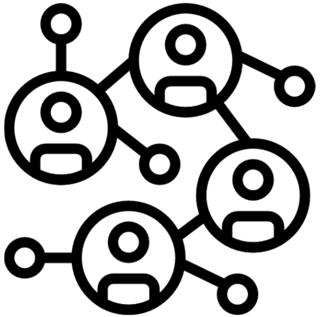
Blaze:
Holistic Caching for Iterative Data Processing

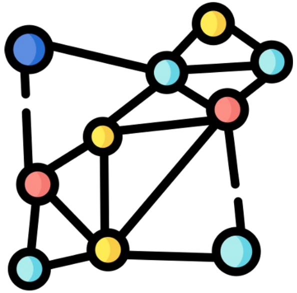
Won Wook SONG, Jeongyoon Eo, Taegeon Um, Myeongjae Jeon, Byung-Gon Chun

Seoul National University, Samsung Research, UNIST, FriendliAI

Caching is Essential in Complex Iterative Data Processing

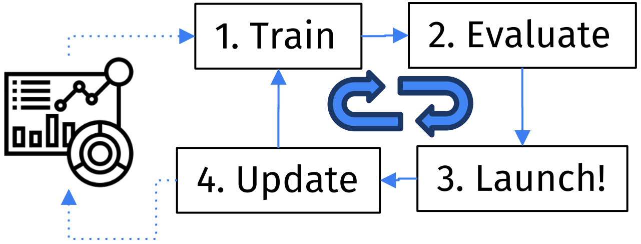
Graph Processing





e.g., PageRank, Connected Components

Iterative Machine Learning

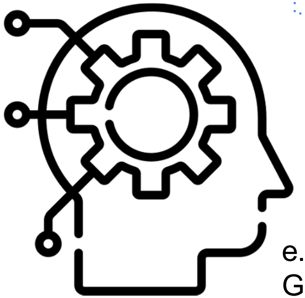


1. Train

2. Evaluate

3. Launch!

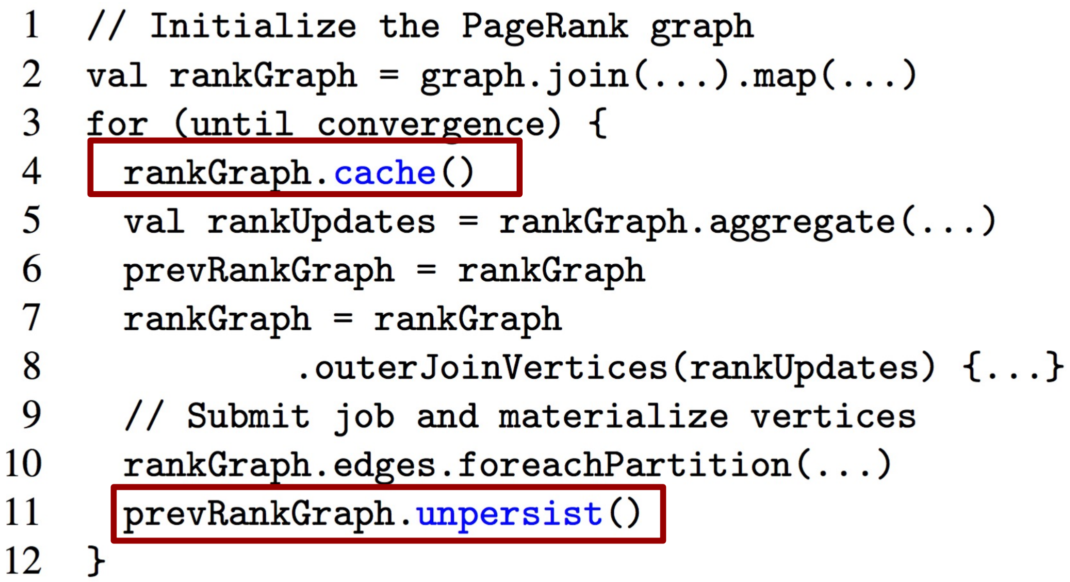
4. Update



e.g., Logistic Regression, K-Means Clustering, Gradient Boosted Trees, SVD++

Without caching, intermediate data must be recomputed every time by default

Cache Hints for Indicating Data to Cache

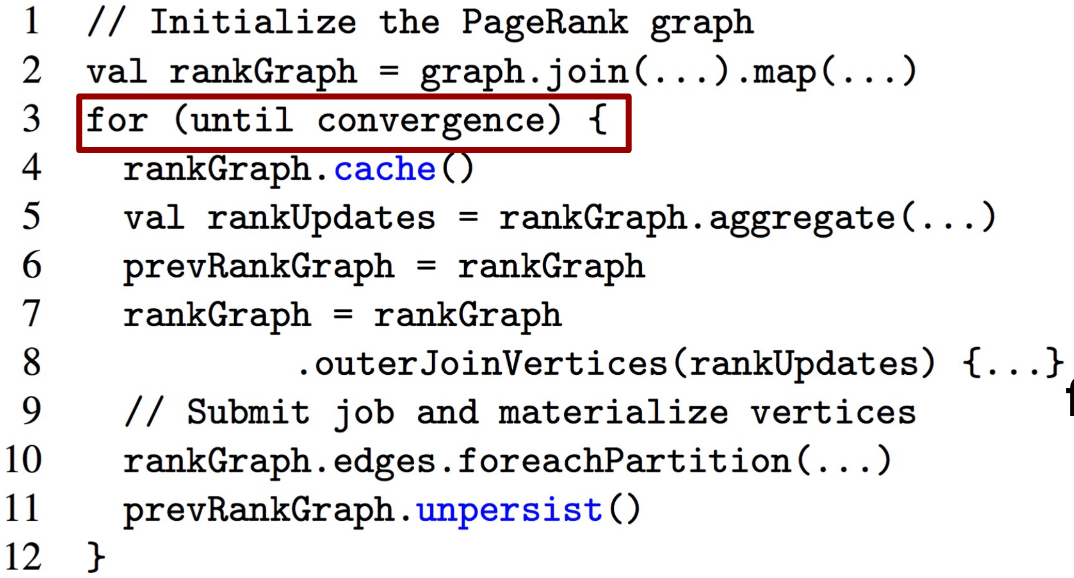


```
// Initialize the PageRank graph
val rankGraph = graph.join(...).map(...)
for (until convergence) {
  rankGraph.cache()
  val rankUpdates = rankGraph.aggregate(...)
  prevRankGraph = rankGraph
  rankGraph = rankGraph
           .outerJoinVertices(rankUpdates) {...}
  // Submit job and materialize vertices
  rankGraph.edges.foreachPartition(...)
  prevRankGraph.unpersist()
}
```

Caching decisions are made by users on an operator dataset level

→ **Caches all partitions of the dataset**

Caching is Done Repeatedly over Iterations until Convergence

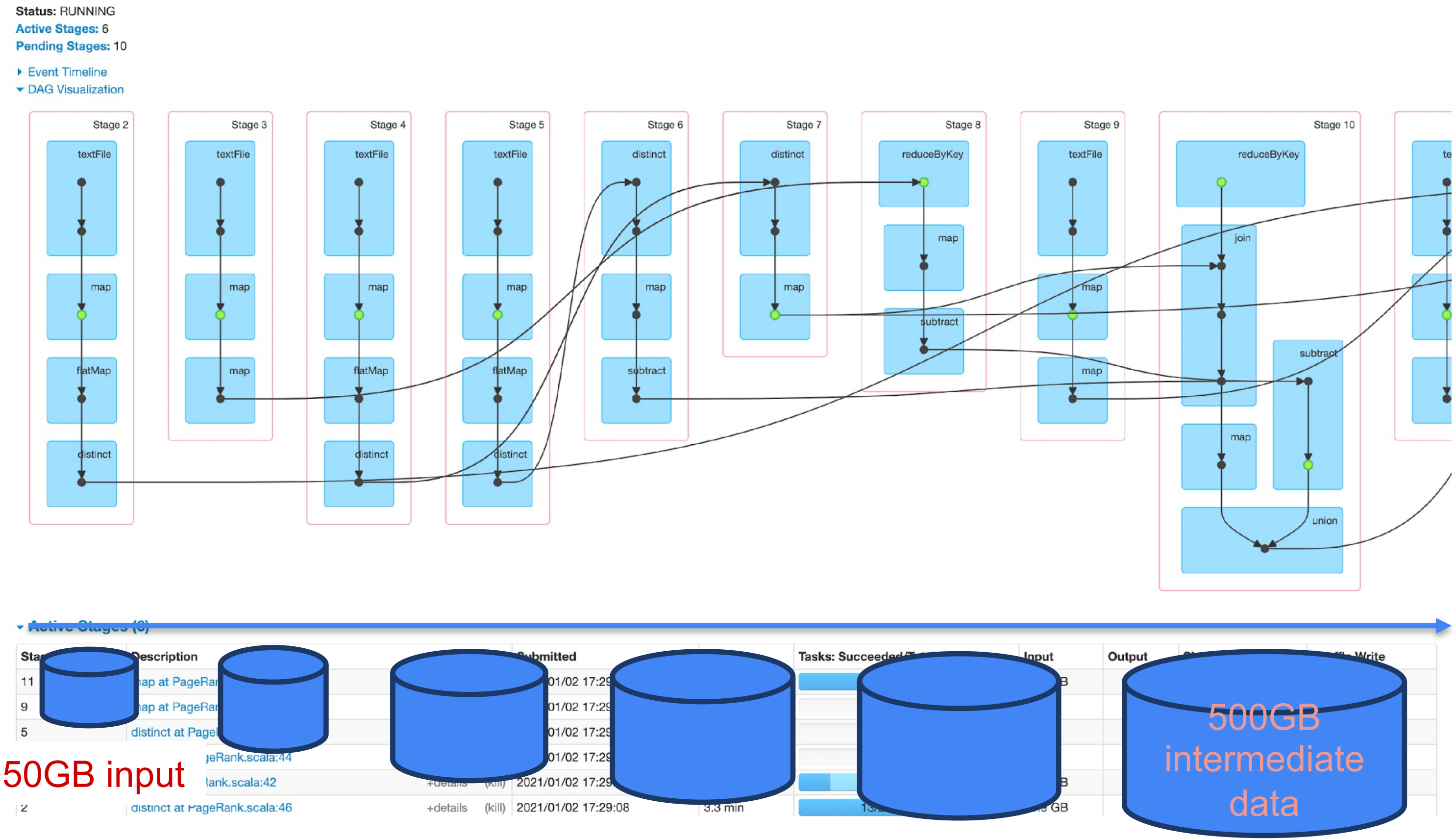


```
// Initialize the PageRank graph
val rankGraph = graph.join(...).map(...)
for (until convergence) {
  rankGraph.cache()
  val rankUpdates = rankGraph.aggregate(...)
  prevRankGraph = rankGraph
  rankGraph = rankGraph
          .outerJoinVertices(rankUpdates) {...}
  // Submit job and materialize vertices
  rankGraph.edges.foreachPartition(...)
  prevRankGraph.unpersist()
}
```

The complete job DAG is unknown
until convergence
(one job == one iteration)

→ Caching is done in a greedy fashion, due to unknown future data references and dependencies

Cache Data Size Increases Over Iterations



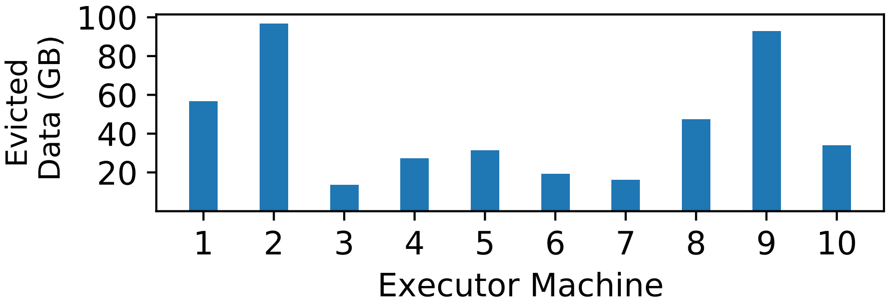
Status: RUNNING
Active Stages: 6
Pending Stages: 10

