Boosting Spark Performance on Many-Core Machines

Qifan Pu
Sameer Agarwal (Databricks)
Reynold Xin (Databricks)
Ion Stoica

amplab
UC BERKELEY

Me

Ph.D. Student in AMPLab, advised by Prof. Ion Stoica

- Spark-related research projects
    - e.g., how to run Spark in a geo-distributed fashion
- Big data storage (e.g., Alluxio)
    - how to do memory management for multiple users

Intern at Databricks in the past summer

- Spark SQL team: aggregates, shuffle

This project

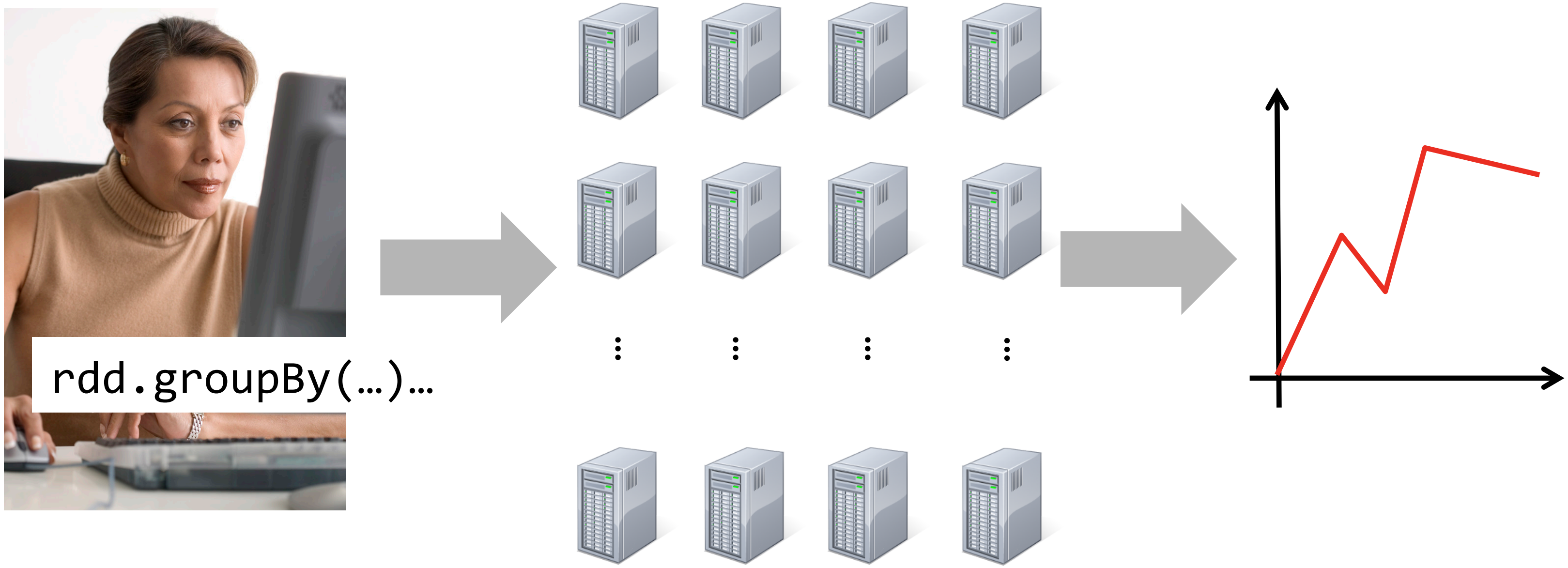
Spark performance on many-core machines

- Ongoing research, feedbacks are welcome
- Focus of this talk:
    - understand shuffle performance
    - Investigate and implement In-memory shuffle

Moving beyond research

- Hope is to get into Spark (but no guarantee yet)

Why do we care about many-core machines?



Spark started as a **cluster** computing framework

Spark on cluster

- Spark's one big success has been high scalability
    - Largest cluster known is 8000
    - Winner of 2014 Daytona GraySort Benchmark

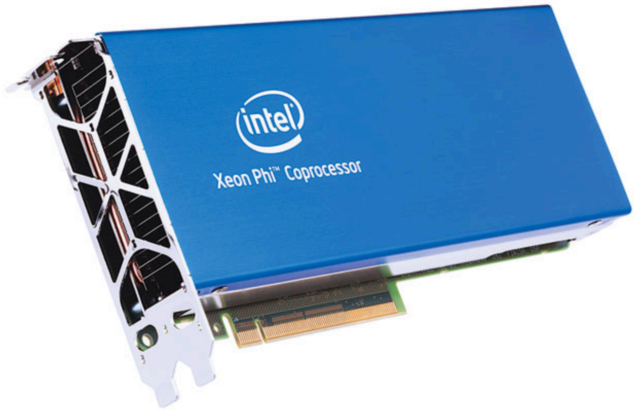
- Typical cluster sizes in practice:

"Our observation is that companies typically experiment with cluster size of under 100 nodes and expand to **200 or more nodes** in the *production* stages."

-- Susheel Kaushik (Senior Director at Pivotal)

Increasingly powerful single node machines

- More cores packed on single chip
  - Intel Xeon Phi: 64-72 cores
  - Berkeley FireBox Project: ~100 cores



- Larger instances in the Cloud
  - Various 32-core instances on EC2, Azure & GCE
  - EC2 X-Instance with 128 cores, 2TB (May 2016)

Cost of many-core nodes

	Memory (GB)	vCPUs	Hourly Cost ($)	Cost/100GB	Cost/8vCPU
x1.32xlarge	1952	128	13.338	0.68	0.83
g2.2xlarge	15	8	0.65	4.33	0.65
i2.2xlarge	61	8	1.705	2.80	1.70
m3.2xlarge	30	8	0.532	1.78	0.53
c3.2xlarge	15	8	0.42	2.80	0.42

1 x1.32xlarge instance is a small cluster
(with more memory, fast inter-core network)

Spark's design was based on many nodes

Focus of this talk

- Data communication (a.k.a. shuffle)
  - Store intermediate data on disk
  - Serialization/deserialization needed across nodes
  - Now: much memory to spare, intra-node shuffle

Ongoing work

- Resource management
  - Designed to handle moderate amount on each node
  - Now: 1 executor for 100 cores + 2TB memory?

Can we improve shuffle on single, multi-core machines?

1, Memory is fast
2, We can use memory for shuffle

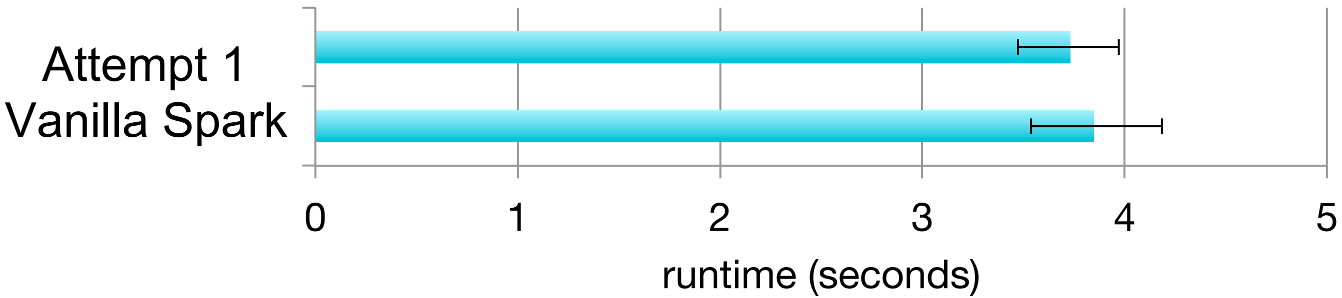
3, Therefore,
Shuffle will be fast

Will this “common sense” work?

