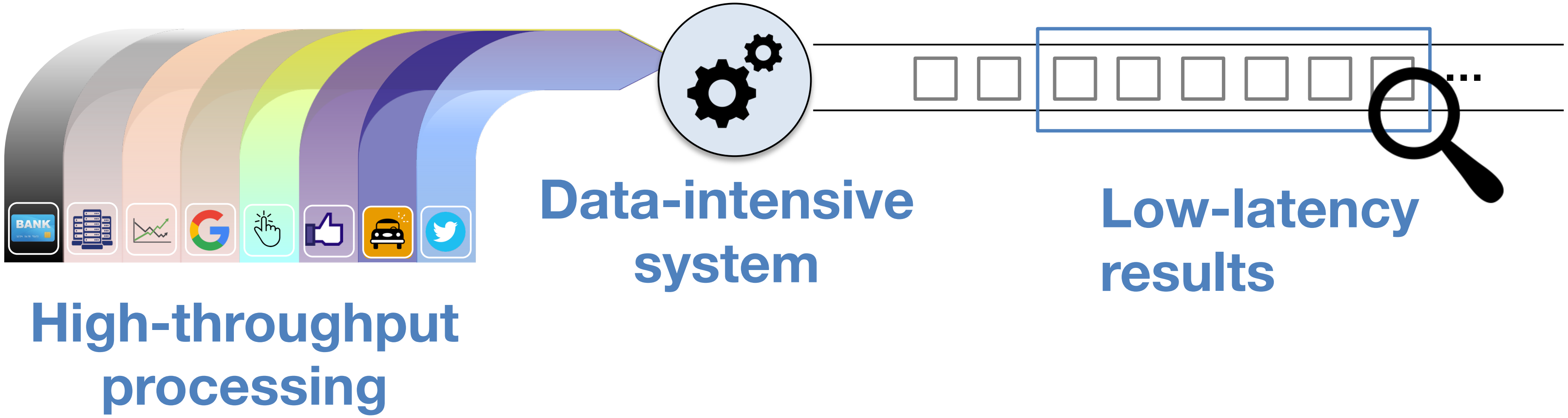
LSDS
Large-Scale Data & Systems Group

Imperial College London

Scalable and Fault-Tolerant Data Stream Processing on Multi-Core Architectures

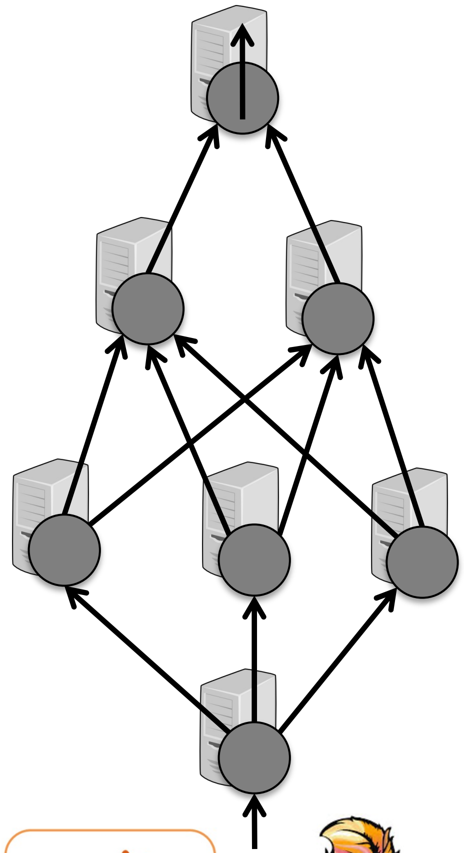
George Theodorakis
Systems Research Group, Neo4j
(was Imperial College London)

Throughput and Result Freshness Matter



Facebook Insights:	**12 GB generated content/s**	**< 10 sec latency**
Feedzai:	**24M credit card transactions/user**	**< 10 ms latency**
Uber:	**PB data/day**	**< 1 ms latency**
NovaSparks:	**150M trade options/s**	**< 1 ms latency**