- Event Timeline
- DAG Visualization

50GB input

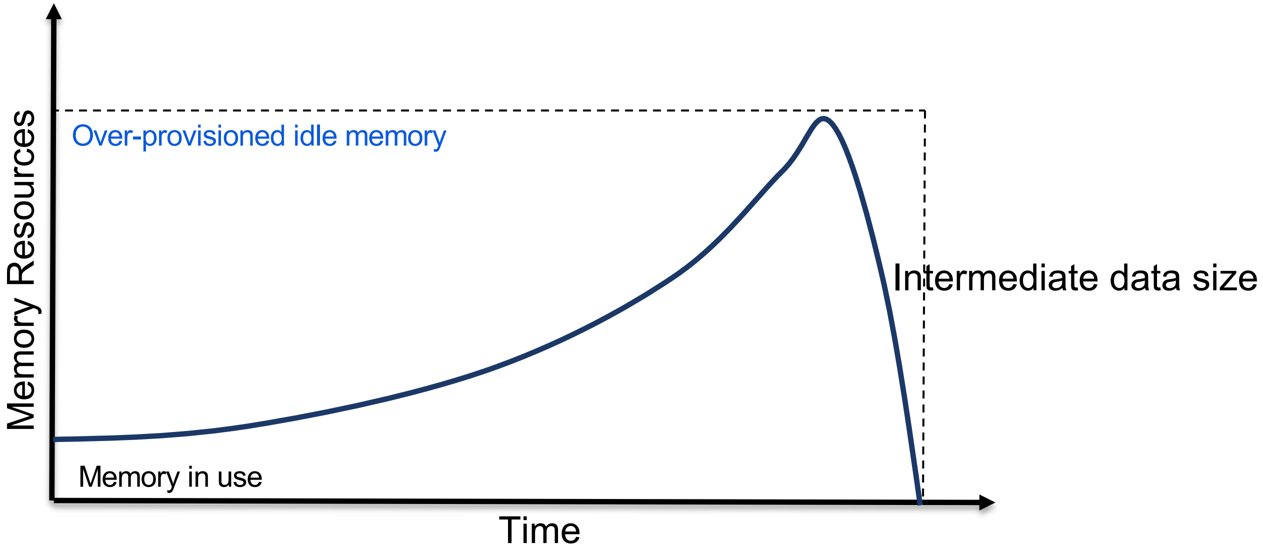
500GB intermediate data

Cache Data Size is Inconsistent Among Executors



Inconsistent memory usage → Bottleneck executors make optimizations tricky

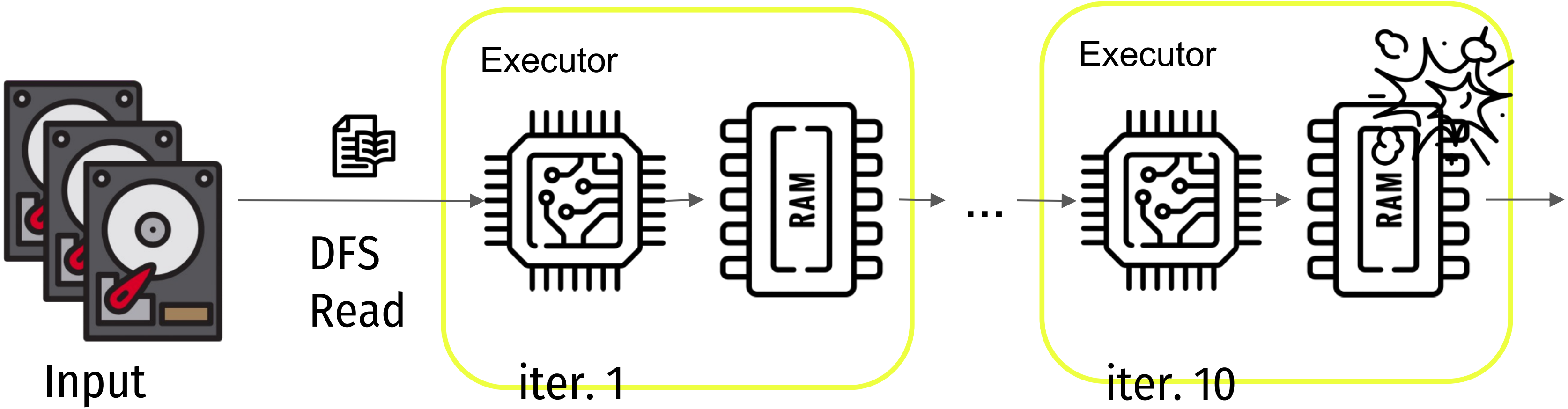
A Naïve Solution: Over-provisioning Memory



Simplest solution, but reserving memory of *>10x the input size* is *costly*

Plus, we cannot *foresee* when the iterations will *conclude*

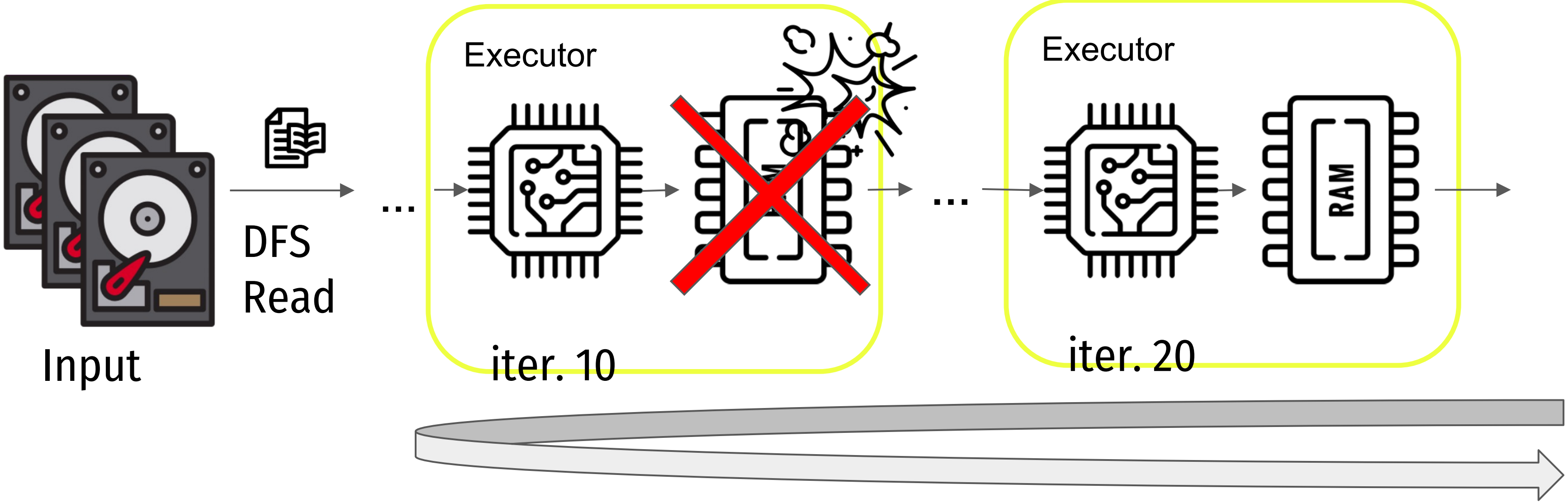
Problem: Memory Space is Constrained



Also, caching in memory is difficult to scale

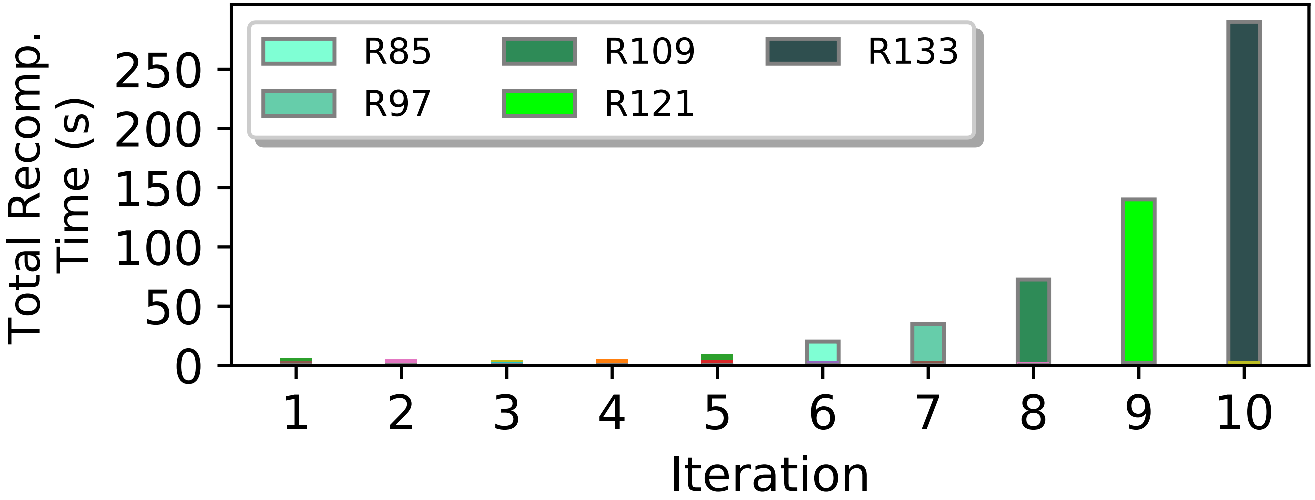
(Need to increase # of VMs)

Goal: Reducing Recomputation Overhead



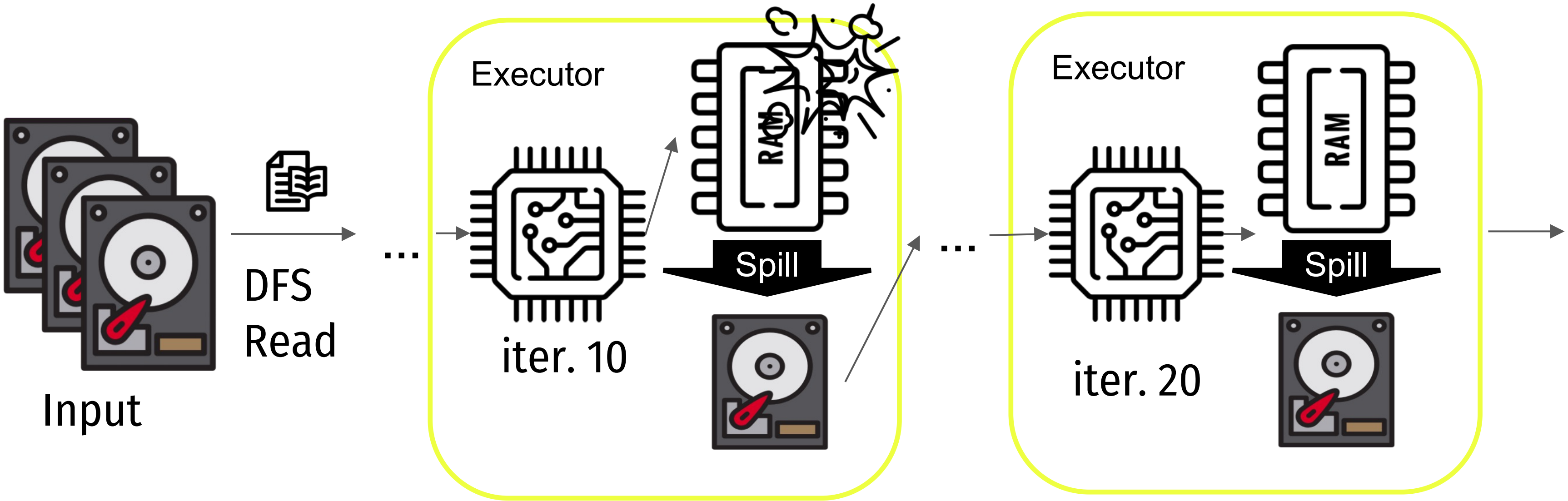
Recomputation overhead is nontrivial

Why is this a problem in real-world environments?



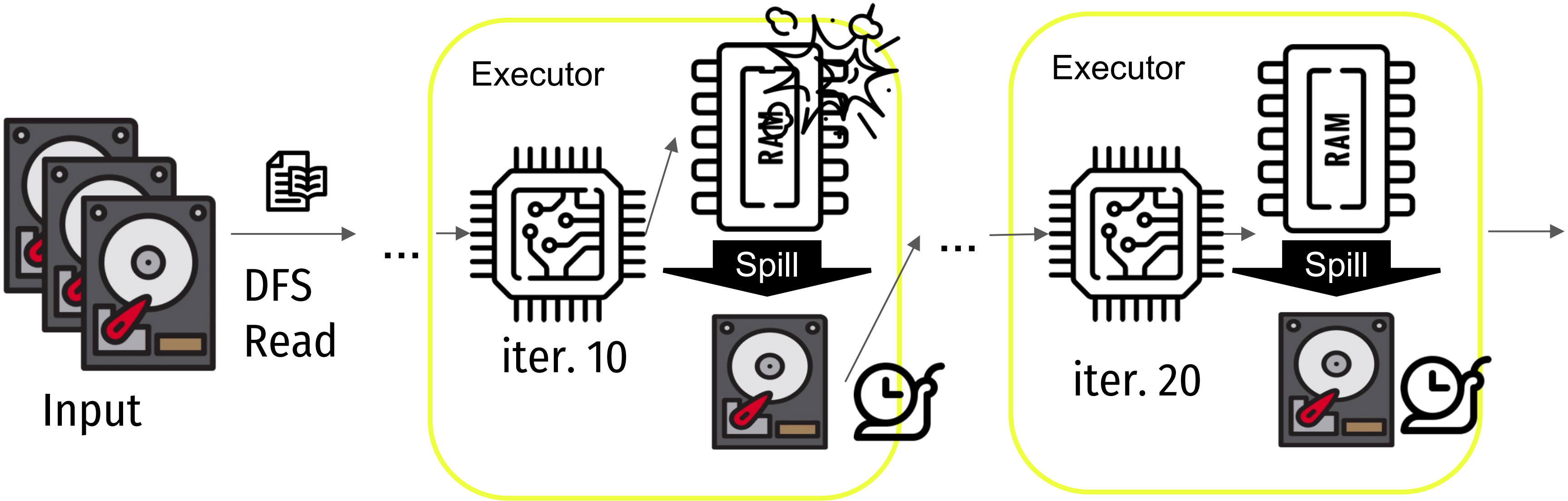
Recomputation costs increase exponentially across iterations upon cache recovery

