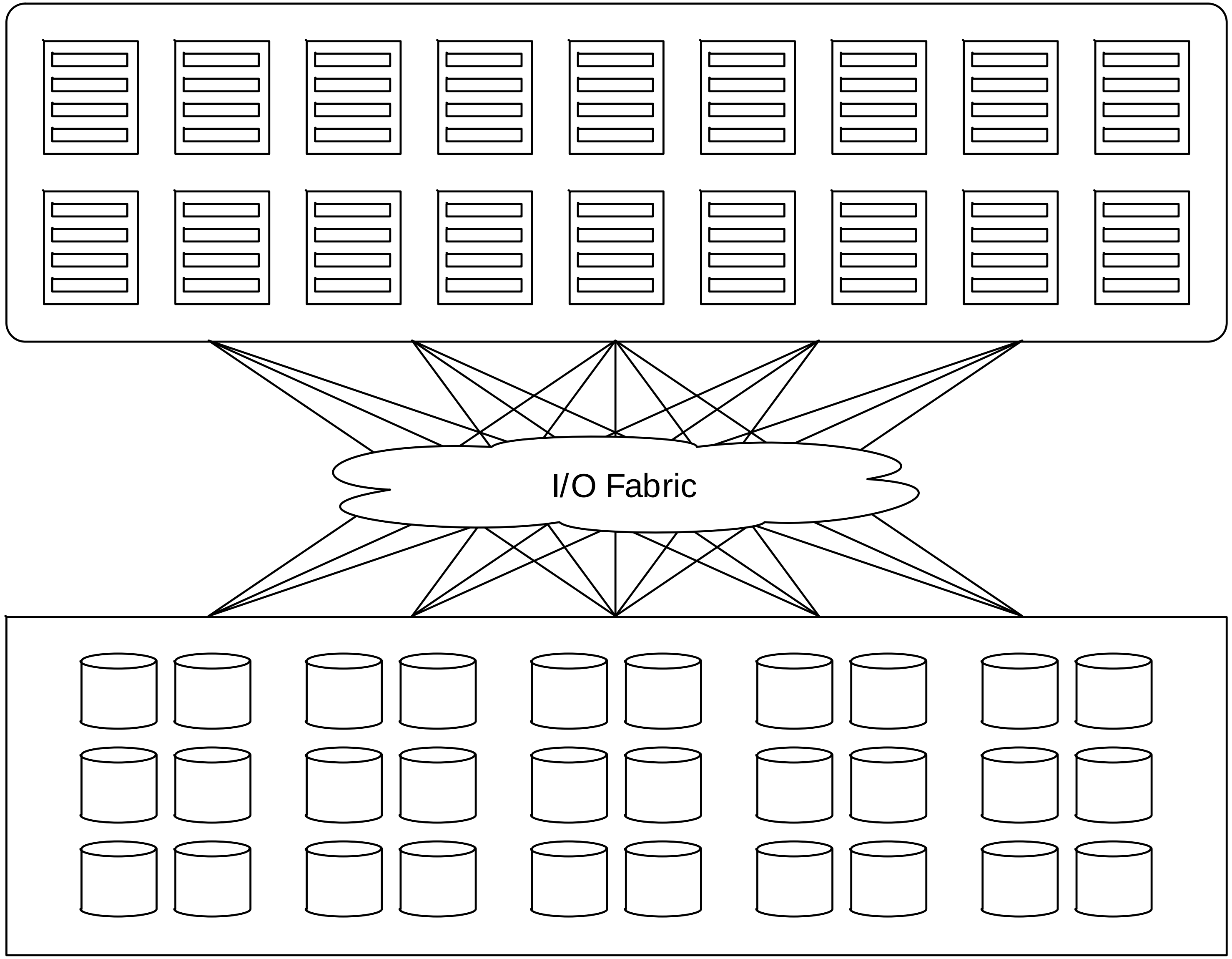
Backtesting with Spark

Patrick Angeles, Cloudera
Sandy Ryza, Cloudera
Rick Carlin, Intel
Sheetal Parade, Intel

