-world and synthetic datasets.

Real-world Datasets: We use four large-scale production data pipeline graphs from Alibaba, anonymized as G1—G4, where G1 is the full view of consolidated data pipelines in Alibaba. Their detailed statistics are presented in Table 5. The datasets exhibit scale-free characteristics and the degree of each node in these graphs follows a power-law distribution, with an average degree of each entity 2.5. A graphic view of the degree distribution is shown in Fig. 5. We open-source these datasets, and they represent the first publicly available real-world data pipeline graphs in data-intensive enterprises, to the best of our knowledge.

	#Vertices	#Arcs	#DPs
G1	28,181,925	66,758,378	10,003,135
G2	11,061,395	36,345,530	5,805,784
G3	712,772	3,808,320	309,852
G4	581,853	3,341,002	252,812

Table 5: Statistics of experimental graphs.

Synthetic Datasets: To systematically evaluate performance under controlled conditions, we use synthetic graphs with varying parameters, including number of vertices, edge density (average degree), and the number of injected feedback arcs. To simulate the degree distribution of the scale-free real-world datasets (as in Fig. 5), the graph was generated using a Barabási–Albert model [8]. Unless otherwise specified, $|V| = 10^6$ and $|A| = 3|V|$ to simulate the scale and density of real-world pipelines.

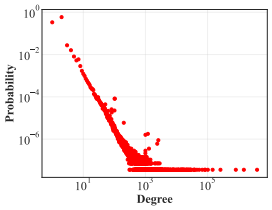


Figure 5: Degree distribution of G1.

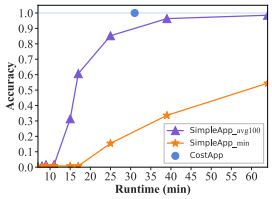


Figure 6: Exactness evaluation on G1.

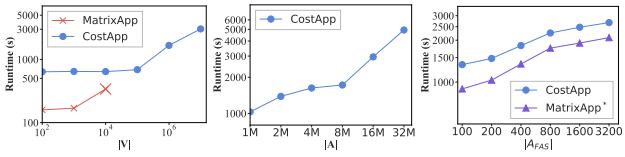
5.2 Exactness and Scalability

To answer **RQ1**, we evaluate the exactness and scalability of CostApp compared to the basic solutions.

5.2.1 Exactness. With theoretical guaranteed exactness, CostApp is expected to outperform the approximate SimpleApp in terms of accuracy. We evaluate this by measuring the accuracy of the computed MC of DPs on the real-world dataset G1. The accuracy metric is defined as the ratio of the estimated MC to the exact MC obtained from CostApp. We track accuracy over time. Fig. 6 presents the results. CostApp completes in approximately 30 minutes, delivering a exact cost result for all DPs. In contrast, the iterative baseline SimpleApp exhibits shortfalls. While the average accuracy of the 100 worst-performing data products (SimpleApp_avg100) appears to converge, the minimum accuracy (SimpleApp_min) remains critically low, reaching only 0.55 even after 60 minutes. This discrepancy highlights the unreliability of approximate methods for financial applications such as product costing.

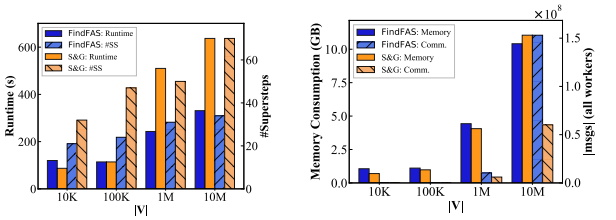
5.2.2 Scalability. We evaluate the scalability of CostApp by analyzing the runtime performance with respect to various graph characteristics, including the number of vertices ($|V|$), the number of arcs ($|A|$), and the number of injected feedback arcs ($|A_{FAS}|$). The results are shown in Fig. 7.

Fig. 7(a) demonstrates the scalability advantage of CostApp. While the baseline MatrixApp suffers from cubic complexity and runs out of memory (OOM) on graphs exceeding 10^4 vertices, CostApp exhibits near-linear scalability, successfully processing graphs with 10^7 vertices. Similarly, Fig. 7(b) shows that the runtime scales linearly with the number of arcs ($|A|$). This behavior is consistent with the communication complexity of our Pregel-based modules, where message passing dominates. It also confirms that



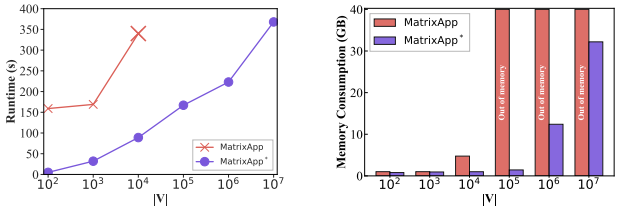
(a) Time vs $|V|$ (b) Time vs $|A|$ (c) Time vs $|A_{FAS}|$

Figure 7: Comparison of scalability of algorithms.



(a) Runtime and supersteps (b) Memory consumption and communicated messages

Figure 8: Evaluation of Pregel-based modules under varying |V|.



(a) Runtime (b) Memory consumption

Figure 9: Performance comparison of MatrixApp and MatrixApp*.

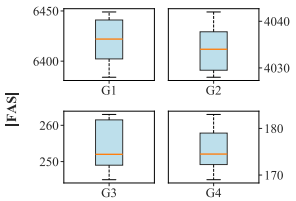


Figure 10: Sizes of FAS on G1—G4.

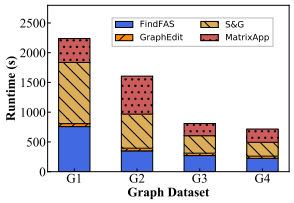


Figure 11: Overall evaluation on G1—G4 and time breakdown.

the distributed aggregation phase effectively handles massive graph data.

Fig. 7(c) isolates the impact of cycles by varying $|A_{FAS}|$ while keeping $|V|$ and $|A|$ constant. The results reveal that the total runtime grows with $|A_{FAS}|$, with a growth curve parallel to that of the MatrixApp* component. The gap between the two curves represents the overhead of the distributed graph processing stages (S&G). Since $|V|$ and $|A|$ are fixed, this distributed overhead remains constant, confirming that the marginal increase in runtime is driven solely by the matrix operations on the reduced graph. This validates that our hybrid design successfully confines the computationally expensive, cycle-dependent workload to a compact algebraic step, preventing the cubic complexity from dominating the entire pipeline.