Using Disks as Secondary Caching Storages



Data can be evicted from memory and spilled on disk, if the memory is full

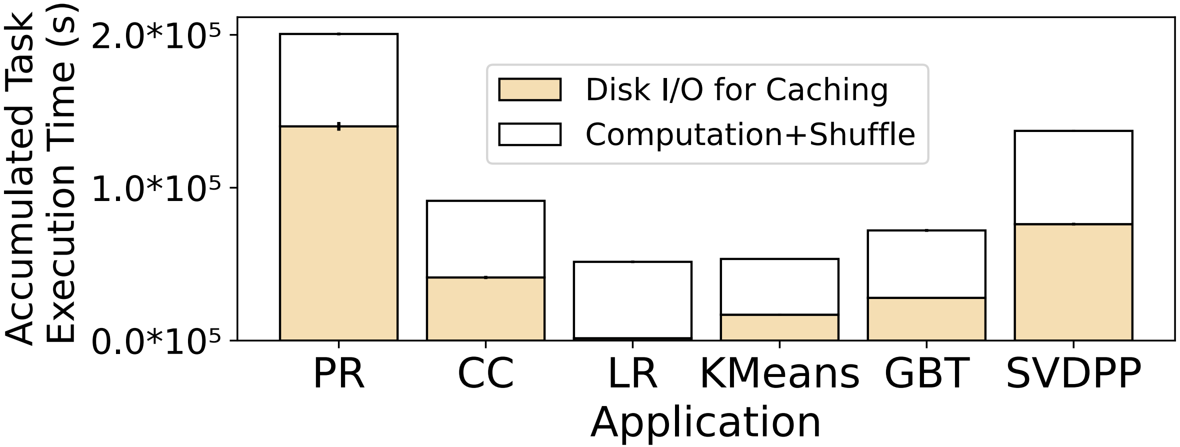
Goal: Reducing Disk I/O Overhead



But incurs serialization/deserialization overheads and disk read/write overheads,

which can be be larger than recomputation overheads

Recomputation vs. Disk I/O Overhead

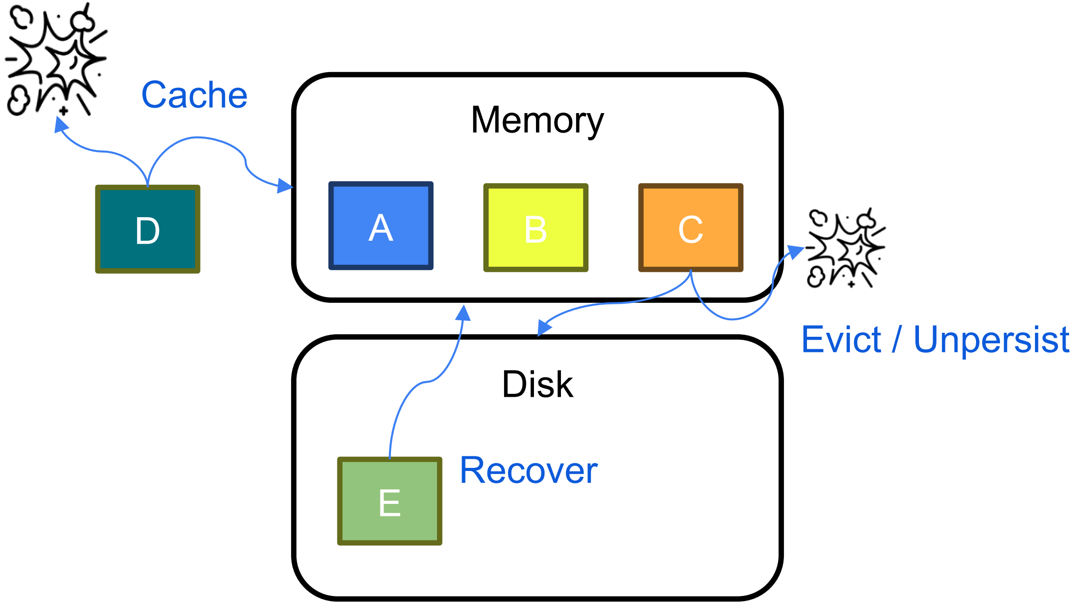


Recomputation overheads increase in **latter** iterations,

Disk I/O overheads increase with **larger** intermediate data,

→ Recomputation cost and disk I/O costs ***differ*** according to application

Separate Caching/Eviction/Recovery Layers



Caching in existing systems occur in three separate operational layers:

1) **caching** layer, 2) **eviction** layer, 3) **recovery** layer

Our Goal

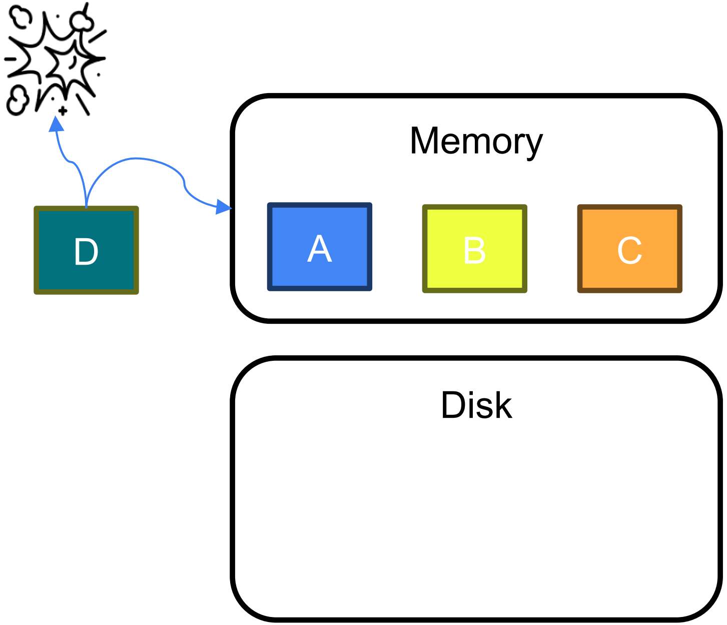
Existing Works

- **Separated** caching mechanism
- **Greedy** cache management
- Based on **pre-defined rules**
- Heuristics based on **past usages**
- **User annotation**-based caching in **dataset** level

Blaze

- **Unified** caching mechanism
- Derive a **blueprint** of the optimal cache state **in advance**
- Based on **dynamically** measured **metrics**
- **Cost predictions** based on **current** and **future** usages of **cached data partitions**
- **Automatic caching** in **partition** level within the data processing system

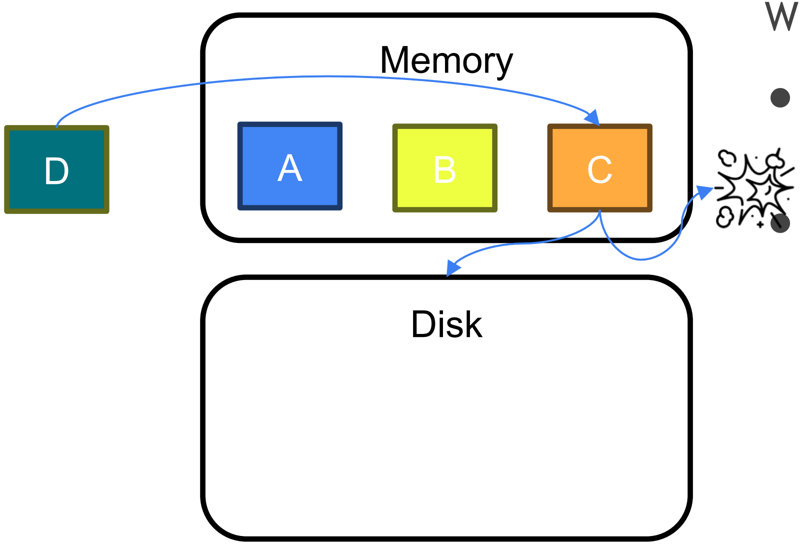
To Cache, or Not To Cache?



Whether or not to cache D if memory is full?

- Default action: Spark **always caches D** according to **user annotation**
  → May lead to **unnecessary** caching
- Proposed action: if D incurs smaller overhead compared to A, B, C, we **shouldn't cache D** in the first place

To Evict, or Not To Evict?

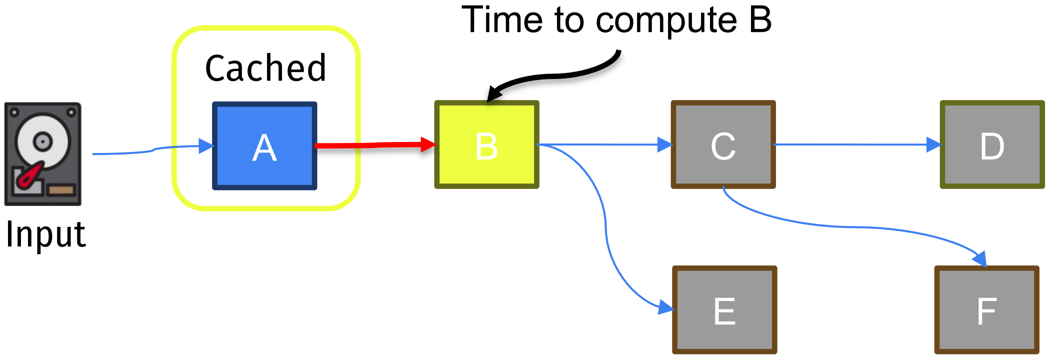


Which partition to evict if memory is full?

- Default action: Spark always **evicts partitions** based on the **eviction policy** (e.g., LRU)
- Proposed action: evict the partition with the **smallest potential cost** = $\min(\text{recomputation cost}, \text{disk cost})$
  - Each partition has different size and potential cost
  - Example:
    - Recomp(A): 3s, Disk(A): 5s, Size(A): 500MB
    - Recomp(B): 5s, Disk(B): 2s, Size(B): 200MB
    - **Recomp(C): 1s**, Disk(C): 5s, Size(C): 500MB

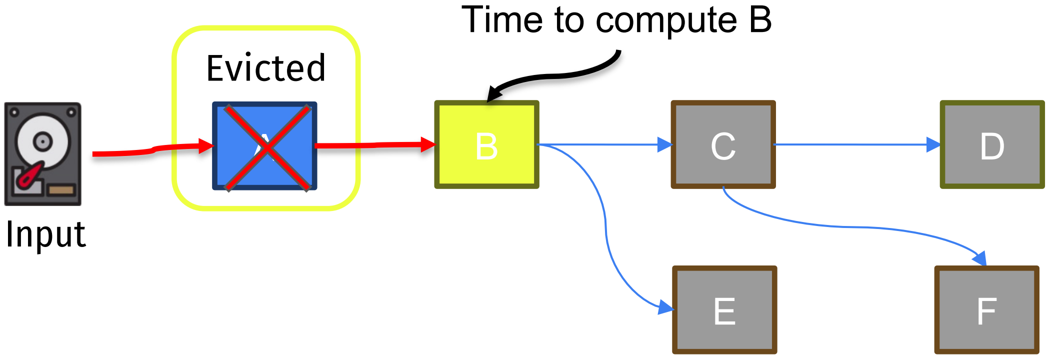
      → Evict C!

Data Dependency Changes Dynamically and Unpredictably



Recomputation cost for partition B = Computation cost [A → B]

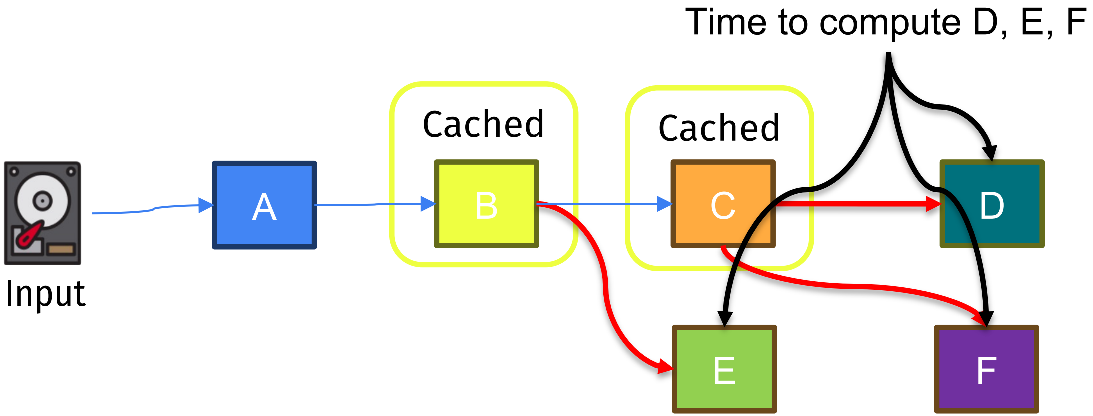
Data Dependency Changes Dynamically and Unpredictably



Recomputation cost for partition B = Computation cost [Input → A → B]

→ Recomputation cost/length **varies** on the cached partitions

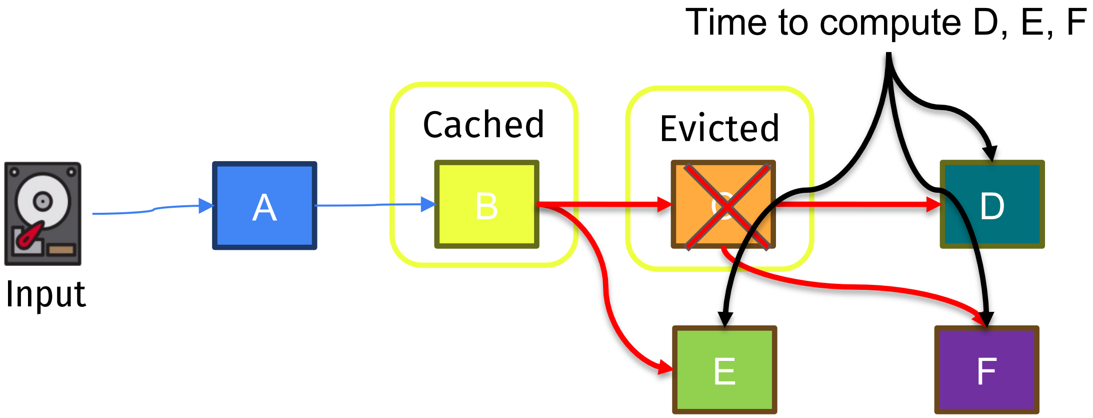
Data Dependency Changes Dynamically and Unpredictably



Cached partition B is referenced once, by E

Cached partition C is referenced twice, by D and F

Data Dependency Changes Dynamically and Unpredictably



Cached partition B is referenced 3 times, by D, F (through C), and E

→ # of cache references varies on the cached data

Blaze Design Principles

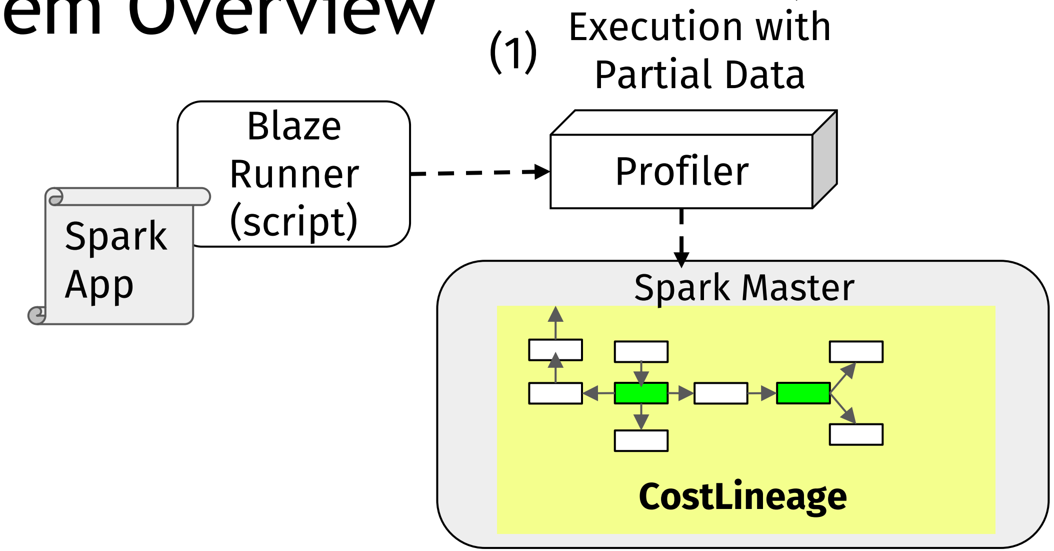
1. Workload **DAG profiling** and **dynamic** metric measurements to induce **potential costs** and references of each partitions

