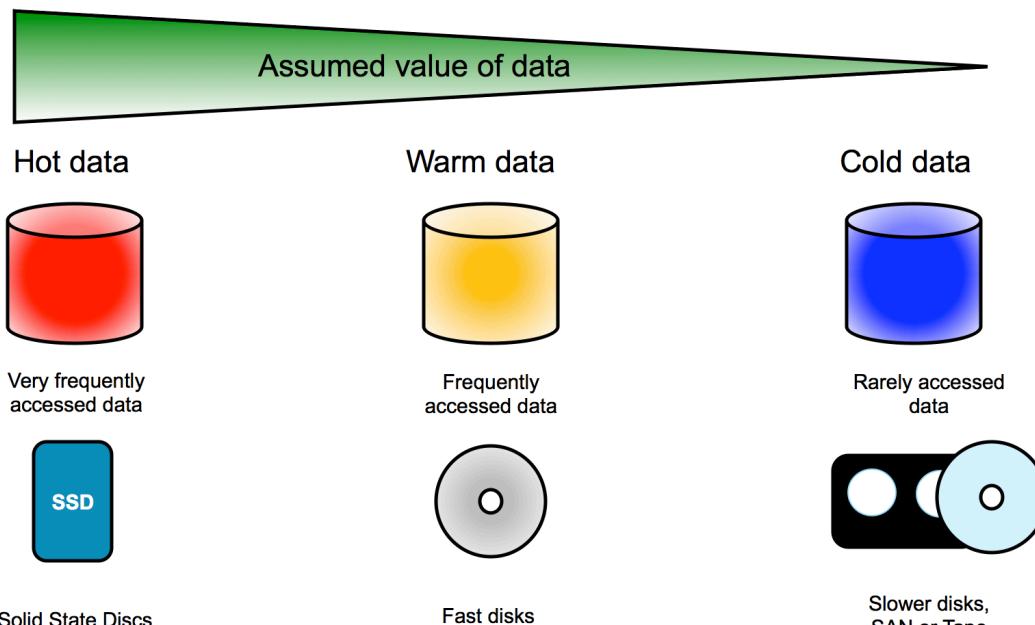


# Cold Storage

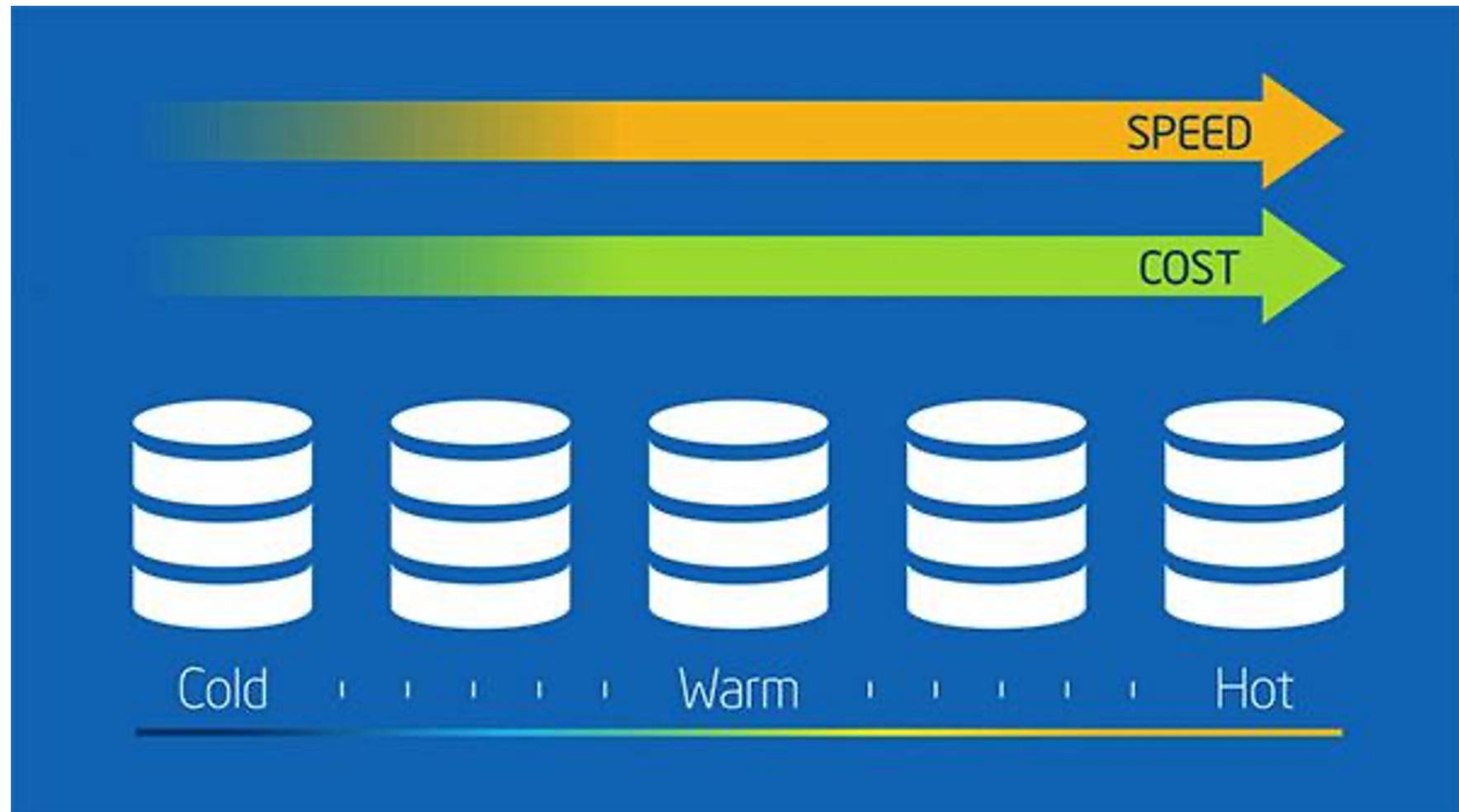
# Data – Size and temperature!

- Data grows in size
- Most of the data gets **COLD**
  - Infrequently accessed
- Objective:
  - High **Performance**
  - Low **Cost**
    - ➔ Price/Performance tradeoff

