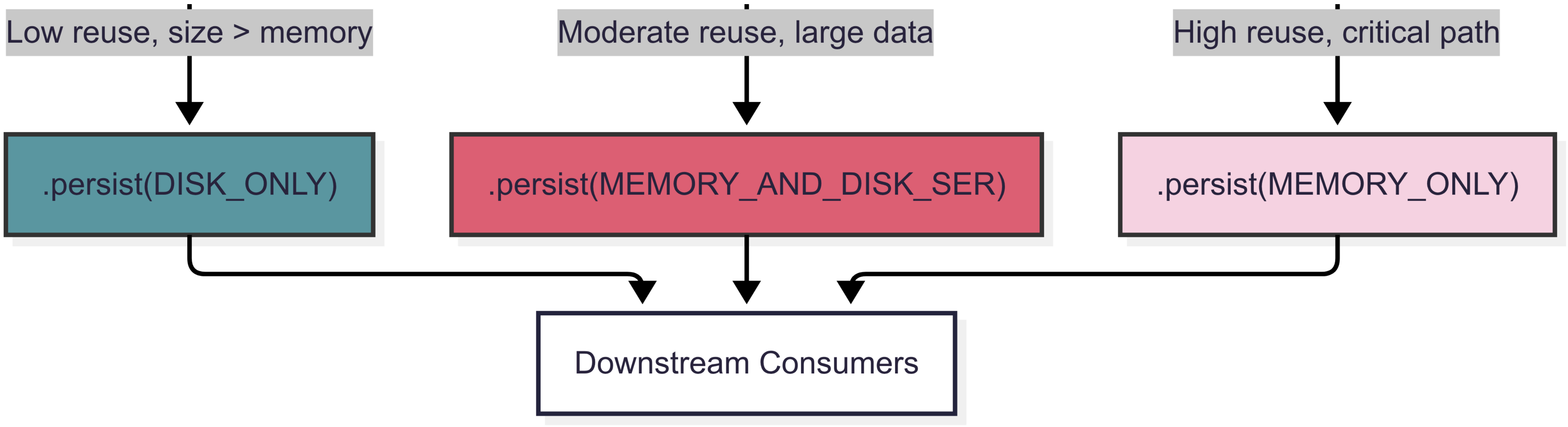
Performance Optimization in Apache Spark

Technical guide to .persist()