2. Incorporating **potential** data caching efficiency into a **unified cost optimization function**

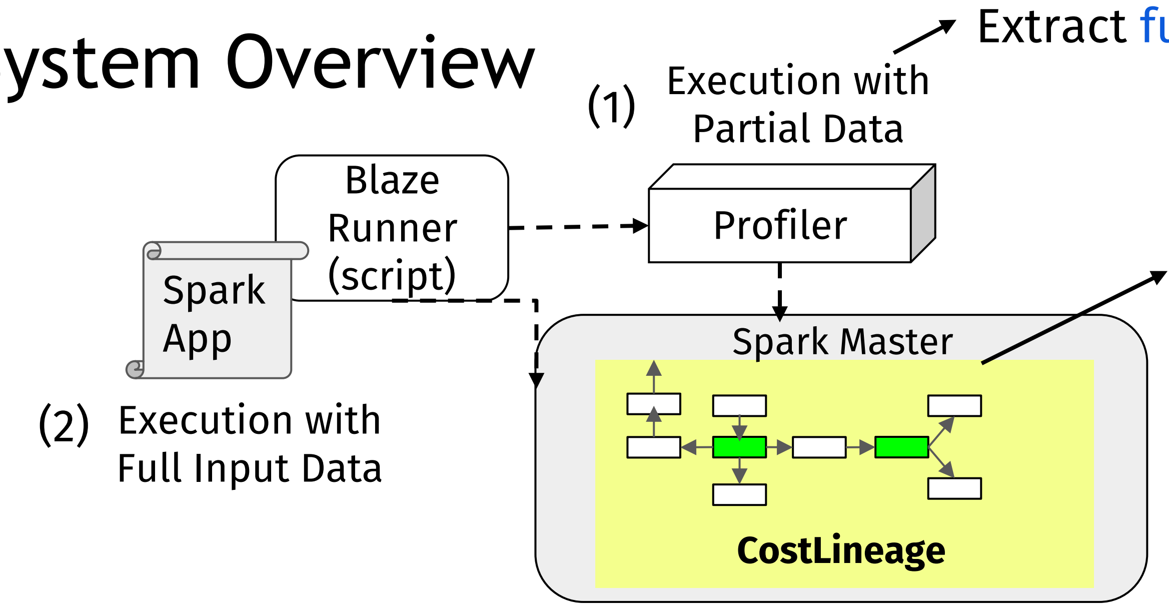
3. **Automatic caching decisions** based on fast ILPs

System Overview

Extract future dependencies



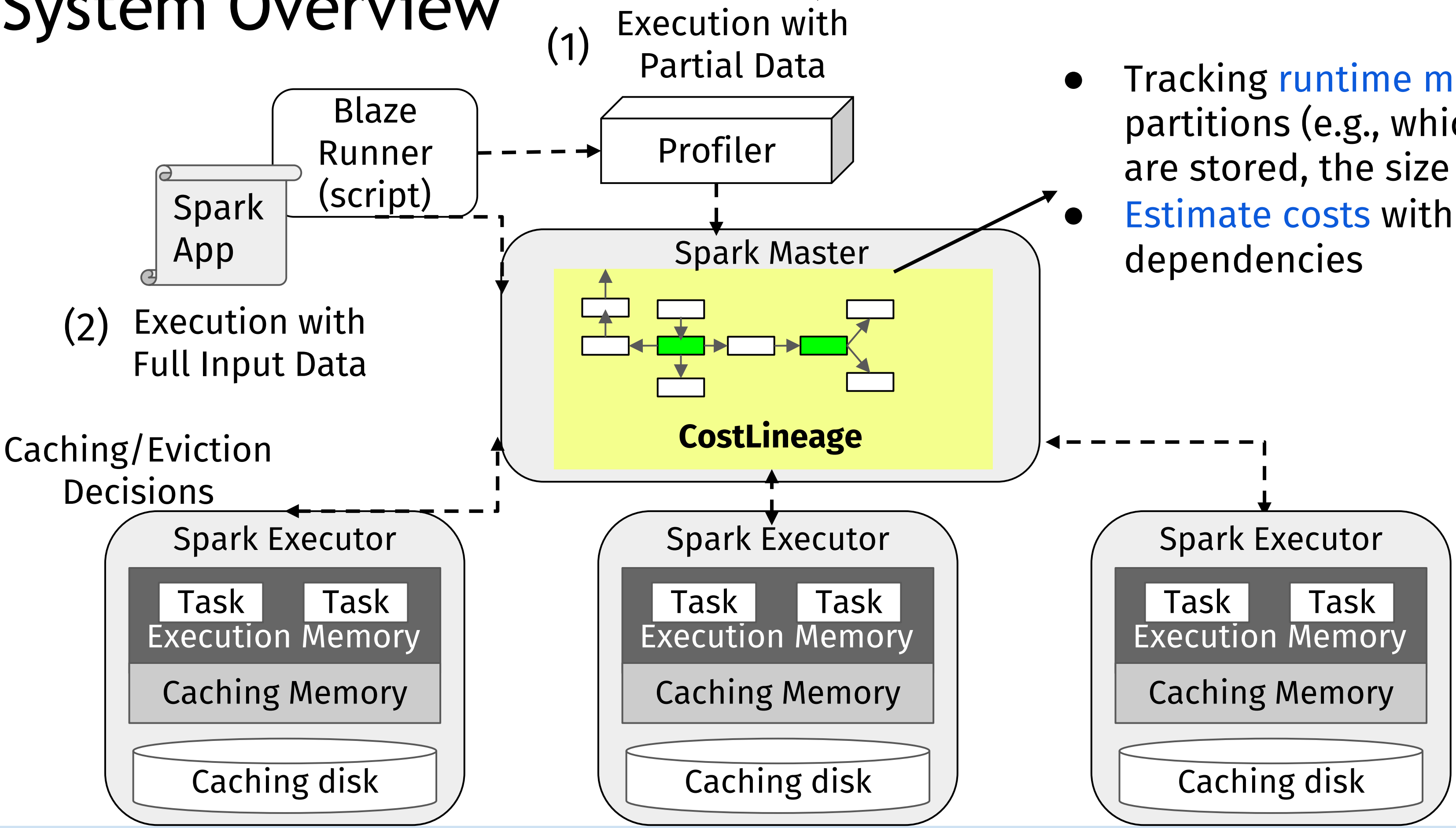
System Overview


(1) Execution with Partial Data
Blaze Runner (script)
Profiler
Spark App
(2) Execution with Full Input Data
Spark Master
CostLineage

Extract future dependencies

- Tracking runtime metrics of partitions (e.g., which data are stored, the size of data)
- Estimate costs with data dependencies

System Overview



(1) Execution with Partial Data

Extract future dependencies

Blaze Runner (script)

Spark App

Profiler

Spark Master

CostLineage

- Tracking runtime metrics of partitions (e.g., which data are stored, the size of data)
- Estimate costs with data dependencies

(2) Execution with Full Input Data

Caching/Eviction Decisions

Spark Executor

Task

Task

Execution Memory

Caching Memory

Caching disk

Spark Executor

Task

Task

Execution Memory

Caching Memory

Caching disk

Spark Executor

Task

Task

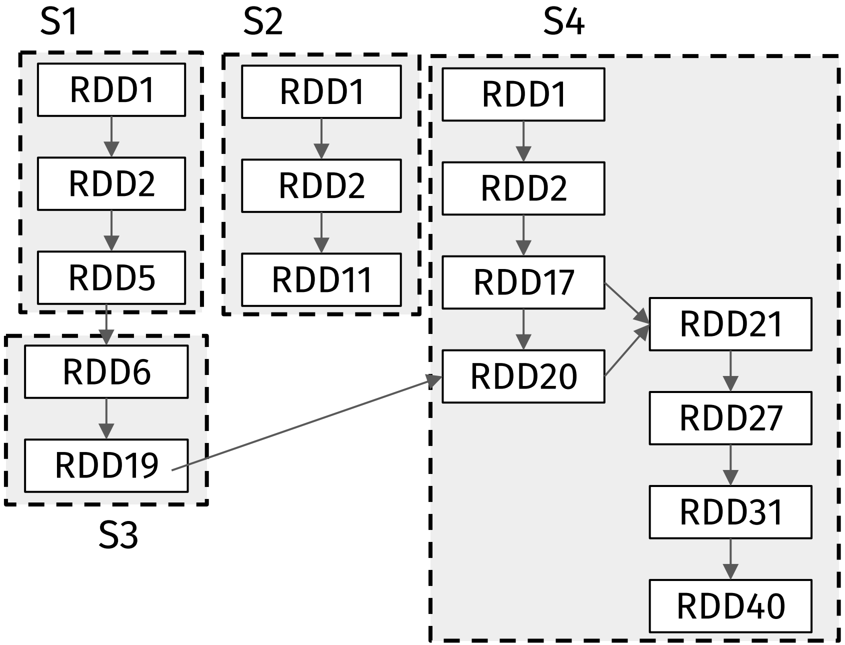
Execution Memory

Caching Memory

Caching disk

Collecting Potential Future References

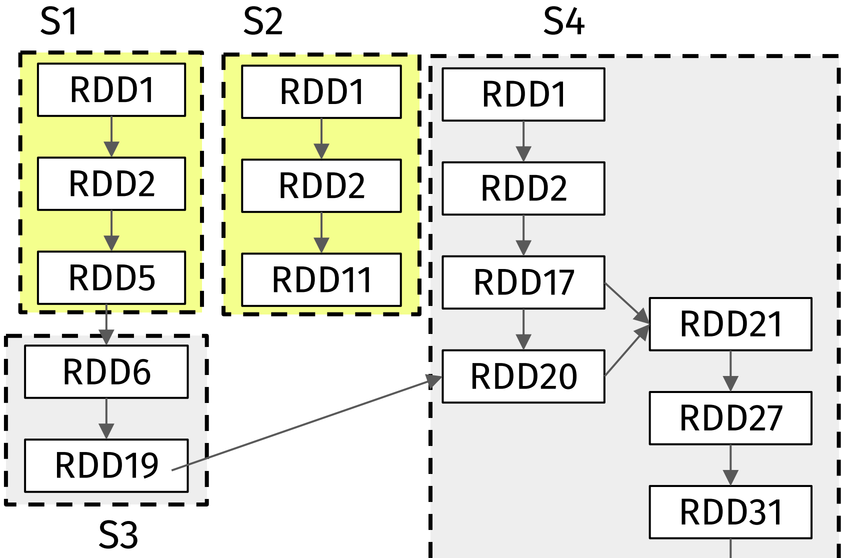
Submitted jobs (profiling phase)



Cost Lineage

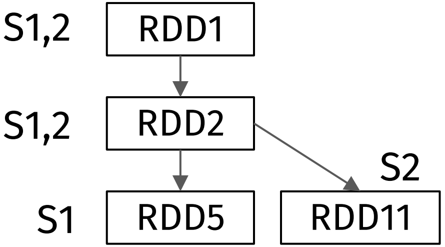
Collecting Potential Future References

Submitted jobs (profiling phase)



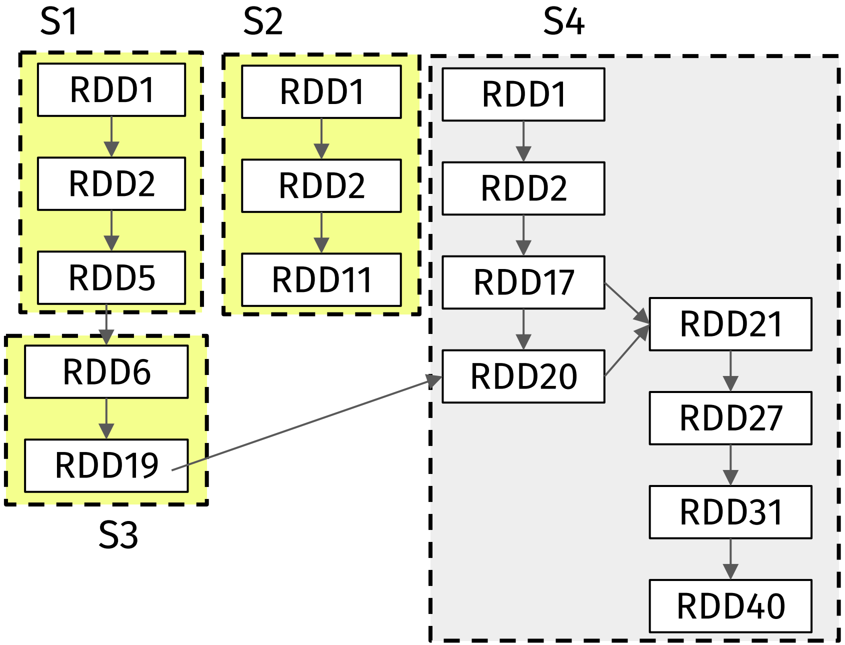
Cost Lineage

1. Keep stage information



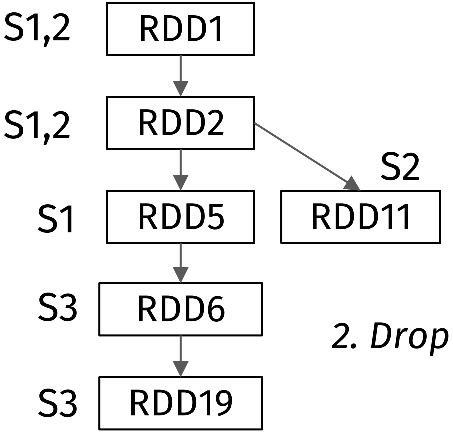
Collecting Potential Future References

Submitted jobs (profiling phase)



