Performance Evaluation II

Distributed systems and web applications

Steffen Viken Valvåg

Microsoft

Just because the wheels are spinning, it doesn't mean the car is moving!

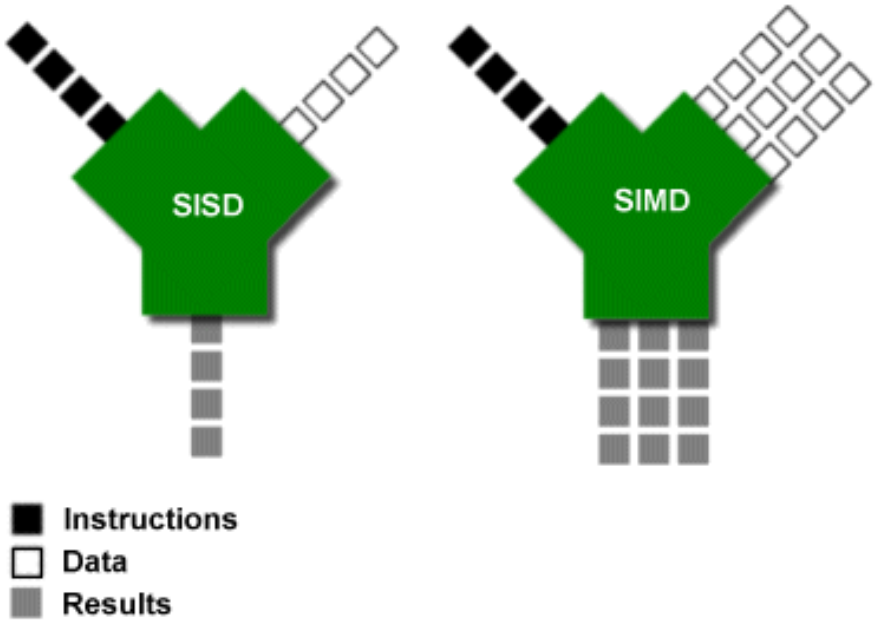
- In a distributed system there are many moving parts.
- To monitor performance, we need to measure the end-to-end performance, using meaningful metrics.
- We also need to monitor the performance of each part to identify problems. (Is the wheel jammed?)

Data Parallelism

- Bake many pizzas in one oven
- (Don’t use many ovens to bake one pizza.)

Data Parallel Hardware Architectures

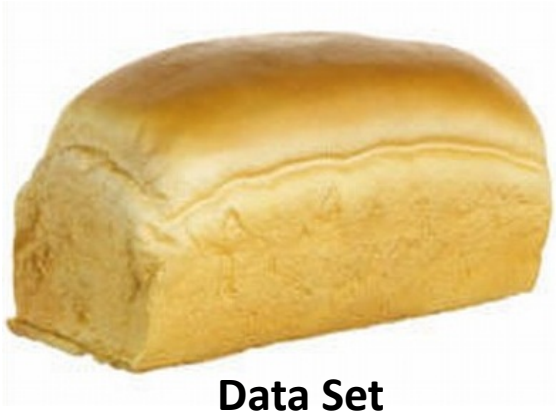
- SIMD [Flynn 1966]
  - Apply the same instruction to multiple data streams



Data Parallel Software Architectures

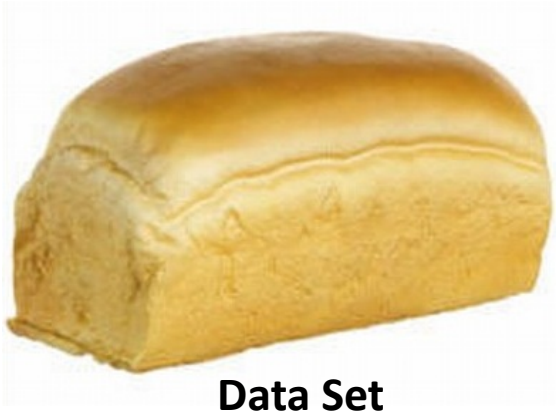
- Execute the same (sequential) code on multiple distinct pieces of data in parallel
  - One independent *task* to execute per piece of data
  - Typically designed for a shared-nothing cluster
- Run-time systems to orchestrate:
  - Storage and partitioning of the input data set(s)
  - Distribution of data to relevant machines
  - Parallel execution (load-balanced and fault-tolerant)
  - Collection/collation of output/results