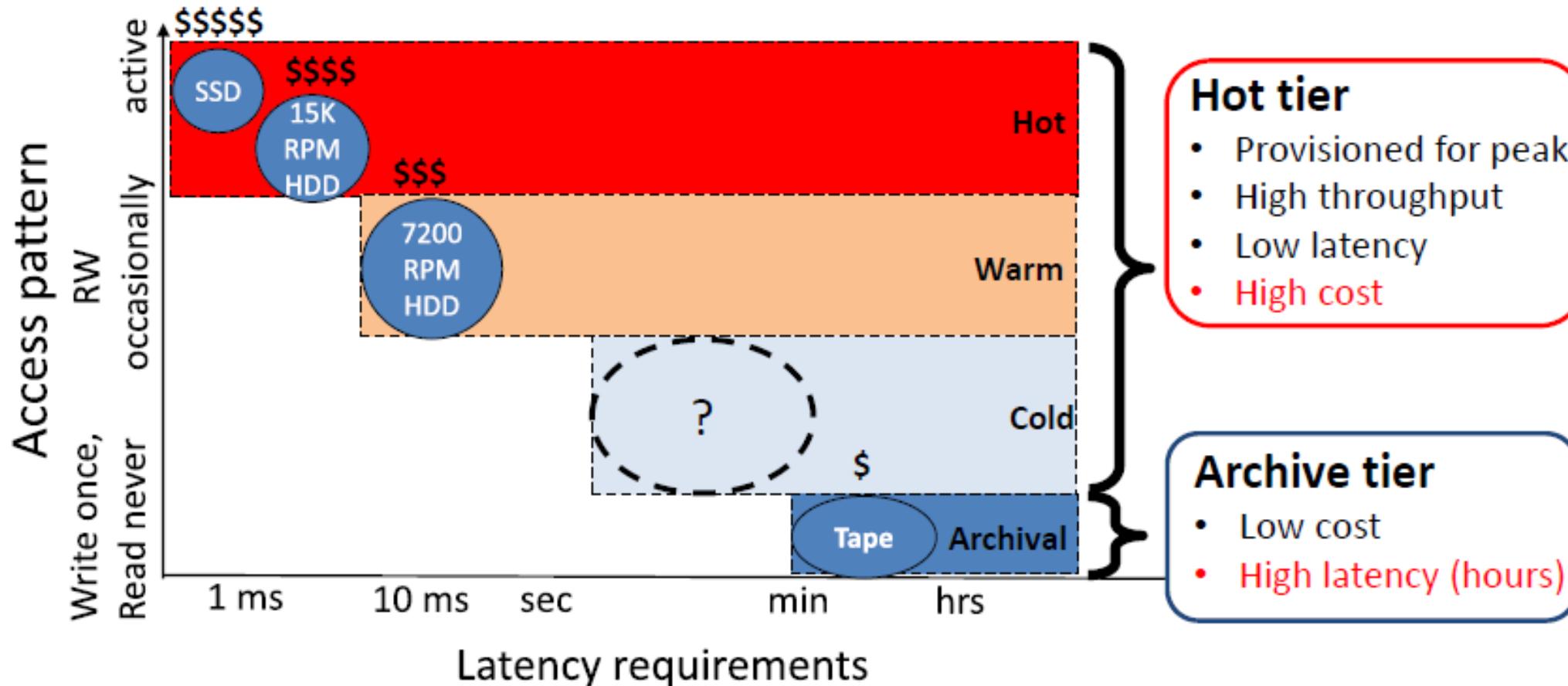


\* Image retrieved from: <https://goo.gl/o7GGcp>

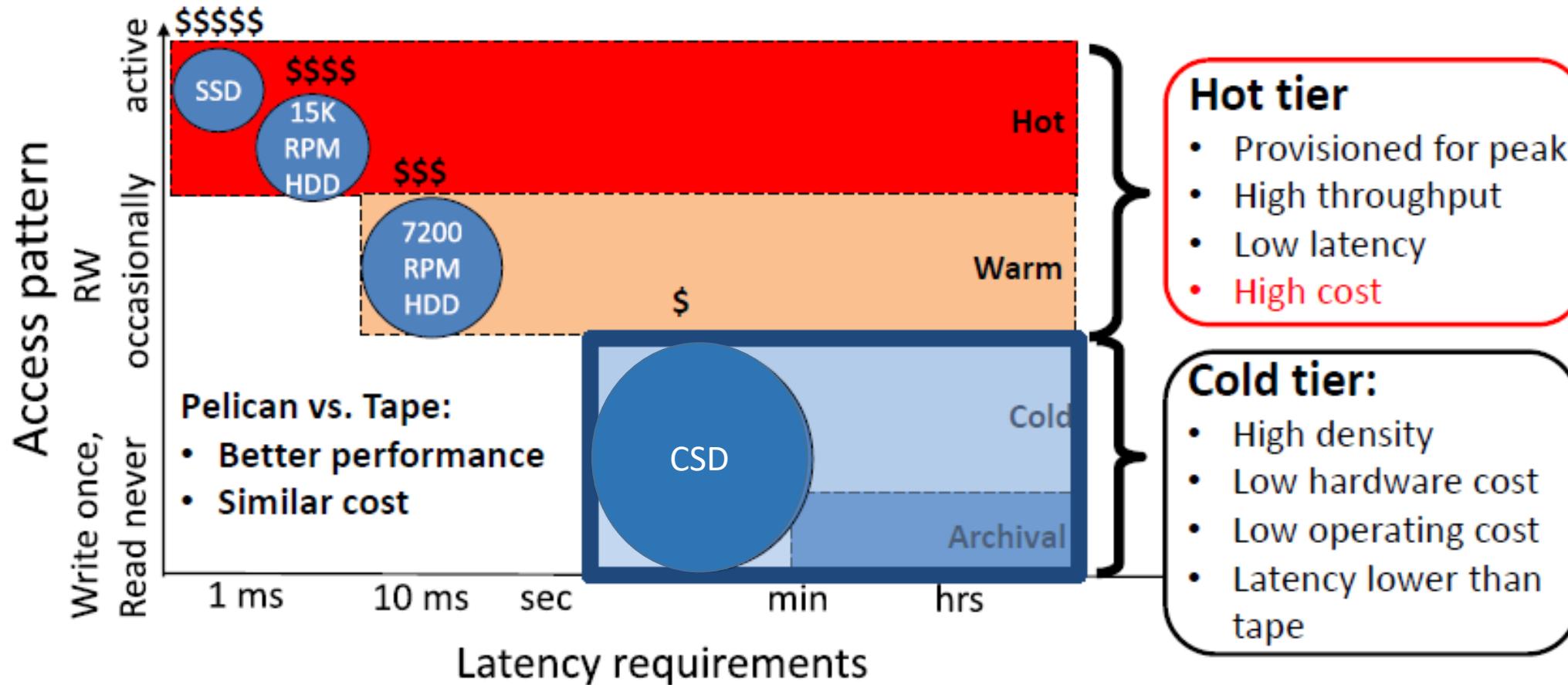
# Data in the cloud



# Price versus Latency



# Price versus Latency



# Adequate Provisioning

Provision resources **just** for the cold data workload:

- Disks:
  - Archival and SMR instead of commodity drives

- Power
- Cooling
- Bandwidth



Enough for the required workload  
**Not to keep all disks spinning**

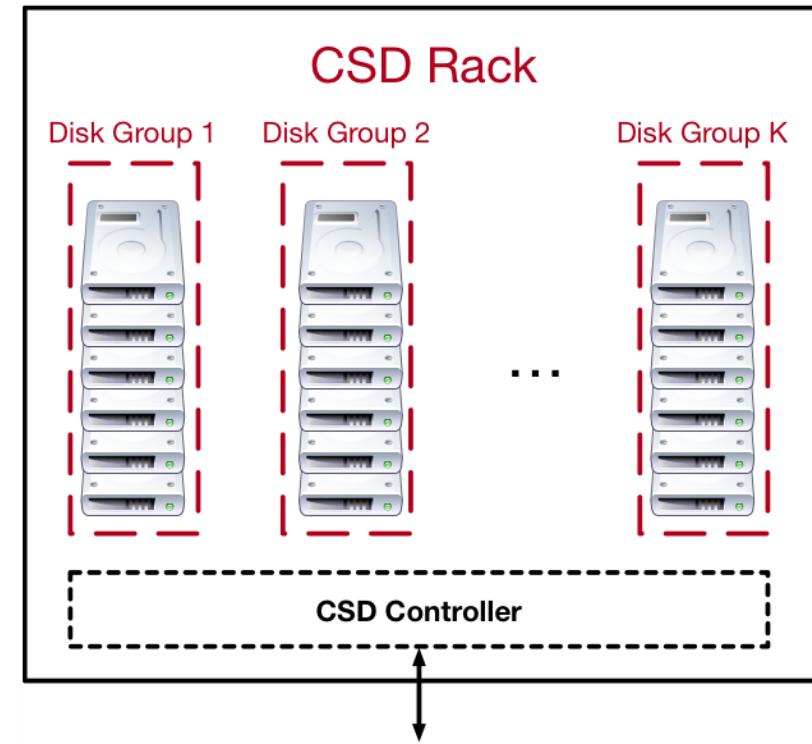
# Advantages

Benefits of removing unnecessary resources:

- High density of storage
- Low hardware cost
- Low operating cost (capped performance)

# Cold Storage Device

- Limited power & cooling facilities
  - Only one disk group is spun up
    - E.g. in Pelican:  $1/K^{\text{th}}$  = 8% of disks are active
  - Disk switch latency: 10-30 seconds
- Example Systems:
  - Microsoft Pelican
  - OpenVault's Knox storage
  - Facebook Cold Storage
  - Amazon Glacier
- Workload: Write Once – Read Occasionally (WORO)



# Pelican

# The Pelican Rack

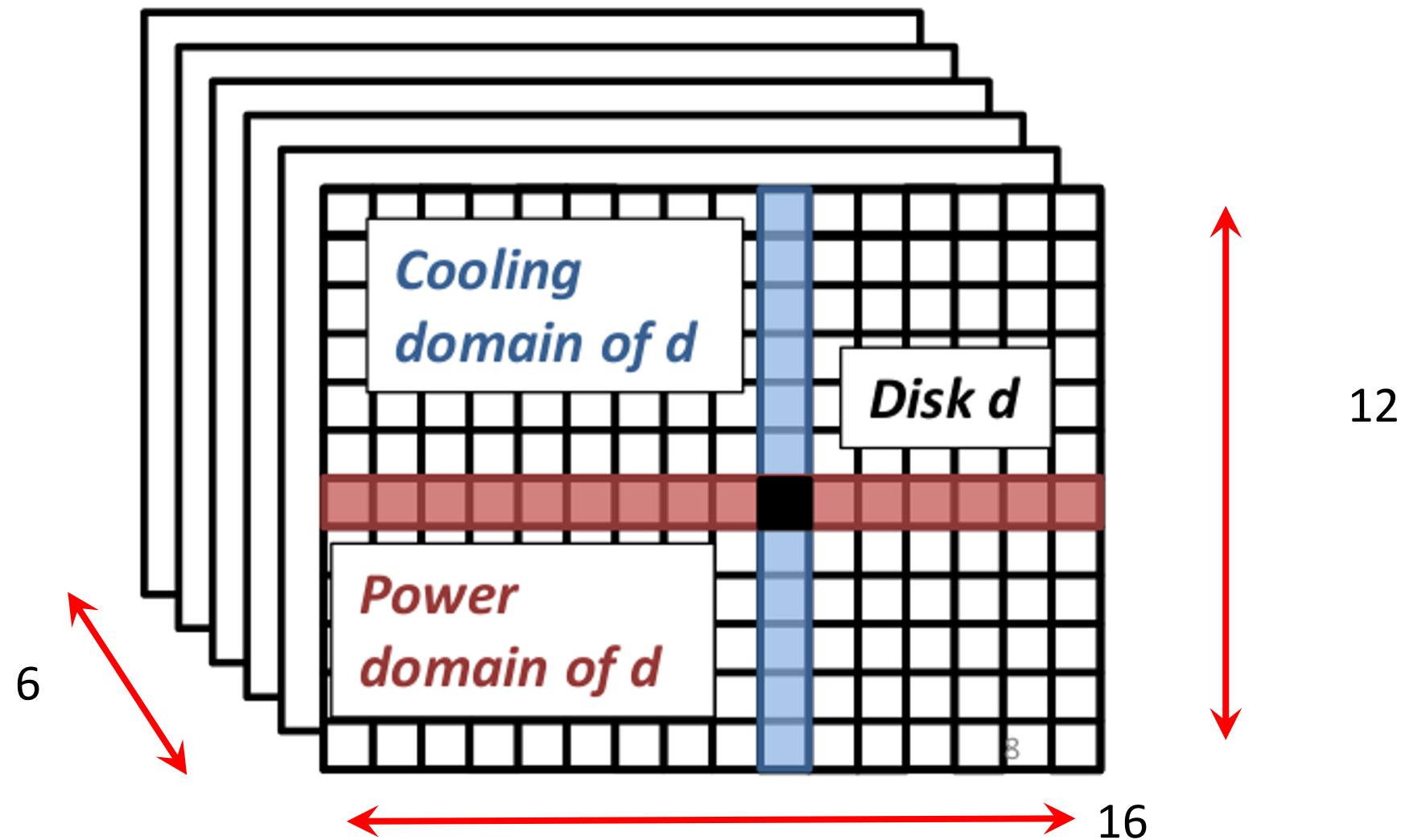
- Mechanical, hardware and storage software stack co-designed.
- Right-provisioned for cold data workload
- 52U rack with 1152 archival class 3.5" SATA disks.
- Average size of 4.55 TB to provide a total of 5 PB of storage
- Only 8% of the disks are spinning concurrently.
- Designed to store blobs which are infrequently accessed.



# Resource Domain

- Each domain is only provisioned to supply its resource to a subset of disks
- Each disk uses resources from a set of resource domains
- ***Domain-conflicting*** - Disks that are in the same resource domain.
- ***Domain-disjoint*** - Disks that share no common resource domains.
- Pelican domains
  - Cooling, Power, Bandwidth

# Schematic representation

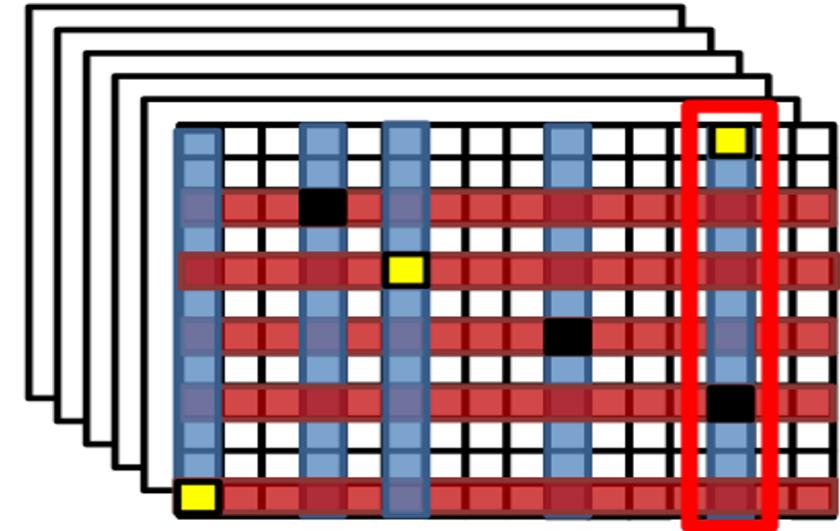


# Data Layout

- Objective - maximize number of requests that can be concurrently serviced while operating within constraints.
- Each blob is stored over a set of disks.
- It is split into a sequence of 128kB fragments. For each “k” fragments, additional “r” fragments are generated.
- The  $k+r$  fragments form a stripe
- In Pelican they statically partition disks into groups and disks within a group can be concurrently active. Thus they concentrate all conflicts over a few sets of disks.