5.3 Modular Evaluation

To answer **RQ2**, we profile the main components of CostApp.

Pregel-based modules. Fig. 8 presents the performance characteristics of the core Pregel-based modules of CostApp: FindFAS and S&G.

As shown in Fig. 8a, both modules demonstrate near-linear scalability with respect to the number of vertices ($|V|$). Notably, the S&G module consistently requires nearly double the execution time of FindFAS. Fig. 8b explains this performance gap through resource consumption. While FindFAS is more communication-intensive in terms of *message count* (due to label propagation), S&G incurs significantly higher overhead per message. This is because S&G employs a map-based mechanism for cost state propagation, which consumes greater computation and communication bandwidth compared to the simple scalar labels in FindFAS. Consequently, despite generating fewer messages, S&G becomes the dominant factor in the distributed phase. Memory usage for both modules scales moderately, remaining within a manageable 12GB per worker even for graphs with 10^7 vertices.

Linear system solver MatrixApp*. Fig. 9a illustrates the performance gains of our hybrid approach in handling the matrix inversion task. The baseline MatrixApp suffers from cubic time complexity and quadratic space complexity relative to $|V|$. As shown in Fig. 9b, its memory footprint explodes on large graphs, leading to OOM failures for graphs exceeding 10^4 vertices. In contrast, MatrixApp* (the solver within CostApp) scales gracefully to graphs with 10^7 vertices, maintaining a negligible memory footprint and runtime. This dramatic improvement validates our dimensionality reduction strategy: by transforming the problem from inverting a massive $|V| \times |V|$ matrix to a tiny $|A_{\text{FAS}}| \times |A_{\text{FAS}}|$ matrix (*e.g.*, , for $|V| = 10^6$, $|A_{\text{FAS}}| = 312 \ll |V|$), we effectively eliminate the computational bottleneck of direct algebraic solutions.

5.4 Real-world Pipeline Evaluation

To validate the performance of CostApp (**RQ3**) on practical pipeline graphs, we conducted experiments on four large-scale production data pipelines at Alibaba. These graphs, anonymized as G1-G4, represent diverse and complex data pipeline workflows.

Sizes of FAS. As the size of FAS identified by FindFAS directly impacts the efficiency of the following steps, we first evaluate $|A_{FAS}|$ of G1—G4 as shown in Fig. 10.Since FindFAS employs randomized label initialization, the size of the identified FAS ($|A_{FAS}|$) exhibits slight variance. We report the average of 10 runs. Crucially, we observe that $|A_{FAS}|$ is consistently lower than 0.05% of $|A|$. This result confirms that the cycles in real-world data pipeline graphs are indeed *sparse*, justifying the design of CostApp that leverages the sparsity of cycles. The small size of FAS is crucial for ensuring the efficiency of the MatrixApp* step.

Overall Performance and Breakdown. We evaluate the performance of CostApp on four large-scale production data pipelines from Alibaba, anonymized as G1-G4, with results shown in Fig. 11.

The empirical results demonstrate the practical efficiency of CostApp on industrial-scale graphs. On the largest graph, G1, the

end-to-end execution completes in approximately 2250 seconds, a runtime that well within the strict time windows required for daily billing cycles in production environments. As the graph size decreases from G1 to G4, the total runtime scales down gracefully, confirming the algorithm's scalability.

The breakdown of execution time provides deep insight into why CostApp scales. The distributed graph processing stages (FindFAS and S&G) dominate the workload, accounting for over 80% of the total runtime on G1. In contrast, the MatrixApp* step consumes a minor fraction ($< 20\%$) of the time. This distribution is highly favorable. It indicates that the total runtime is dominated by the *near-linear* distributed components, while the computationally expensive cubic matrix inversion has been successfully confined to a tiny reduced system. This confirms that CostApp effectively transforming the intractable large cyclic problem into a reduced solvable one, achieving the best of both worlds: the scalability of distributed graph processing and the exactness of solving a linear system.

Overall, experimental results demonstrate that our approach achieves both exactness and scale to massive data pipelines for MCP problem. CostApp can be completed within 1 hour on graphs of Alibaba's real pipeline with millions of vertices, demonstrating its practical applicability in production environments.

6 RELATED WORKS

6.1 Data Pipelines

Data pipelines, particularly ETL (Extract-Transform-Load) workflows, have been widely studied as the backbone of data warehousing and analytics [6, 12, 25, 50–52]. They enable the automated extraction of data from diverse sources, support complex transformations to ensure data quality, and facilitate loading into target stores for subsequent analysis or selling. Many big companies embrace data pipelines to build data manufacturing lines and provide business data pipeline platforms, *e.g.*, Alibaba [52], Meta [43], Google Cloud Dataflow [5], etc. A typical data pipeline is executed in scheduled batch mode [35], which minimizes the load on source systems by operating during off-peak hours. At the same time, a real-time pipeline [52] aims to improve latency and provide more timely data results.

6.2 Cost of Digital Product in Economy

The cost structure of digital products differs fundamentally from that of physical goods, due to major reductions in search, replication, transportation, tracking, and production costs [37]. Search and transportation are made efficient by the Internet, enabling easy discovery and access. Replication costs approach zero, allowing scale, customization, and new models such as bundling and subscriptions [31, 39]. Low tracking costs facilitate personalized pricing, while raising concerns about privacy [18, 36]. Due to free-duplication nature of data, the production cost of data only contains fixed cost that is not related to the sharing amount. Thus, the unit production cost can approach zero through sharing as long as sufficient reuse and sales volume. These facilitate the development of innovative pricing strategies and the evolution of data marketplaces [28, 38, 41, 47, 53]. Different from prior works that focus on the cost structure and pricing strategies of digital products, our work aims to product costing in data pipelines, which is orthogonal to cost structure and a prerequisite for pricing.