Toaster Analogy



Data Set

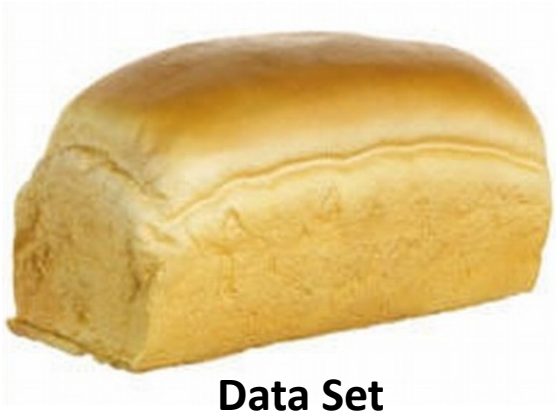
Toaster Analogy



Data Set

Data Partitioning

Toaster Analogy



Data Set

Data Partitioning

Parallel Processing Engine

MapReduce Programming Model

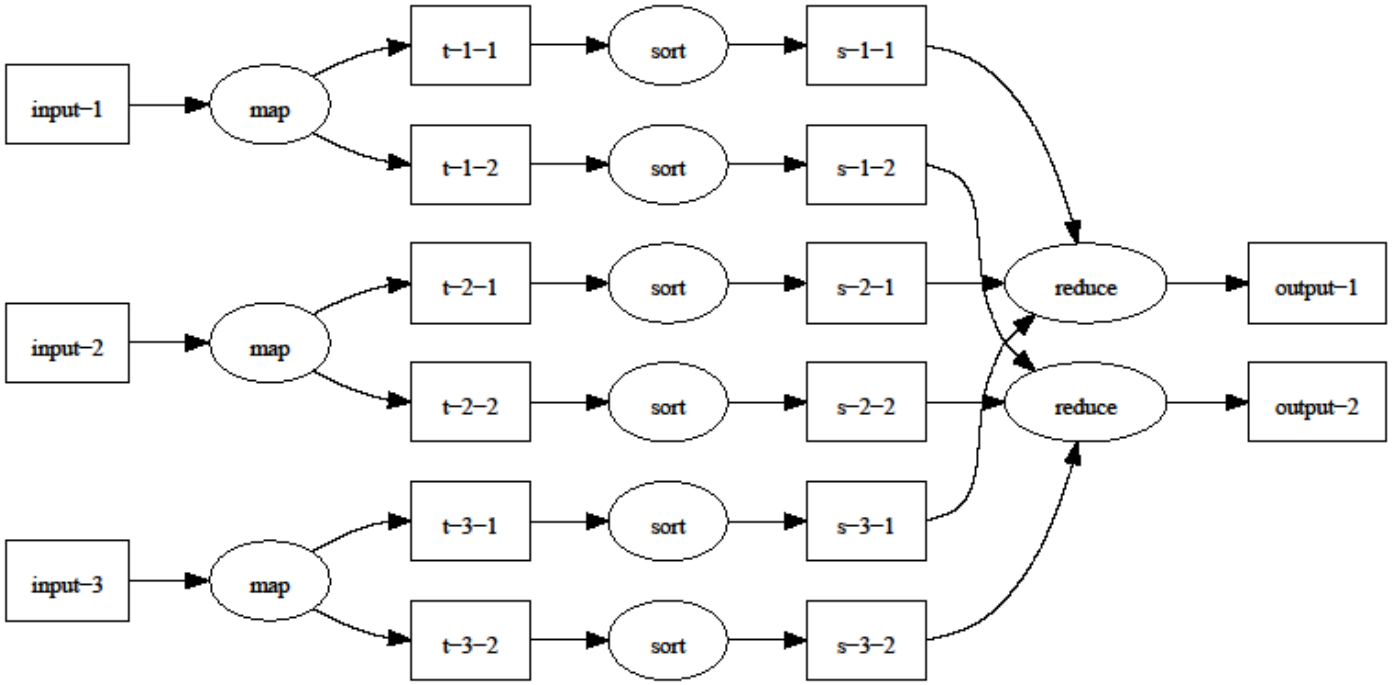
- Specify a computation in terms of:
  - A *map* function to apply to each input record
  - An intermediary key space that determines how to group records output by the map function
  - A *reduce* function that defines how to aggregate groups of intermediate records for the final output
- Map and reduce functions are typically sequential
  - Map executes in parallel for different input partitions; invoked once per input record
  - Reduce executes in parallel for different partitions of the intermediate key space; invoked once per unique intermediate key
- Execution amounts to executing a certain number of map tasks, followed by a certain number of reduce tasks
  - Two-phase execution dictated by data dependencies