Distributed Stream Processing Engines Face Challenges



Spark Streaming

Flink

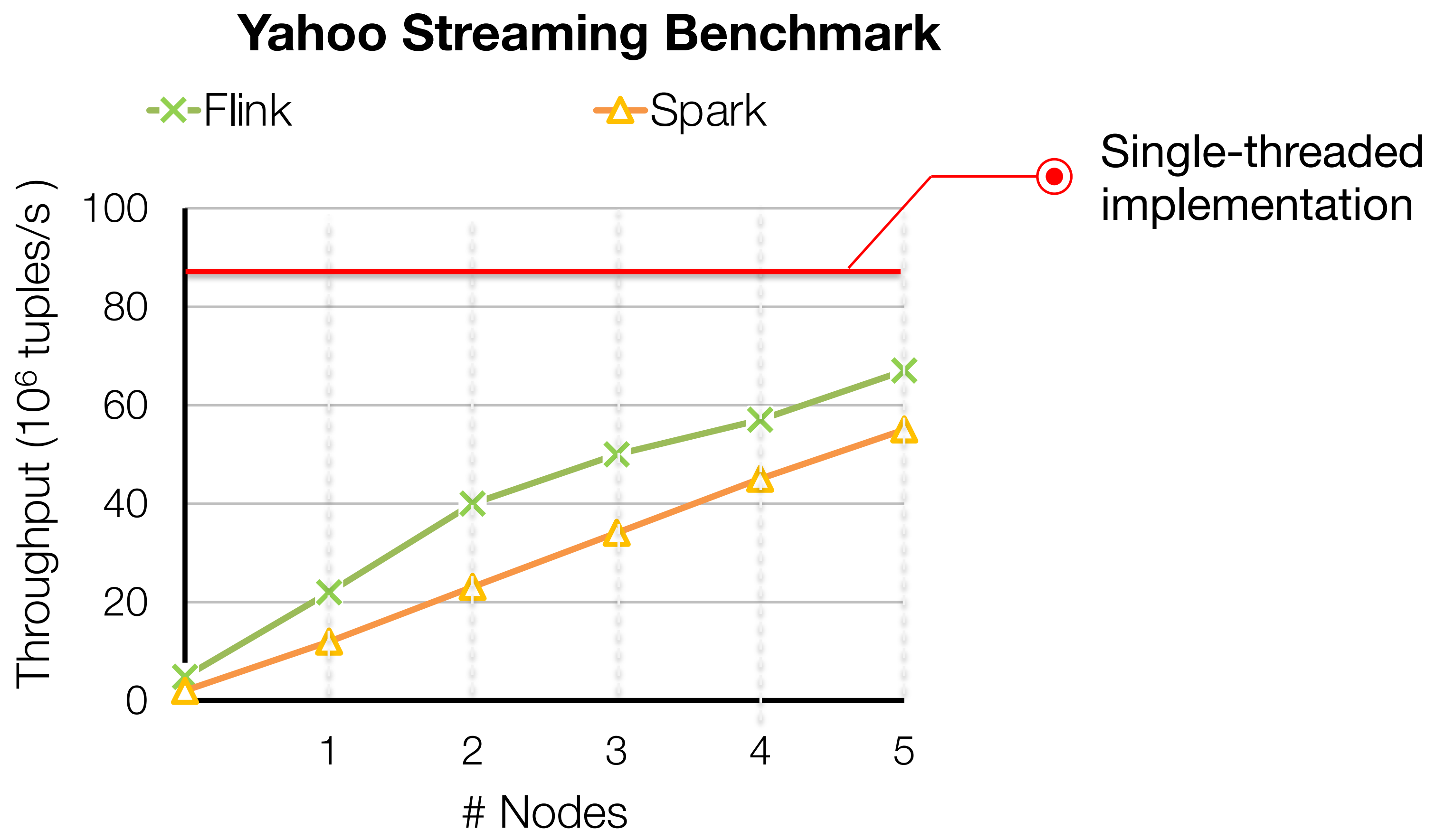
Pros

> Complex analytics scalability
> Fault-tolerance

Cons

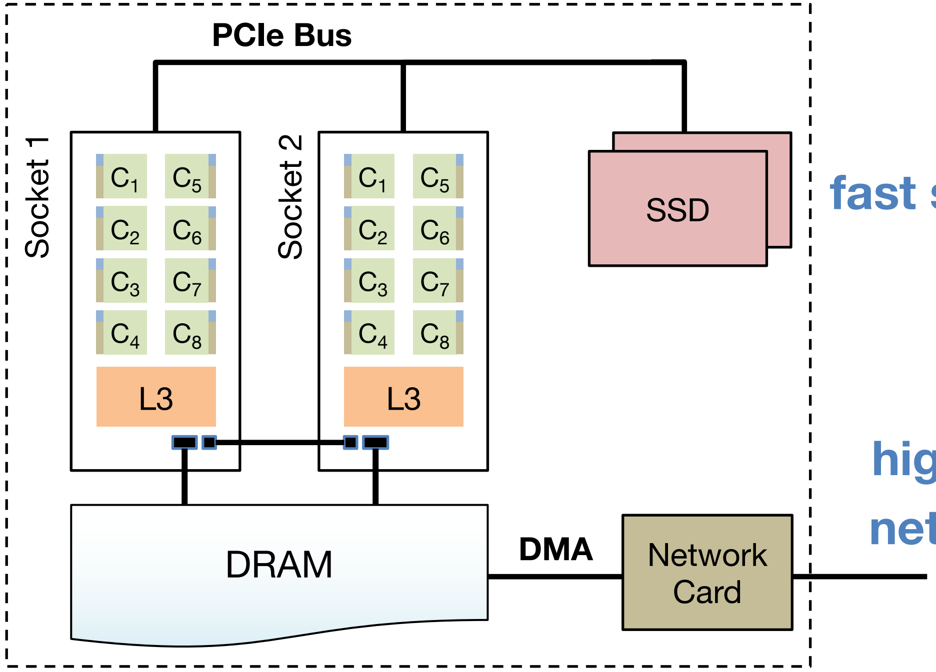
> Cross-process and network overheads
> Unpredictable latency guarantees
> Inefficient execution strategies

COST of Distributed Execution



Highly-Parallel Scale-up Architectures in Data Centers

10s of
CPU cores



fast storage

high-speed
networking

☛ **Are scale-up systems a practical alternative for scalability and fault tolerance?**

High-Performance Streaming and Fault-Tolerance is Hard

Scale-up Stream Processing Engine

Hardware-efficient single-core execution	Multi-core parallelization	Reliable stream processing
Computation sharing over windows	Parallel window execution strategy	Fault-tolerant window operators

Scale-up Stream Processing Engine

Hardware-efficient single-core execution

Computation sharing over windows

Scale-up Stream Processing Engine

Hardware-efficient single-core execution

Computation sharing over windows

HammerSlide

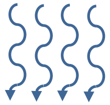
SlideSide

Multi-core parallelization

Parallel window execution strategy

Scaling Window Operators on Multi-Core Processors

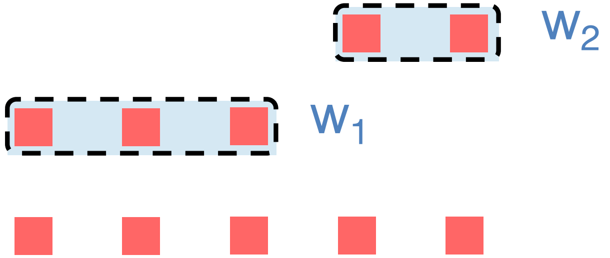
Performance = Incremental Computation + Parallelism