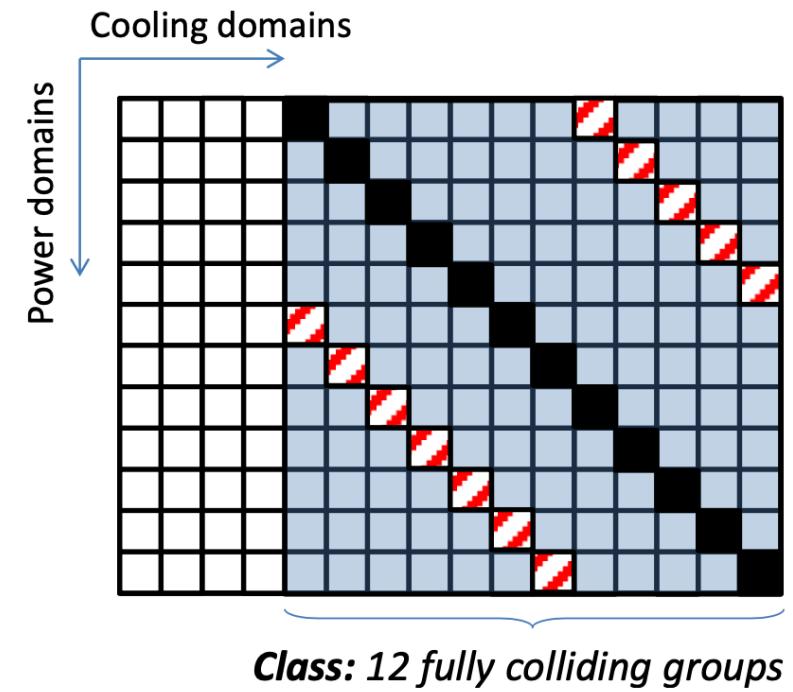
# Data Placement

First approach: random placement



- Disks of blob 1
- Disks of blob 2

Rack: 3D array of disks



# IO Scheduler

- Traditional disks are optimized to reorder IOs to minimize seek latency.
- Pelican - reorder requests in order to minimize the impact of spin up latency.
- Four independent schedulers. Each scheduler services requests for its class and reordering happens at a class level.
- Each Scheduler uses two queues – one for reorder operations and one for other operations.
- Reordering is done to amortize the group spin up latency over the set of operations.
- Rate limiting is done to manage the interference between reorder and other operations.

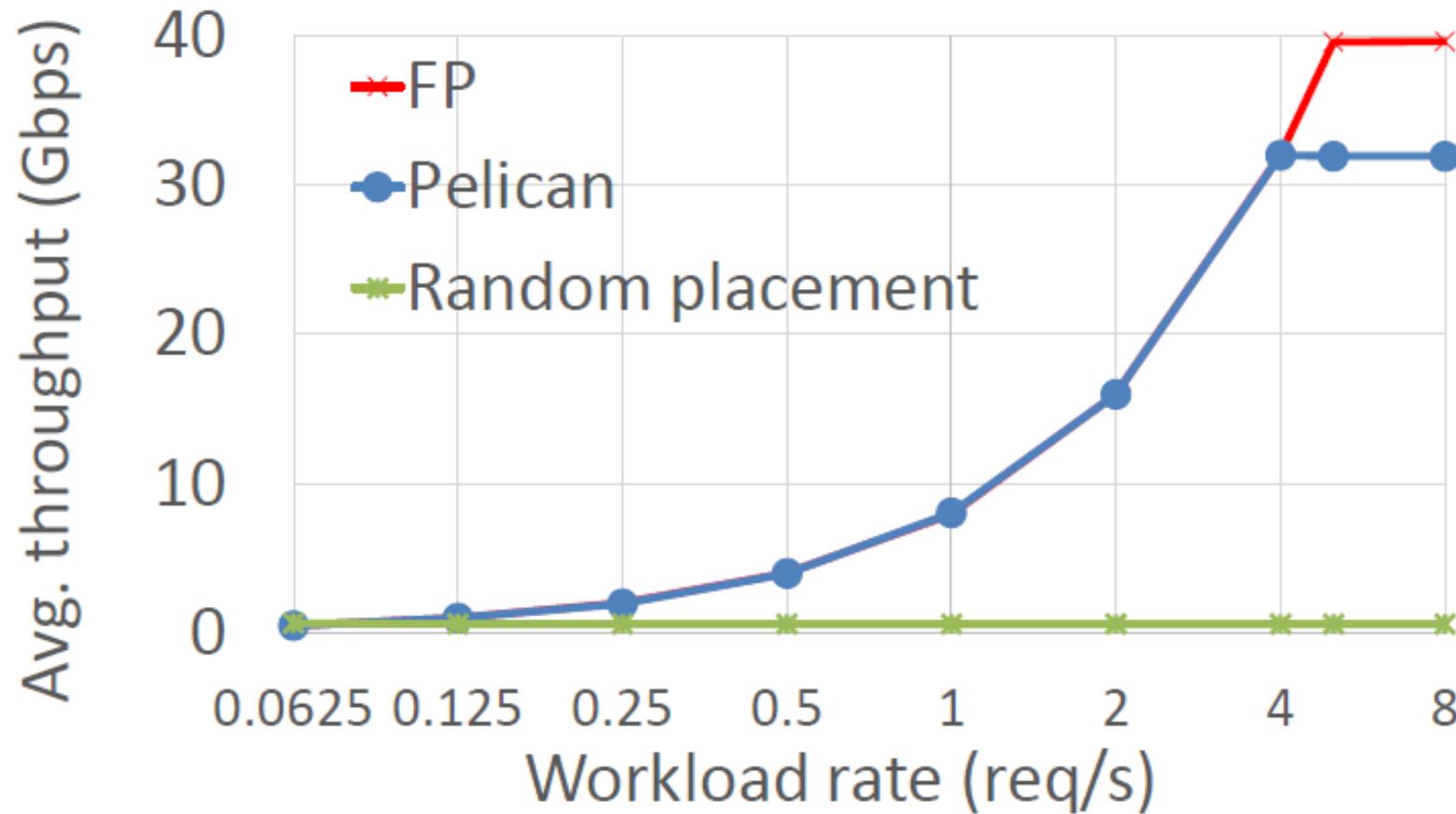
# Implementation Considerations

- Power Up in Standby is used to ensure disks do not spin without the Pelican storage stack managing the constraints.
- Group abstraction is exploited to parallelize the boot process.
  - Initialization is done on the group level.
  - If there are no user requests, then all 4 groups are concurrently initialized
  - Takes around 3 minutes to initialize the entire rack.