Dryad/DryadLINQ

- Compose a flexible communication graph with customizable *vertices* passing data over *channels*
  - Vertex code is typically sequential, invoked with a set of input and output channels
  - Higher-level data-parallel abstractions akin to MapReduce available through DryadLINQ
- Executed as a collection of tasks, where each task executes one vertex, and independent tasks may execute in parallel
  - Split into *stages* that manifest synchronization barriers

Data Dependency Graphs

- Tasks require certain input data, and produce certain output data that other tasks may depend on
  - Below is a MapReduce graph with 3 map tasks and 2 reduce tasks; Dryad allows more flexible graph topologies
  - **Virtualized** execution plan

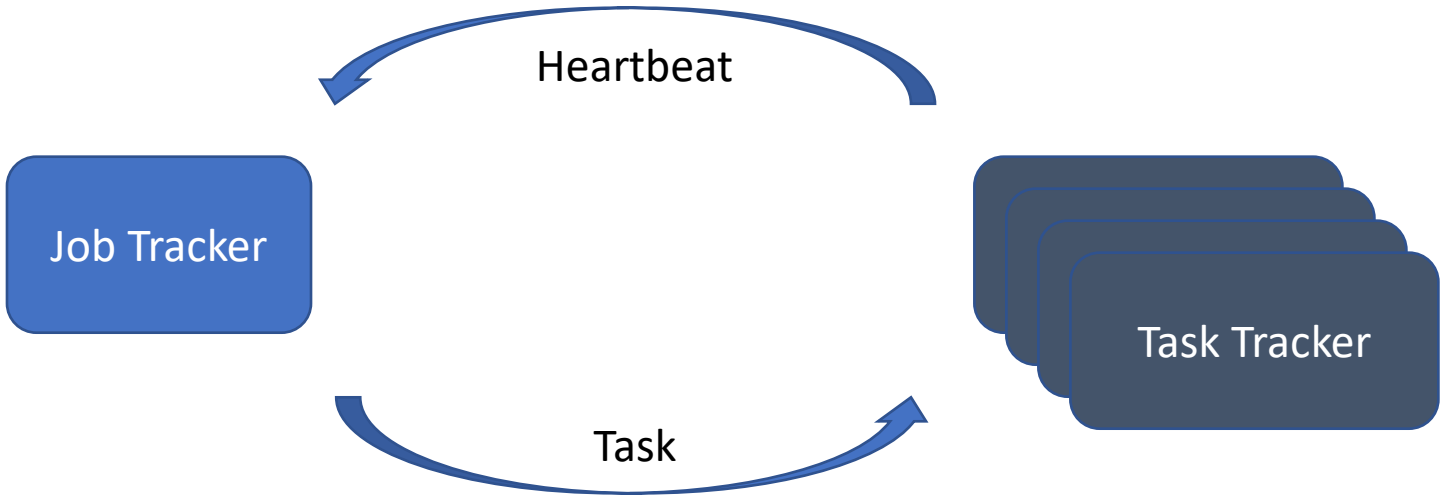


Apache Hadoop

- Hadoop is a widely deployed implementation of MapReduce
  - Also a popular research vehicle
- Jobs are submitted to a central *job tracker* component
  - Makes all scheduling decisions, tracking multiple concurrent jobs
- Every node runs a *task tracker* that communicates regularly with the job tracker to obtain scheduling decisions
  - Each task tracker has a number of task execution *slots,* bounding the number of concurrent tasks on a node
  - Free slots are filled by requesting additional tasks from the job tracker; the received tasks may belong to any ongoing job
- Input and output data are stored in a block-based distributed file system (HDFS)
  - Intermediate data stored locally, outside HDFS