Traditional Grid

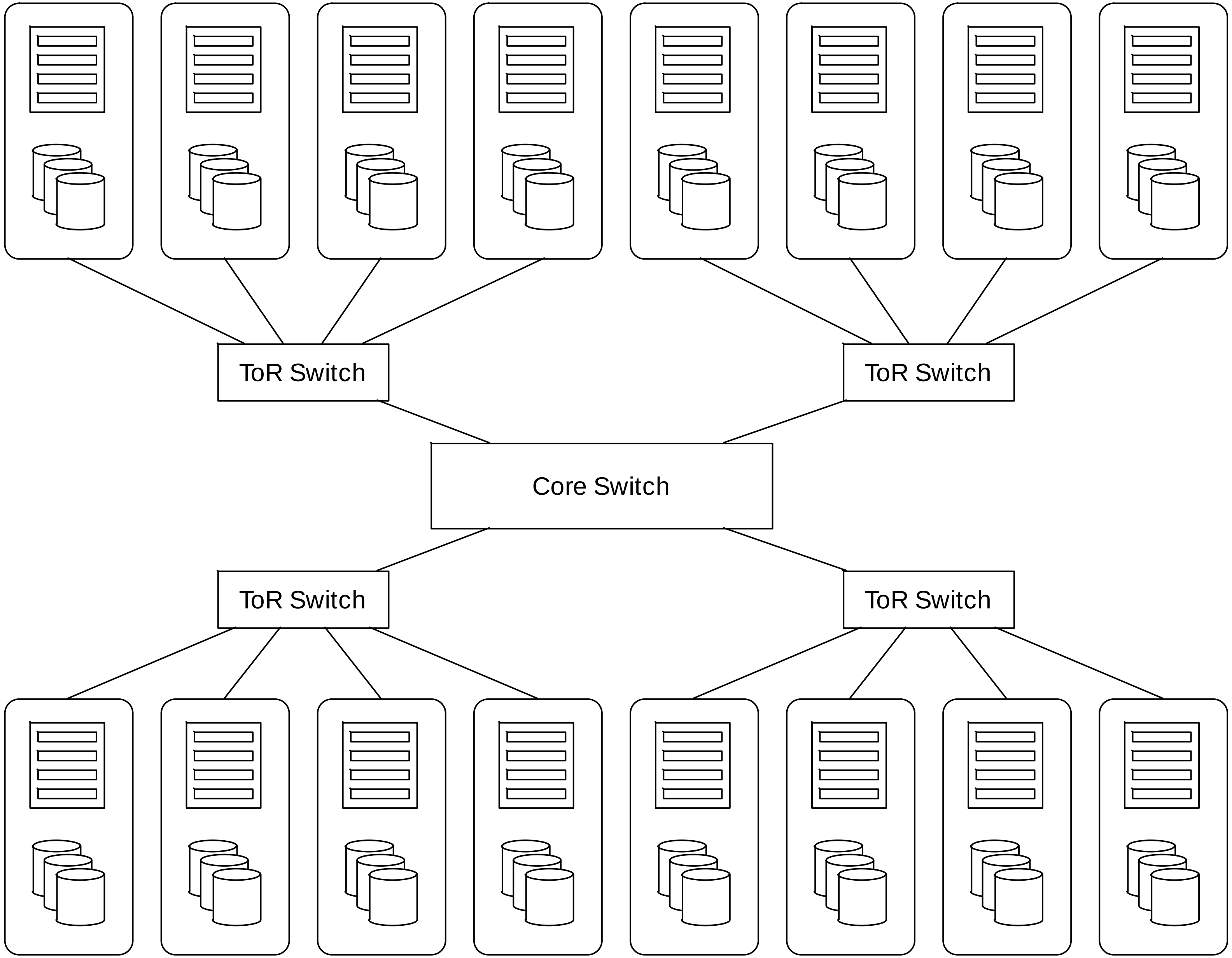
- Shared storage
- Storage and compute scale independently
- Bottleneck on I/O fabric
- Typically exotic hardware
- Proprietary schedulers
- Homegrown application frameworks



cloudera | intel

Hadoop Cluster

- Shared nothing
- Storage and compute scale together
- Hierarchical architecture minimizes data transfer
- Typically commodity hardware
- Open source scheduler and frameworks