Put in practice…

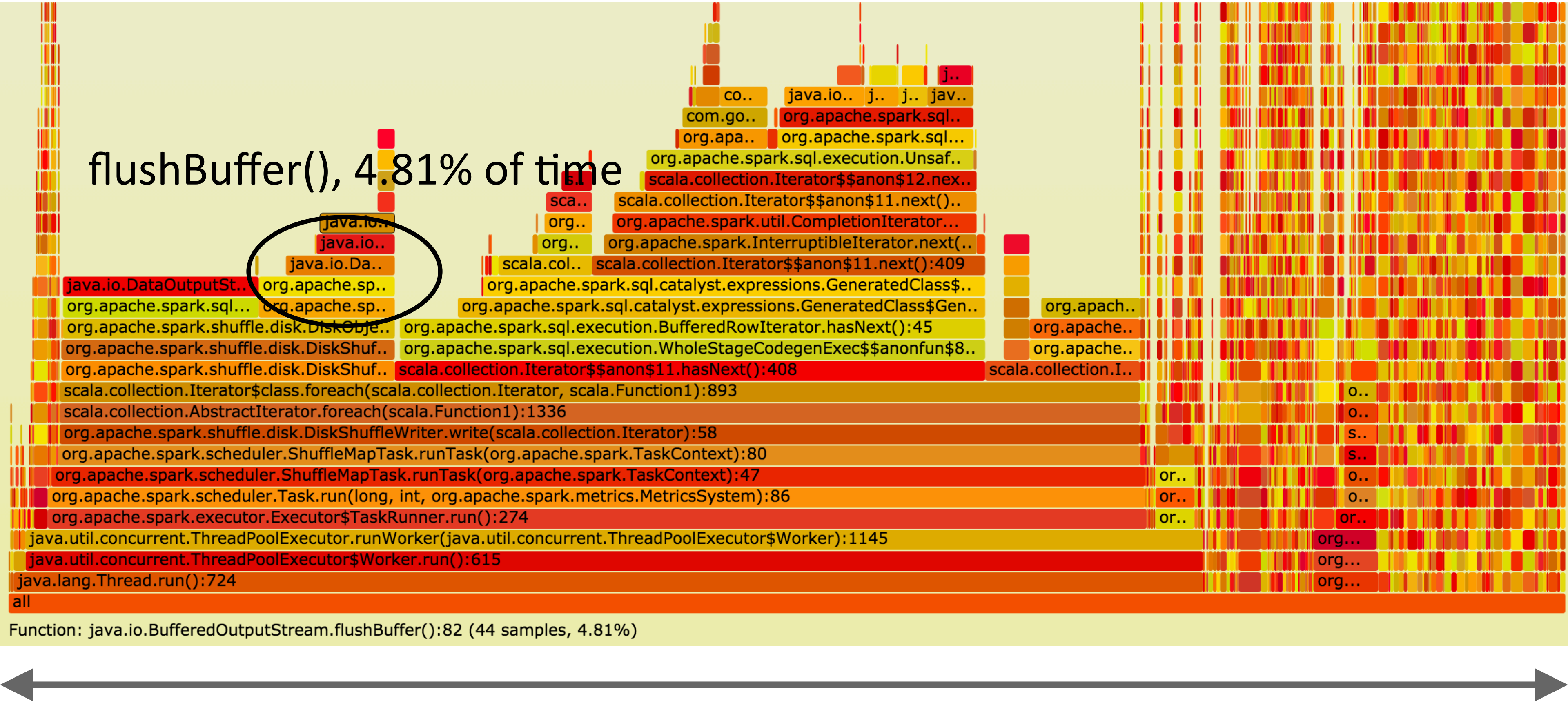
- Spark: write to a file stream, and save all bytes to disk
- Solution: …, …. to memory (bytes on heap)

spark.range(16M).repartition(1).selectExpr("sum(id)").collect()



Why zero improvement?

Why zero improvement?



Entire duration of a job (100%)

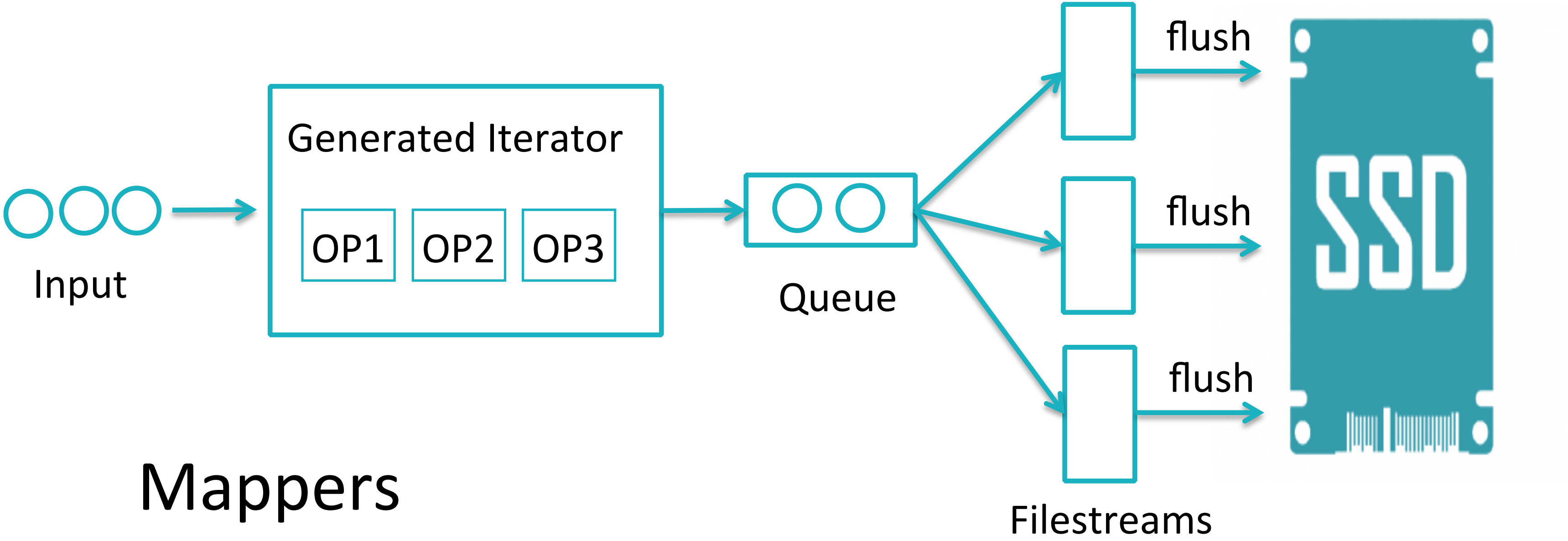
Why zero improvement?

1, I/O throughput is not the bottleneck (in this job)

2, Buffer cache:

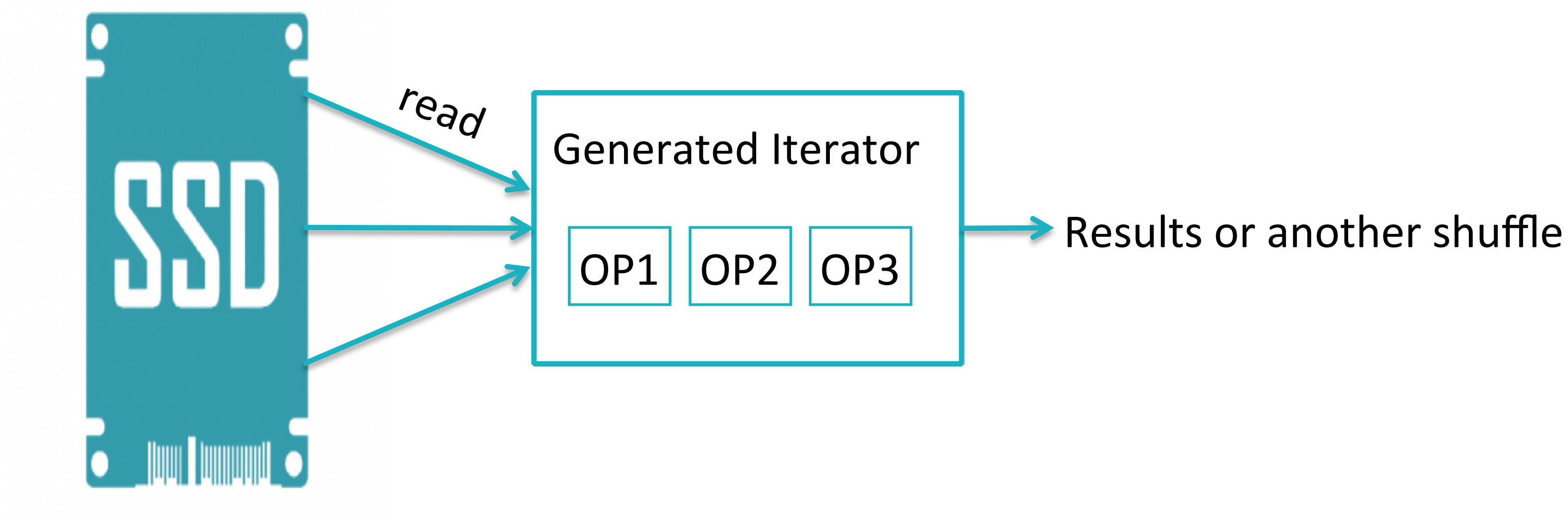
memory is being exploited by disk-based shuffle