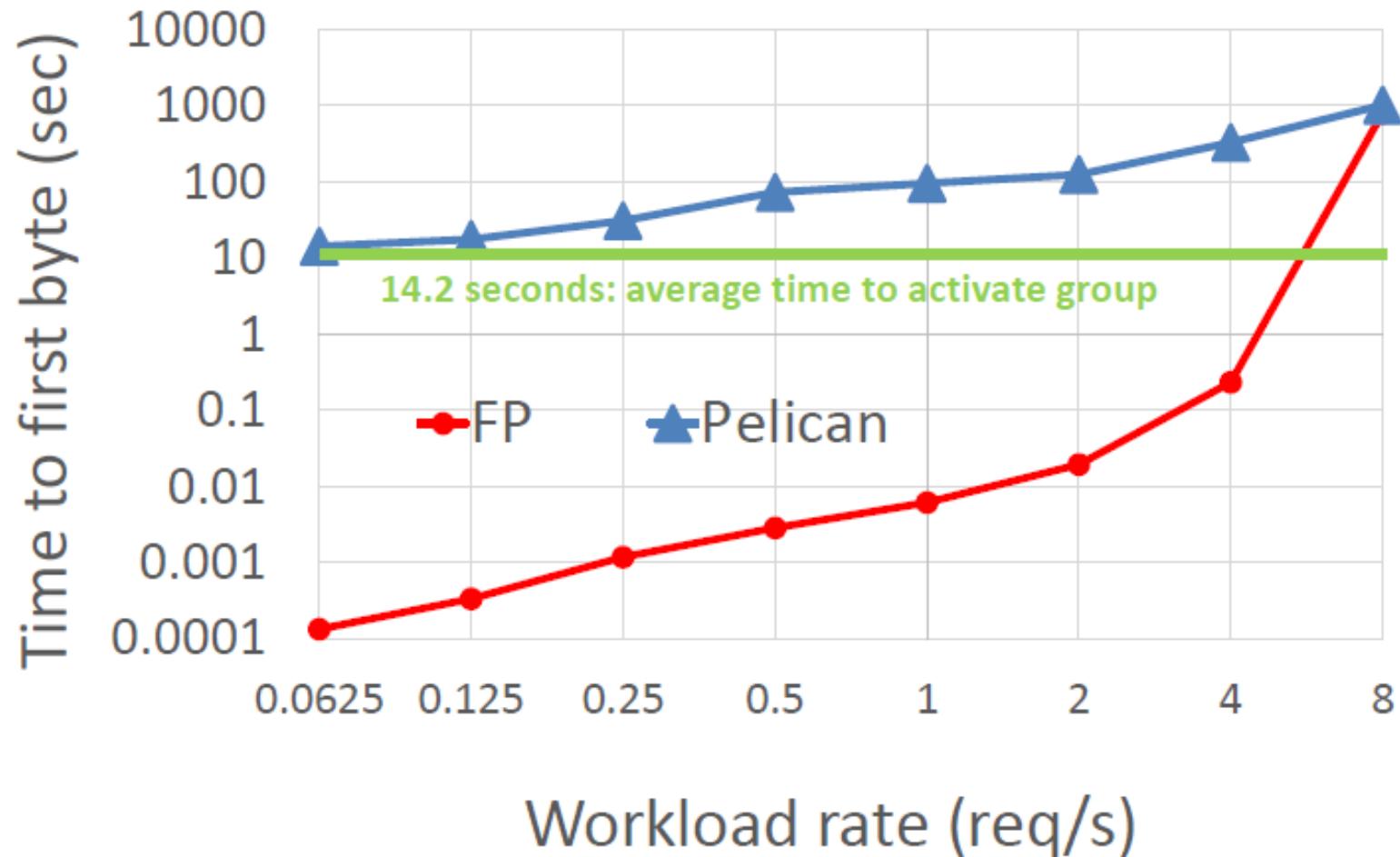
# Evaluation

- Comparison against a system organized like Pelican but with full provisioning for power and cooling.
- The FP uses the same physical internal topology but disks are *never spun down*.

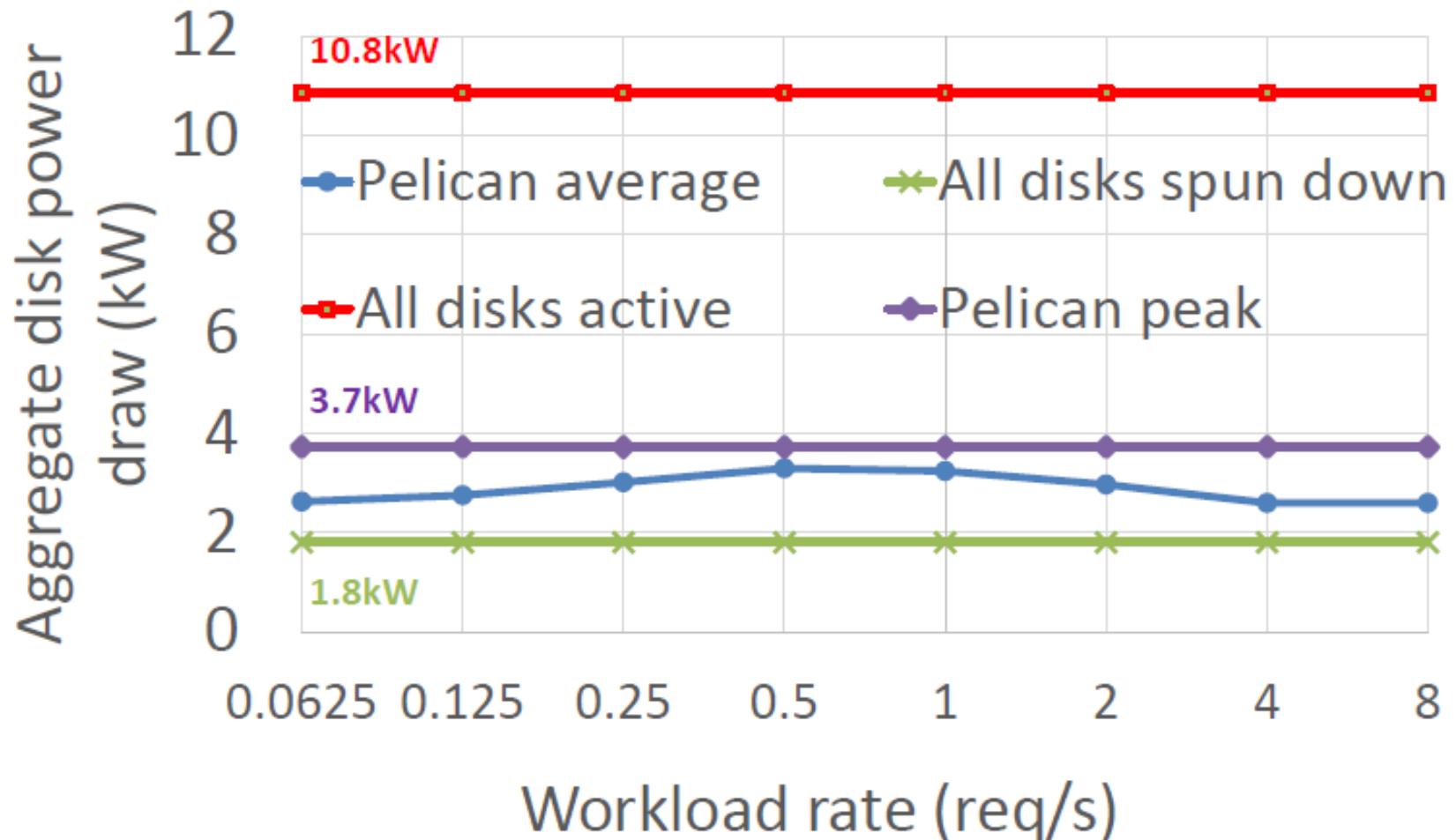
# Performance – Rack Throughput



# Time to first byte



# Power Consumption



# Pros

- Reduced
  - Cost
  - Power Consumption
- Erasure codes for fault tolerance
- Hardware abstraction simplifies IO schedulers work.

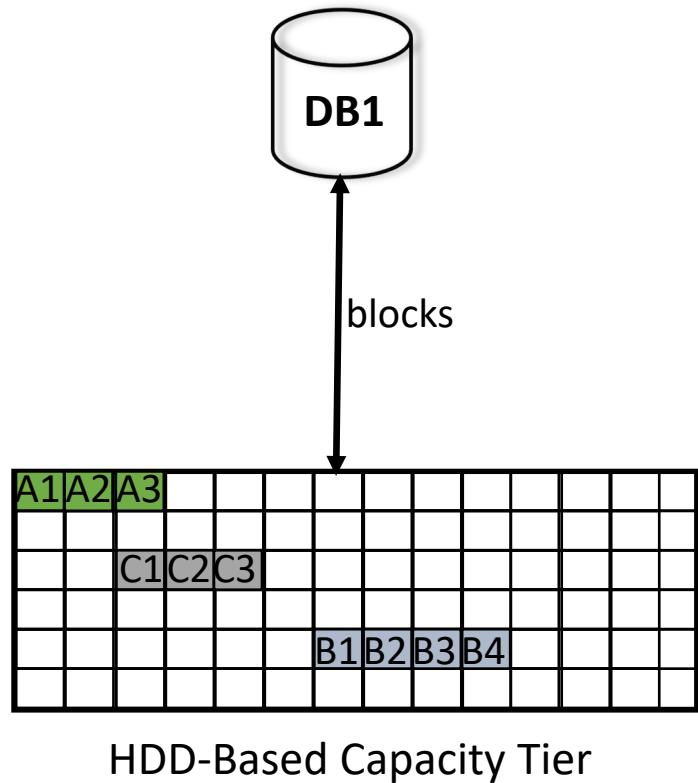
# Cons

- Tight constraints – less flexible to changes
- Sensitive to hardware changes
- No justification to some of the configuration decisions made.
- Not sure if it is an optimal design