Implementation Choices

```
import datetime

class Issue():
    """TODO write docs here"""
    def __init__(self, **kwargs):
        # TODO: Validate input
        self.__dict__.update(kwargs)

    def publish(self):
        return ('This is the {0.pubdate:%B
                'It is {0.pages:,} pages lo
                'costs ${0.price:.5}. '
                'It is about {0.subject}.')
```

Storage Engine

- HBase
- HDFS + CSV
- HDFS + Parquet
- HDFS + AvroFile

Processing Language

- SQL
- Native: C / C++
- JVM: Java, Scala
- Python

Processing Framework

- Hive / Impala
- MR4C
- MapReduce
- Spark

cloudera | intel

Spark in 60 Seconds

- Started in 2009 by Matei Zaharia at UC Berkeley AMPLab
- Advancements over MapReduce:
  - DAG engine
  - Takes advantage of system memory
  - Optimistic fault tolerance using lineage
  - 10x – 100x faster
- Supports applications written in Scala, Java and Python
- Rich ecosystem: SparkStreaming, SparkR, SparkSQL, MLLib, GraphX
- Strong community: ~40 committers, 100s of contributors, multi-vendor support

BLASH: Algorithm Implementation

- Data layout: one file per symbol for a year.
- Pipelining to avoid re-reading the data.
  - Process order book for all symbols.
  - Sort and filter results in the end.
- Unit Testing
  - Separation of concerns – parallelization from algorithm.
  - Automated verification for correctness.
- Optimizations
  - Use trending to reduce expensive method invocation.
  - Keep memory in check – process an order at a time.
- ~2 weeks effort. Includes coding, data generation and running benchmarks.

Setup

Hardware

- 1 master, 1 mgmt node
- 12 workers
    - 2 x E5-2695 v2 @ 2.40GHz
    - 24 physical cores
    - 96GB RAM
    - 8 x 1TB SAS drives
    - 10Gb Ethernet