Understanding Shuffle Performance



Understanding Shuffle Performance

Reducers

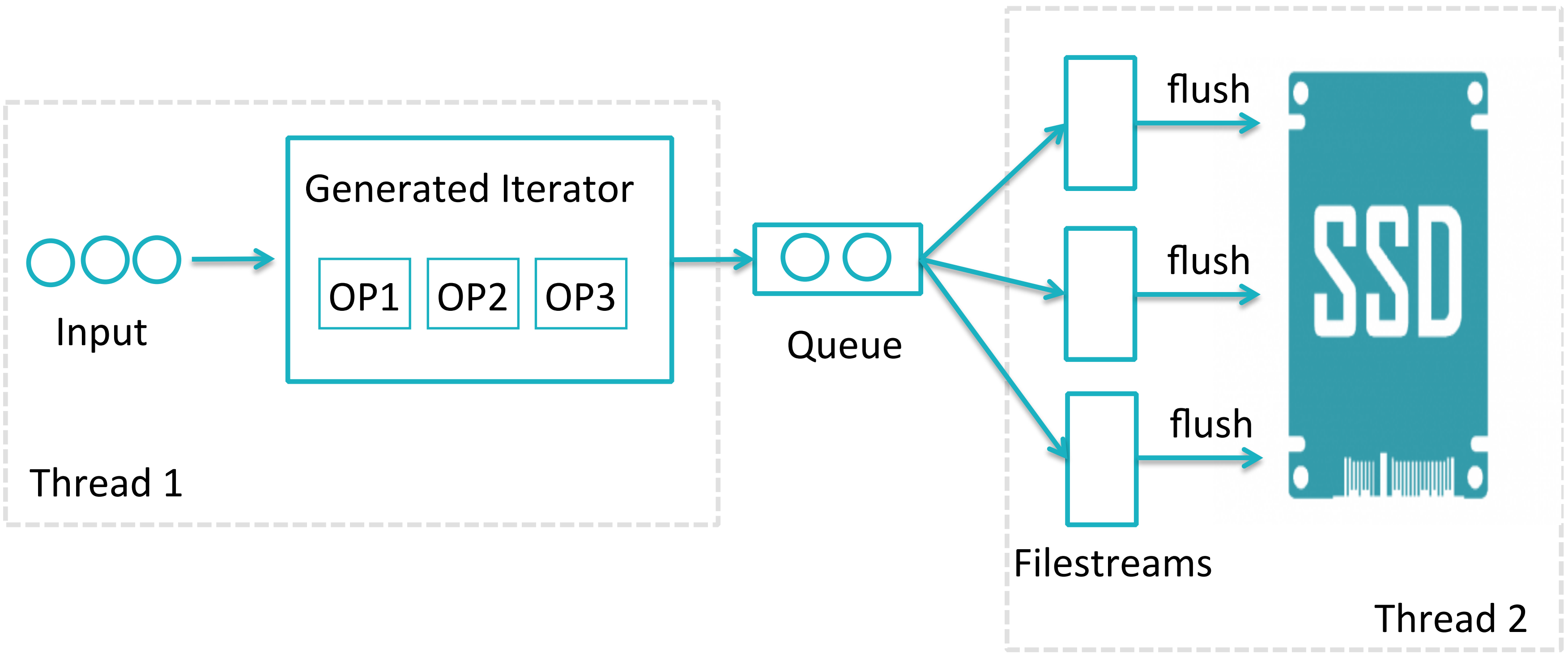


More complications

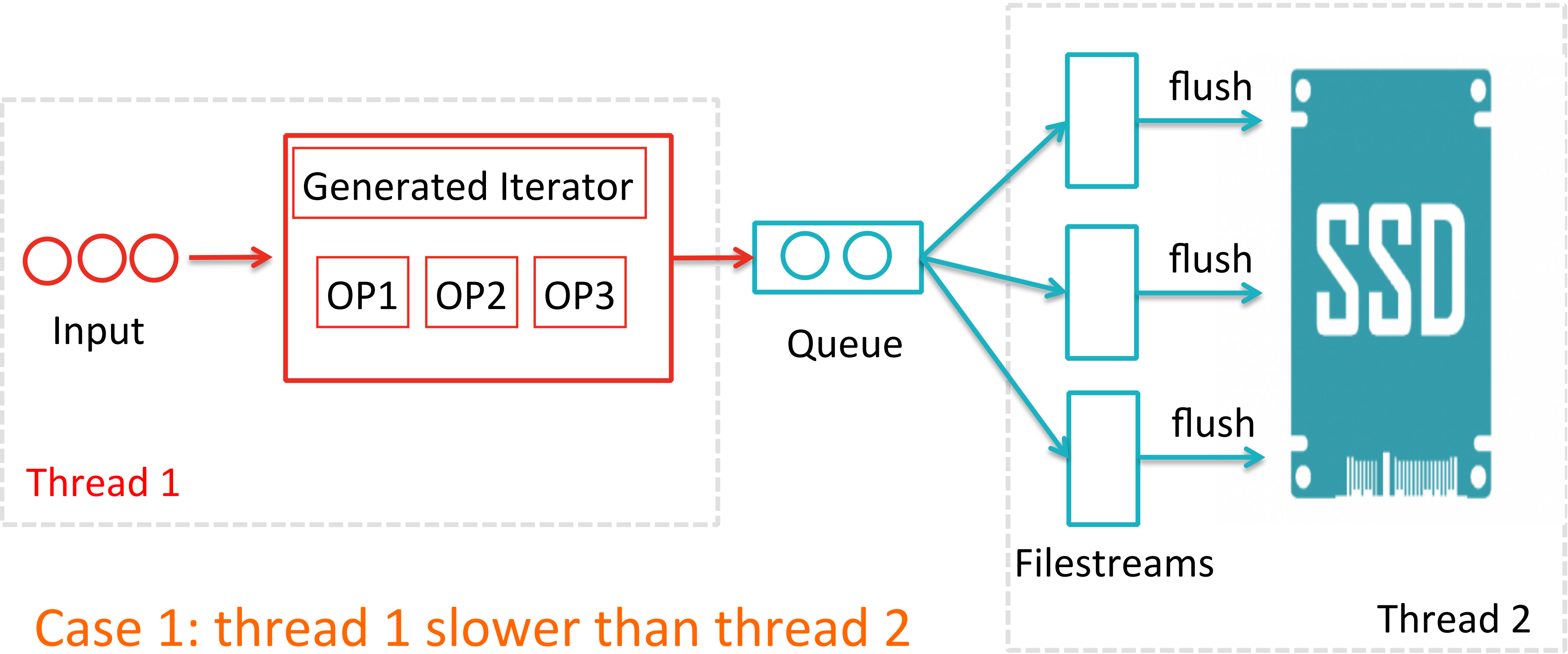
- Sort vs. hash-based shuffle
- Spill when memory runs out
- Clear data after shuffle