Tension Between Parallelism & Incremental Computation

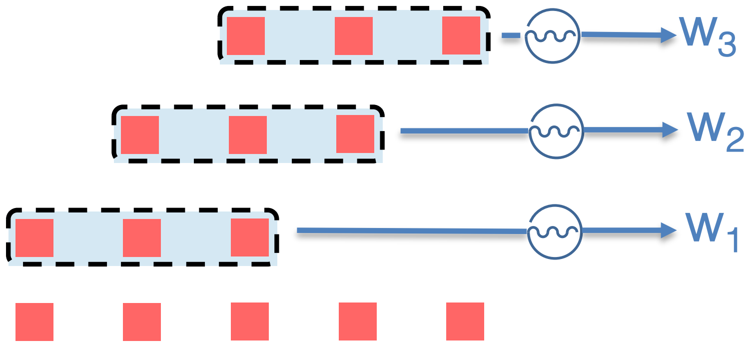
Tumbling Windows

Nothing to optimize



Sliding Windows

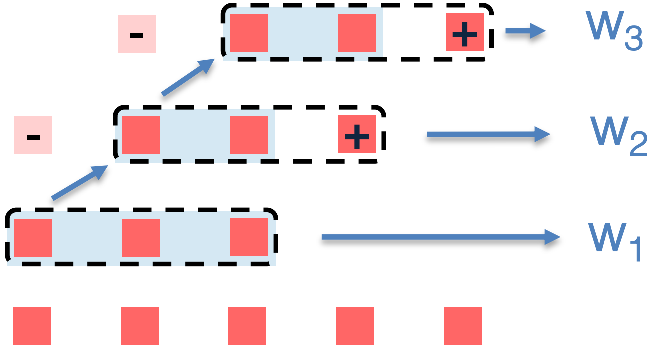
Parallel Execution



+ Parallel

- Work Efficient

Incremental Execution



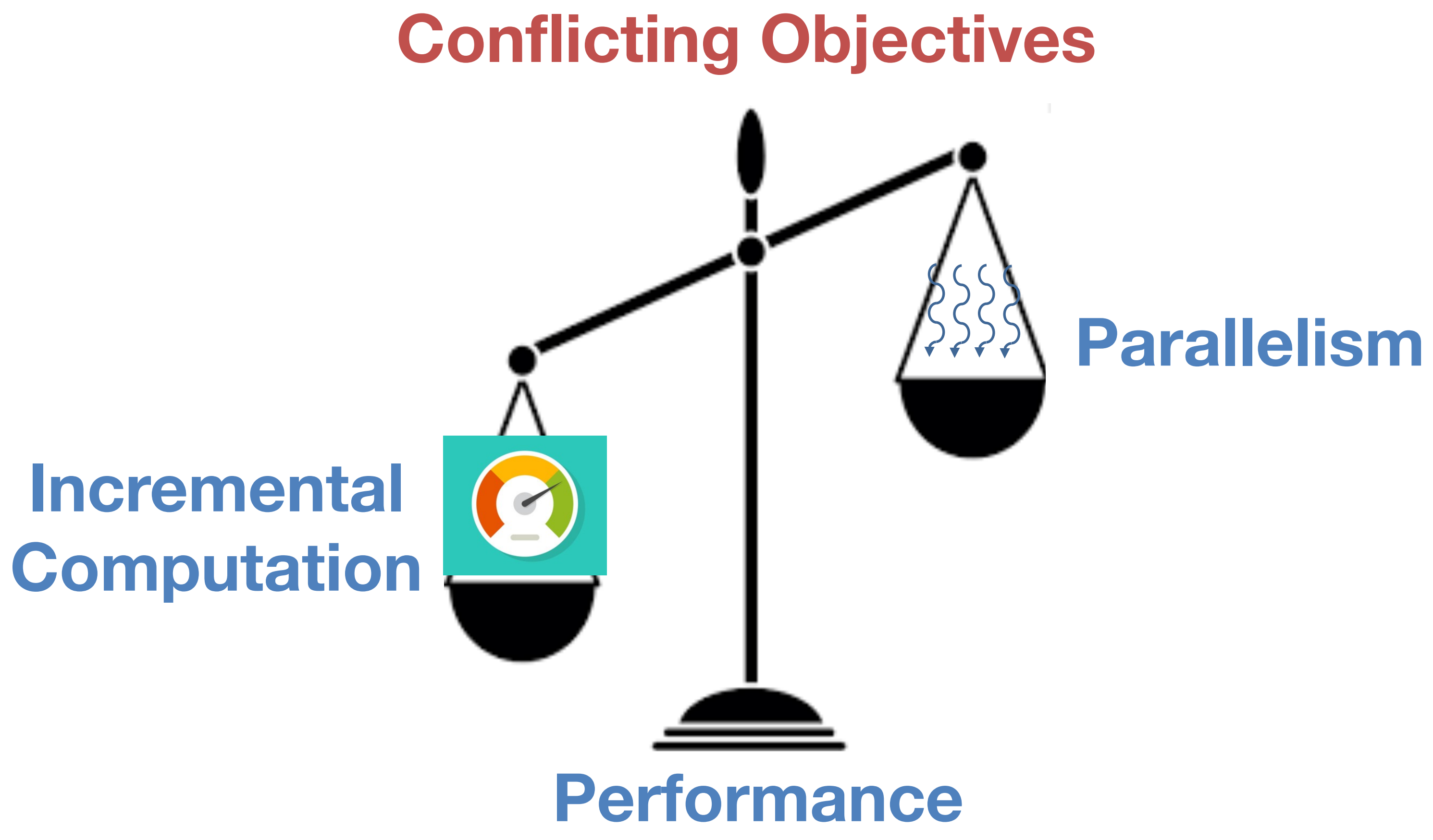
- Sequential

+ Work Efficient

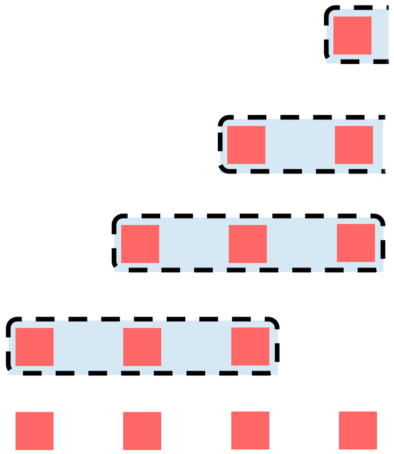
Existing System Implement Ad-Hoc Solutions