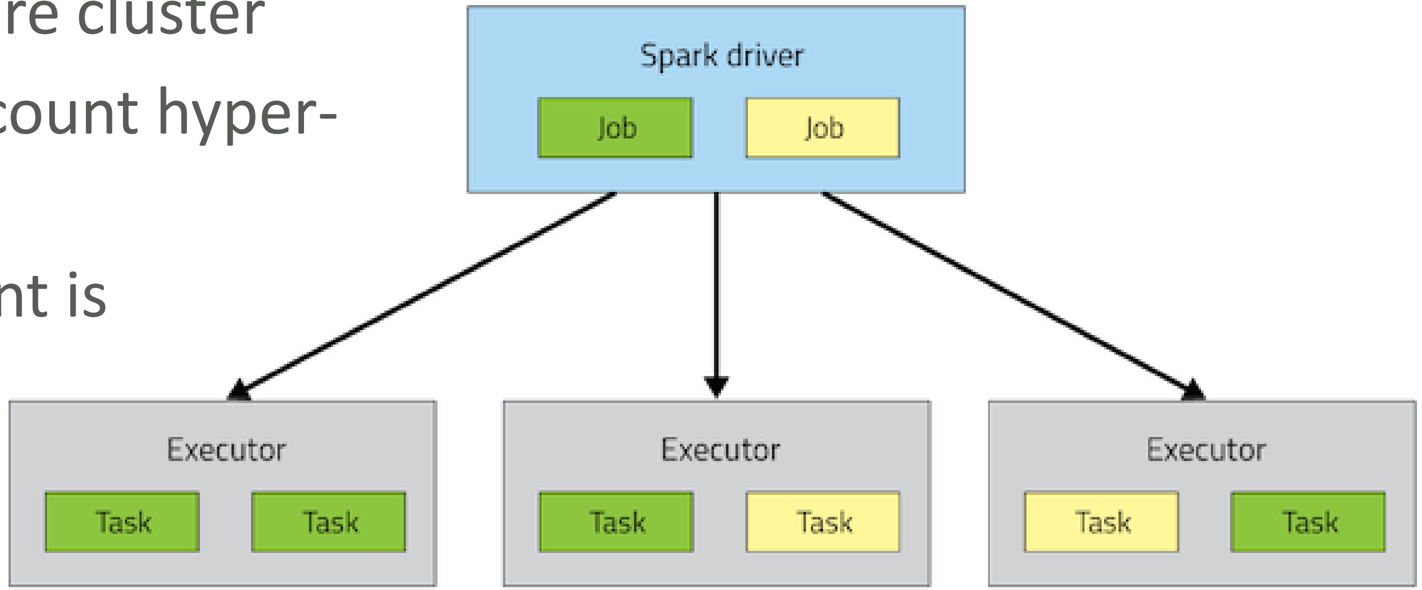
Software

- RHEL 6.6
- CDH 5.4.0
    - Apache Hadoop 2.6.0
    - Apache Spark 1.3
    - Apache Parquet 1.5

Setup

Spark Settings

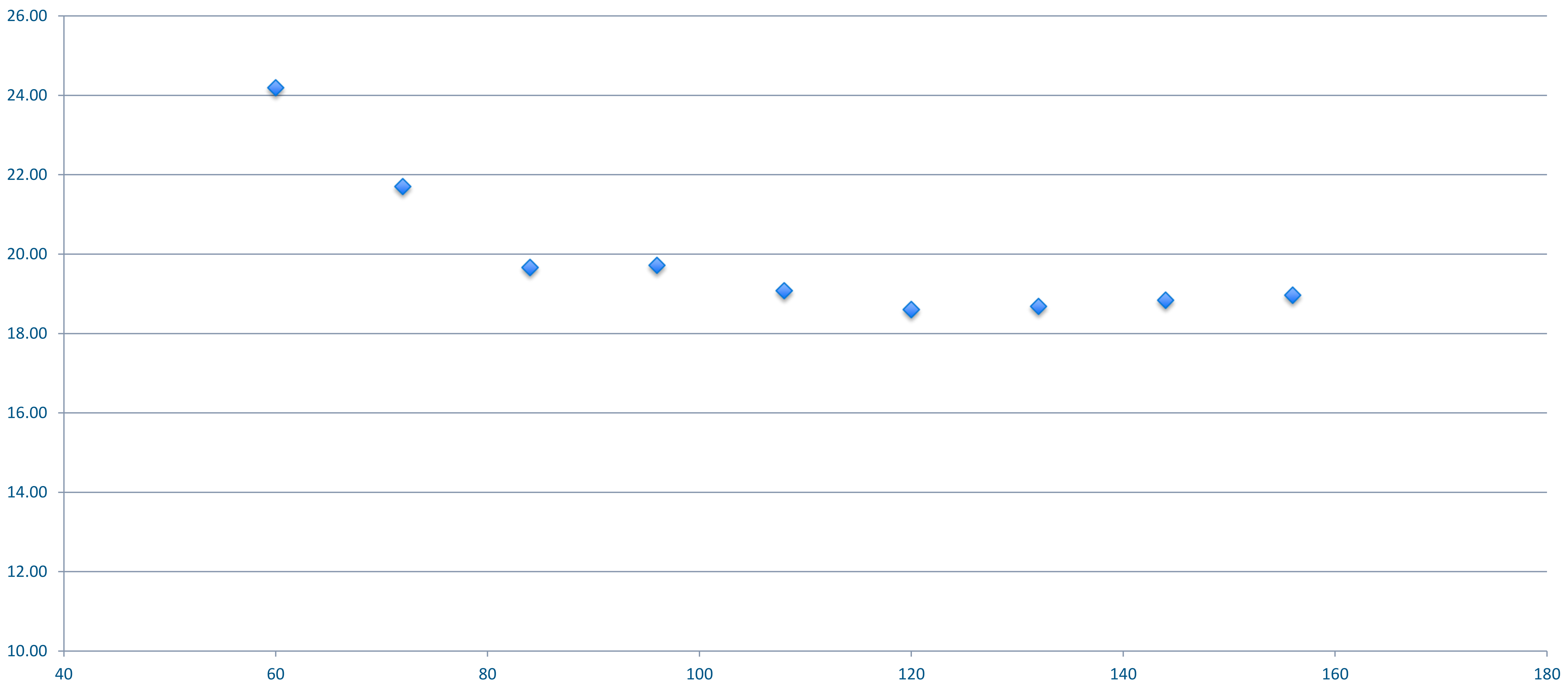
- 4 cores, 4GB per executor
- Given 288 total physical cores, theoretical max of 72 executors for the entire cluster
- Or 144 executors taking into account hyper-threading
- In reality, the effective core count is somewhere in between