Cost Lineage

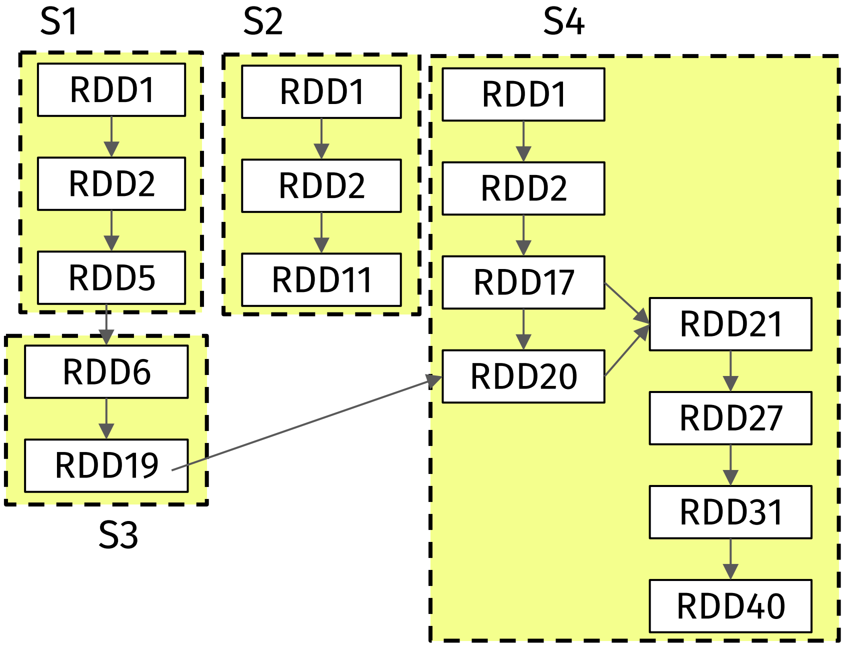
1. Keep stage information



2. Drop duplicate RDDs

Collecting Potential Future References

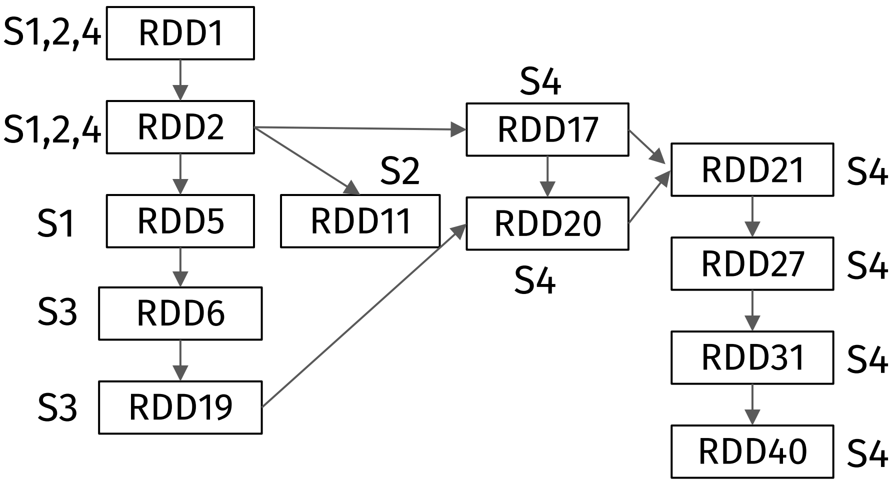
Submitted jobs (profiling phase)



Cost Lineage

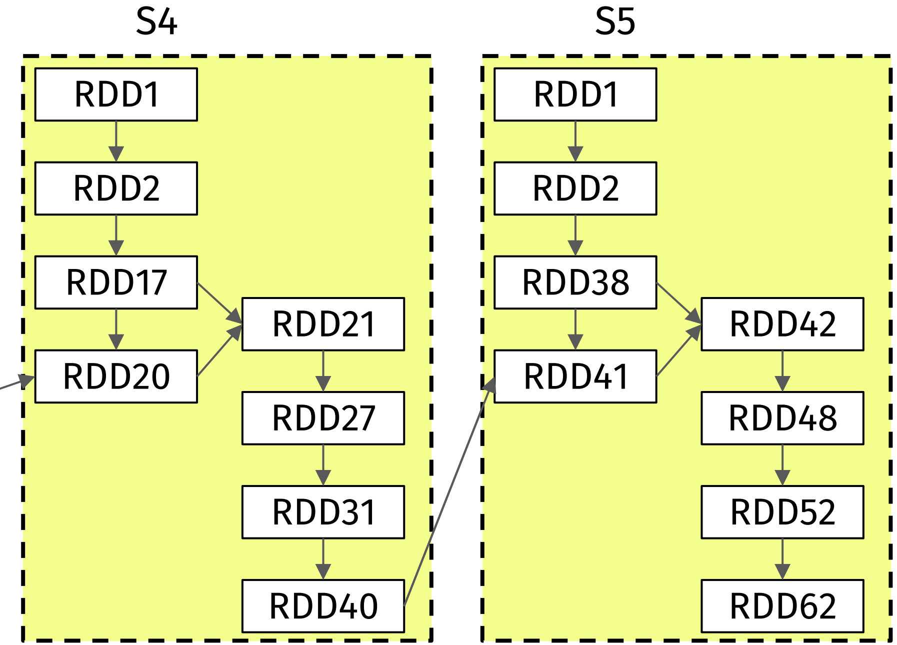
1. Keep stage information

2. Drop duplicate RDDs



Collecting Potential Future References

Submitted jobs (profiling phase)

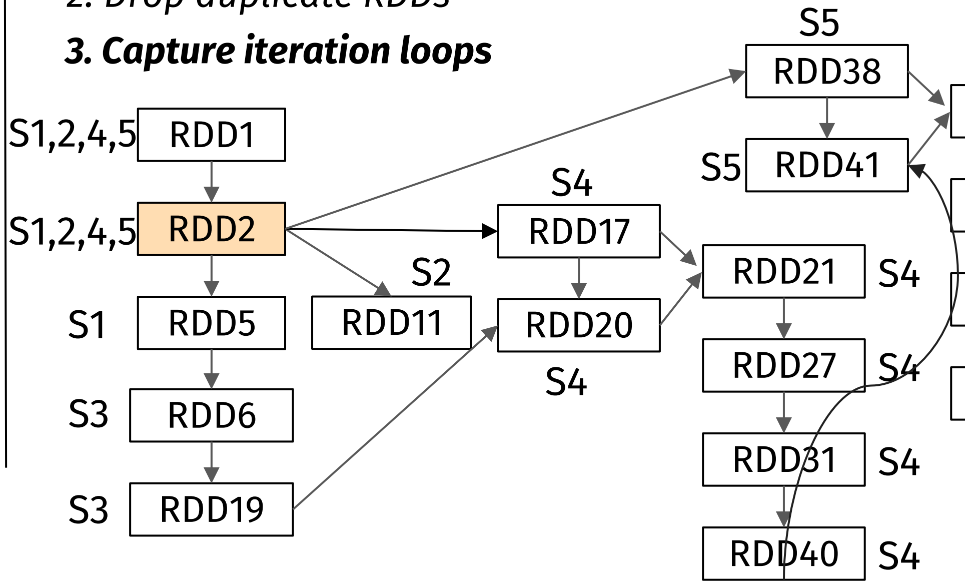


Cost Lineage

1. Keep stage information

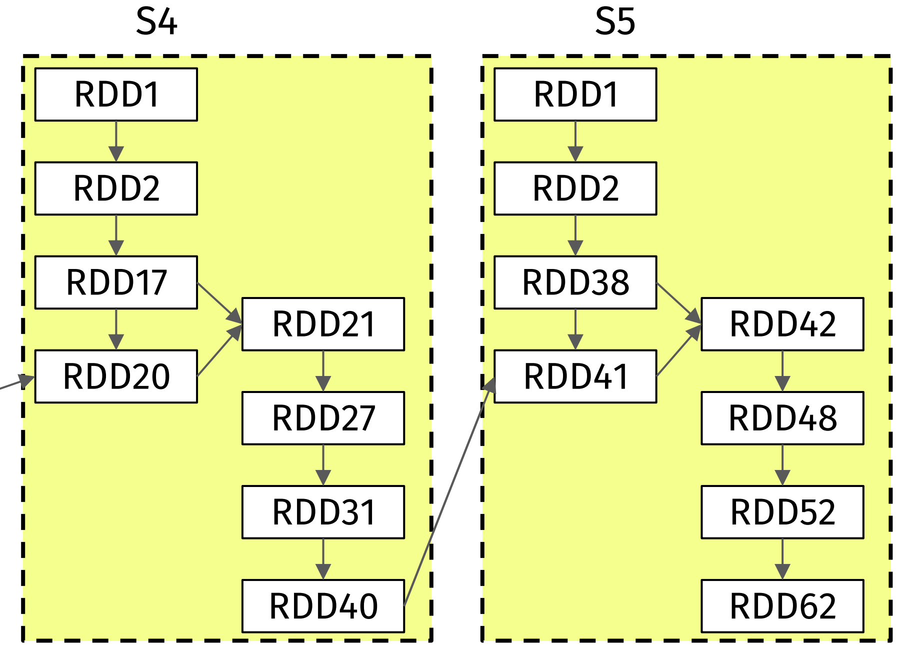
2. Drop duplicate RDDs

3. Capture iteration loops



Collecting Potential Future References

Submitted jobs



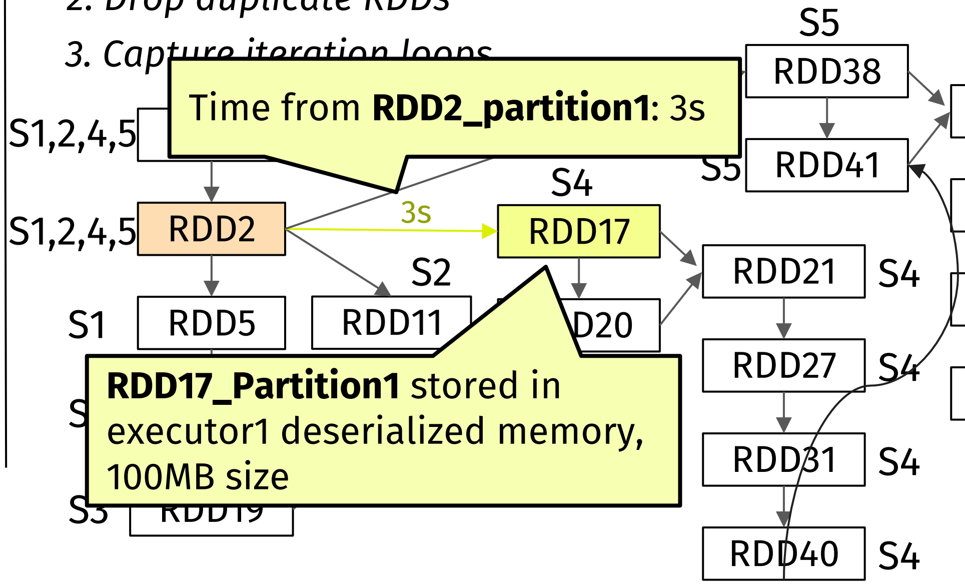
Cost Lineage (initial stages)