…

Case 1

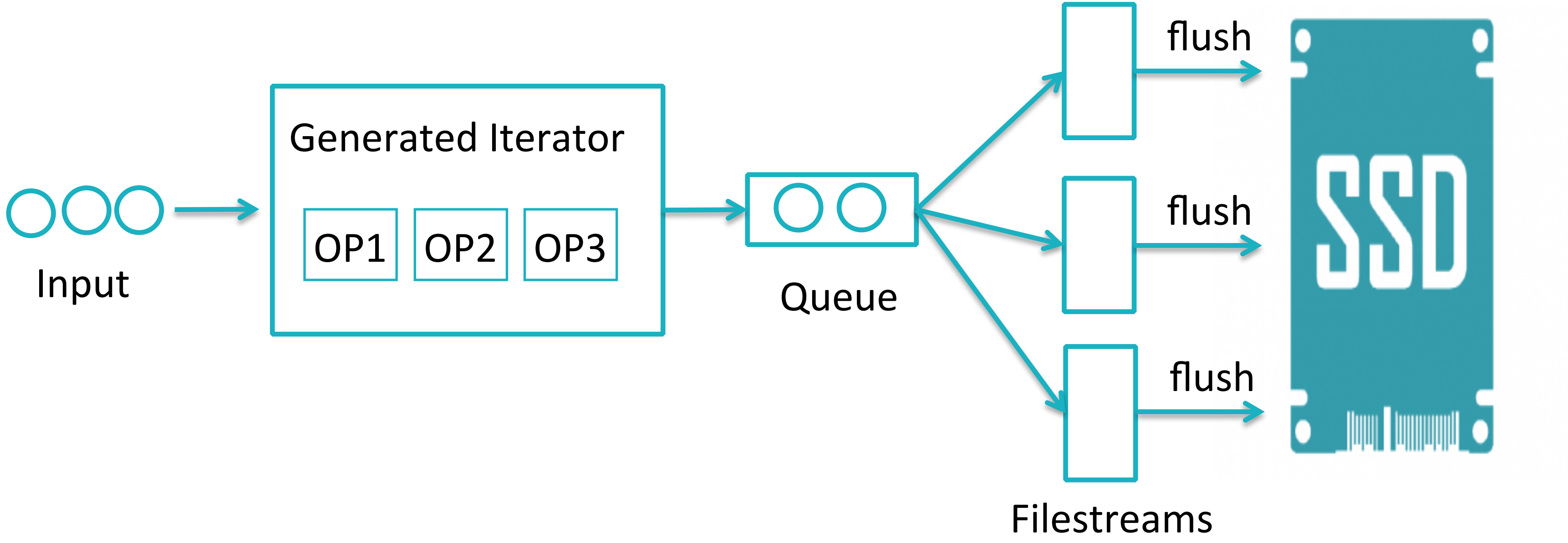

Generated Iterator
OP1
OP2
OP3
Input
Thread 1
Queue
flush
flush
flush
SSD
Filestreams
Thread 2