Performance = Incremental Computation + Parallelism

Existing System Implement Ad-Hoc Solutions



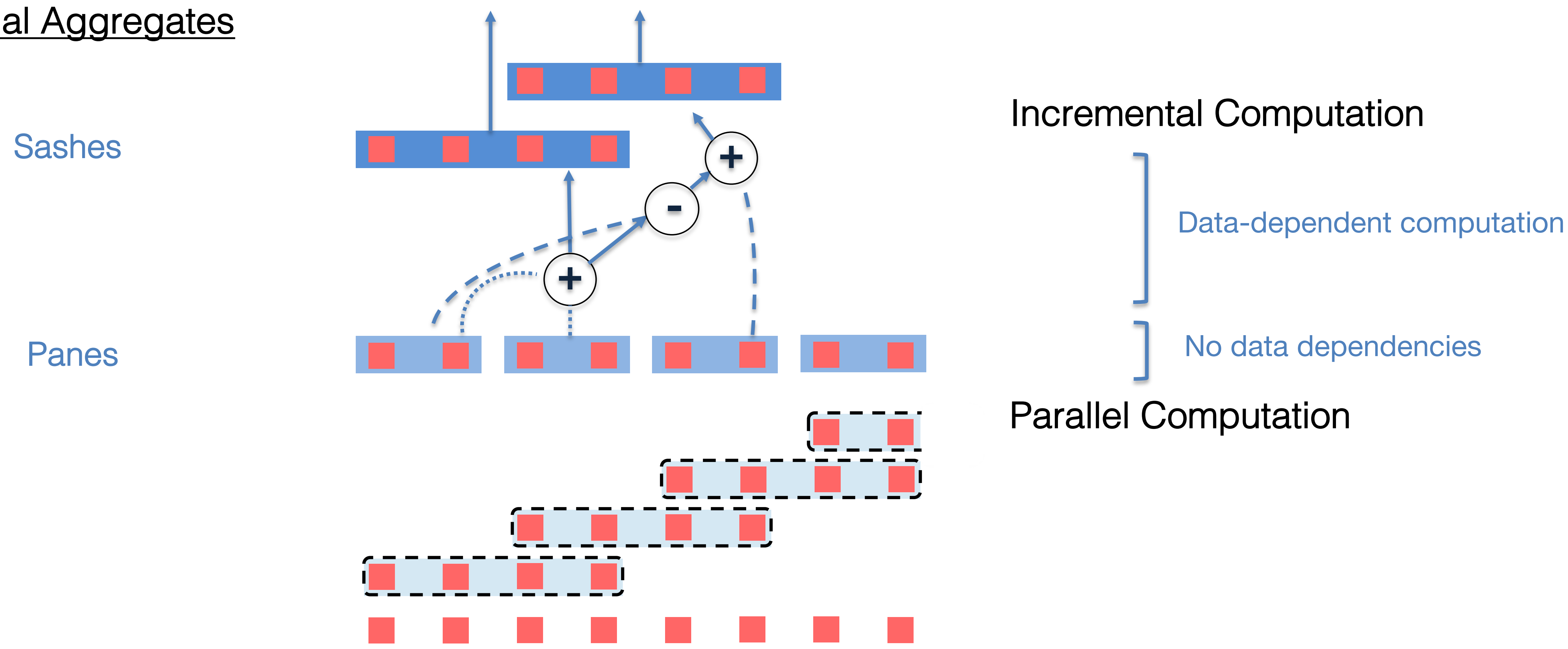
Let’s Double the Window Slide!



Let’s Double the Window Slide!