7 CONCLUSION AND FUTURE WORK

In this paper, we addressed the critical challenge of product costing in large-scale data pipelines. We formalized the MCP problem to account for the non-rivalrous nature of data. To solve this, we proposed CostApp, a hybrid algorithm that combines distributed graph processing with matrix inversion. By strategically isolating cyclic dependencies via a feedback arc set, CostApp reduces the computational complexity of exact costing from cubic to near-linear relative to the graph size. Extensive experiments on real-world production pipelines at Alibaba demonstrate that our approach achieves both result exactness and high scalability.

Future work. We present some directions to inspire future work.

(1) Explainable product costing in data pipelines. For fine-grained cost optimization, enterprises require a detailed understanding of how each upstream data entity contributes to the cost of a final data product. This can be achieved by maintaining a map that tracks the propagation of monetary costs from each component through the pipeline. This method yields a detailed breakdown of each data product, attributing its total cost to its constituent components and enabling explainability. This idea will incur additional storage and communication overhead. We leave the concrete design and optimization of solutions to this explainability as future work.

(2) Exploration of potential applications of CostApp. Although motivated by data pipelines, CostApp's core capability, which is solving large-scale sparse linear systems with cyclic structures, has the potential to apply to broader problems. One potential application is solving Bellman equations in Markov Decision Processes (MDPs) with sparse state transition matrixs. In an MDP, the value function $\mathbf{v}$ satisfies the Bellman equation $\mathbf{v} = \mathbf{R}+\gamma\mathbf{P}\mathbf{v}$, where $\mathbf{R}$ is the reward vector, $\mathbf{P}$ is the state transition matrix, and γ is the discount factor. Rearranging this equation yields $\mathbf{v} = (\mathbf{I} - \gamma\mathbf{P})^{-1} \cdot \mathbf{R}$, which aligns with the form in §3.3.2. A common method for evaluating the value function is through value iteration [44], which can be proved to be equivalent to the cost iteration of SimpleApp for $L(G)$. The experimental results in §5 suggest that CostApp obtains exact solutions faster than SimpleApp achieves solutions of acceptable accuracy, indicating its potential to accelerate value function computation in MDPs with sparse transitions and large state spaces. Another potential application is input-output economic modeling [26, 27, 46], where industries are interdependent with cyclic supply chains. In this model, $\mathbf{x}$ denotes the total production required and can be obtained by resolving a Leontief inverse: $\mathbf{x} = (\mathbf{I} - \mathbf{A})^{-1} \cdot \mathbf{d}$, where $\mathbf{d}$ is a demand vector and $\mathbf{A}$ is an input-output matrix, which is naturally large-scale and sparse. Thus, CostApp can be directly applied to efficiently compute $\mathbf{x}$ on large-scale economic networks without approximations. These examples demonstrate that CostApp can inspire the optimization of numerical solvers for large-scale sparse linear systems, making it valuable future work to explore more potential applications.

REFERENCES

[1] [n.d.]. The audit of assertions. https://www.accaglobal.com/us/en/student/exam-support-resources/fundamentals-exams-study-resources/f8/technical-articles/assertions.html. Accessed: 2025-06-30.

[2] [n.d.]. Breaking Down The Numbers: How Much Data Does The World Create Daily in 2024? http://edgedelta.com/company/blog/how-much-data-is-created-per-day. Accessed: 2025-09-22.

[3] [n.d.]. Building a Simple Data Pipeline. https://airflow.apache.org/docs/apache-airflow/stable/tutorial/pipeline.html. Accessed: 2025-06-30.

[4] [n.d.]. From ETL to ELT: Modernising Pipelines for High-Volume Metrics. https://www.alibabacloud.com/blog/build-and-run-an-etl-data-pipeline-and-bi-with-luigi-and-metabase-on-alibaba-cloud_598152. Accessed: 2025-06-30.

[5] [n.d.]. Google Cloud Dataflow. https://cloud.google.com/dataflow/docs/overview. Accessed: 2025-06-30.

[6] [n.d.]. What is ETL? https://cloud.google.com/learn/what-is-etl?hl=en. Accessed: 2025-06-30.

[7] Karthik V Aadithya, Balaraman Ravindran, Tomasz P Michalak, and Nicholas R Jennings. 2010. Efficient computation of the shapley value for centrality in networks. In *International workshop on internet and network economics*. Springer, 1–13.

[8] Réka Albert and Albert-László Barabási. 2002. Statistical mechanics of complex networks. *Reviews of modern physics* 74, 1 (2002), 47.

[9] Alibaba Cloud. 2025. MaxCompute. https://www.alibabacloud.com/en/product/maxcompute?_p_lc=1. Accessed: 2025-01-10.

[10] Juliana Ventura Amaral and Reinaldo Guerreiro. 2019. Factors explaining a cost-based pricing essence. *Journal of Business & Industrial Marketing* 34, 8 (2019), 1850–1865.

[11] Michael Backes, Niklas Grimm, and Aniket Kate. 2015. Data lineage in malicious environments. *IEEE Transactions on Dependable and Secure Computing* 13, 2 (2015), 178–191.

[12] Chengliang Chai, Jiayi Wang, Yuyu Luo, Zeping Niu, and Guoliang Li. 2022. Data management for machine learning: A survey. *IEEE Transactions on Knowledge and Data Engineering* 35, 5 (2022), 4646–4667.

[13] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *The Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society, 120–121.

[14] Zicun Cong, Xuan Luo, Jian Pei, Feida Zhu, and Yong Zhang. 2022. Data pricing in machine learning pipelines. *Knowledge and Information Systems* 64, 6 (2022), 1417–1455.

[15] Pierluigi Crescenzi, Viggo Kann, and M Halldórsson. 1995. A compendium of NP optimization problems.

[16] Yingwei Cui and Jennifer Widom. 2003. Lineage tracing for general data warehouse transformations. *the VLDB Journal* 12, 1 (2003), 41–58.

[17] Guy Even. 1995. Joseph (Seffi) Naor, Baruch Schieber, and Madhu Sudan. *Approximating minimum feedback sets and multi-cuts in directed graphs. Integer Programming and Combinatorial Optimization* (1995), 14–28.