Case 1



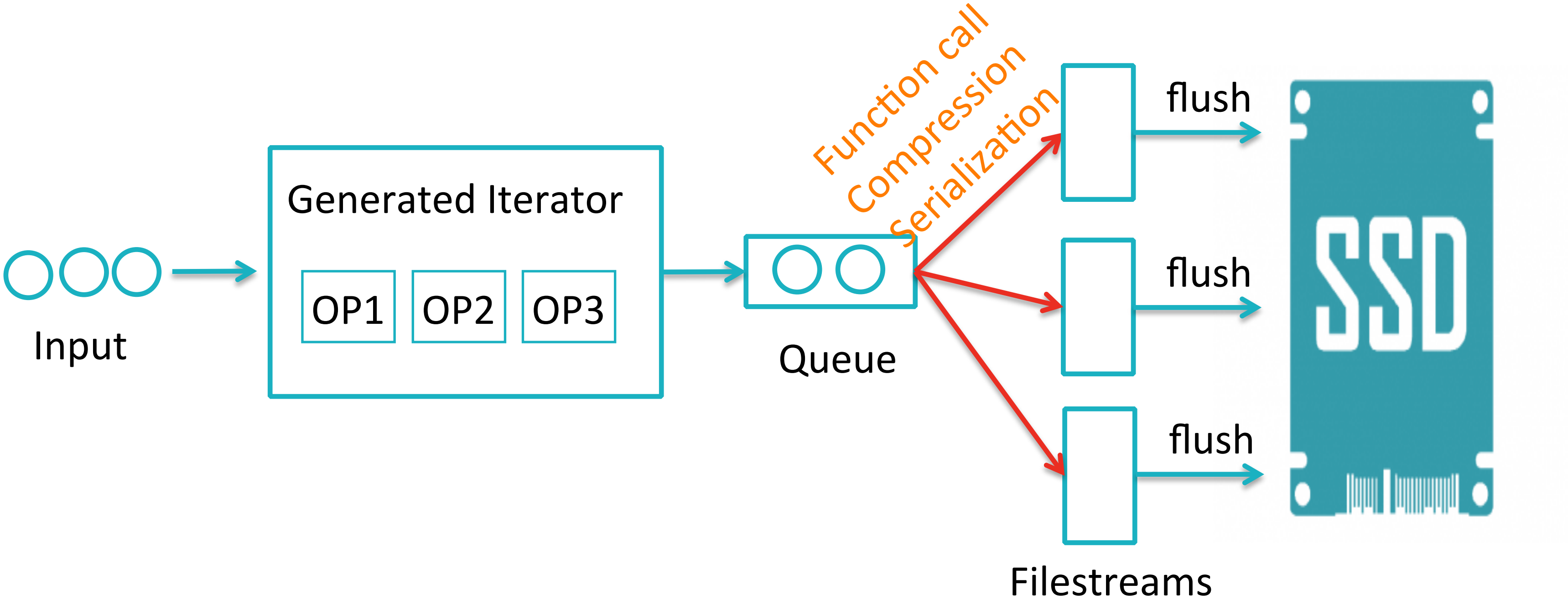
Case 1: thread 1 slower than thread 2

Case 2



Input
Generated Iterator
OP1
OP2
OP3
Queue
flush
flush
flush
SSD
Filestreams

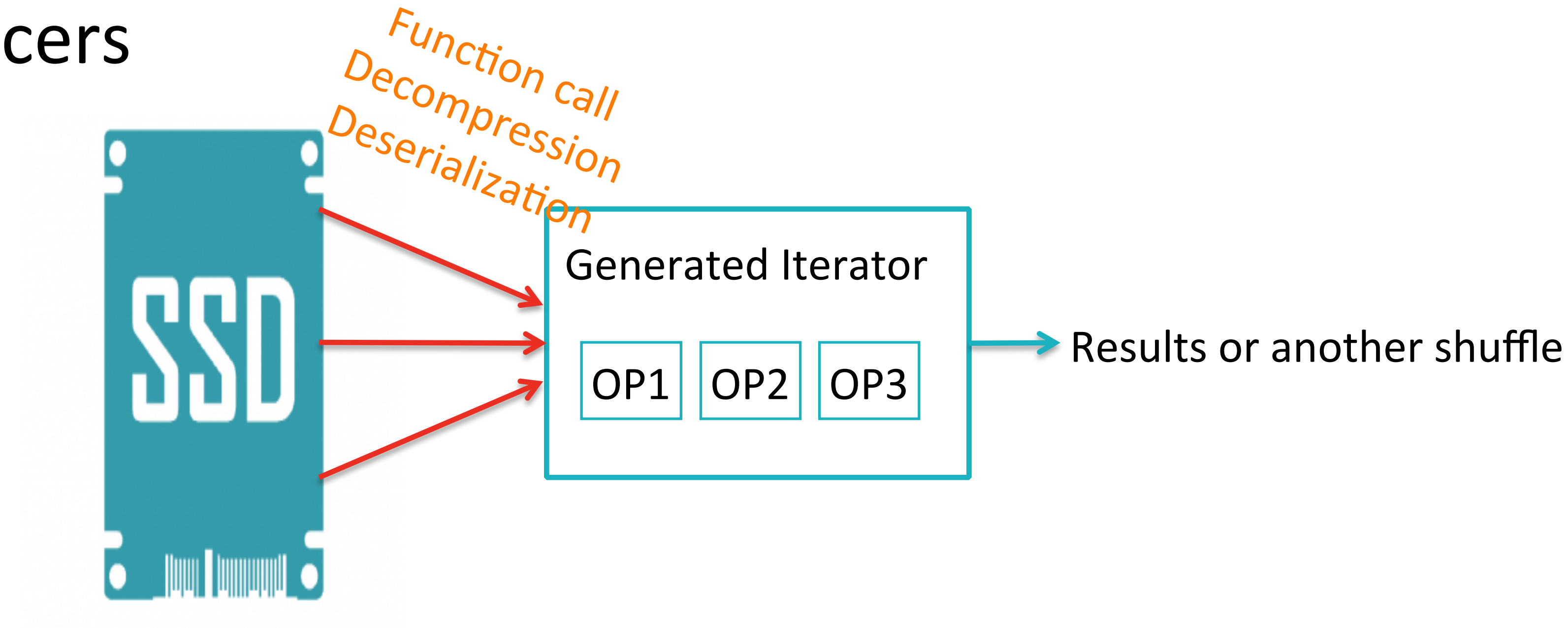
Case 2



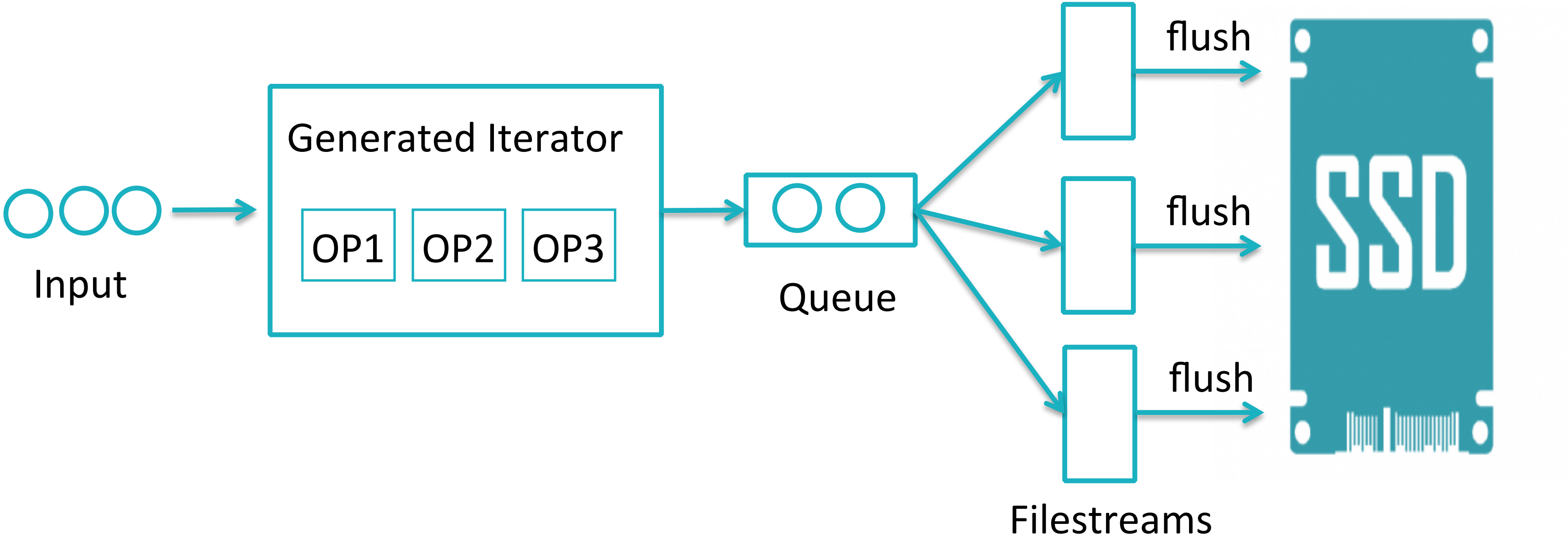
Case 2: writing/reading file streams is slow