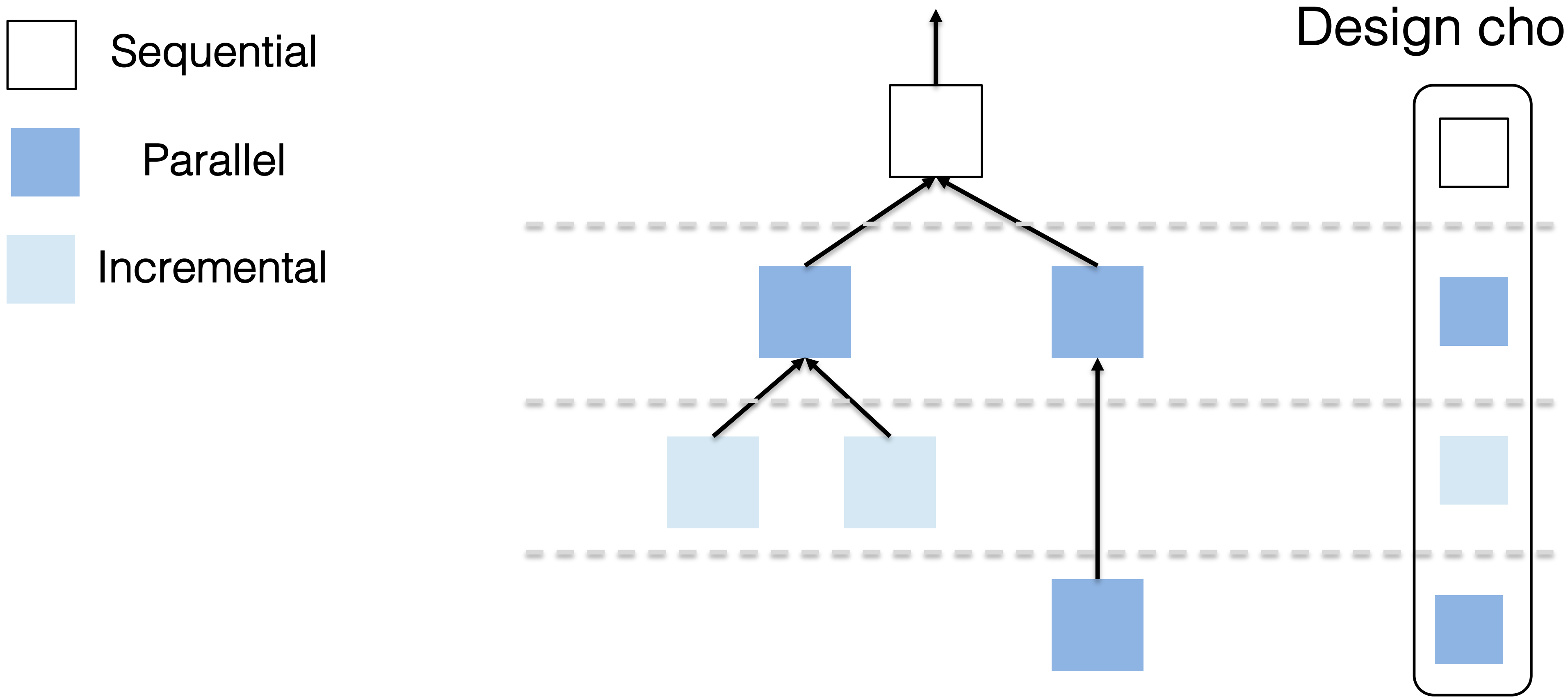
Two Sides of the Same Coin

Partial Aggregates

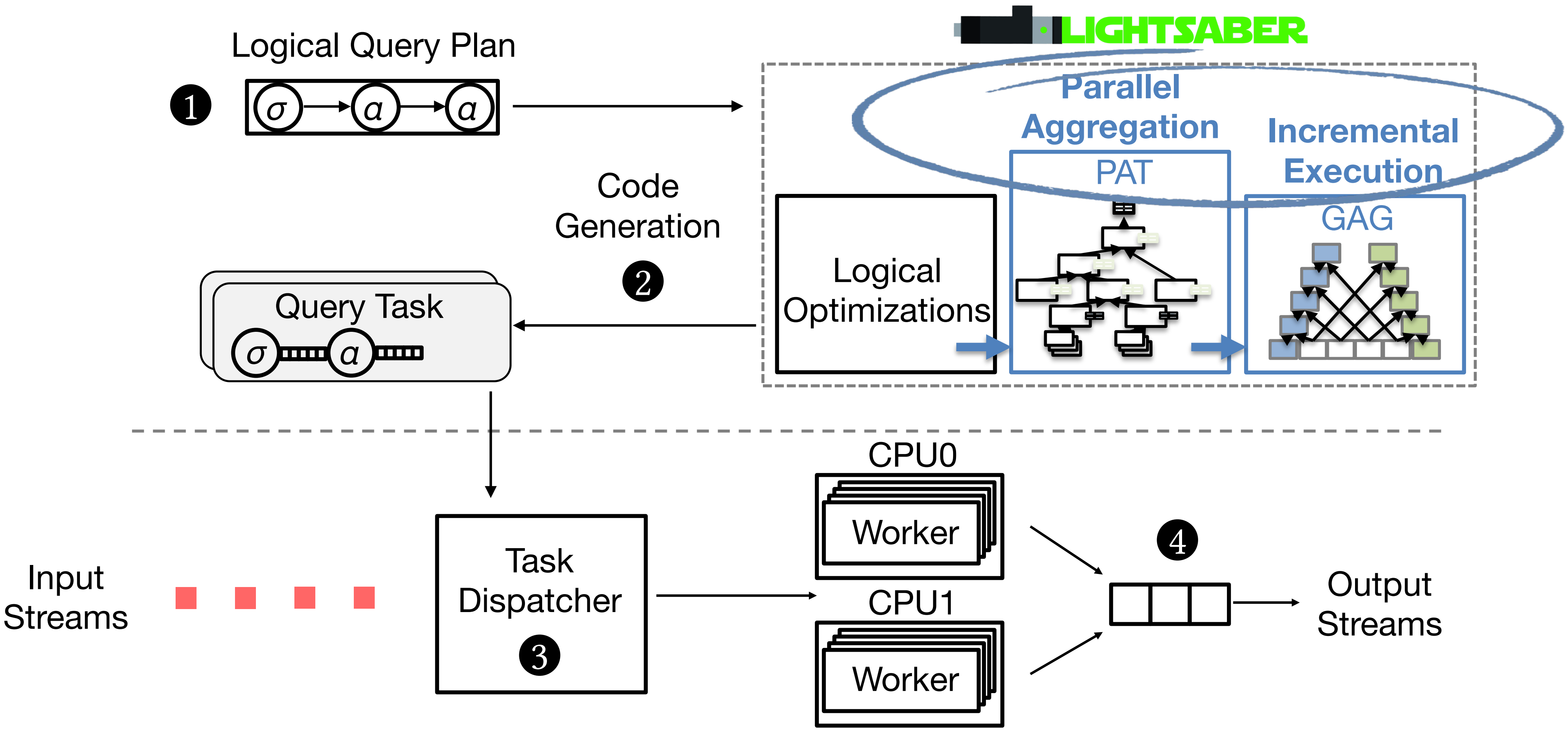


☛ **How to partition streams into intermediate steps?**

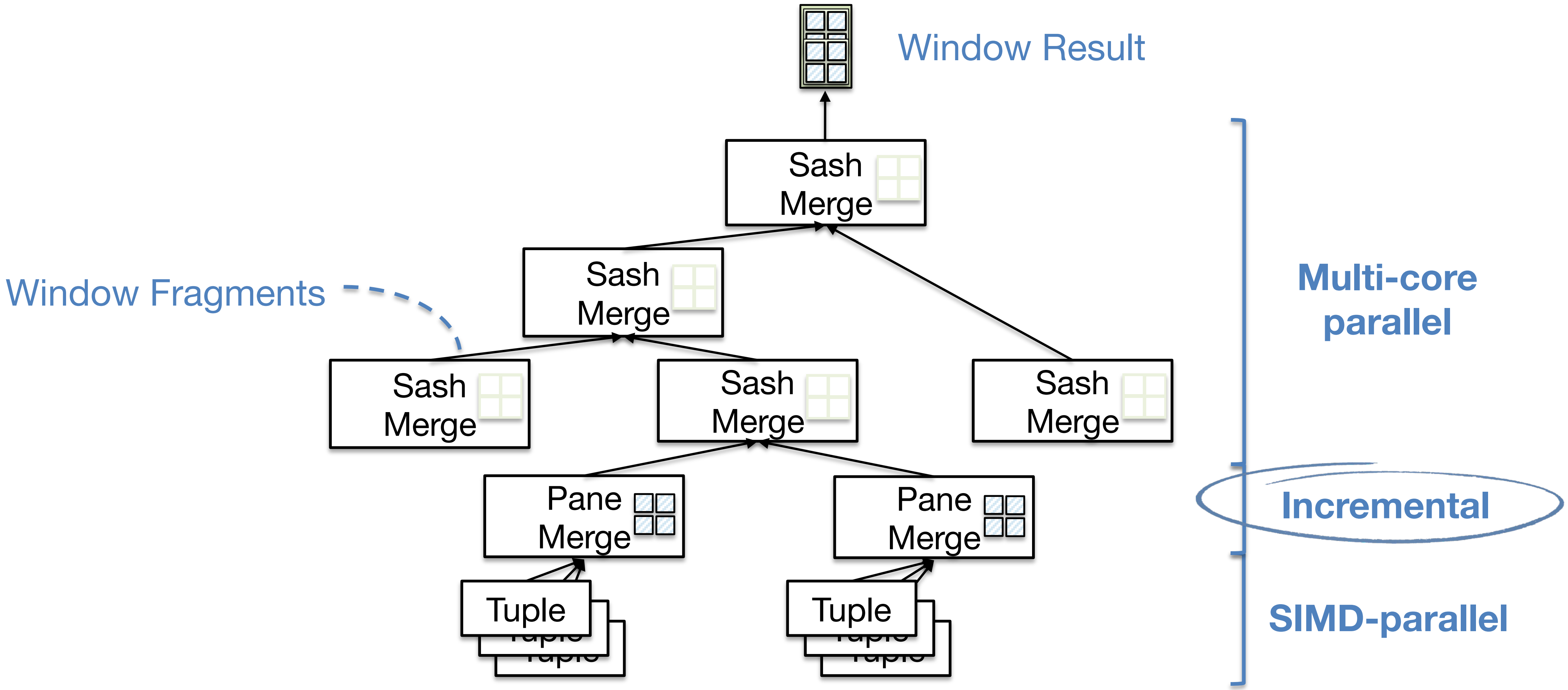
Create a Model That Splits Aggregation Into Steps



LightSaber: Combine Parallelism With Incremental Execution



Parallel Aggregation Tree: Multi-level Window Aggregation



How to Generate Efficient Code for Incremental Execution

Pane Merge

Incremental Algorithms

SoE SlideSide
Two-Stacks
SlickDeque

?