Hadoop Architecture

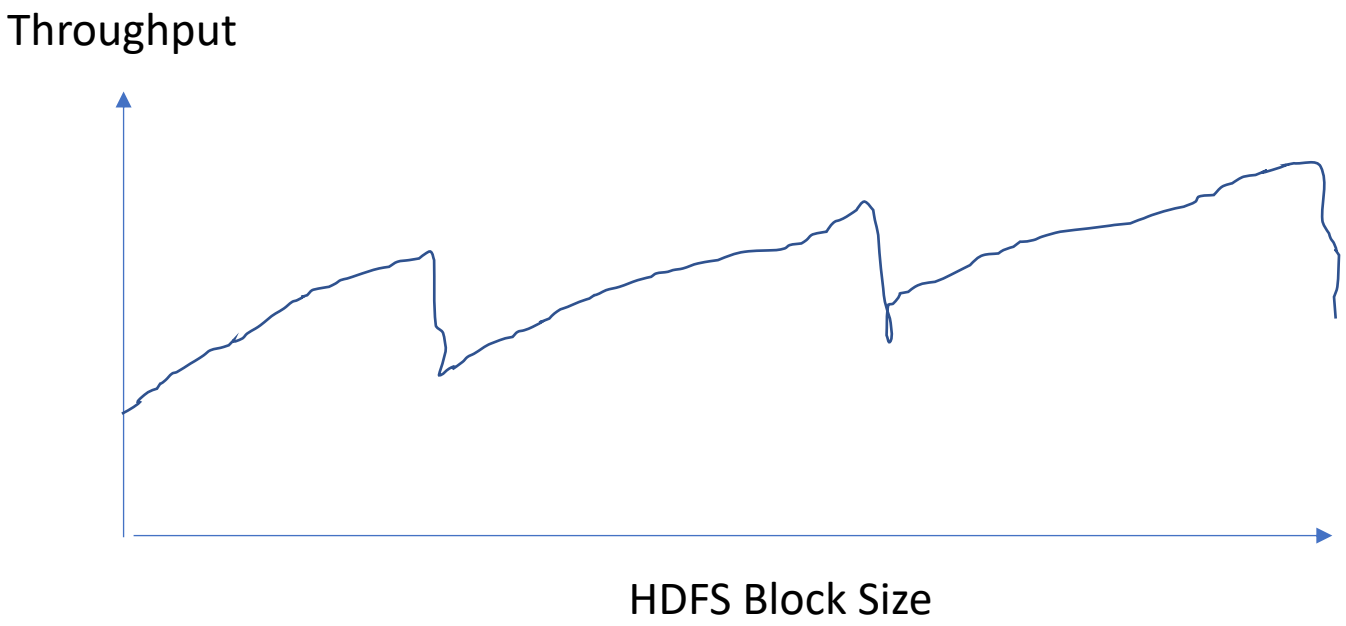
- Master/worker pattern with task trackers that are loosely coupled to the job tracker
  - Communicating through heartbeat RPCs
  - Same architecture for the underlying HDFS



Evaluating the performance of Hadoop

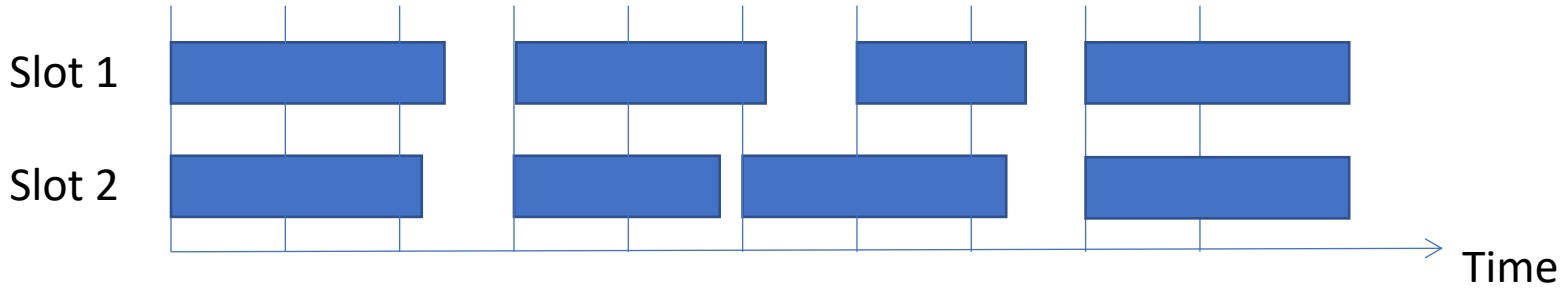
- 100+ configuration options for MapReduce
- 100+ configuration options for the underlying HDFS
- Selecting factors was a nightmare!
- I relied on folklore and recommendations in the documentation for “good” configurations, and used those as starting points.