[18] Drew Fudenberg and J Miguel Villas-Boas. 2012. Price discrimination in the digital economy. (2012).

[19] Robert Ikeda and Jennifer Widom. 2009. Data lineage: A survey. *Stanford University Publications. http://ilpubs. stanford. edu* 8090, 918 (2009), 1.

[20] Gabriela Jacques-Silva, Evangelia Kalyvianaki, Katriel Cohn-Gordon, Adham Meguid, Huy Nguyen, Danny Ben-David, Carl Nayak, Varun Saravagi, George Stasa, Ioannis Papagiannis, et al. 2025. Unified Lineage System: Tracking Data Provenance at Scale. In *Companion of the 2025 International Conference on Management of Data*. 457–470.

[21] Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nick Hynes, Nezihe Merve Gürel, Bo Li, Ce Zhang, Dawn Song, and Costas J Spanos. 2019. Towards efficient data valuation based on the shapley value. In *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 1167–1176.

[22] Charles I Jones and Christopher Tonetti. 2020. Nonrivalry and the Economics of Data. *American Economic Review* 110, 9 (2020), 2819–2858.

[23] Nicola Jones. 2024. The AI revolution is running out of data. What can researchers do? *Nature* 636, 8042 (2024), 290–292.

[24] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

[25] Bilal Khan, Saifullah Jan, Wahab Khan, and Muhammad Imran Chughtai. 2024. An Overview of ETL Techniques, Tools, Processes and Evaluations in Data Warehousing. *Journal on Big Data* 6 (2024).

[26] Wassily Leontief. 1974. Structure of the world economy: Outline of a simple input-output formulation. *The American Economic Review* 64, 6 (1974), 823–834.

[27] Wassily Leontief. 1986. *Input-output economics*. Oxford University Press.

[28] Jinfei Liu, Jian Lou, Junxu Liu, Li Xiong, Jian Pei, and Jimeng Sun. 2021. Dealer: an end-to-end model marketplace with differential privacy. *Proceedings of the VLDB Endowment* 14, 6 (2021).

[29] Rui Liu, Kwanghyun Park, Fotis Psallidas, Xiaoyong Zhu, Jinghui Mo, Rathijit Sen, Matteo Interlandi, Konstantinos Karanasos, Yuanyuan Tian, and Jesús Camacho-Rodríguez. 2023. Optimizing Data Pipelines for Machine Learning in Feature Stores. *Proceedings of the VLDB Endowment* 16, 13 (2023), 4230–4239.

[30] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, et al. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. In *VLDB 2022-48th International Conference on Very Large Databases*, Vol. 15.

[31] Jeffrey K MacKie-Mason, Juan F Riveros, and Robert S Gazzale. 2000. Pricing and bundling electronic information goods: Experimental evidence. MIT Press.

[32] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD)*. 135–146.

[33] Aliomar Lino Mattos, JosÃ Oyadomari, Fernando Nascimento Zatta, et al. 2021. Pricing Research: State of the Art and Future Opportunities. *SAGE Open* 11, 3 (2021).

[34] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[35] Aiswarya Raj Munappy, Jan Bosch, and Helena Homström Olsson. 2020. Data pipeline management in practice: Challenges and opportunities. In *Product-Focused Software Process Improvement: 21st International Conference, PROFES 2020, Turin, Italy, November 25–27, 2020, Proceedings 21*. Springer, 168–184.

[36] Andrew Odlyzko. 2003. Privacy, economics, and price discrimination on the internet. In *Proceedings of the 5th international conference on Electronic commerce*. 355–366.

[37] Jian Pei. 2020. A survey on data pricing: from economics to data science. *IEEE Transactions on knowledge and Data Engineering (TKDE)* 34, 10 (2020), 4586–4608.

[38] Jian Pei, Raul Castro Fernandez, and Xiaohui Yu. 2023. Data and ai model markets: Opportunities for data and model sharing, discovery, and integration. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3872–3873.

[39] Carl Shapiro. 1999. *Information rules: A strategic guide to the network economy*. Harvard Business School Press.

[40] Carl Shapiro and Hal R Varian. 1998. Versioning: the smart way to sell information. *Harvard business review* 107, 6 (1998), 107.

[41] Wei Shuyue, Yongxin Tong, Zimu Zhou, Tianran He, and Yi Xu. 2025. Efficient Data Valuation Approximation in Federated Learning: A Sampling-Based Approach. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2922–2934.

[42] Tapan Srivastava and Raul Castro Fernandez. 2024. Saving Money for Analytical Workloads in the Cloud. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3524–3537.

[43] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, et al. 2023. Presto: A decade of SQL analytics at Meta. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[44] R.S. Sutton and A.G. Barto. 2018. *Reinforcement Learning, second edition: An Introduction*. MIT Press. https://books.google.com.br/books?id=sWV0DwAAQBAJ

[45] Jan Vincent Szlang, Sebastian Bress, Sebastian Cattes, Jonathan Dees, Florian Funke, Max Heimel, Michel Oleynik, Ismail Oukid, and Tobias Maltenberger. 2025. Workload Insights from the Snowflake Data Cloud: What Do Production Analytic Queries Really Look Like? *Proceedings of the VLDB Endowment* 18, 12 (2025), 5126–5138.

[46] Thijs Ten Raa. 2009. *Input-output economics: Theory and applications-featuring Asian economies*. World Scientific.

[47] Yongxin Tong, Libin Wang, Zimu Zhou, Lei Chen, Bowen Du, and Jieping Ye. 2018. Dynamic pricing in spatial crowdsourcing: A matching-based approach. In *Proceedings of the 2018 international conference on management of data*. 773–788.

[48] Shuyue Wei, Yongxin Tong, Zimu Zhou, and Tianshu Song. 2020. Efficient and fair data valuation for horizontal federated learning. In *Federated Learning: Privacy and Incentive*. Springer, 139–152.

[49] Caesar Wu, Rajkumar Buyya, and Kotagiri Ramamohanarao. 2019. Cloud pricing models: Taxonomy, survey, and interdisciplinary challenges. *ACM Computing Surveys (CSUR)* 52, 6 (2019), 1–36.