Case study 2

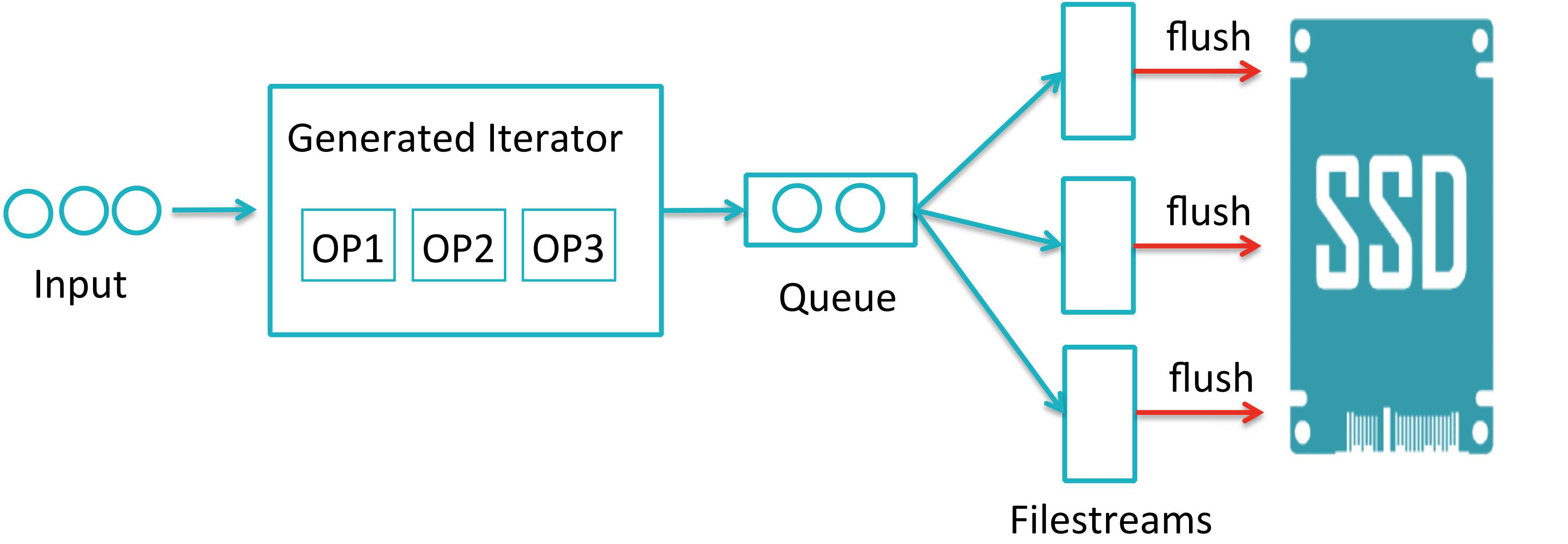
Reducers



Case 3


Input
Generated Iterator
OP1
OP2
OP3
Queue
flush
flush
flush
SSD
Filestreams

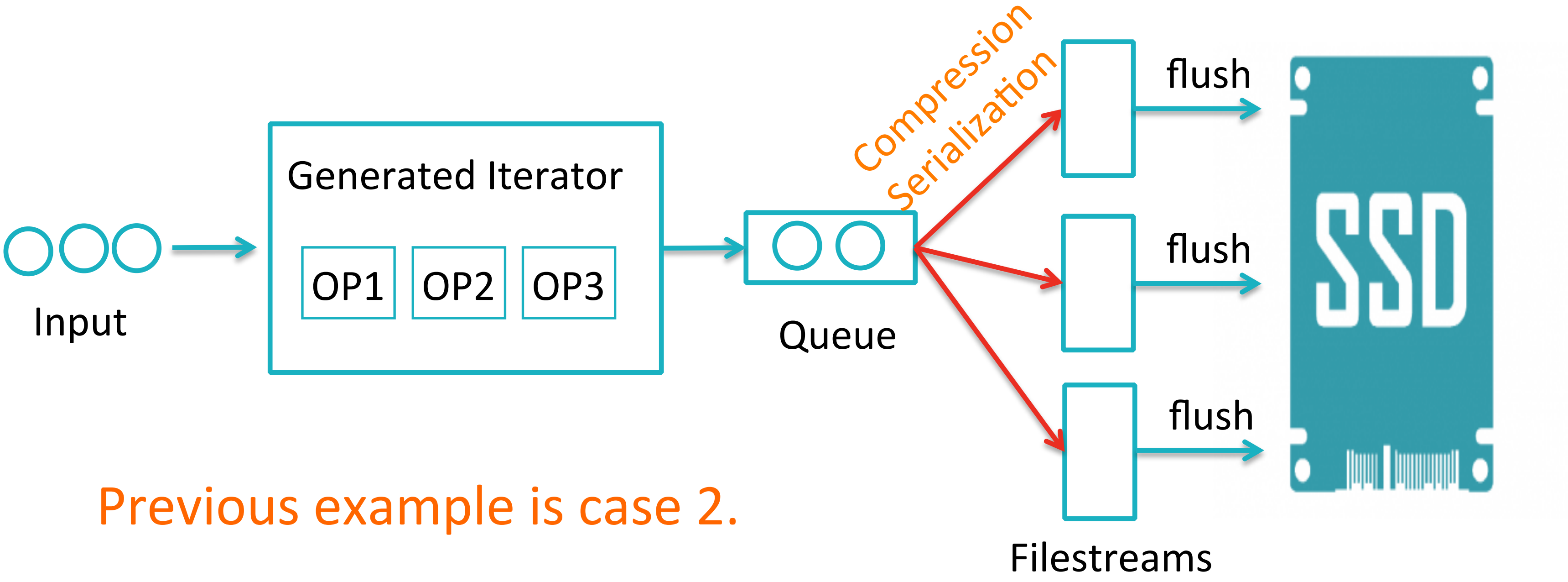
Case 3



Case 3: I/O contentions

Our first attempt should work in this case!

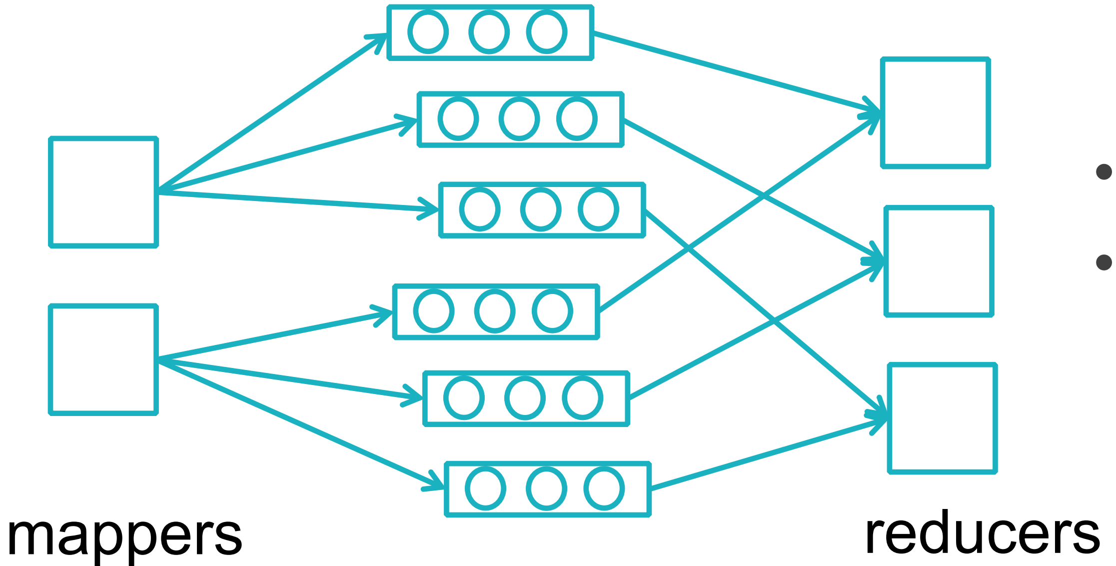
Can we improve case 2?



Previous example is case 2.

Attempt 2: get rid of ser/deser, etc

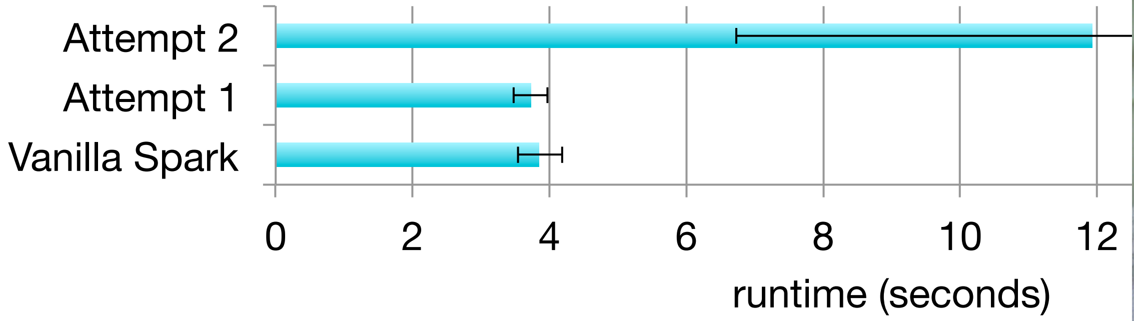
- Attempt 2: create NxM queues (N=mappers, M=reducers)
  push corresponding records into queues

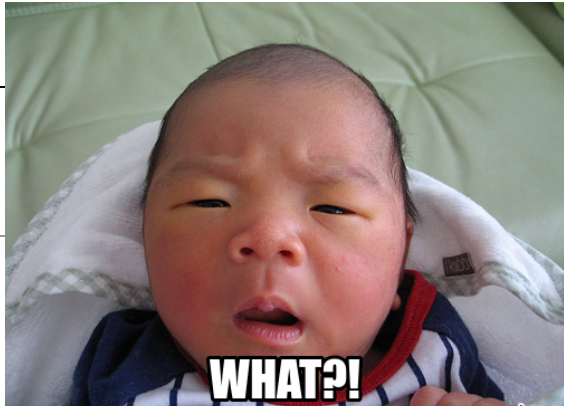


- No serialization
- No copy (data structure shared by both sides)

Attempt 2: get rid of ser/deser, etc

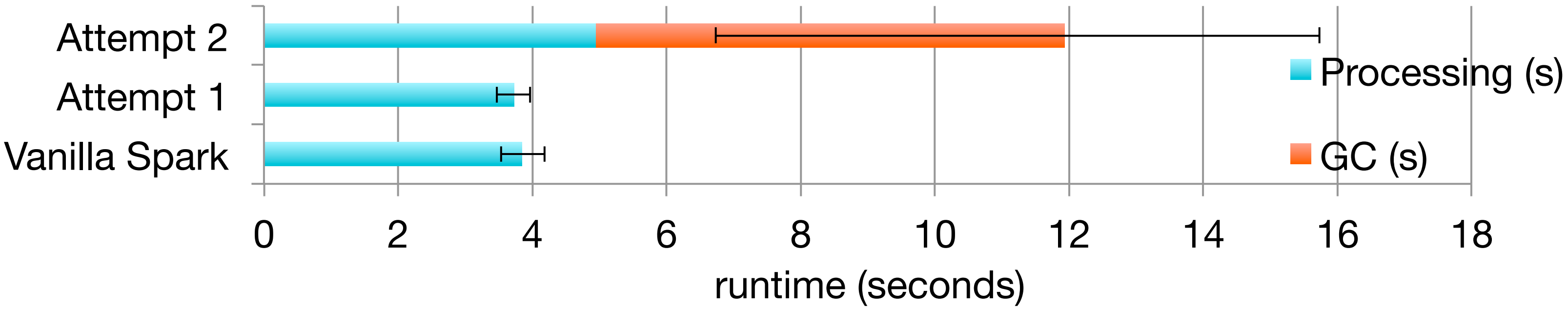
- Attempt 2: create NxM queues (N=mappers, M=reducers)
  push corresponding records into queues