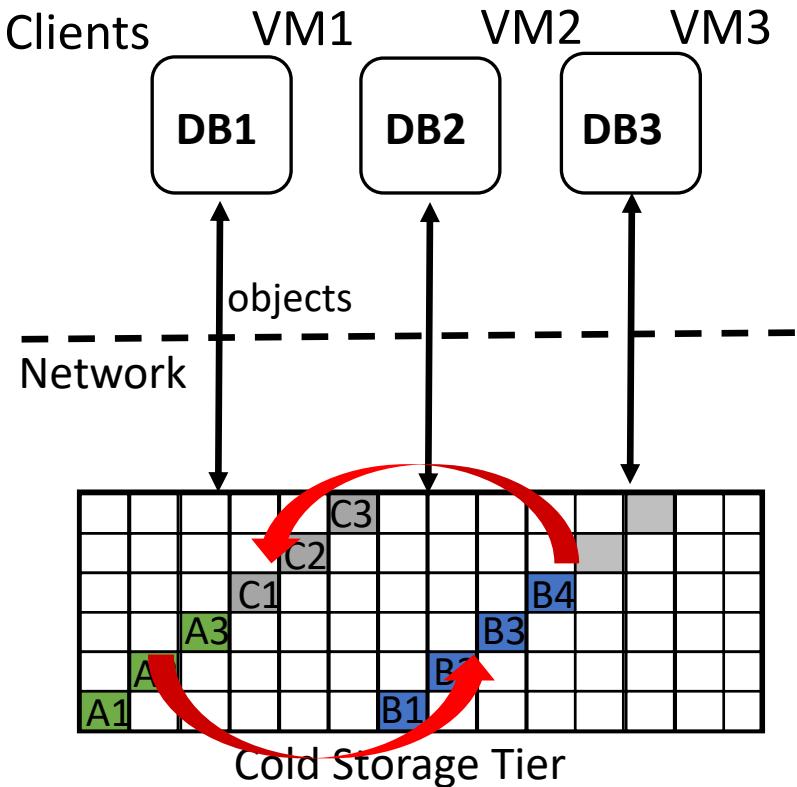
# Data Processing

# Query execution over CSD

Traditional setting



Virtualized enterprise data center

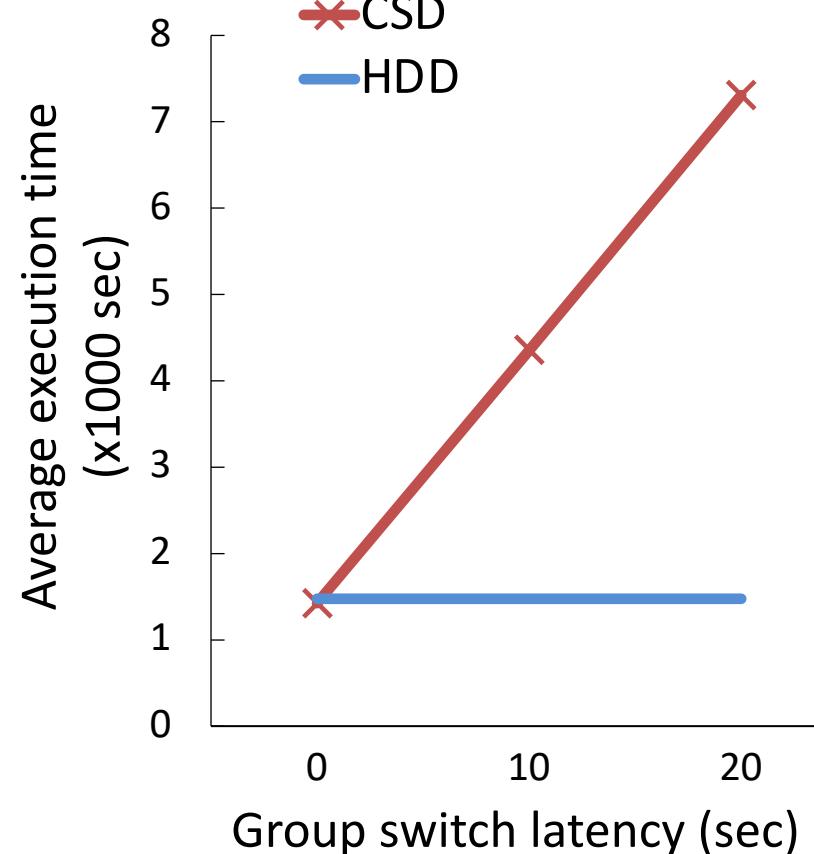
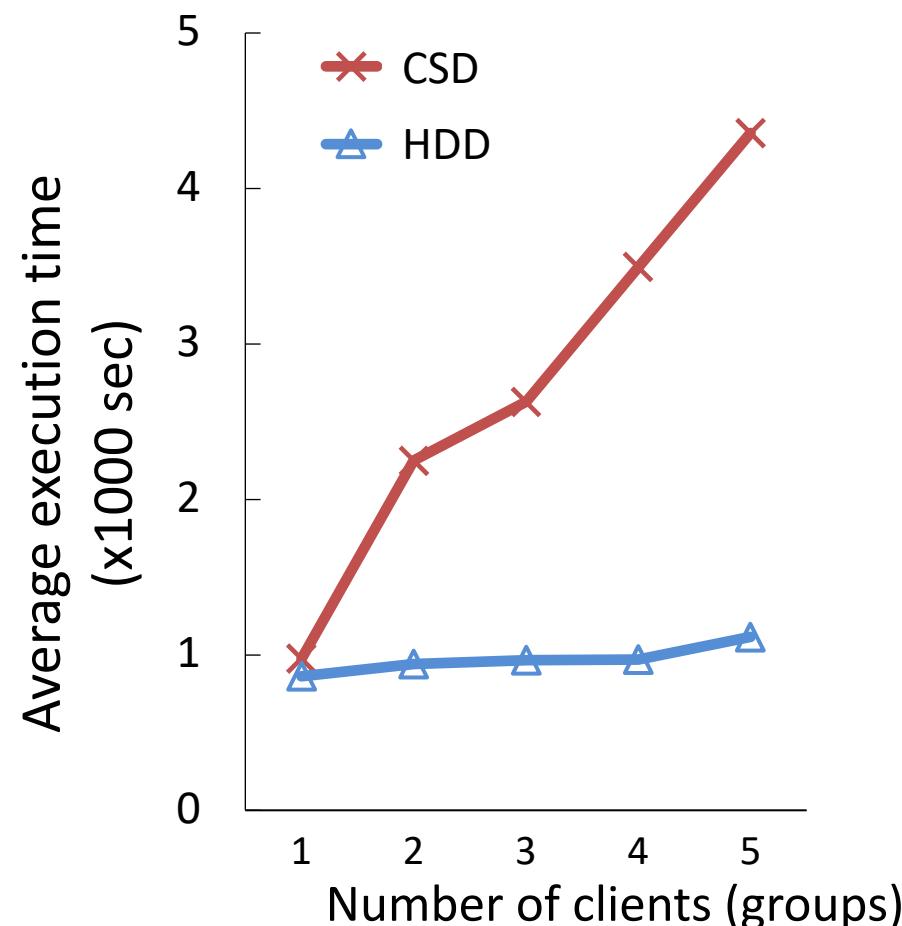