1. Keep stage information

2. Drop duplicate RDDs

3. Capture iteration loops

4. Keep partition metrics



Time from **RDD2_partition1**: 3s

RDD17_Partition1 stored in executor1 deserialized memory, 100MB size

Collecting Potential Future References

Cost Lineage

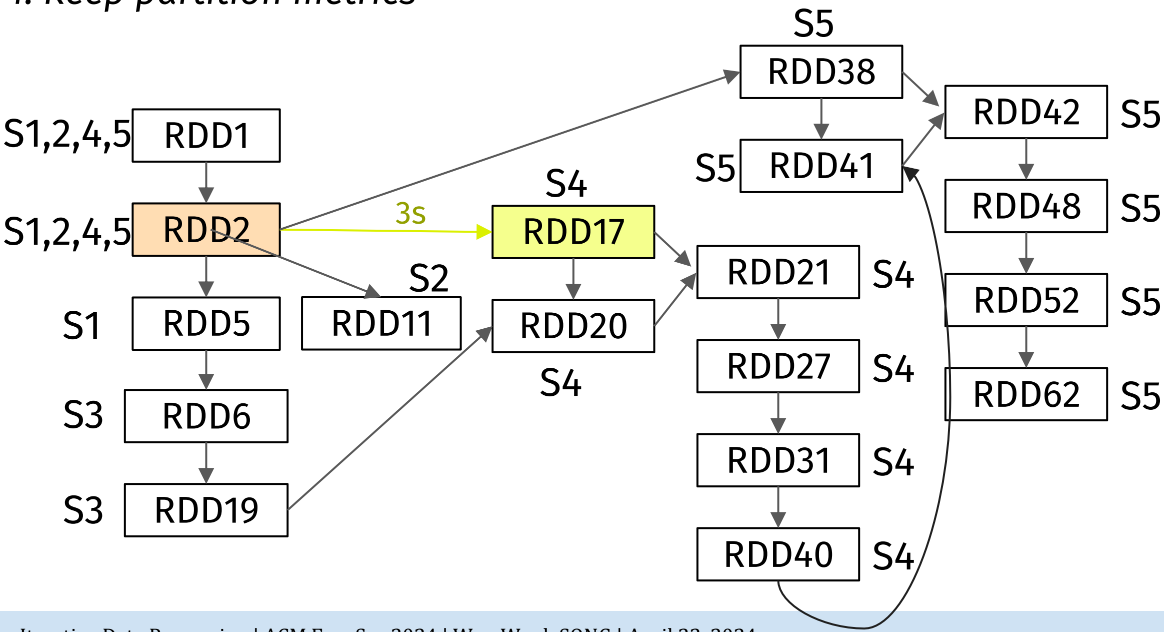
1. Keep stage information

2. Drop duplicate RDDs

3. Capture iteration loops

4. Keep partition metrics

5. Perform induction on future iterations



Collecting Potential Future References

Cost Lineage

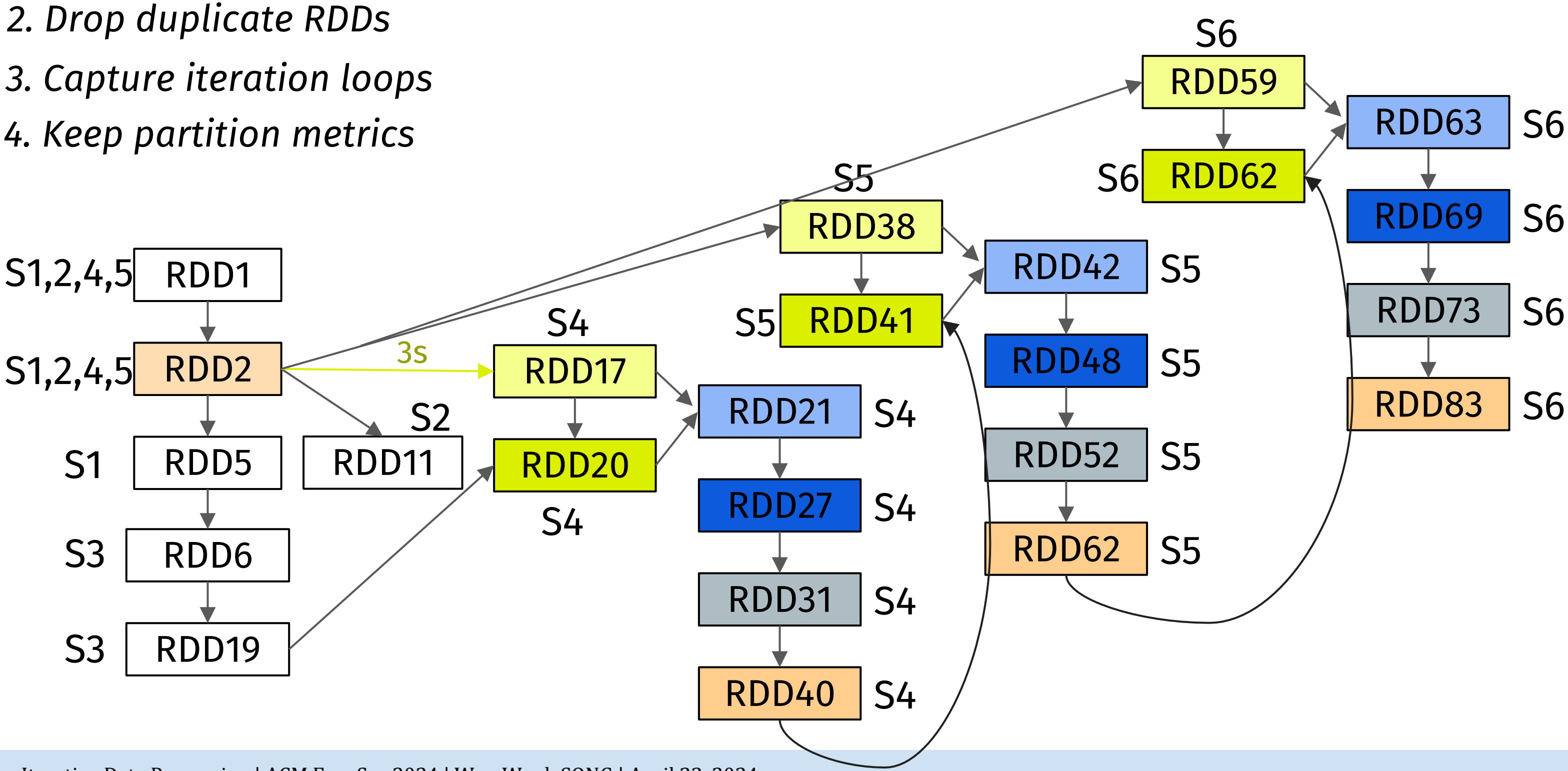
1. Keep stage information

2. Drop duplicate RDDs

3. Capture iteration loops

4. Keep partition metrics

5. Perform induction on future iterations

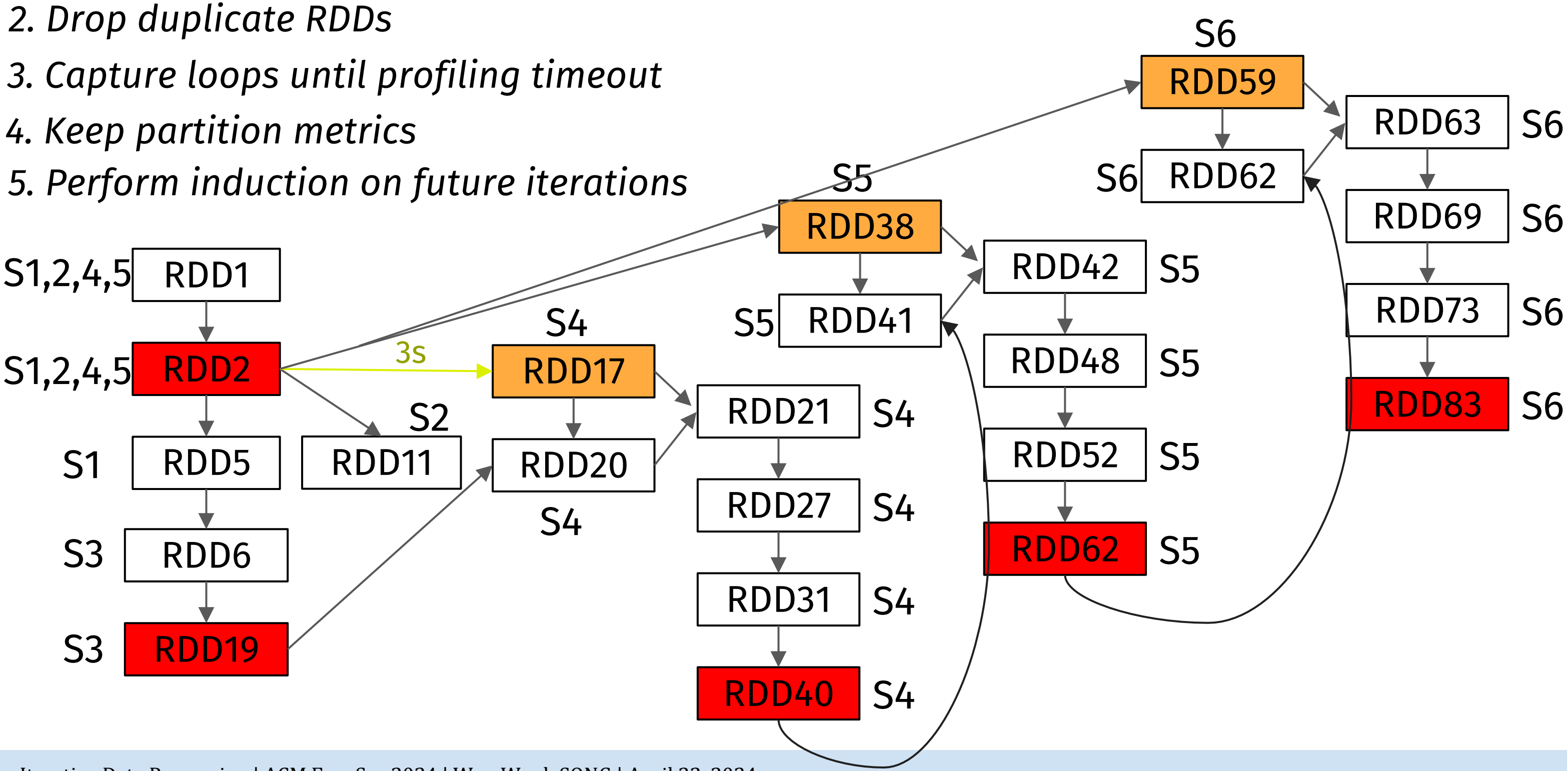


Collecting Potential Future References

Cost Lineage

1. *Keep stage information*
2. *Drop duplicate RDDs*
3. *Capture loops until profiling timeout*
4. *Keep partition metrics*
5. *Perform induction on future iterations*

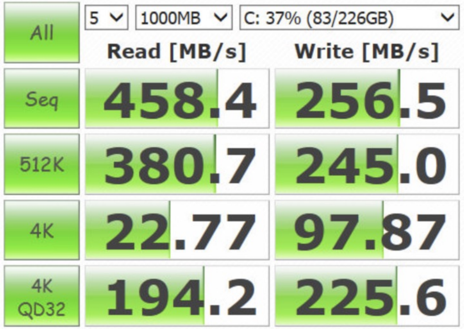
6. Automatic caching decisions (Triggered after each stage execution)



Potential Cost Estimation

Disk Cost

$$cost_{i_d} = \frac{size_i}{throughput_{disk}}$$

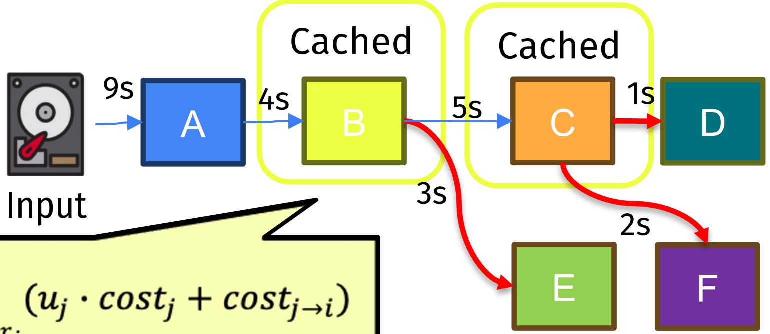


5 | 1000MB | C: 37% (83/226GB)

All	Read [MB/s]	Write [MB/s]
Seq	458.4	256.5
512K	380.7	245.0
4K	22.77	97.87
4K QD32	194.2	225.6

<span style="color:blue">Linearly increases</span> with the data size
(Depends on disk performance, like RPM)

Recomputation Cost



$$cost_{i_r} = \max_{j \in P_{ancestor_i}} (u_j \cdot cost_j + cost_{j \to i})$$

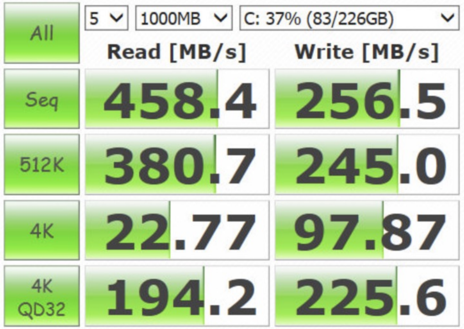
Depends on future data dependencies

Depends on cached ancestor data,
which varies at runtime

Potential Cost Estimation

Disk Cost

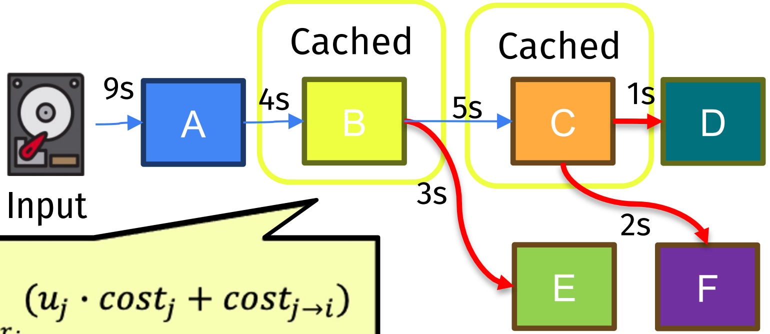
$$cost_{i_d} = \frac{size_i}{throughput_{disk}}$$



All	5	1000MB	C: 37% (83/226GB)
	Read [MB/s]	Write [MB/s]	
Seq	458.4	256.5	
512K	380.7	245.0	
4K	22.77	97.87	
4K QD32	194.2	225.6	

Linearly increases with the data size
(Depends on disk performance, like RPM)

Recomputation Cost



$$cost_{i_r} = \max_{j \in P_{ancestor_i}} (u_j \cdot cost_j + cost_{j \to i})$$

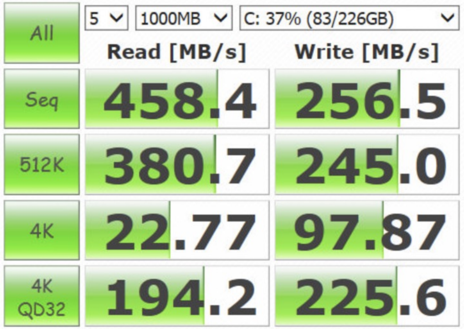
Depends on future data dependencies

Depends on cached ancestor data,
which varies at runtime

Potential Cost Estimation

Disk Cost

$$cost_{i_d} = \frac{size_i}{throughput_{disk}}$$

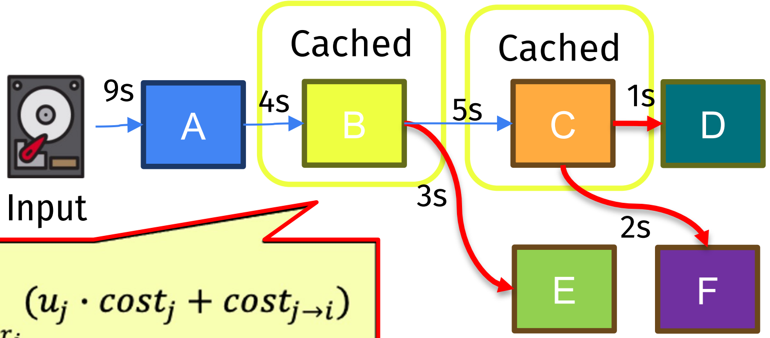


5 | 1000MB | C: 37% (83/226GB)

All	Read [MB/s]	Write [MB/s]
Seq	458.4	256.5
512K	380.7	245.0
4K	22.77	97.87
4K QD32	194.2	225.6

Linearly increases with the data size
(Depends on disk performance, like RPM)

Recomputation Cost

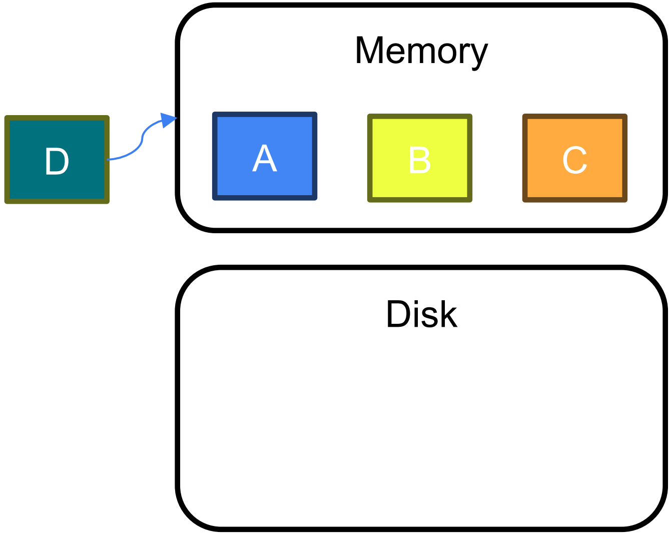


$$cost_{i_r} = \max_{j \in P_{ancestor_i}} (u_j \cdot cost_j + cost_{j \to i})$$

Depends on future data dependencies

Depends on cached ancestor data,
which varies at runtime

Decision Making Algorithm (Trigger)