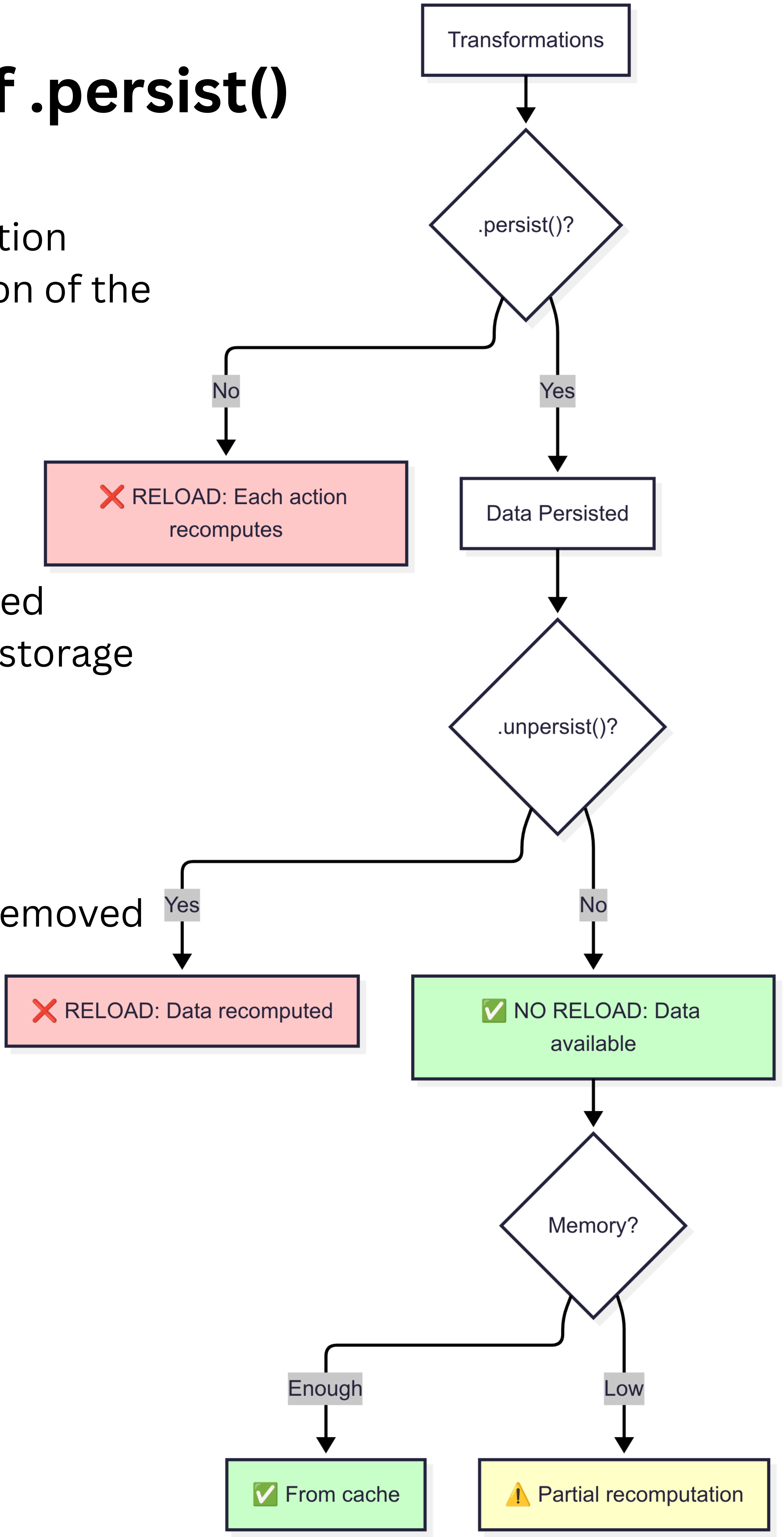


Understanding Load Behavior of .persist()

Without .persist(): Each action triggers a full recomputation of the DataFrame.

With .persist(): Data is stored according to the specified storage level.

After .unpersist(): Data is removed from storage and must be recomputed



Transformations

.persist()?

No

❌ RELOAD: Each action recomputes

Yes

Data Persisted

.unpersist()?

Yes

❌ RELOAD: Data recomputed

No

✅ NO RELOAD: Data available

Memory?