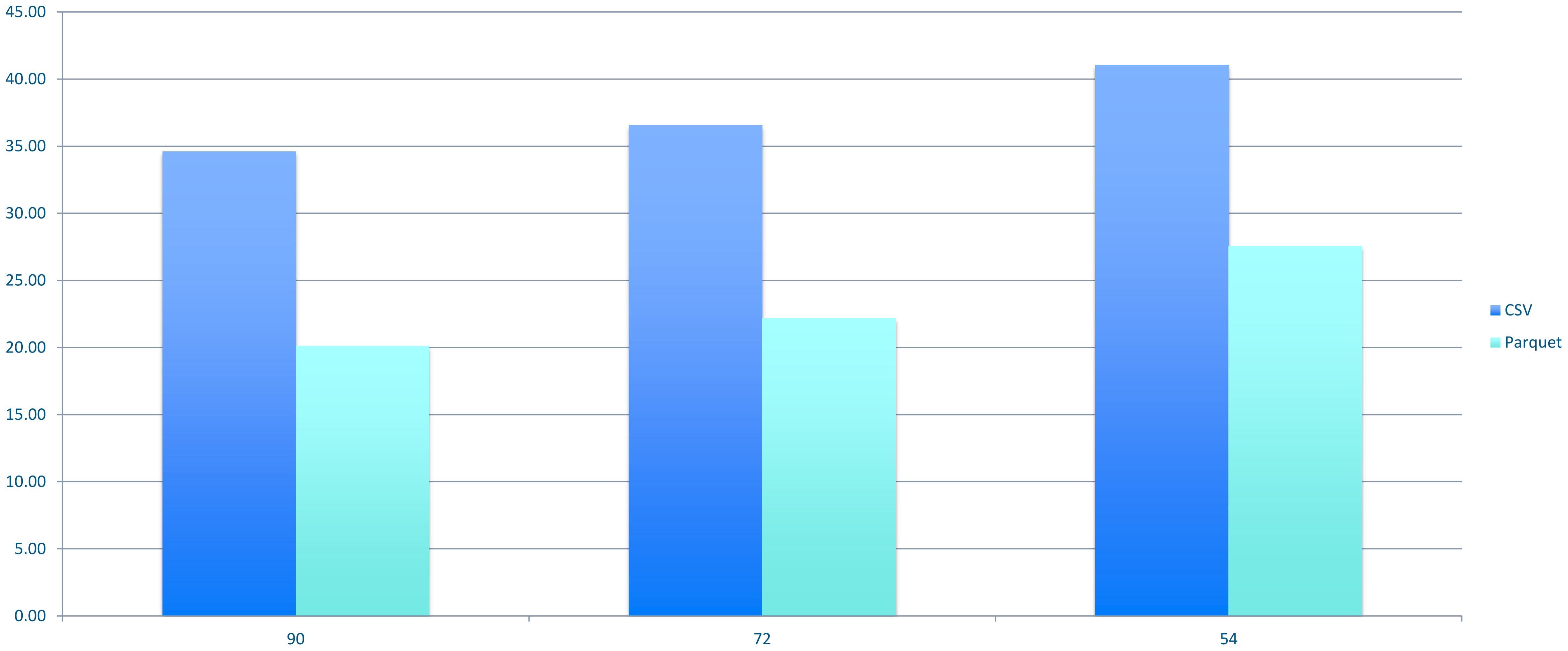
Data set

	CSV	Parquet + Snappy
Avg File Size (Hi / Low)	7.5 GB 550 MB	2.2 GB 164 MB
Total Size, 1 year	8.23 TB	2.46 TB
Data Specs	Full market symbols 251 trading days (1 year) Simulated high and low volume instruments Simulated high volume trading days	