Algorithm triggered after each iteration when:

1) size of new data to be cached exceeds free memory

$$size_{new} > free\_capacity_{mem}$$

AND

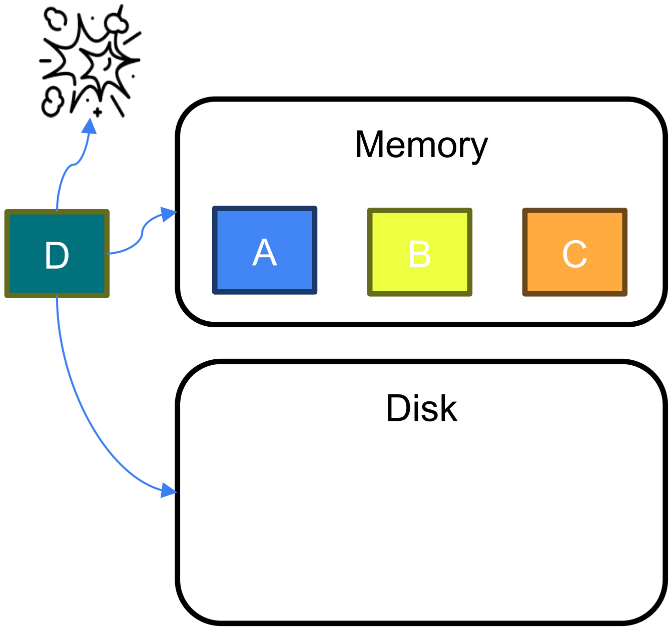
2) expected potential recovery overhead of new data to be prevented (min of recomputation cost and disk cost)

$$cost_{new} = \min(cost_{new_r},\ cost_{new_d})$$

is larger than the potential overhead of any data in memory:

$$cost_{new} > \min_{d \in D_{mem}} cost_d$$

Decision Making Algorithm (ILP)



ILP Solver

Action space: mem-cached / disk-cached / un-cached state, for each partition

$$\sum_{i\in P} m_i + \sum_{i\in P} d_i + \sum_{i\in P} u_i = |P| \text{ AND } \forall i \in P,\ m_i + d_i + u_i = 1$$

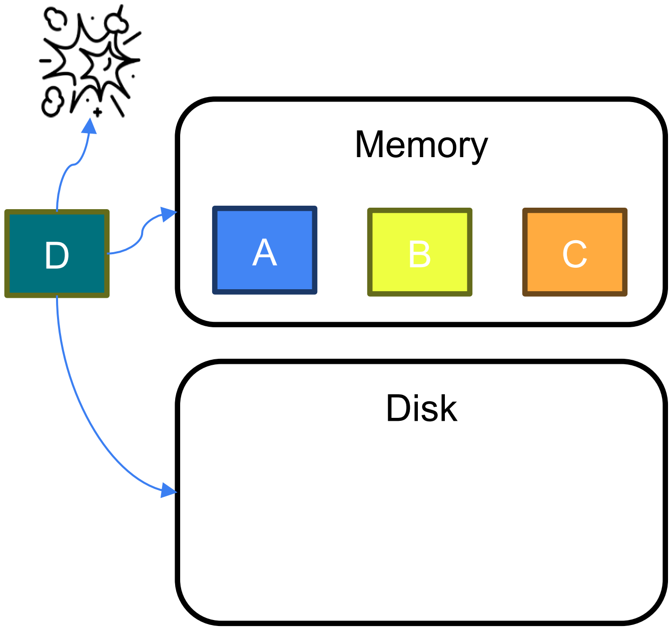
Constraint: memory space does not exceed sum of mem-cached partition data

$$\sum_{i\in P} size_i \cdot m_i \leq capacity_{mem}$$

Objective: minimize the sum of potential recovery overhead and JCT for the RDDs in the current & next job J (upcoming iterations)

$$\operatorname{argmin} \sum_{i\in J} cost_i = \operatorname{argmin} \sum_{i\in J} (u_i \cdot cost_{i_r} + d_i \cdot cost_{i_d})$$

Decision Making Algorithm



ILP Solver

Action space: mem-cached / disk-cached / un-cached state, for each partition

$$\sum_{i\in P} m_i + \sum_{i\in P} d_i + \sum_{i\in P} u_i = |P| \text{ AND } \forall i \in P,\ m_i + d_i + u_i = 1$$

Constraint: memory space does not exceed sum of mem-cached partition data

$$cost_{i_r} = \max_{j\in P_{ancestor_i}} (u_j \cdot cost_j + cost_{j\to i})$$

y_{mer}

$$cost_{i_d} = \frac{size_i}{throughput_{disk}}$$

Objective: minimize the s tential recovery overhead JCT for the RDDs in the current & ne h J (upcoming iteratio

$$\underset{}{\text{argmin}} \sum_{i\in J} cost_i = \text{argmin} \sum_{i\in J} (u_i \cdot cost_{i_r} + d_i \cdot cost_{i_d})$$

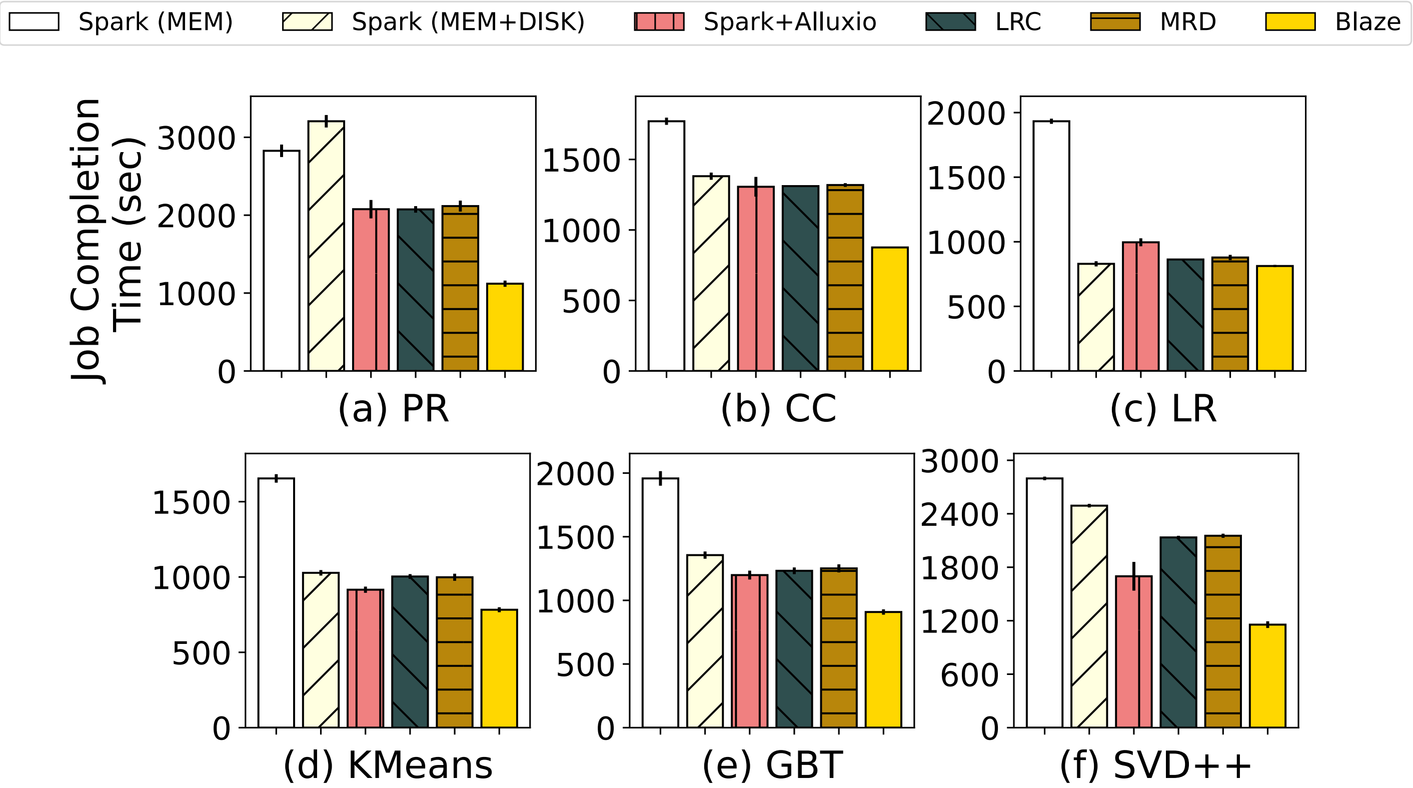
Blaze Implementation

- Data processing runtime
  - Spark 3.3.2, modified and added 6K lines of Scala 2.12 code
- Blaze profiler & cache optimizer
  - ~500 lines of Bash script
- ILP solver
  - Gurobi optimizer 10.0.1

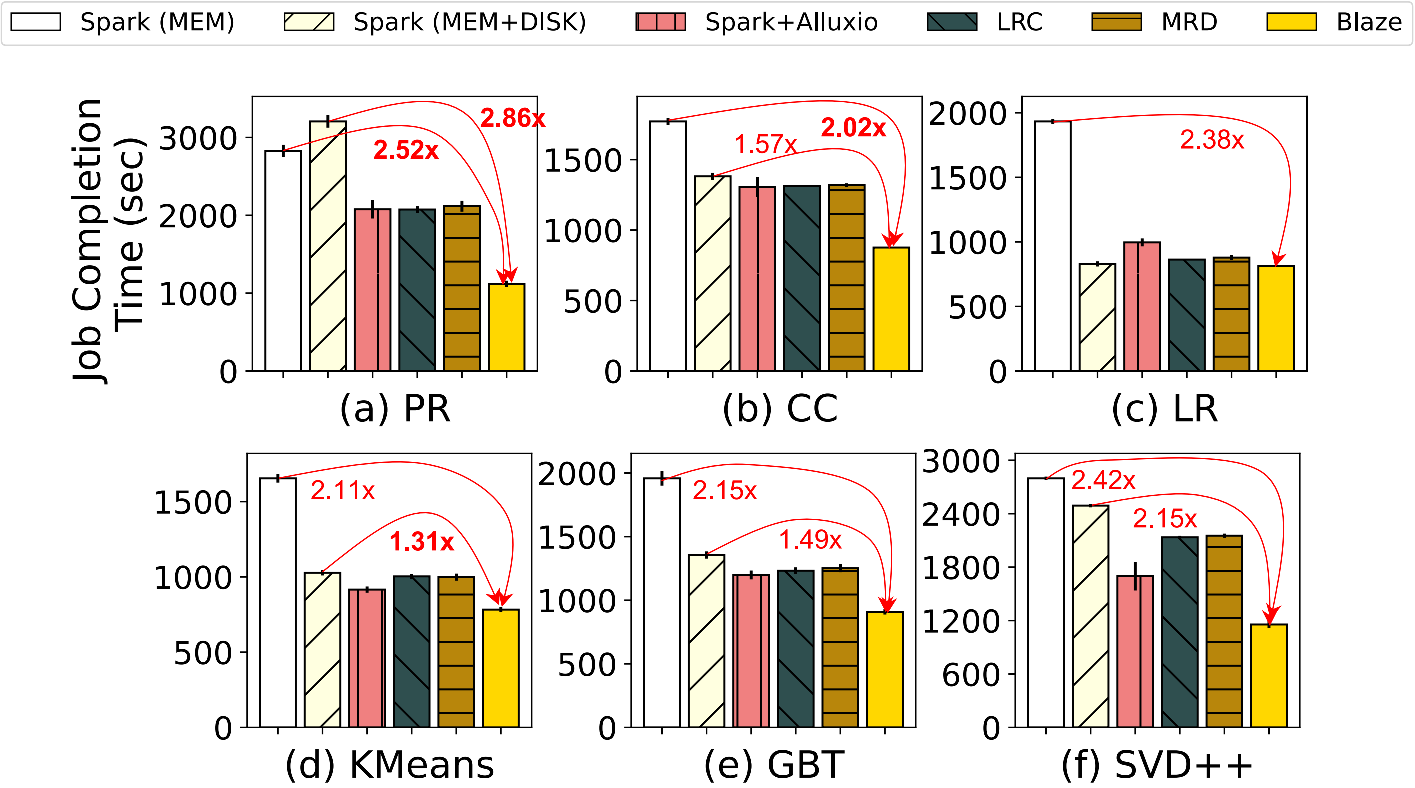
Evaluation Setup

- 11 instances (1 as master, 10 as executors)
  - AWS EC2 r5.2xlarge instances (8vCores, 64GB memory, 10Gbps network, each)
    - → A total of **80** executor vCores, **500GB** executor memory (**170GB** memory used as **cache memory**).
  - **100GB SSD** (gp2) on each instance for caching stores (disks) → A total of **1TB SSD** for a **disk** caching store
- Workloads
  - **PageRank (PR)**
  - **Connected Components (CC)**
  - **Logistic Regression (LR)**
  - **K-Means Clustering (KMeans)**
  - **Gradient Boosted Trees (GBT)**
  - **Singular Vector Decomposition++ (SVD++)**