[50] Xike Xie, Xingjun Hao, Torben Bach Pedersen, Peiquan Jin, and Jinchuan Chen. 2016. OLAP over probabilistic data cubes I: Aggregating, materializing, and querying. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 799–810.

[51] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. 2015. Lenses: An on-demand approach to etl. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1578–1589.

[52] Fangyuan Zhang, Mengqi Wu, Chunlei Xu, Yunong Bao, Jiyu Qiao, Yingli Zhou, Hua Fan, Caihua Yin, Wenchao Zhou, and Feifei Li. 2025. Streaming View: An Efficient Data Processing Engine for Modern Real-Time Data Warehouse of Alibaba Cloud. *Proceedings of the VLDB Endowment* 18, 12 (2025), 5153–5165.

[53] Jiayao Zhang, Yuran Bi, Mengye Cheng, Jinfei Liu, Kui Ren, Qiheng Sun, Yihang Wu, Yang Cao, Raul Castro Fernandez, Haifeng Xu, et al. 2024. A Survey on Data Markets. *arXiv preprint arXiv:2411.07267* (2024).

[54] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. 2024. cedar: Optimized and Unified Machine Learning Input Data Pipelines. *Proceedings of the VLDB Endowment* 18, 2 (2024), 488–502.

[55] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2022. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th annual international symposium on computer architecture*. 1042–1057.

A PROOF OF THEOREM 2

We prove Theorem 2 as follows.

Proof. We begin with the total direct cost of the system. By rearranging Eq. 2, we can express the direct cost of any single vertex v as $C_{dc}(v) = C_{mc}(v) - C_{ic}(v)$. Summing over all vertices in the graph $V(G)$:

$$\begin{aligned}\sum_{v\in V(G)} C_{dc}(v) &= \sum_{v\in V(G)} (C_{mc}(v) - C_{ic}(v)) \\ &= \sum_{v\in V(G)} C_{mc}(v) - \sum_{v\in V(G)} C_{ic}(v)\end{aligned} \tag{6}$$

Let us analyze the total input cost term, $\sum_{v\in V(G)} C_{ic}(v)$. By the definition of input cost $C_{ic}(v)$, we get:

$$\sum_{v\in V(G)} C_{ic}(v) = \sum_{v\in V(G)} \sum_{u\in U_G^1(v)} w(u,v)\cdot C_{mc}(u)$$

This double summation iterates over all directed arcs $(u, v) \in A(G)$. We can change the order of summation to first sum over all potential source nodes $u \in V(G)$ and then over their direct downstream neighbors $v \in D_G^1(u)$:

$$\sum_{v\in V(G)} C_{ic}(v) = \sum_{u\in V(G)} \sum_{v\in D_G^1(u)} w(u,v)\cdot C_{mc}(u)$$

We can factor out $C_{mc}(u)$ from the inner summation, as it is constant with respect to v:

$$\sum_{v\in V(G)} C_{ic}(v) = \sum_{u\in V(G)} \left(C_{mc}(u)\cdot \sum_{v\in D_G^1(u)} w(u,v)\right)$$

By the cost conservation property (Definition 1), the inner sum $\sum_{v\in D_G^1(u)} w(u,v)$ equals 1 for any non-leaf node $u \in V(G) \setminus L(G)$. For any leaf node $p \in L(G)$, its set of downstream neighbors $D_G^1(p)$ is empty, so the inner sum is 0. Therefore, the expression simplifies to the sum of manufacturing costs over all non-leaf nodes:

$$\sum_{v\in V(G)} C_{ic}(v) = \sum_{u\in V(G)\setminus L(G)} C_{mc}(u)\cdot 1 = \sum_{u\in V(G)\setminus L(G)} C_{mc}(u)$$

Substituting this result back into Eq. 6:

$$\sum_{v\in V(G)} C_{dc}(v) = \sum_{v\in V(G)} C_{mc}(v) - \sum_{v\in V(G)\setminus L(G)} C_{mc}(v)$$

The right-hand side is the sum of manufacturing costs over all vertices minus the sum of manufacturing costs over all non-leaf vertices. This difference is exactly the sum of manufacturing costs over the leaf vertices, $L(G)$.

$$\sum_{v\in V(G)} C_{dc}(v) = \sum_{p\in L(G)} C_{mc}(p)$$

This completes the proof. □

B THE THREASHOLDABLE VARIANT OF SIMPLEAPP

We detail an algorithm variant of SimpleApp to handle cycles in the data pipeline graph in Alg. 5, termed SimpleApp-T. It limits the number of iterations to a fixed step threshold t. It is designed to ensure the termination of cost propagation while still allowing for an approximate cost accumulation at the leaves.

Algorithm 5: SimpleApp-T (Pregel-style pseudocode)

Input: Graph $G = (V, A)$, operation costs $C_{dc}(v)$, arc weights $w_{u,v}$, step threshold t
Output: $C_1(v)$

```
Each vertex v ∈ V sets C_1(v) ← C(v) ;          // Initialization
n ← 0 ;                                          // Superstep counter
while any non-leaf vertex v is active and n < t do
    n ← n + 1;
    foreach active vertex v do in parallel
        /* CA operation in Pregel style                                  */
        compute(msgs)
            update C_1(v) ← C_1(v) + Σ_{m_i ∈ msgs} m_i;
            sends w(v, v') · C_1(v) to each v' ∈ D_G^1(v);
            updates C_1(v) ← 0;
            voteToHalt() // Reactivated upon receiving a message
return C_1(v)
```

C PROOF OF THEOREM 3

Proof. Let $C^k(v)$ denote the cost held by vertex v at the beginning of superstep k, with $C^1(v) = C_{dc}(v)$. The algorithm terminates when $C^k(v) = 0$ for all non-leaf vertices $v \in V \setminus L(G)$. The proof proceeds in two parts: termination and correctness.

1. Termination. In a DAG, the cost propagation is unidirectional. A cost contribution from an initial $C_{dc}(u)$ can only reach a vertex v via a path from u to v. After one superstep, this cost can reach vertices at a path distance of 1. After k supersteps, it can reach vertices at a path distance of at most k.