Enough

✅ From cache

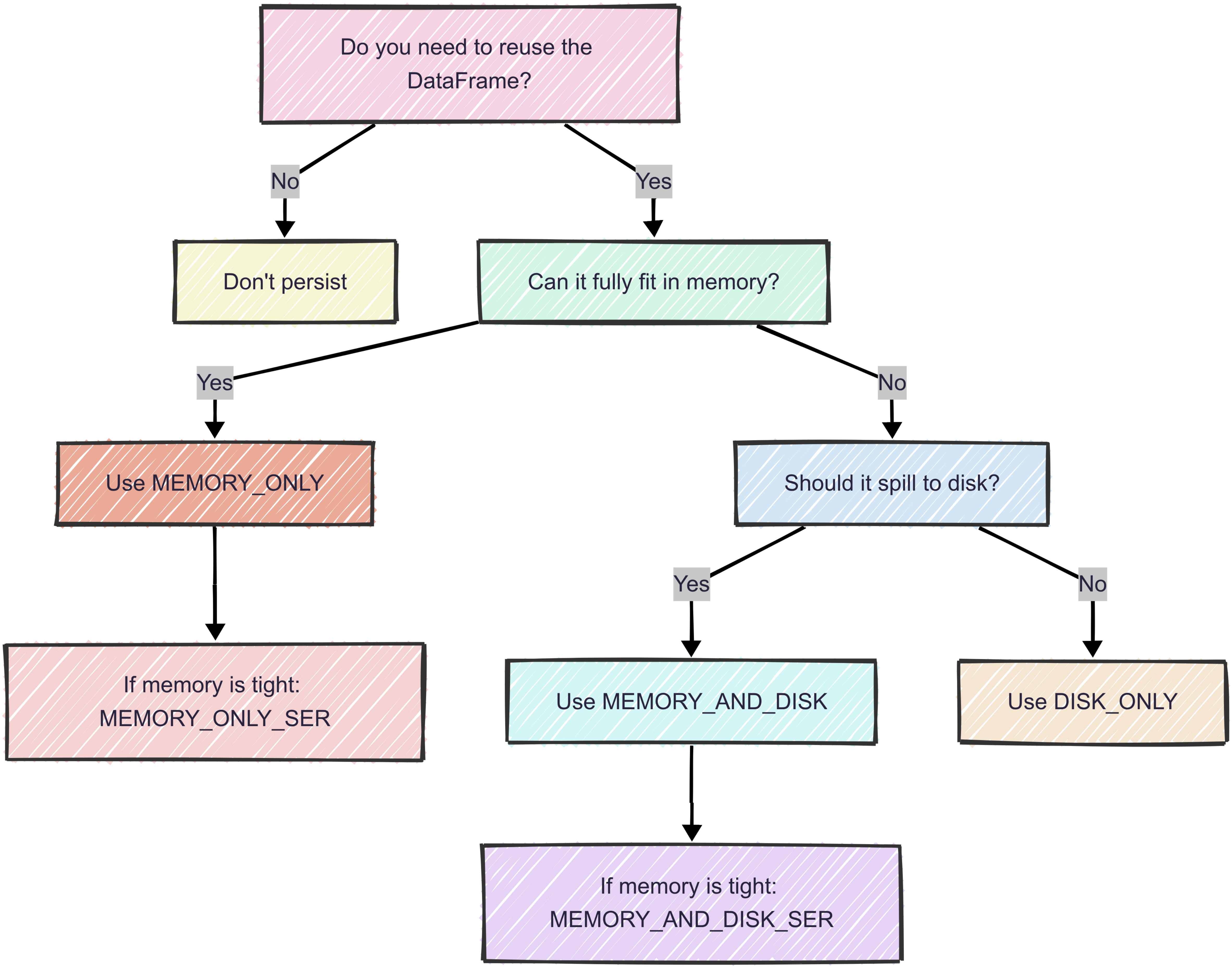
Low

⚠️ Partial recomputation

Understanding Levels

of .persist()

Spark provides multiple storage levels for .persist(), allowing you to control how data is cached based on reuse, memory limits, and system design needs.



Do you need to reuse the DataFrame?
No
Yes
Don't persist
Can it fully fit in memory?
Yes
No
Use MEMORY_ONLY
Should it spill to disk?
If memory is tight: MEMORY_ONLY_SER
Yes
No
Use MEMORY_AND_DISK
Use DISK_ONLY
If memory is tight: MEMORY_AND_DISK_SER

Architectural Considerations of .persist()

From a system design perspective, the .persist() method serves as a critical performance optimization tool in the Spark execution model.

Memory Management Allows system designers to control how data flows through memory resources

Performance Tuning Enables explicit control over the trade-off between computation speed and memory usage

Resource Allocation Functions as a mechanism to prioritize specific DataFrames/RDDs in complex data pipelines

System Boundaries Helps manage the boundaries between in-memory and on-disk processing

Performance Considerations of .persist()

From a system design perspective, the .persist() method serves as a critical performance optimization tool in the Spark execution model.

	IMPACT	DESIGN CONSIDERATION
Latency	Can reduce computation time by avoiding recomputation	Identify reused datasets and apply appropriate storage level
Throughput	Improves overall system throughput for repeated operations	Balance with memory constraints of the cluster
Resource Utilization	Can lead to memory pressure if overused	Monitor heap usage and implement unpersist() when data is no longer needed
Execution Stability	Prevents cascade failures from repeated heavy computations	Use with checkpointing for critical path operations

Best Practices & Anti-Patterns of .persist()

Be mindful of the Anti-patterns and best practices to get the most out of performance tuning.

ANTI-PATTERNS	BEST PRACTICES
Over-persistence: Persisting everything without consideration for reuse patterns	Profile first: Measure dataset sizes before choosing storage levels
Neglecting unpersist(): Failing to release memory when datasets are no longer needed	Document decisions: Comment code with rationale for persistence choices
Wrong storage level: Using MEMORY_ONLY for datasets larger than available memory	Strategic persistence: Focus on datasets at branch points in execution DAG
Ignoring serialization: Not considering serialized options for large objects	Consider partitioning: Adjust partition count to optimize memory usage