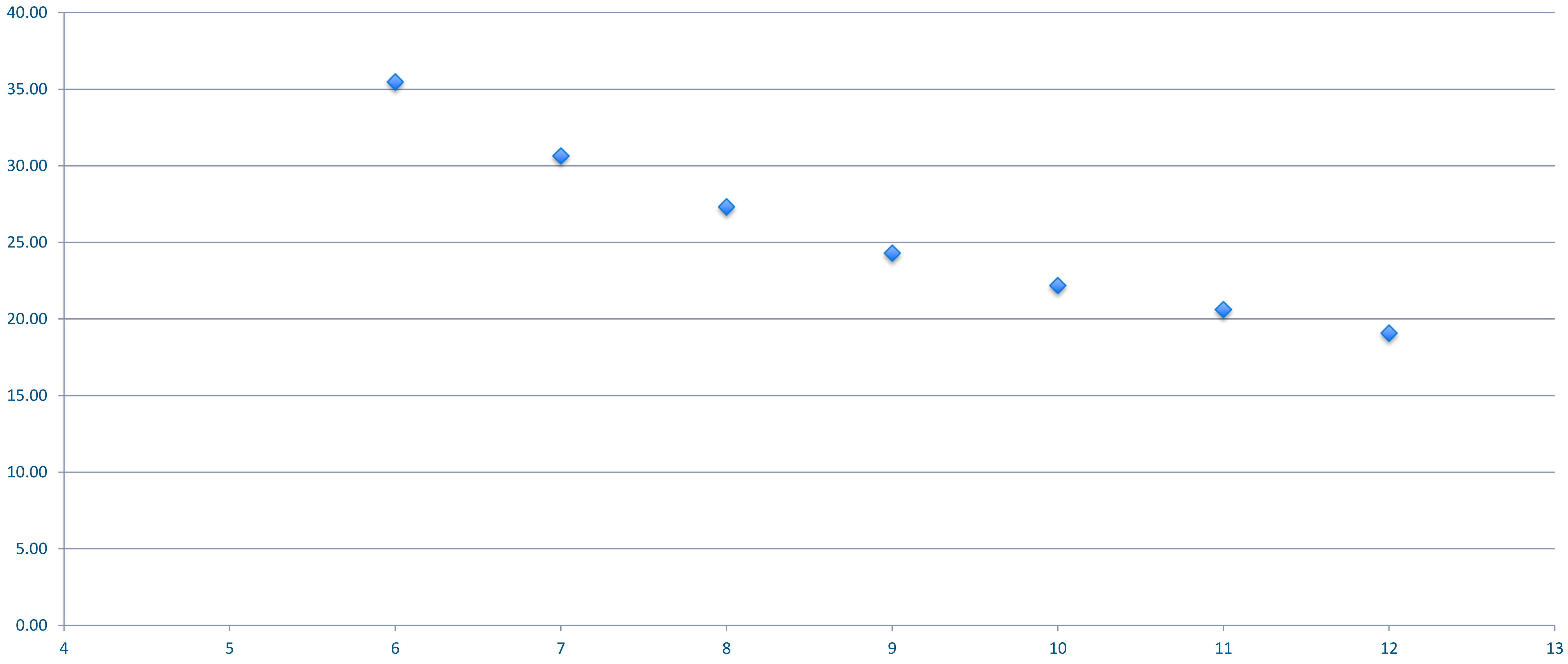
Vertical Scaling Test



Parquet vs CSV



Horizontal Scaling Test



Observations

- Lots of room for optimization
  - Code refactor (avoiding expensive operations)
  - Pre-processing of common data like order books, moving averages, bars, etc.
- No built-in way of dealing with split time-series data
  - Processing is local only for the first split
  - Workaround: use bigger HDFS block sizes
  - Better: API to process file splits sequentially, ability to pass intermediate state to the next task

Thank you!

@patrickangeles