HDFS Block Size



Anomaly: Idle Time

- Hadoop's task trackers communicate with the central job tracker using heartbeat RPCs
  - Heartbeats occur at most every 3 seconds, and task completion is only reported then
  - Consequently, **task trackers may go idle** if tasks are short-lived
  - Since tasks tend to start at the same time (upon receiving a heartbeat response), they also tend to finish at the same time



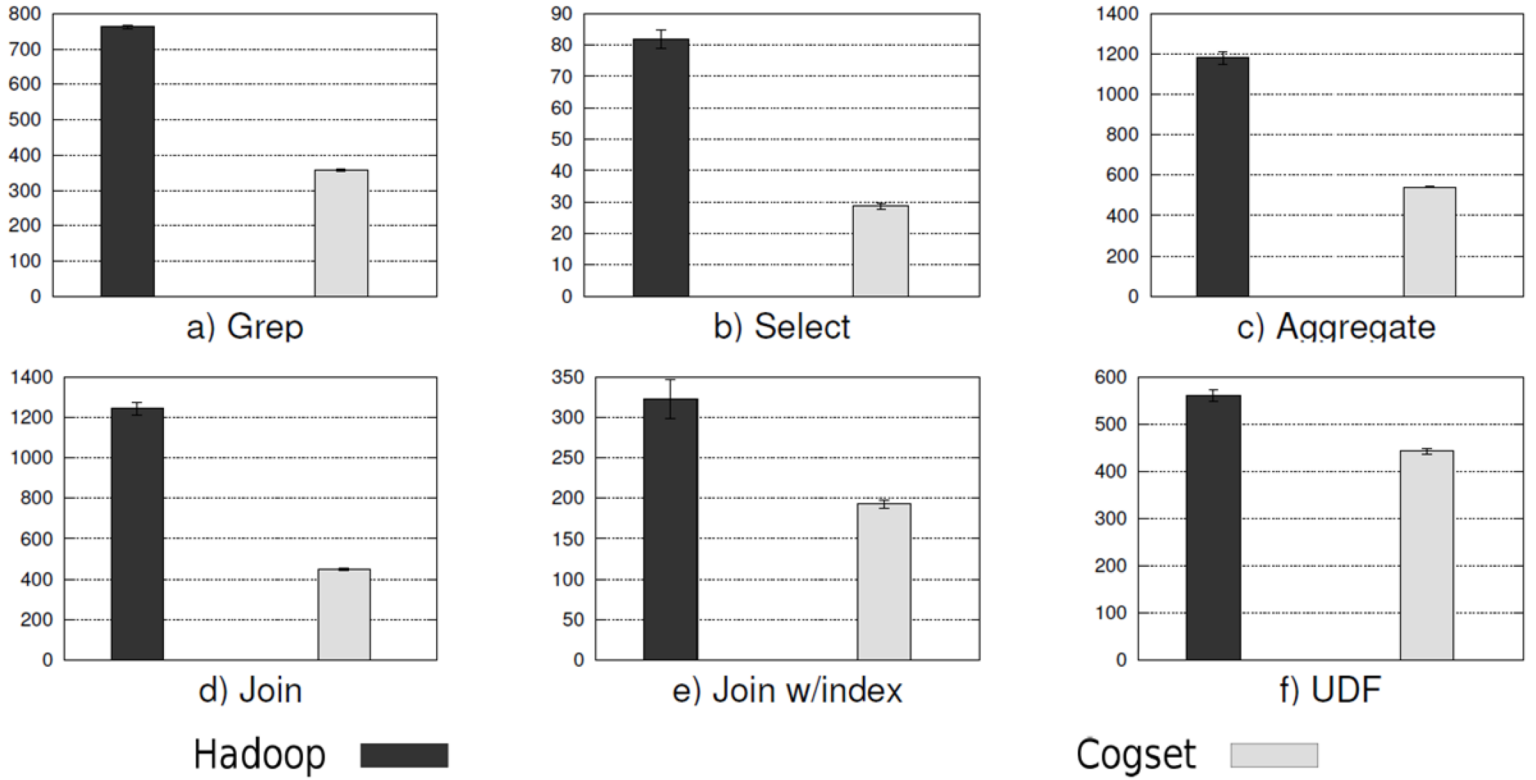
Anomaly: Idle Time

- Unexpected interaction with HDFS block size
  - Bigger block size => more work per mapper => less idle time
- For Grep, task trackers were idle **34%** of the time using the default Hadoop configuration
  - A simple patch allowed completed tasks to be reported immediately
- Hadoop 0.21 introduced a new option that may help
  - mapreduce.tasktracker.outofband.heartbeat
  - Enable this to send out-of-band heartbeats upon task completion