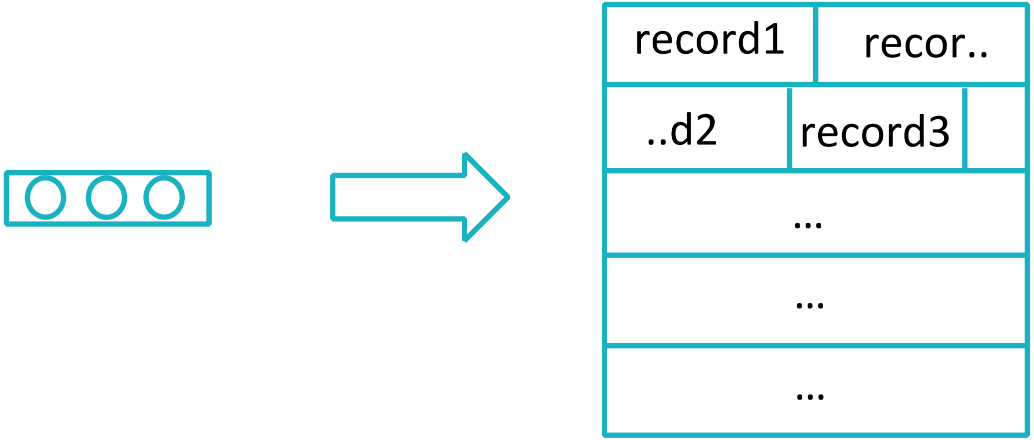
Attempt 2: get rid of ser/deser, etc

- Attempt 2: create NxM queues (N=mappers, M=reducers)
  push corresponding records into queues



Attempt 3: avoiding GC

- Attempt 3: instead of queue, copy records to memory pages



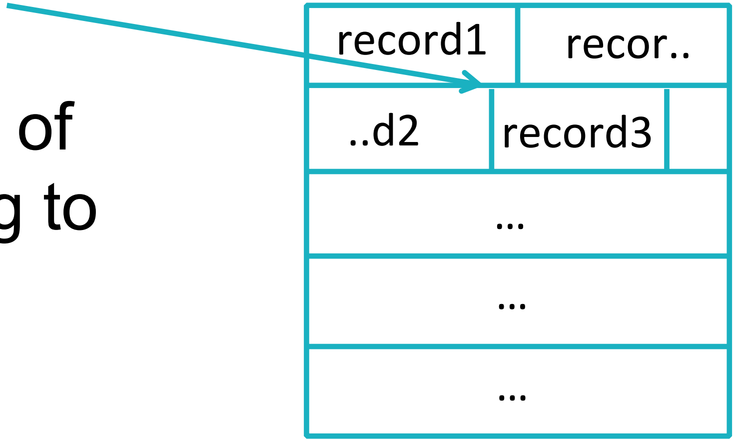
Number of objects: ~records → ~pages

NxM pages, or alternatively, one page per reducer

Spark SQL

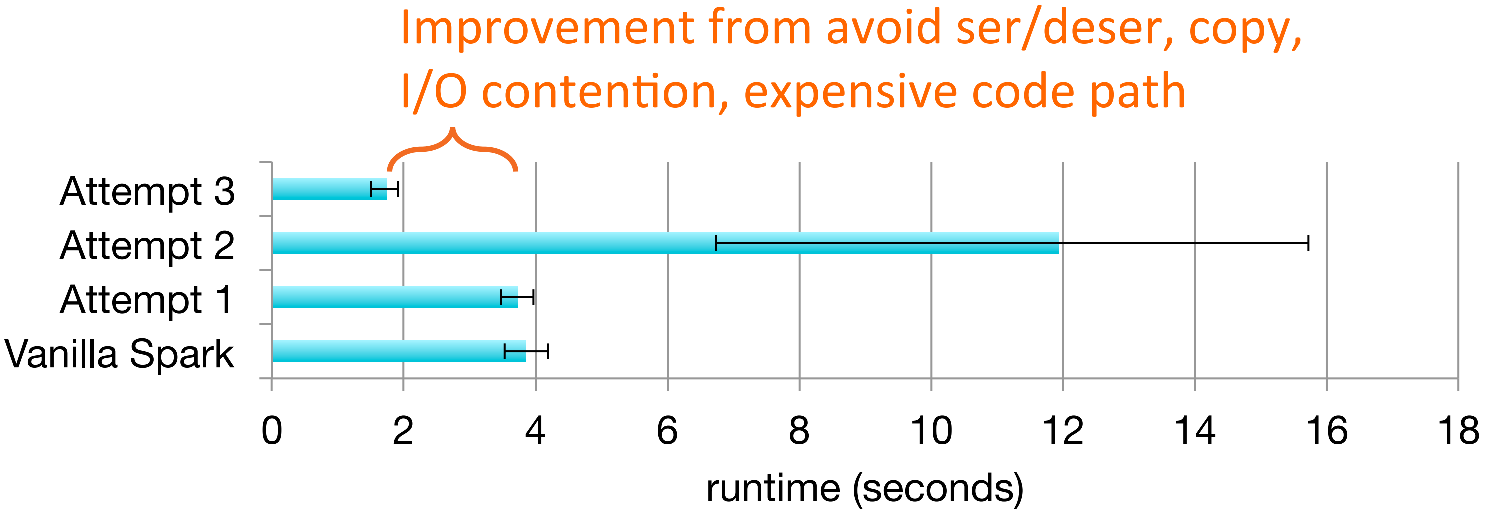
- Unsafe Row (a buffer-backed row format):
- row.pointTo(buffer)

Instantaneous creation of unsafe rows by pointing to different offsets In the page