Let $P = \max_{u,v\in V}\{\text{length of longest path from } u \text{ to } v\}$ be the longest path length in G. After P supersteps, any cost originating from any vertex u will have propagated along all possible paths of length up to P. Since no path in the graph is longer than P, all costs must have passed through all intermediate (non-leaf) vertices on their respective paths and accumulated at the leaf vertices.

Consequently, at the beginning of superstep $P + 1$, no non-leaf vertex $v \in V \setminus L(G)$ can receive any message, as all upstream cost flows have already terminated at leaves. Since all non-leaf vertices also distributed their own costs in prior supersteps, $C_{P+1}(v) = 0$ for all $v \in V \setminus L(G)$. This satisfies the termination condition, proving the algorithm halts in at most P supersteps.

2. Correctness. Let $C^{final}(p)$ be the final cost at a leaf $p \in L(G)$ upon termination. We must show $C^{final}(p) = C_{mc}(p)$.

First, observe that the total cost is conserved. In any superstep, for any active non-leaf vertex v, the cost it sends out, $\sum_{v'\in D_G^1(v)} w_{v\to v'} C^k(v)$, equals $C^k(v)$ because $\sum_{v'\in D_G^1(v)} w_{v\to v'} = 1$ (cost conservation). Thus, $\sum_{v\in V} C^{k+1}(v) = \sum_{v\in V} C^k(v)$. Upon termination,

$$\sum_{p\in L(G)} C^{final}(p) = \sum_{v\in V} C^1(v) = \sum_{v\in V} C_{dc}(v),$$

satisfying cost conservation.

Let $C^*(v)$ be the total cost that flows through or terminates at vertex v throughout the algorithm's execution. This is the sum of its direct cost and all costs received from upstream neighbors over all supersteps.

$$C^*(v) = C_{dc}(v) + \sum_{u \in U_G^1(v)} (\text{total cost received from } u) \tag{7}$$

The total cost received by v from an upstream neighbor u is the proportion $w(u, v)$ of the total cost that flowed through u, which is precisely $w(u, v) \cdot C^*(u)$. Substituting this gives:

$$C^*(v) = C_{dc}(v) + \sum_{u \in U_G^1(v)} w(u, v) \cdot C^*(u) \tag{8}$$

This system of linear equations defining $C^*(v)$ for all $v \in V$ is identical to the one defining the manufacturing cost $C_{mc}(v)$ in Equations 2. Since G is a DAG, this system has a unique solution. Therefore, $C^*(v) = C_{mc}(v)$ for all $v \in V$.

For a leaf node $p \in L(G)$, it only accumulates cost and never distributes it. Its final cost $C^{final}(p)$ is the total cost that has flowed into it, which is $C^*(p)$. Thus:

$$C^{final}(p) = C^*(p) = C_{mc}(p) \tag{9}$$

This confirms that the algorithm correctly computes the manufacturing cost for all data products. □

D MATRIX FORM DERIVATION

To derive the matrix form of MC from the scalar equations, we start with the definition of MC for a single vertex v_i:

$$C_{mc}(v_i) = C_{dc}(v_i) + \sum_{v_j \in U_G^1(v_i)} w(v_j, v_i) \cdot C_{mc}(v_j)$$

This equation holds for all n vertices in the graph $V = \{v_1, \ldots, v_n\}$. We can express this system of n linear equations in matrix form as follows:

Let $\mathbf{c}$ be the $n \times 1$ column vector of manufacturing costs, where the i-th element is $\mathbf{c}_i = C_{mc}(v_i)$. Let $\mathbf{d}$ be the $n \times 1$ column vector of direct costs, where the i-th element is $\mathbf{d}_i = C_{dc}(v_i)$. Let $\mathbf{W}$ be the $n \times n$ weighted adjacency matrix, where $\mathbf{W}[i, j] = w(v_i, v_j)$ if an arc exists from v_i to v_j, and 0 otherwise.

The summation term represents the total input cost for vertex v_i. Let's analyze the matrix-vector product $\mathbf{W}^\mathsf{T}\mathbf{c}$. The i-th element of this product vector is computed as:

$$(\mathbf{W}^\mathsf{T}\mathbf{c})_i = \sum_{j=1}^{n} (\mathbf{W}^\mathsf{T})_{ij} \cdot \mathbf{c}_j = \sum_{j=1}^{n} \mathbf{W}[j, i] \cdot \mathbf{c}_j = \sum_{j=1}^{n} w(v_j, v_i) \cdot C_{mc}(v_j)$$

Since $w(v_j, v_i)$ is non-zero only when v_j is a direct predecessor of v_i (i.e., $v_j \in U_G^1(v_i)$), this sum is exactly equivalent to the input cost term in the original scalar equation.

We can now rewrite the scalar equation for each vertex v_i using this vector notation:

$$\mathbf{c}_i = \mathbf{d}_i + (\mathbf{W}^\mathsf{T}\mathbf{c})_i$$

Since this relationship holds for every element $i = 1, \ldots, n$, we can express it for the entire system of equations in vector form:

$$\begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \\ \vdots \\ \mathbf{c}_n \end{pmatrix} = \begin{pmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \vdots \\ \mathbf{d}_n \end{pmatrix} + \begin{pmatrix} (\mathbf{W}^\mathsf{T}\mathbf{c})_1 \\ (\mathbf{W}^\mathsf{T}\mathbf{c})_2 \\ \vdots \\ (\mathbf{W}^\mathsf{T}\mathbf{c})_n \end{pmatrix}$$

This simplifies to the final matrix equation:

$$\mathbf{c} = \mathbf{d} + \mathbf{W}^\mathsf{T}\mathbf{c}$$

E PROOF OF MATRIX INVERTIBILITY

We prove that $(\mathbf{I} - \mathbf{W}^\mathsf{T})$ is invertible by showing that the spectral radius of $\mathbf{W}$ satisfies $\rho(\mathbf{W}) < 1$. Since a matrix and its transpose share the same eigenvalues, this suffices to guarantee that all eigenvalues of $(\mathbf{I} - \mathbf{W}^\mathsf{T})$ are strictly positive, and thus the matrix is nonsingular.