Anomaly: Multi-Core CPU Utilization

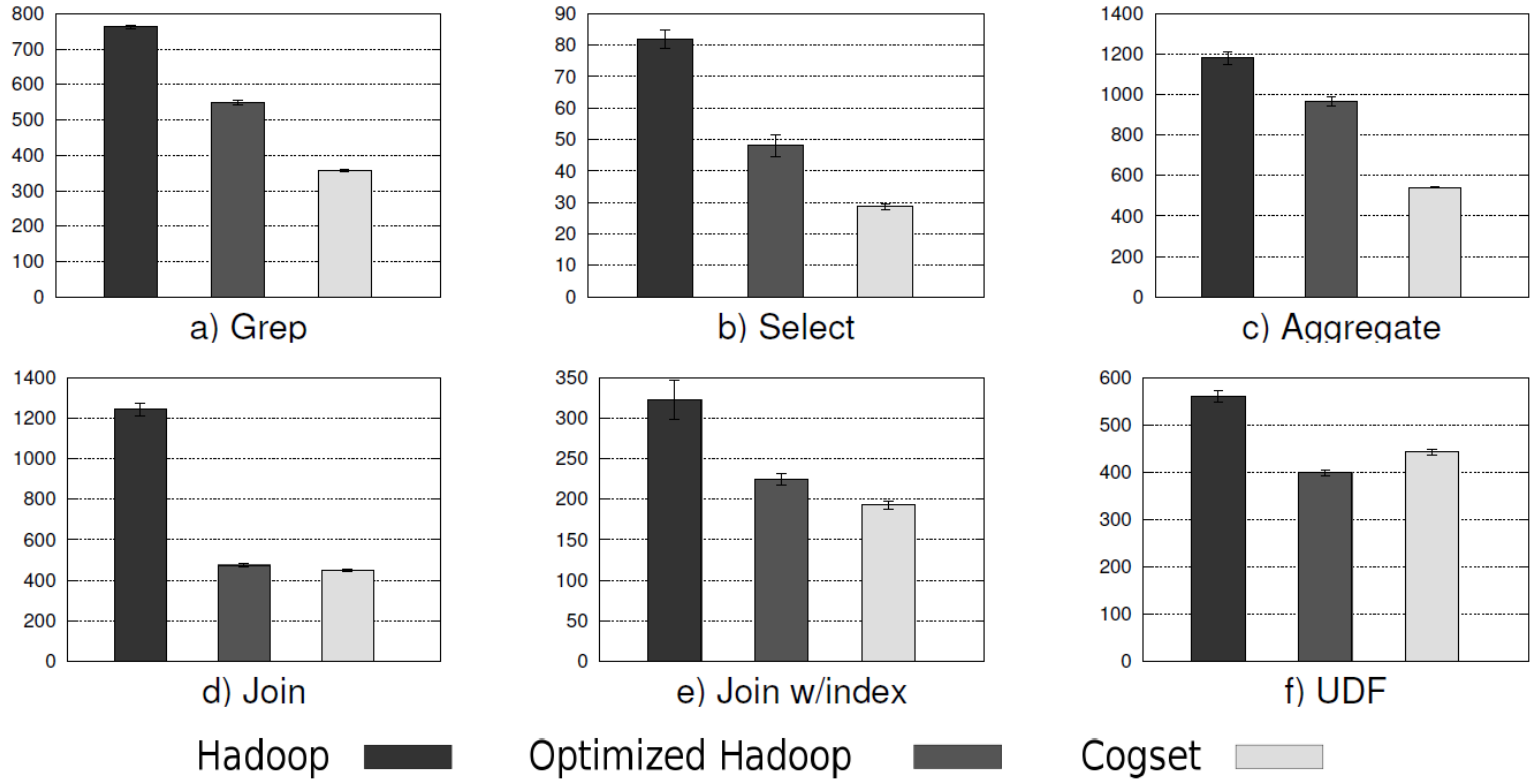
- For sequential scanning of data, and whenever costly UDFs are invoked, Hadoop quickly becomes CPU bound
  - Multiple cores are not well utilized, so there may well be spare CPU cycles that go unused
  - Increasing the number of concurrent processes is ineffective, because of memory footprint and less optimal I/O access patterns
- Remedy: employ multiple threads to read, parse and process records in parallel
  - Fully exploits all cores when costly UDFs are employed
- By implementing a similar approach in Hadoop, plugging in multi-threaded **Cogset** code as a custom input format, performance was greatly improved