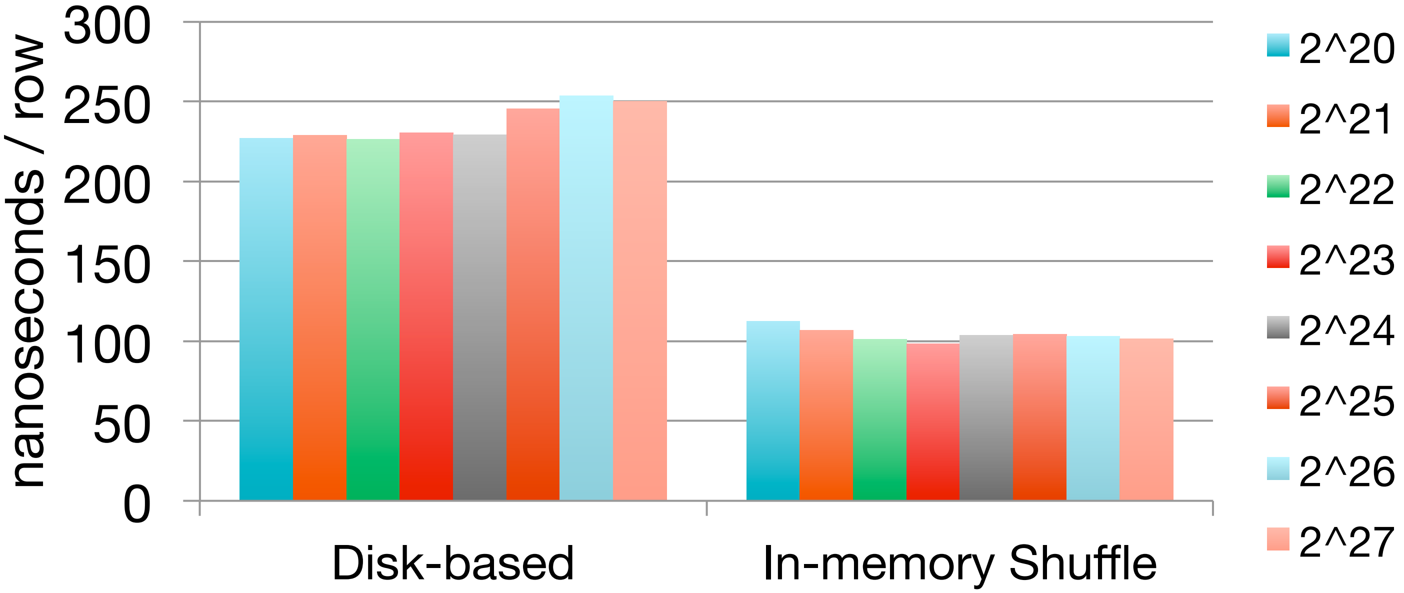


Attempt 3: avoiding GC

- Attempt 3: copy records onto large memory pages

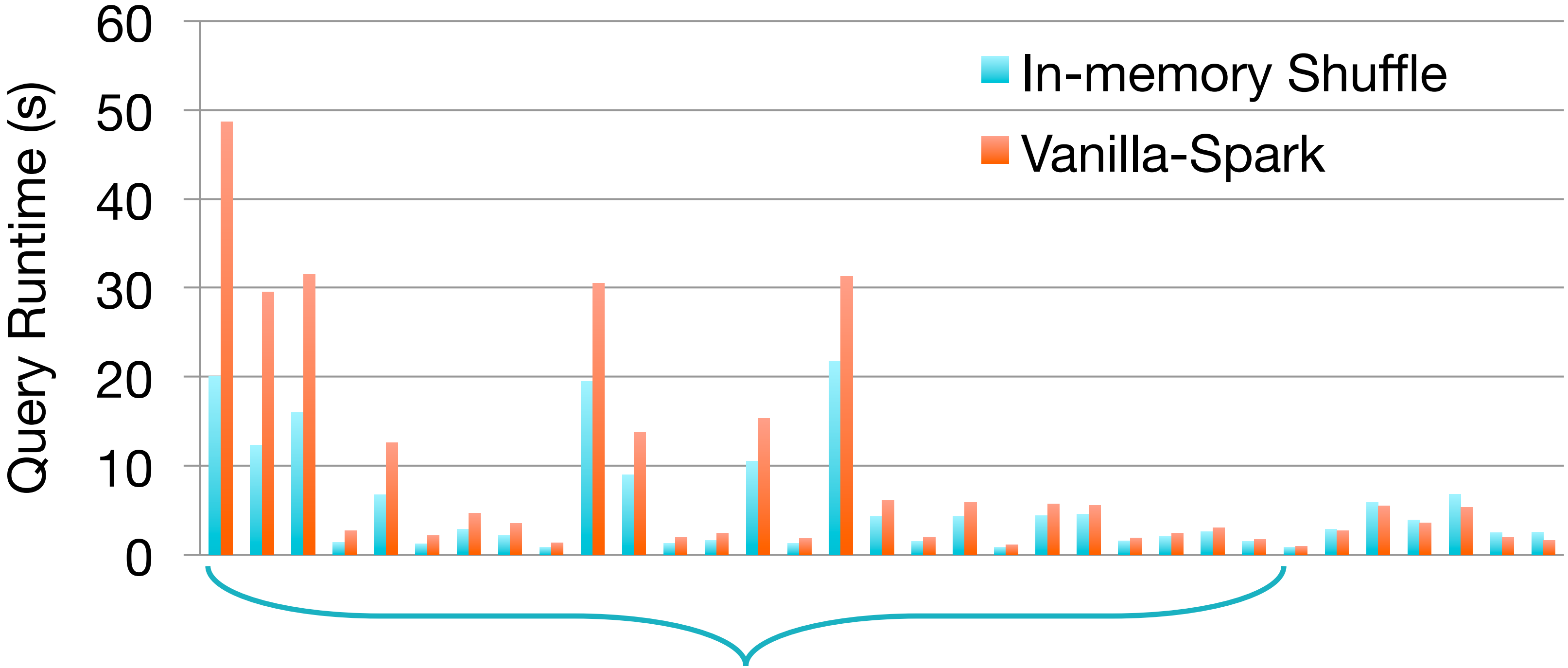


Consistent improvement with varying size



spark.range(N).repartition().selectExpr("sum(id)").collect()
Use N from 2^ 20 to 2^27

TPC-DS performance (single node)



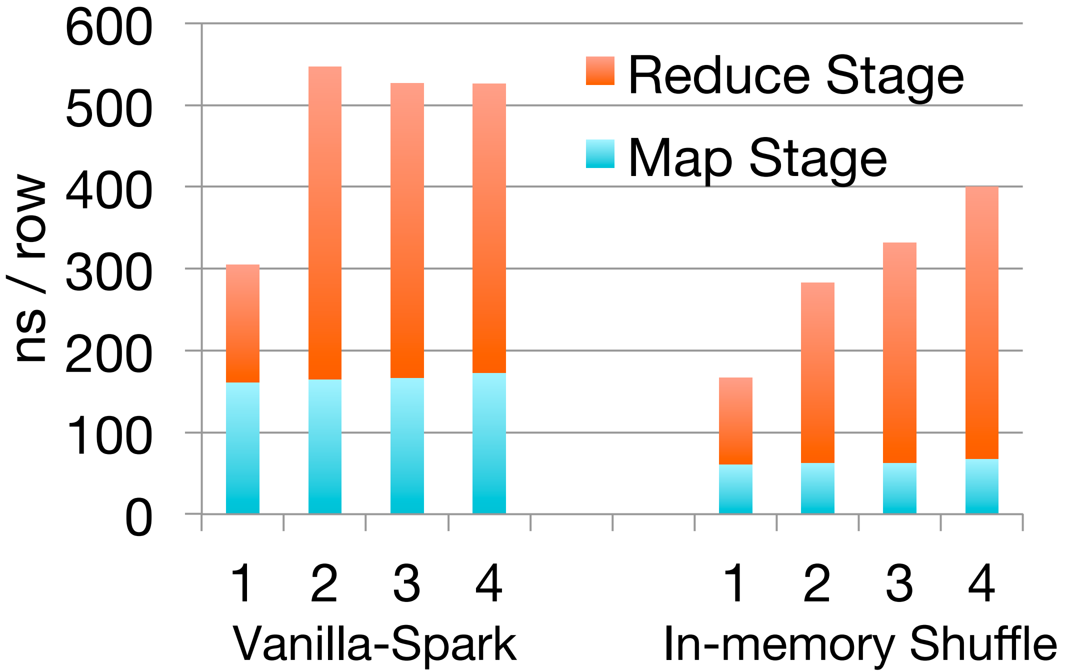
27/33 queries improve with a median of 31%

Extending to multiple nodes

- Implementation
  - All data goes to memory
  - For remote transfer, copy from memory to network buffer

- A more memory-preserving way…
  - Local transfer goes to memory
  - Remote transfer goes to disk
  - Cons1: have to enforce stricter locality on reducers
  - Cons2: cannot avoid I/O contentions

Simple shuffle job



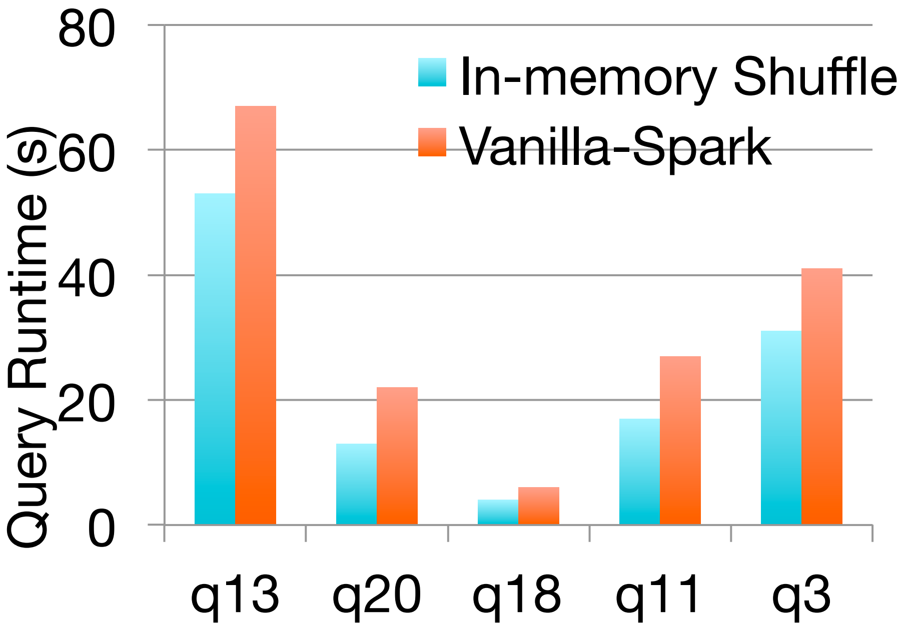
Map:
Consistent improvement

Reduce:
Improvement decreases with more nodes

```
spark.range(N).repartition().selectExpr("sum(id)").collect()
```

TPC-DS performance (x1.xlarge32)

- SF=100
- Pick top 5 queries from single node experiment
- Best of 10 runs



Many other performance bottlenecks need investigation!

Summary

- Spark on many-core requires many architectural changes

- In-memory shuffle
  - How to improve shuffle performance with memory
  - 31% improvement over Spark

- On-going research
  - Identify other performance bottlenecks

Thank you

Qifan Pu
qifan@cs.berkeley.edu