Generate workload- & query- specific code

open proceedings

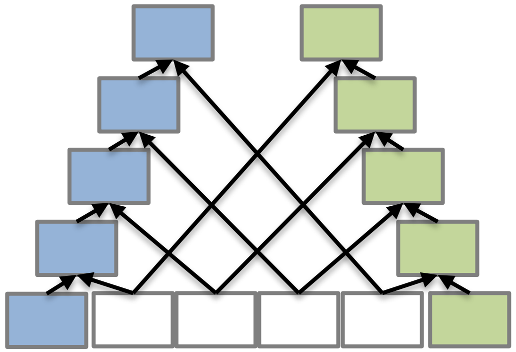
SlideSide: A fast Incremental Stream Processing Algorithm for Multiple Queries

Georgios Theodorakis   Peter Pietzuch   Holger Pirk

No best approach

General Aggregation Graph: Capture Low-Level Dependencies

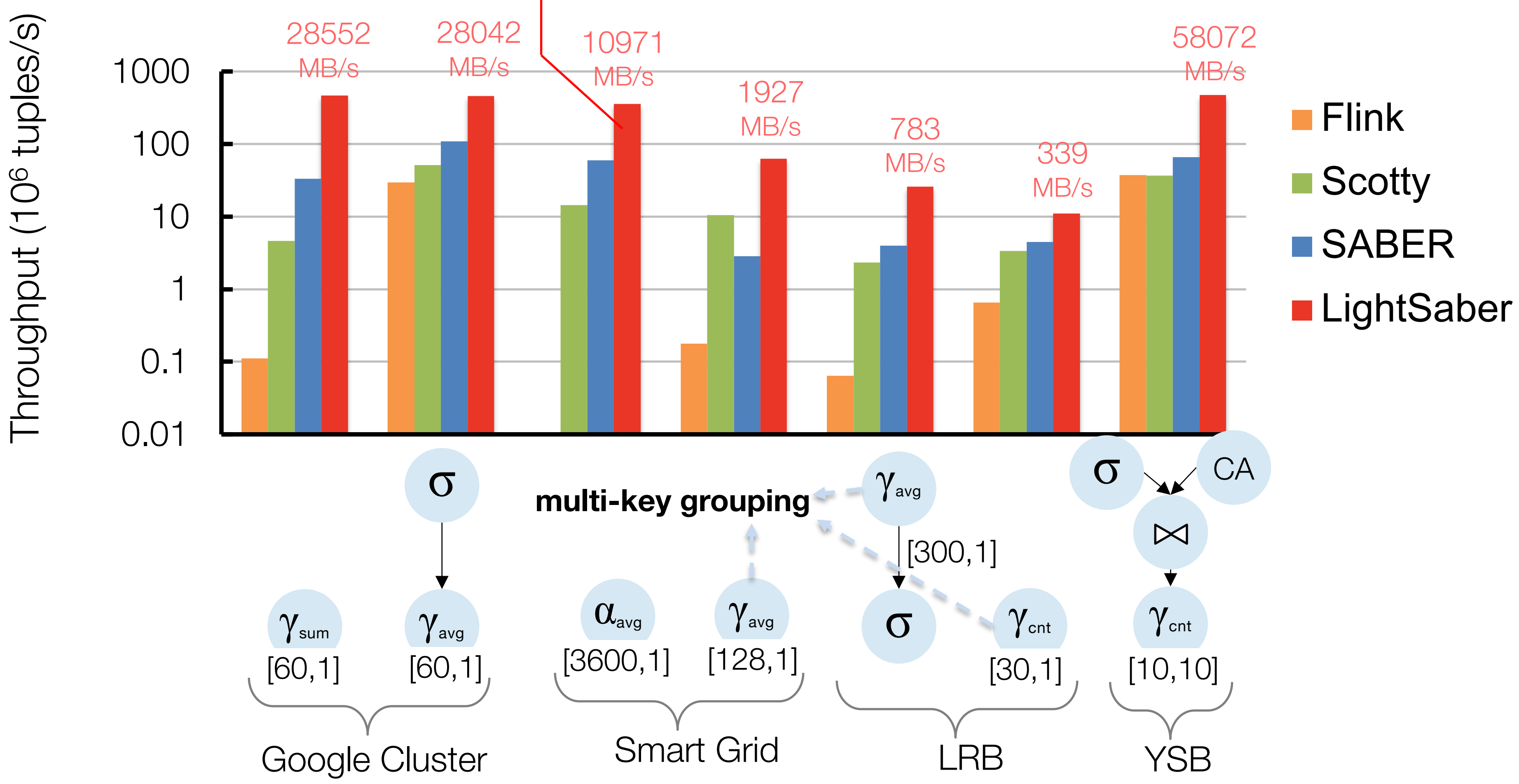
- Aggregation functions
- Window Types



```
int leafIter = 0;
for (auto &t: input) {
  if (leafIter == WINDOW_SIZE) {
    for (int i = 0;i<WINDOW_SIZE;++i)
      s[i+1] = min(ss[i],
                   l[WINDOW_SIZE-1-i]);
    leafIter = 0; ps = INT_MAX;
  }
  ps = min(ps, t);
  emit_result(min(ps,
            ss[WINDOW_SIZE-leafIter]));
  leafIter++;
}
```