Recall that $\mathbf{W}$ is the cost allocation matrix defined in §3.3.2. By construction:

(1) $\mathbf{W}$ is non-negative: $\mathbf{W}[i, j] \geq 0$ for all i, j.
(2) For every non-leaf vertex $u \in V(G) \setminus L(G)$, the outgoing weights satisfy the conservation constraint
$$\sum_{v \in D_G^1(u)} w(u, v) = 1,$$
so the corresponding row of $\mathbf{W}$ sums to 1.
(3) For every leaf vertex $p \in L(G)$, we have $D_G^1(p) = \emptyset$, hence the entire row of $\mathbf{W}$ is zero.

Thus every row sum of $\mathbf{W}$ is at most 1, and at least one row sum (any leaf row) is strictly less than 1.

Step 1: $\rho(\mathbf{W}) \leq 1$. Let λ be an eigenvalue of $\mathbf{W}$ with eigenvector $\mathbf{x} \neq \mathbf{0}$, so $\mathbf{W}\mathbf{x} = \lambda\mathbf{x}$. Let k be an index such that

$$|x_k| = \max_i |x_i| > 0.$$

Consider the k-th coordinate of the eigen-equation:

$$\lambda x_k = \sum_{j=1}^{n} \mathbf{W}[k, j] x_j.$$

Taking absolute values and using the triangle inequality,

$$|\lambda|\,|x_k| = \left| \sum_{j=1}^{n} \mathbf{W}[k, j] x_j \right| \leq \sum_{j=1}^{n} \mathbf{W}[k, j] |x_j| \leq \left(\sum_{j=1}^{n} \mathbf{W}[k, j] \right) |x_k| \leq |x_k|;$$

Since $|x_k| > 0$, we obtain $|\lambda| \leq 1$. Hence $\rho(\mathbf{W}) \leq 1$.

Step 2: $\rho(\mathbf{W}) < 1$. We now show that $|\lambda| = 1$ is impossible. Suppose, for contradiction, that there exists an eigenvalue λ with $|\lambda| = 1$. For the chain of inequalities above to hold as equalities, we must simultaneously have:

(1) The row sum of row k equals 1, i.e.,
$$\sum_{j=1}^{n} \mathbf{W}[k, j] = 1.$$
Thus vertex k is not a leaf (since leaf rows are all zeros).
(2) For every j with $\mathbf{W}[k, j] > 0$, we must have $|x_j| = |x_k|$; otherwise the second inequality would be strict.

Algorithm 6: Label (Pregel-style pseudocode)

Input: Graph $G = (V, A)$.
Output: The final label $L(v)$ for each vertex $v \in V$.

```
/* Initialization: Assign unique integer labels to roots                */
foreach root vertex v_i ∈ R(G) in parallel: L(v_i) ← i;
foreach non-root vertex v ∈ V \ R(G): L(v) ← ∅;
while any vertex is active do
    foreach active vertex v do in parallel
        compute(msgs)
            label_updated ← false;
            L_best_incoming ← L(v);
            foreach message containing label L_msg do
                if L_msg >_lo L_best_incoming then
                    L_best_incoming ← L_msg;
            if L_best_incoming >_lo L(v) then
                L(v) ← L_best_incoming;
                label_updated ← true;
            if label_updated then
                /* Propagate new labels to downstream neighbors         */
                foreach j-th neighbor v_j ∈ D^1_G(v) do
                    send message with label L(v) + ".j" to v_j;
            voteToHalt();
            // Reactivated upon receiving a message
return final labels L(v) for all v ∈ V;
```

Condition (2) implies that every direct downstream neighbor of k with a positive weight also has the same maximal modulus $|x_j| = |x_k|$. Repeating this argument along any directed path starting from k, we conclude that all vertices reachable from k with positive path weight also satisfy $|x_i| = |x_k|$.

In a valid data pipeline, every non-leaf vertex has at least one path to some leaf data product. Therefore, there exists a leaf vertex $p \in L(G)$ reachable from k via a directed path with positive total weight. By the propagation argument above, this implies

$$|x_p| = |x_k| > 0.$$

However, since p is a leaf, its outgoing weights are all zero, so the p-th row of $\mathbf{W}$ is the zero vector. The eigen-equation at coordinate p is

$$\lambda x_p = \sum_{j=1}^{n} \mathbf{W}[p, j] x_j = 0.$$

With $|\lambda| = 1$, this forces $x_p = 0$, contradicting $|x_p| = |x_k| > 0$.

Therefore no eigenvalue of $\mathbf{W}$ can satisfy $|\lambda| = 1$, and thus

$$\rho(\mathbf{W}) < 1.$$

Step 3: Invertibility of $(\mathbf{I} - \mathbf{W}^\mathsf{T})$. Since $\rho(\mathbf{W}) < 1$ and $\mathbf{W}$ and $\mathbf{W}^\mathsf{T}$ share the same eigenvalues, every eigenvalue of $\mathbf{W}^\mathsf{T}$ also has modulus strictly less than 1. Hence every eigenvalue of $(\mathbf{I} - \mathbf{W}^\mathsf{T})$ is strictly positive, so $(\mathbf{I} - \mathbf{W}^\mathsf{T})$ is nonsingular and invertible. This guarantees that the linear system

$$\mathbf{c} = \mathbf{d} + \mathbf{W}^\mathsf{T}\mathbf{c}$$

has a unique solution $\mathbf{c} = (\mathbf{I} - \mathbf{W}^\mathsf{T})^{-1}\mathbf{d}$.

F THE LABEL ALGORITHM

The Label algorithm, implemented in a Pregel-like paradigm, is presented in Alg. 6. The process begins by assigning unique integer labels to all root vertices (those with in-degree 0). In each subsequent superstep, every active vertex performs an update based on messages from its upstream neighbors. A vertex v adopts a new label if it receives a label from an upstream neighbor that is lexicographically greater than its current label $L(v)$. If its label is updated, it generates new, extended labels (by appending a suffix '.i') and propagates them to its downstream neighbors. This iterative process continues until no more label updates occur, at which point every vertex has a stable, final label.

G PROOF OF LEMMA 1

The proof of Lemma 1 is provided below.