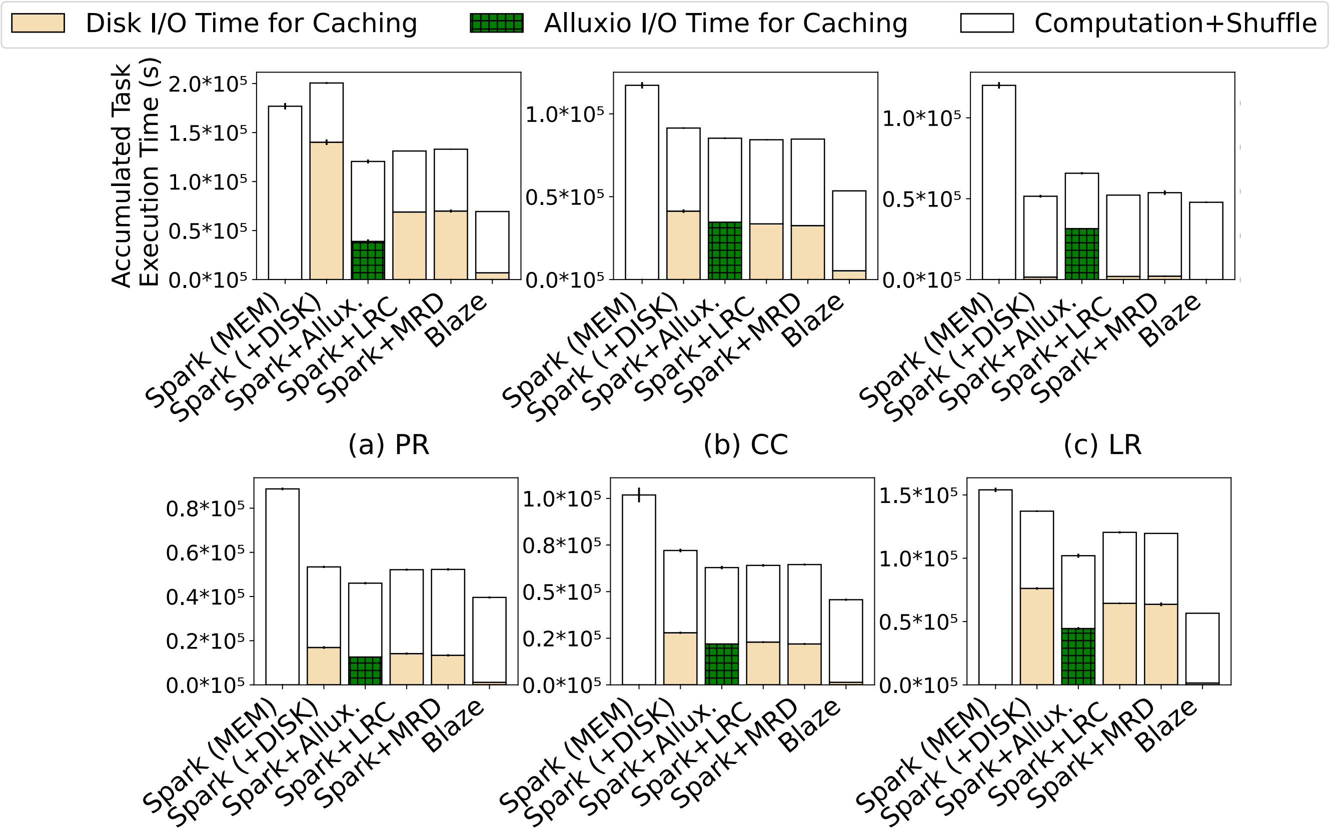
End-to-End Performance Comparison



End-to-End Performance Comparison



Performance Analysis: Computation+Shuffle vs. Disk I/O



(a) PR

(b) CC

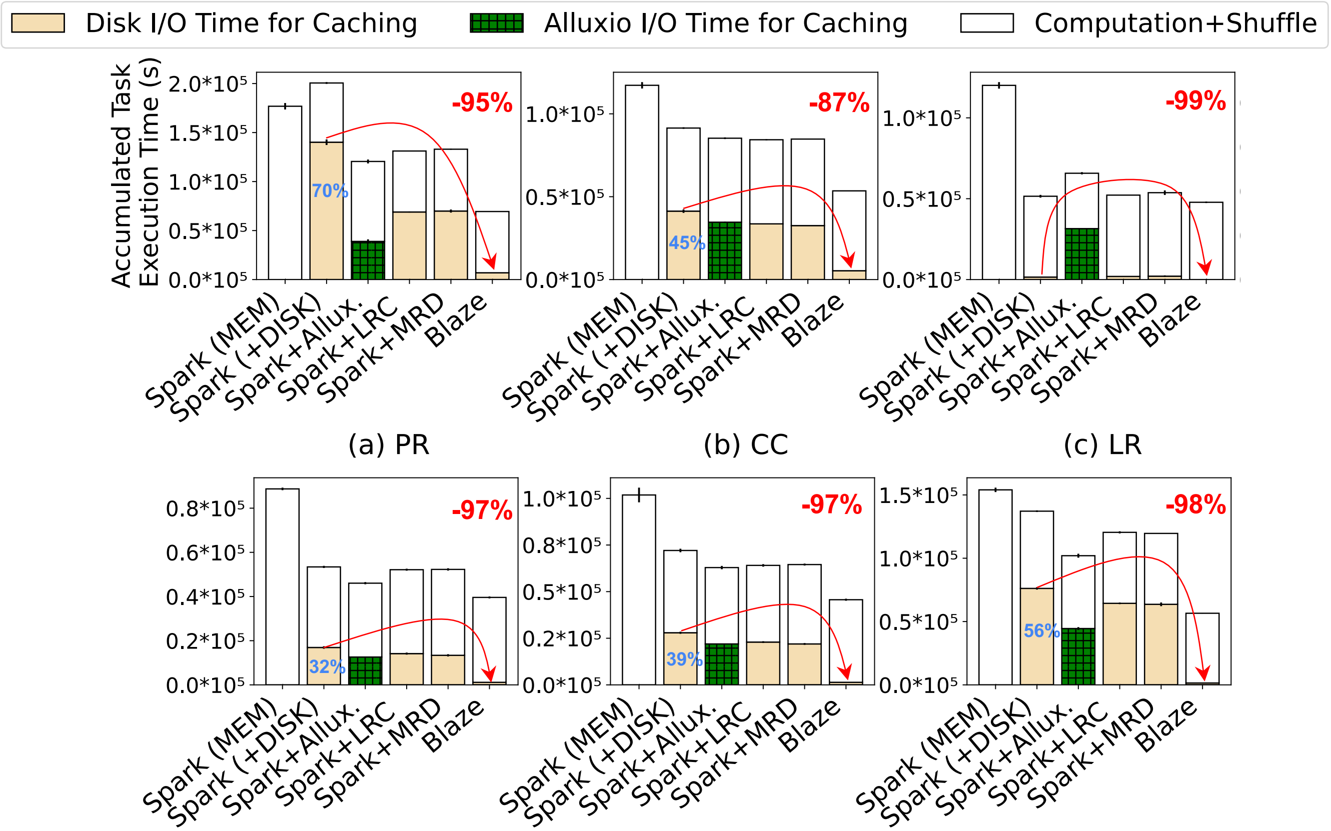
(c) LR

(d) KMeans

(e) GBT

(f) SVD++

Performance Analysis: Computation+Shuffle vs. Disk I/O



(a) PR

(b) CC

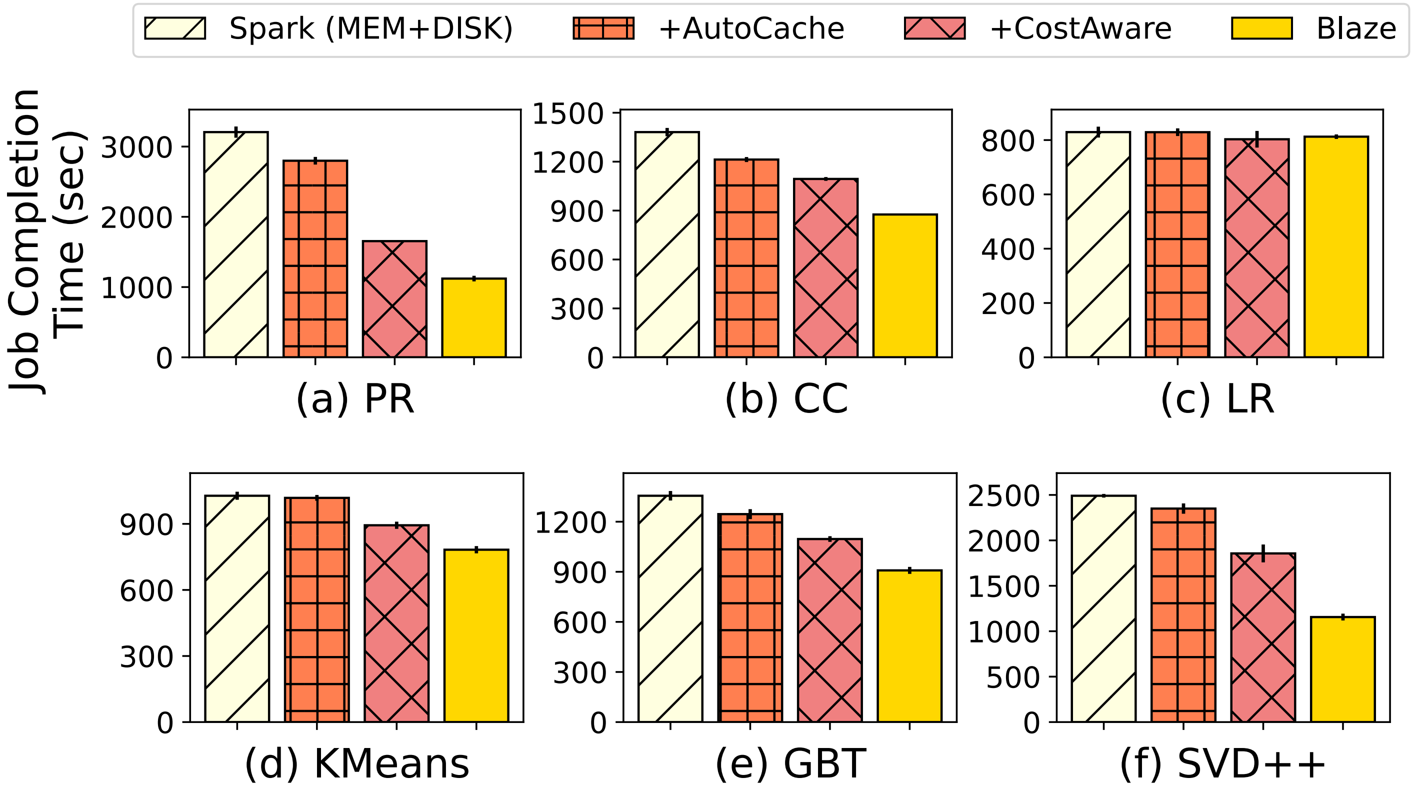
(c) LR

(d) KMeans

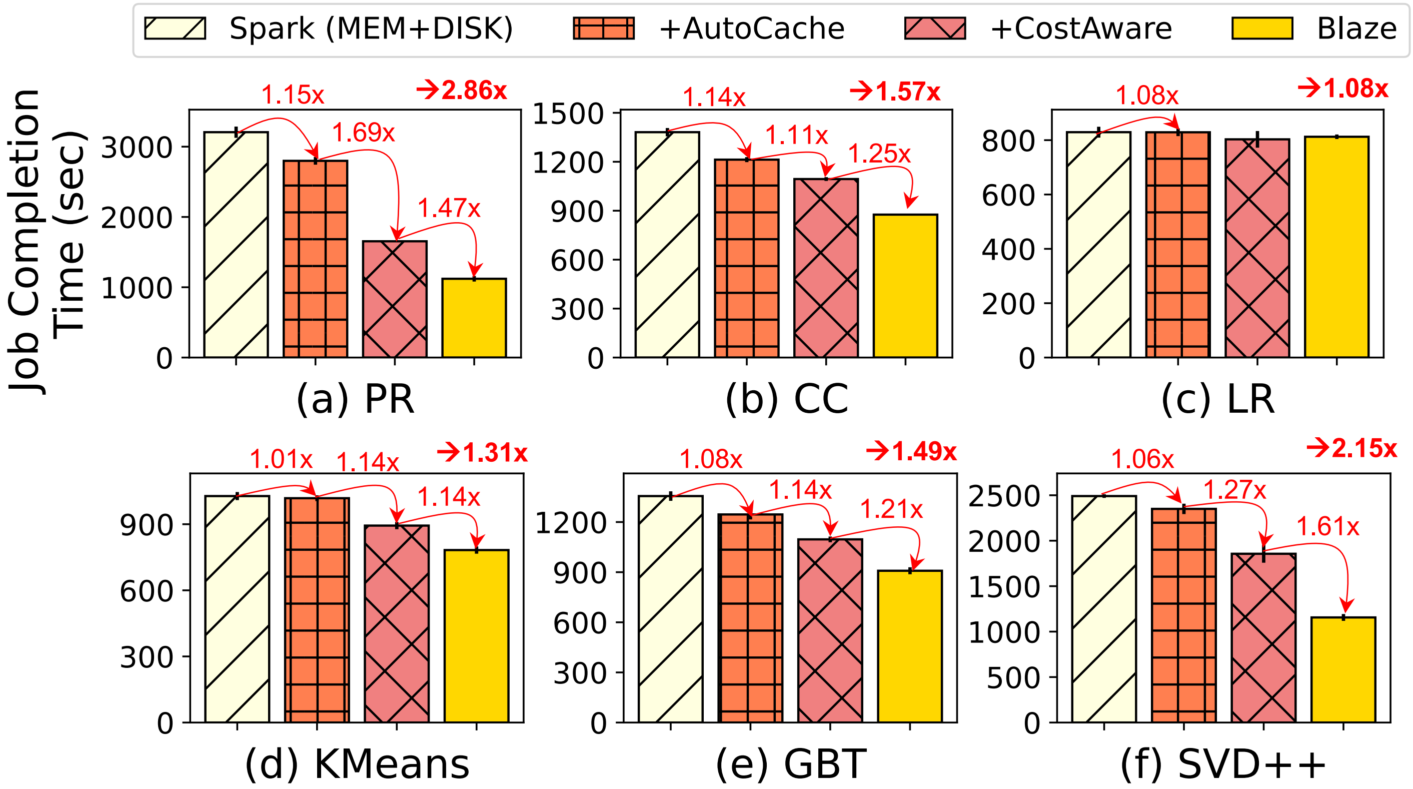
(e) GBT

(f) SVD++

Blaze: Performance Breakdown



Blaze: Performance Breakdown



Blaze Summary

- Caching is crucial in reducing the recomputation costs for iterative data processing workloads (e.g., graph processing, ML)
- Existing separated, greedy mechanisms lead to unnecessary caching and inefficient use of memory space
- Recomputation and disk overheads + potential references have to be tracked dynamically
- Based on the tracked information, Blaze makes sophisticated decisions on which data to cache and to evict with an ILP solver
- Blaze achieves up to 2.86x speedup on end-to-end performance and optimizes cache data by 95% on average

Thank you!

Blaze:

Holistic Caching for Iterative Data Processing

Contact me for further questions

wonook@apache.org

https://wonook.github.io