Efficient Multi-core Execution

5x to more than one order of magnitude better throughput



Scale-up Stream Processing Engine

Hardware-efficient single-core execution

Computation sharing over windows

HammerSlide

SlideSide

Multi-core parallelization

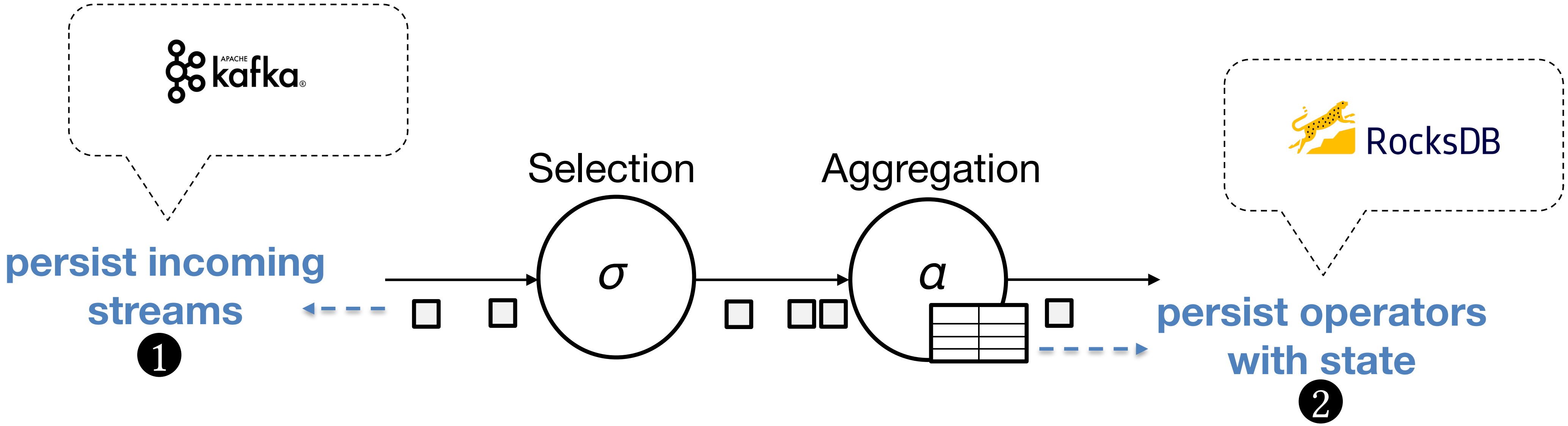
Parallel window execution strategy

LIGHTSABER

Reliable stream processing

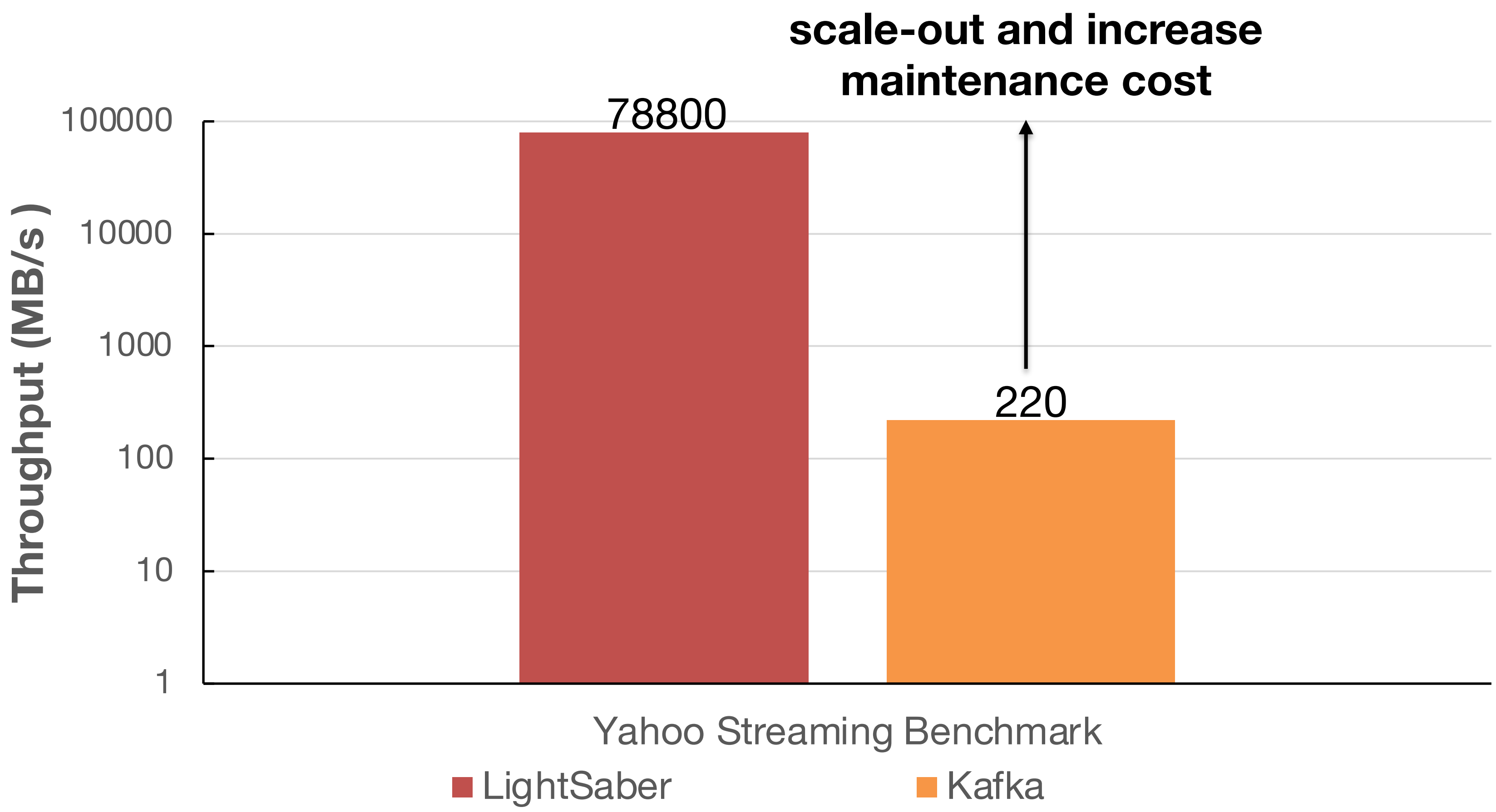
Fault-tolerant window operators

Scale-up Engines Have Limited Adoption due to Lack of Built-in Fault-Tolerance



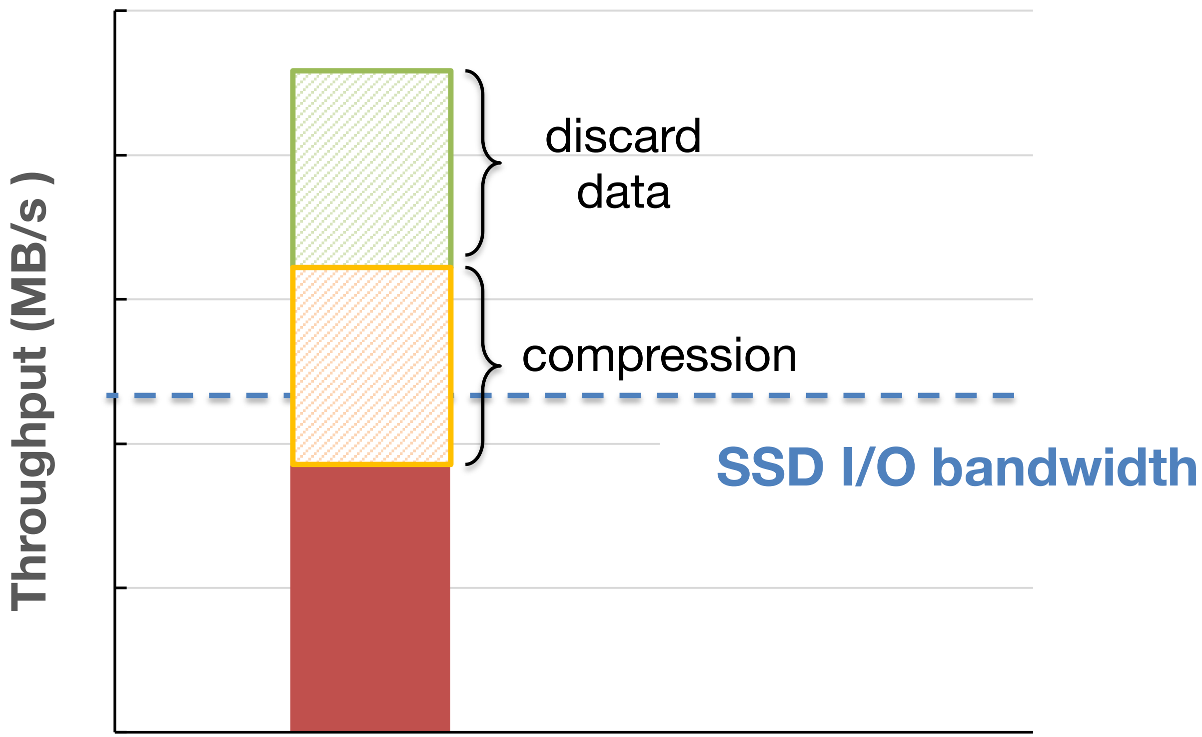
> Fault-tolerance requires persisting data from queries
> Persistence is offloaded to external systems

Kafka Ingestion Trails Scale-up Performance

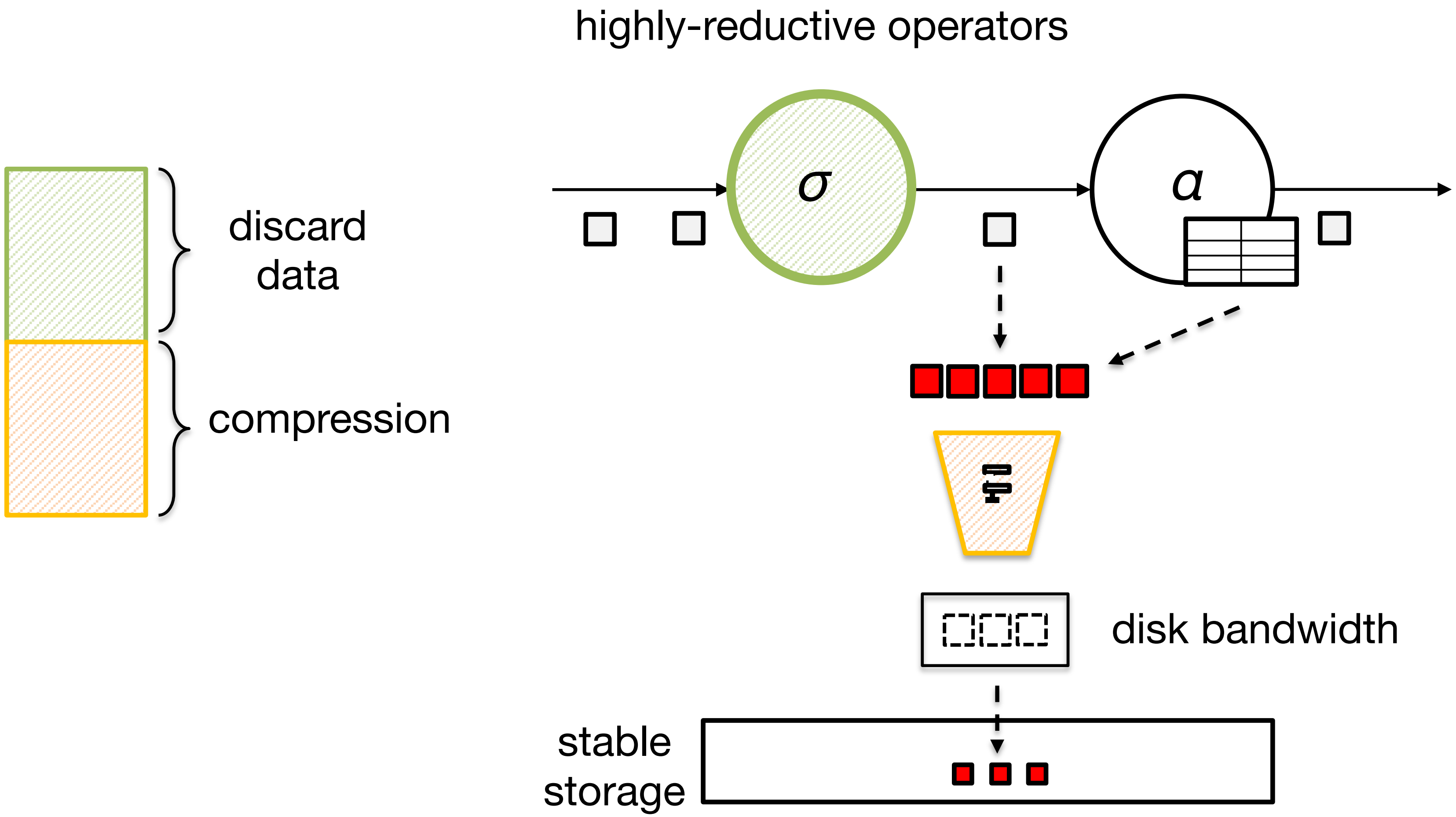


Single-node fault-tolerance without compromising performance!

Key Idea: Reduce Required Disk I/O Bandwidth



Scabbard: Reduce Required Disk I/O Bandwidth



1
decide what/when to persist based on workload

2
compress remaining data

Single-Node Fault-Tolerant Stream Processing

SCABBARD

> Co-optimize persistence and query execution
> JIT compile compression operators at runtime
> Use remote storage (e.g., EBS) and high-speed networking (RDMA)

Scale-up Stream Processing Engine

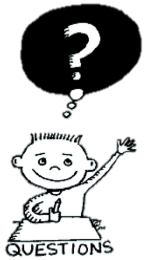
Hardware-efficient single-core execution	Multi-core parallelization	Reliable stream processing
Computation sharing over windows	Parallel window execution strategy	Fault-tolerant window operators
HammerSlide **SlideSide**	LIGHTSABER	SCABBARD

Summary

Single-node SPEs provide a practical alternative for **scalable** and **reliable** stream processing!

LIGHTSABER

https://github.com/lsds/LightSaber



Thank you!
Questions?

George Theodorakis
george.theodorakis@neo4j.com