Uniform access ✓   Control layout ✓  
Static (pull-based) execution ✓

Uniform access ✗   Control layout ✗

# What this means for an enterprise datacenter...

**Setting:** multitenant enterprise datacenter, clients: PostgreSQL , TPCH 50, Q12, CSD: shared, layout: one client per group

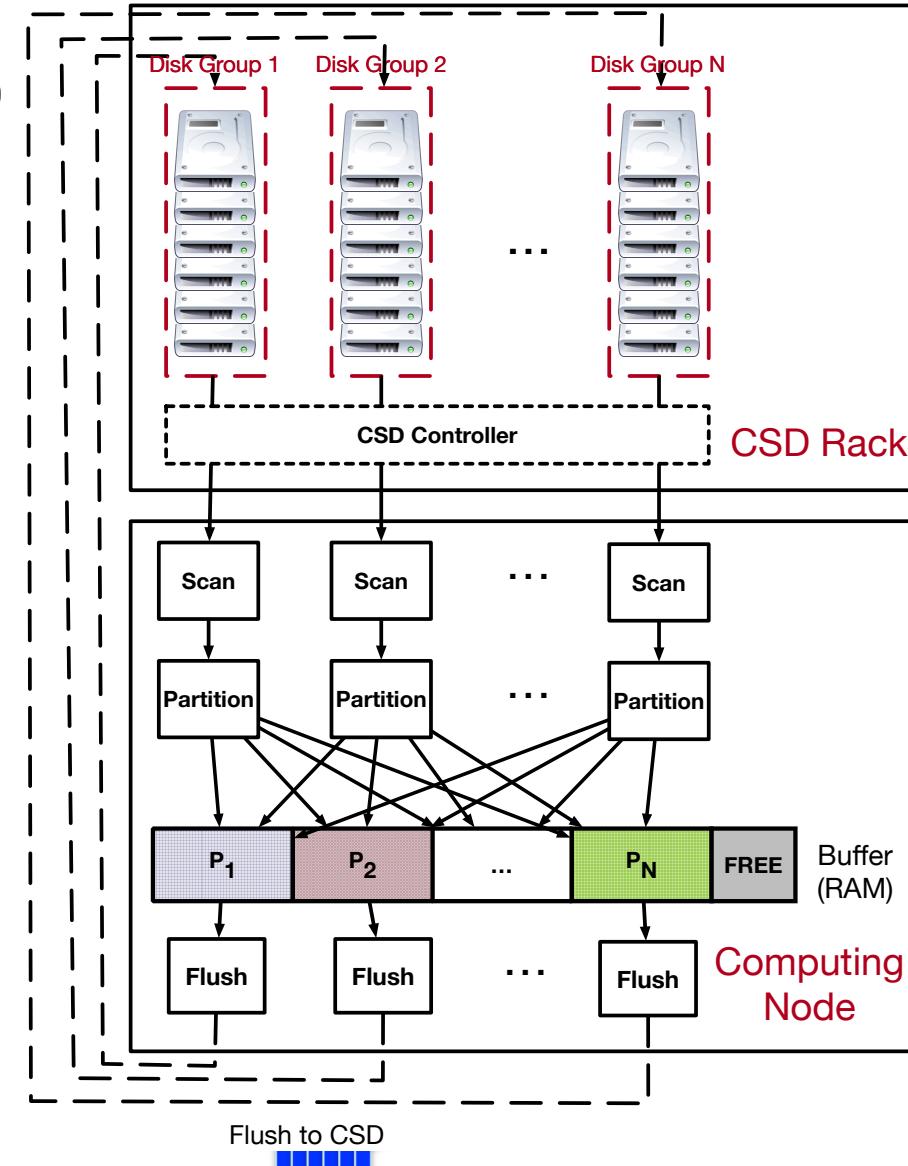


# Need hardware-software Co-design

1. Data access has to be **hardware-driven** to minimize group switches
2. Query execution engine has to process data pushed from storage in **out-of-order** (unpredictable) manner
3. Reduce data round-trips to cold storage by **smart data caching**

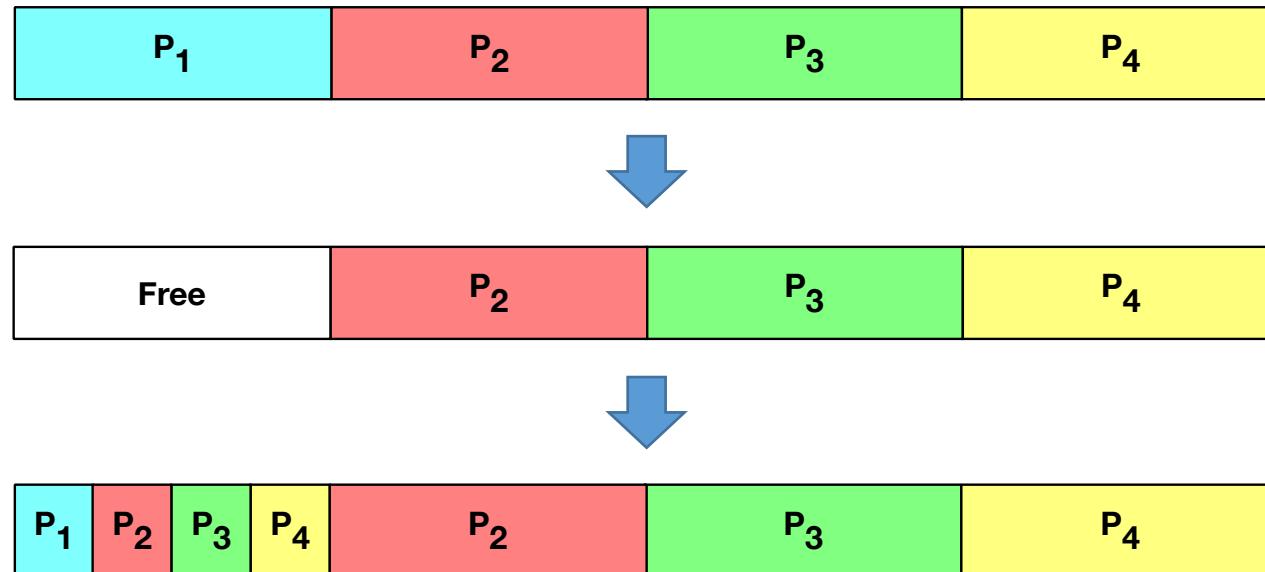
# Batch Processing on CSD

- Common batch processes on cold data:
  - Massive-scale Group-by / Join
  - [Near]-duplicate detection
  - Data Localization
  - In-place Map-Reduce
- Data Partitioning
  - Partition items into K groups
    - Distribute between K disk groups
  - **group\_ID = Partitioner(element)**
  - Various Partitioners:
    - Hash, Range, map(), etc.