MR/DB Results for 25 nodes



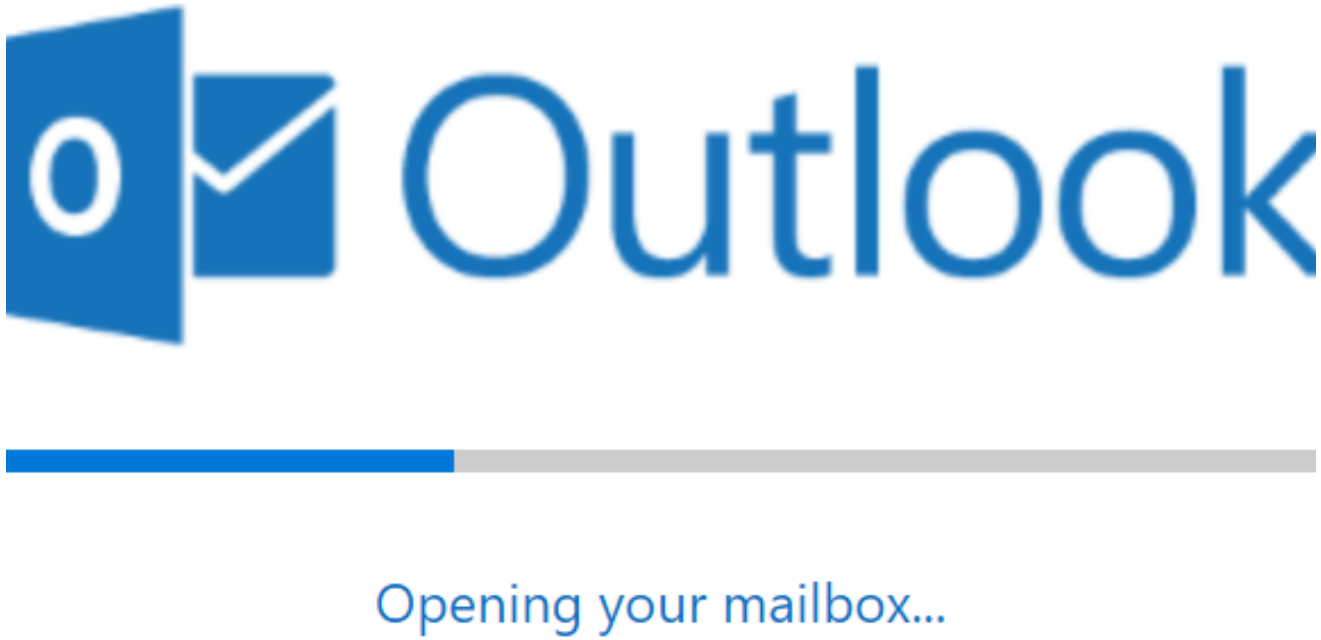
- The Hadoop optimizations close some of the gap
  - Cogset still performs significantly better

MR/DB Results for 25 nodes



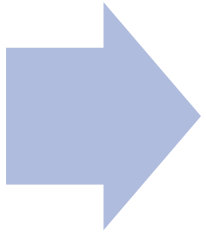
- The Hadoop optimizations close some of the gap
  - Cogset still performs significantly better