Proof Sketch. Consider the graph $G' = (V, A \setminus A_{\text{FAS}})$. For any remaining arc $\langle u, v \rangle$ in G', we have $L(v) >_{lo} L(u)$. This implies that a traversal along any path in G' corresponds to a monotonically increasing sequence of labels. Such a sequence cannot repeat a vertex, proving that G' contains no cycles. □

H A FORMULATION OF GRAPHEDIT

Alg. 7 presents the pseudocode for the GraphEdit algorithm. This algorithm takes as input a directed graph G and a feedback arc set A_{FAS}. For each arc in A_{FAS}, it removes the arc from G and introduces two new auxiliary vertices: a sink vertex and a source vertex. The original arc is replaced by three new arcs: one from the original source to the new sink, one from the new source to the original target, and one connecting the new sink to the new source. The weights of these new arcs are assigned as specified in the algorithm.

Algorithm 7: GraphEdit: Editing G by Arc Replacement

Input: Directed graph $G = (V, A)$, feedback arc set $A_{\text{FAS}} \subseteq A$
Output: Edited graph $G_1 = (V_1, A_1)$, set of new sink vertices V_{sk}, set of new source vertices V_{sr}

```
G_1 ← G, V_sk ← ∅, V_sr ← ∅;
foreach edge f = (u, v, w(u, v)) ∈ A_FAS do
    Remove edge f from G_1;
    Create new vertex v_sk and add to G_1 and V_sk;
    Create new vertex v_sr and add to G_1 and V_sr;
    Add new edge (u, v_sk, w(u, v)) to G_1;
    Add new edge (v_sr, v, 1) to G_1;
    Add new edge (v_sk, v_sr, 1) to G_1;
return (G_1, V_sk, V_sr) ;
```

I PROOF OF THEOREM 5

Proof. Let $\mathbf{c}^*$ and $\mathbf{c}_1^*$ be the unique manufacturing cost vectors that solve the linear systems for graphs G and G_1, respectively. We need to show that for any original vertex $v \in V(G)$, its manufacturing cost is the same in both solutions, i.e., $\mathbf{c}^*(v) = \mathbf{c}_1^*(v)$.

The GraphEdit procedure modifies the graph locally for each feedback arc $(u, v) \in A_{\text{FAS}}$. Let's analyze the effect of a single such transformation. The cost equations for all vertices not involved in any feedback arc remain unchanged. We only need to examine the equations for vertices u, v, and the new vertices v_{sk} and v_{sr}.

Algorithm 8: SimpleApp* (Pregel-style pseudocode)

Input: Graph $G = (V, A, w)$, stateful cost map $C(v)$
Output: $C^{out}(v)$

Each vertex $v \in V$ sets $C^{out}(v) \leftarrow C(v)$; // Initialization
while *any non-leaf vertex v is active* **do**
  foreach active vertex v **do** in parallel
    /* CA operation in Pregel style */
    compute(msgs)
      foreach $m \in msgs$ **do**
        foreach $key \in m$.getKeys() **do**
          update $C^{out}(v)[key] \leftarrow C^{out}(v)[key] + m$.getValue($key$);
      sends $w(v, v') \cdot C^{out}(v)$ to each $v' \in D^1_G(v)$; // Send weighted map to downstream neighbors
      updates $C^{out}(v) \leftarrow \emptyset$;
      voteToHalt();
      // Reactivated upon receiving a message
return $C^{out}(v)$

In the original graph G, the manufacturing cost for vertex v includes a term from u:

$$C_{mc}(v) = C_{dc}(v) + w(u,v) \cdot C_{mc}(u) + \sum_{z \in U^1_G(v) \setminus \{u\}} w_{z \to v} \cdot C_{mc}(z) \tag{10}$$

In the transformed graph G_1, the arc (u, v) is replaced by the path $u \to v_{sk} \to v_{sr} \to v$. The new cost equations are:

$$C_{mc}(v_{sk}) = C_{dc}(v_{sk}) + w_{u \to v_{sk}} \cdot C_{mc}(u) \tag{11}$$

$$C_{mc}(v_{sr}) = C_{dc}(v_{sr}) + w_{v_{sk} \to v_{sr}} \cdot C_{mc}(v_{sk}) \tag{12}$$

$$C_{mc}(v) = C_{dc}(v) + w_{v_{sr} \to v} \cdot C_{mc}(v_{sr}) + \sum_{z \in U^1_G(v) \setminus \{u\}} w_{z \to v} \cdot C_{mc}(z) \tag{13}$$

By construction in Algorithm 7, the new vertices have zero direct cost, i.e., $C_{dc}(v_{sk}) = C_{dc}(v_{sr}) = 0$. The weights are set as $w_{u \to v_{sk}} = w(u, v)$, $w_{v_{sk} \to v_{sr}} = 1$, and $w_{v_{sr} \to v} = 1$.

Substituting Eq. 11 into Eq. 12, we get:

$$C_{mc}(v_{sr}) = 1 \cdot C_{mc}(v_{sk}) = w(u,v) \cdot C_{mc}(u)$$

Now, substituting this result for $C_{mc}(v_{sr})$ into Eq. 13:

$$C_{mc}(v) = C_{dc}(v) + 1 \cdot (w(u,v) \cdot C_{mc}(u)) + \sum_{z \in U^1_G(v) \setminus \{u\}} w_{z \to v} \cdot C_{mc}(z)$$

This equation is identical to the original cost equation for v (Eq. 10).

Since the cost equation for every original vertex $v \in V(G)$ remains unchanged in the transformed system, the solution to the system of linear equations for all original vertices must also be identical. Specifically, the manufacturing costs for all data products $p \in L(G) \subset V(G)$ are preserved. □

J THE SIMPLEAPP* ALGORITHM

We present the SimpleApp* algorithm in Alg. 8, a Pregel-style pseudocode variant of SimpleApp (Alg. 1). This variant utilizes a stateful cost map $C(v)$ at each vertex v to accumulate costs for multiple keys simultaneously. During each superstep, active vertices aggregate incoming cost maps from upstream neighbors into their local cost map and then propagate weighted versions of this aggregated map to their downstream neighbors. This approach allows for efficient handling of multiple cost dimensions in a single pass through the graph.