# Flushing

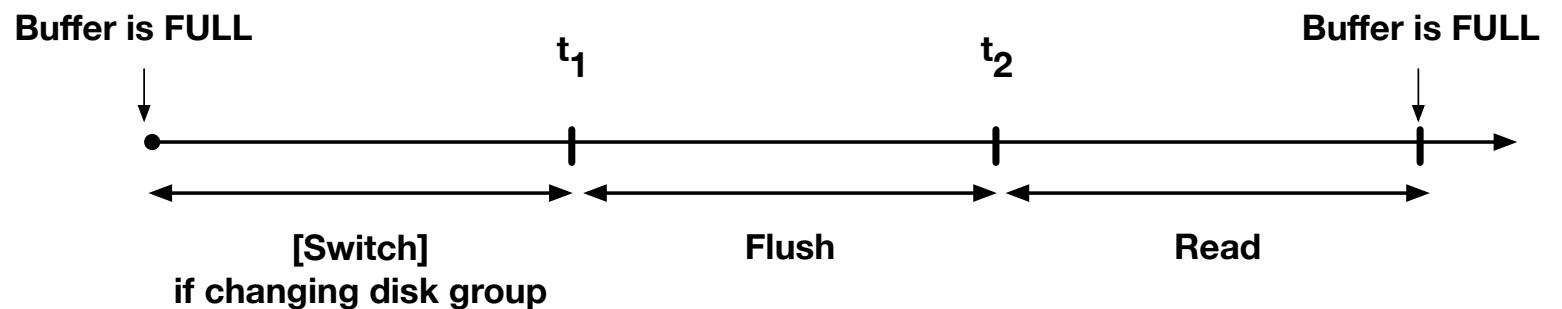
- Buffer is full → flush → into which disk group?



- Naïve approach: Many switches

# Buff-Pack

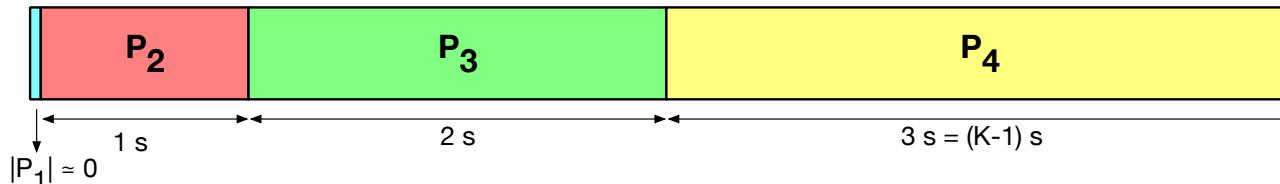
- Greedy approach:



- Try to maximize **throughput (GB/sec)** in the next step(s)

- Intuition:

- Flush into the **current disk group** to avoid switching, if possible.
- Otherwise: switch to disk group with the **largest buffer**



# Off-Pack

- Intuition:

- Available buffer plays a key role
  - Flush the **entire** buffer
  - Write-Offloading: Write data to the '**wrong**' disk group → then transfer.
- When buffer is full (active partition = i):
  - Flush **buffer[i]** into the active disk group
  - For all.  $j \neq i$  flush **buffer[i]** into **offload\_buffer\_i\_j**
- Post processing: move all **offload\_buffer\_i\_j** to disk group j

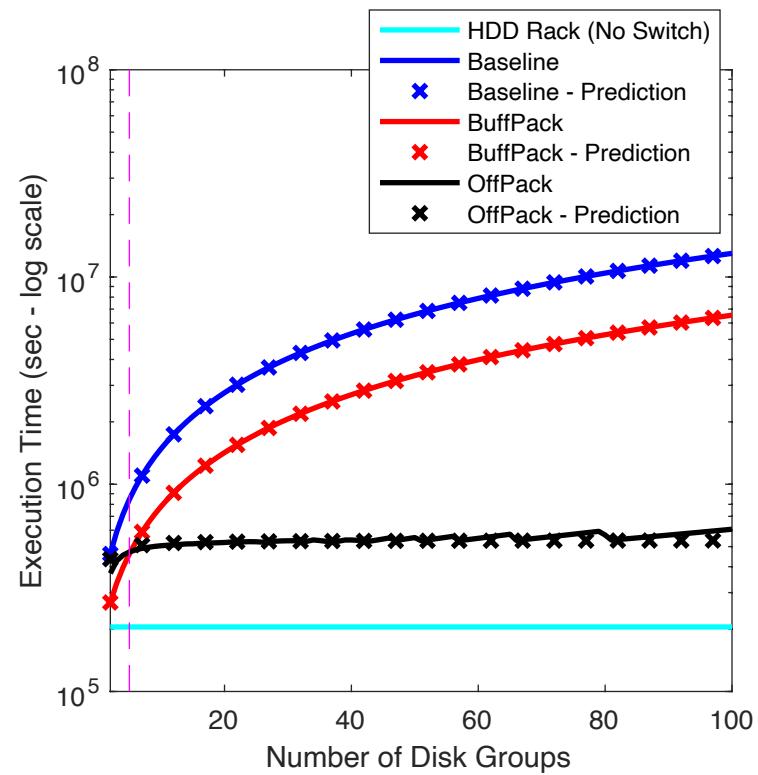
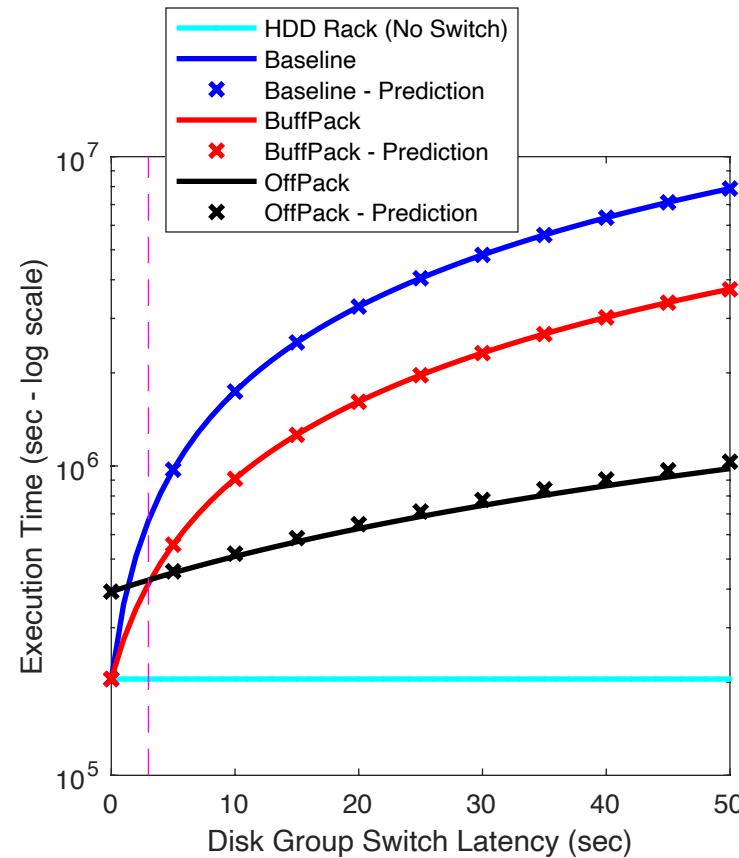
# Which algorithm? - Analytical Model

- Estimates for the **number of disk group switches** and **computation time** of each algorithm

$$T_{total} = T_{switch} + T_{seek} + T_{read} + T_{write}$$

- Intuition:
  - Off-Pack: Fewer switches, more read/write
  - Off-Pack is better for a
    - Smaller **buffer** ↓
    - Higher **# disk groups** ↑
    - Higher **throughput** ↑

# Experiments – CSD Spec.



Parameter	Value
Buffer size	8 GB
# Disk groups	12
Dataset size	100 TB
Throughput	1 GB/sec
Group switch latency	10 sec

## Configuration

Parameter	Value
Buffer size	8 GB
# Disk groups	12
Dataset size	100 TB
Throughput	1 GB/sec
Group switch latency	10 sec

# Experiments – Buffer & Data