Performance in web applications



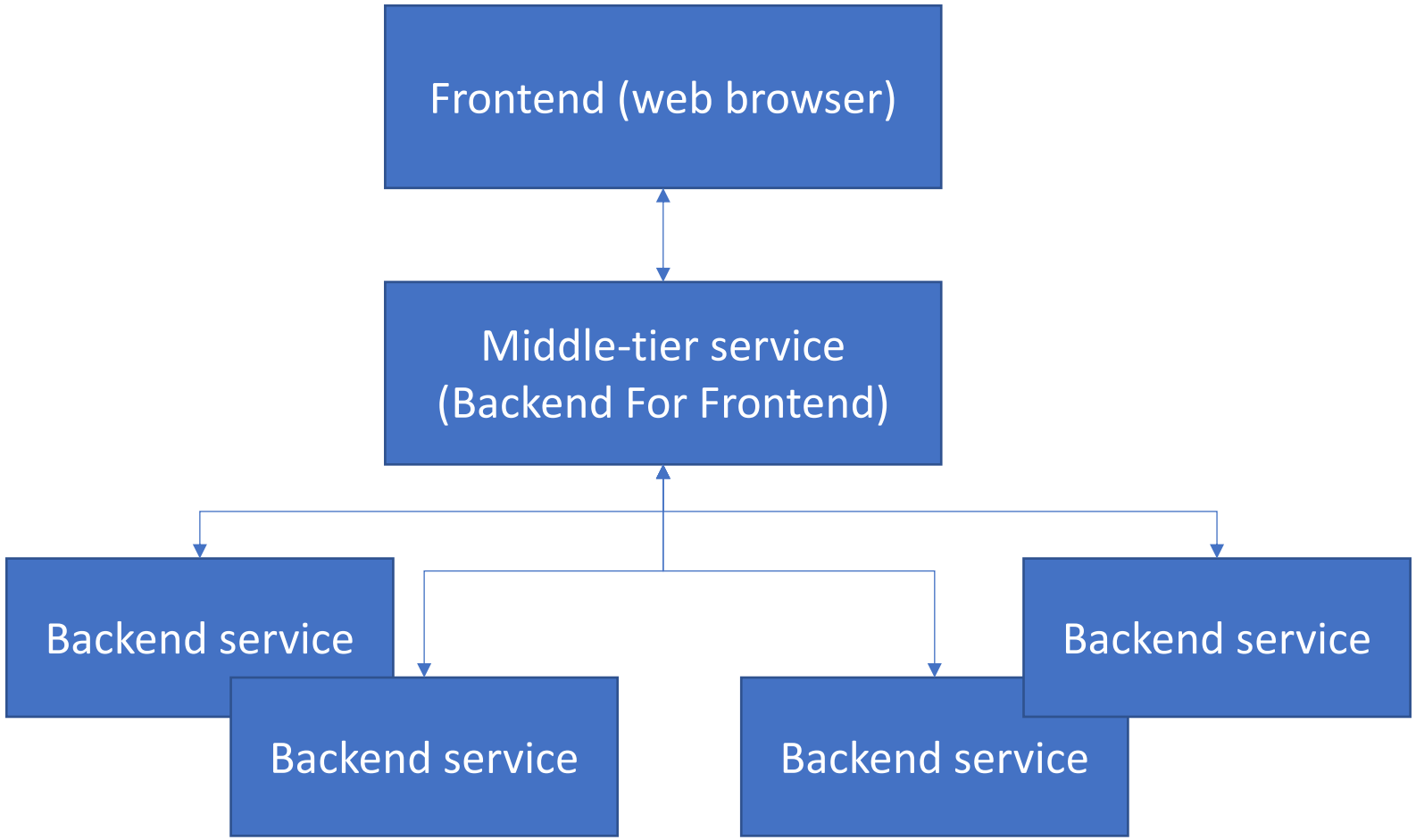
Single-page web applications (SPAs)

Load a single web page into the browser



Dynamically update the page based on user interaction

Typical architecture



Relevant metrics for web applications

- Backend throughput
- Latency for individual requests
- Bandwidth consumption
- Browser frame rate
- End-user perceived latency (EUPL)
  - The time from a click until the UI has finished updating
- Initial page load time
  - Javascript bundle size matters! => minification, compression, staged loading
- Cost of goods sold (COGS)
  - How much money did this cost us

Browser quirks

- Max 6 concurrent connections
  - => domain sharding
  - => pipe requests through a websocket
  - => HTTP/2
- Updating the DOM is expensive
  - => React, a library to efficiently update the DOM tree based on a synthetic DOM tree
- Javascript is single-threaded, but highly concurrent:
  - => Redux, to manage state in a predictable way in the face of concurrency

Correlating events

- A customer reports that the UI says “something went wrong”.
- How do we figure out what went wrong?

- We see that EUPL is high for some users.
  - Why?

- To answer questions like these, we must be able to correlate log events produced by multiple services.
  - => Use randomly generated GUIDs called “correlation IDs” and pass them along with requests, making sure they are logged on both ends.

Demo time