COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 18: Performance

Lecturer: Emery Berger *Scribe(s): Shahar Dahan, SriKrishna Kompella*

18.1 Importance of Performance

Performance is important because:

1. $$Time = Money$$
2. Lots of data requires processing, which leads to information or insights
3. Limited time budget (i.e. couple hours, day, or a week)

18.1.1 Wal-Mart Example

Wal-Mart tracks every sale made by every store to analyze the entire buying experience. Wal-Mart has around 12,000 stores. Let's assume each store makes around 10,000 sales a day. That gives us 120,000,000 items. Should each item be 1KB, that gives us 120 GB of data. This can be saved in disk, so might not require a distributed system. However, Wal-Mart also tracks data on customers as they move throughout the stores, how items get moved or put back, how different layouts affect the amount bought, etc. In addition, this is all regionalized.

Every night, this data is sent to a 'ML black-box' that performs an analysis on the data so that Wal-Mart knows how much to buy, what to buy, and where to put it in the store. This computation needs to occur in 8 hours overnight, and Wal-Mart keeps adding stores as sales keep increasing.

18.1.2 Morgan Stanley Example

Stock traders have massive portfolios of stocks in different companies. To gauge the risk in buying/selling, applications are developed to run simulations based on news, trends and performance of the stocks. The simulations have to be run overnight to calculate the risk and the information should be available before start of trade the next day.

This is another example of a limited time budget

18.2 Performance Factors

Performance depends on a bunch of factors.

Hardware :

1. Number and quality of CPUs, as well as the infrastructure (endianness) (Intel vs AMD)
2. Number of GPUs

   Many general programming GPU (GPGPUs) are coming around. CUDA for nVidia. GPUs are incredibly multithreaded, great for parallel programs. However, they have drawbacks. They expect the same operation on every core (meaning no conditional logic).

   On the same note of GPGPUs, memory transfer is expensive. So, you would ideally have a lot of computation between every step of communication.
3. (esoteric) accelerators (following examples help with ML)
   (a) FPGAs (Azure)
   (b) TPUs (Google)
4. Network
   (a) Network Topology
   Number of jumps between routers would increase latency.
   Amount of bandwidth would determine throughput.
5. Disk
   (a) HDD – latency on seeks is expensive
   (b) SSD – capacity is expensive
   (c) RAID increases bandwidth (Redundant Array of Inexpensive (or Independent) Disks)
6. Memory
   (a) Speeds: Bus Frequency / Bandwidth
   (b) Capcaity – More = Better
   (c) Cache (SRAM): Bigger = Better
   Note on these caches:
   More expensive CPUs have larger caches. So a 4GHz CPU will have a large cache while a 2 GHz CPU will have a small cache.

Software :

1. Garbage Collection: Characteristic of managed languages (i.e. Java / Javascript)
2. Operating System / File System / Networking Stack / Scalability of OS / Optimized OS / Scheduler (Don't want to swap too often, as we'd have only cold caches)

   Linux used to be poor for scalability, but is better now. Mac is still bad.
3. Scheduler

   (a) Don't want to swap too often as we'd have only cold caches. The longer you run code that can reuse cache, the more ammortized is the load time.
   (b) Priority Inversion: If a process is using little CPU and too much I/O every time it is scheduled, the Scheduler starts scheduling it out. This can cause a low priority CPU intensive process to get more CPU time than a low priority I/O intensive process. This is Priority inversion

4. Algorithms: Poor algorithms can lead to bad big-O notation which leads to bad performance.
5. Compilation: a JIT compiler will need a warmup period to optimize the code
6. Java: Starting up a JVM is slow, vs a C/C++ program which is always optimized.
7. Design choices: Do we optimize throughput or latency?

   Normally, we can do one object quickly (so have good latency) but it is usually faster to operate on a batch of objects (which would lead to good throughput). So, these are usually a tradeoff.
8. Multi-Thread vs Multi-Process:
   (a) Multi-Process Pros:
      i. Some languages (NodeJS) only have one thread, and concurrency is done through waiting on I/O. This would require a multiprocess solution.
      ii. Multiprocess'd programs are more fault tolerant.
      iii. Good when there is no shared state.
   (b) Multi-Thread Pros:
      i. Easier to create a thread than a process
      ii. Can take advantage of a cache (i.e. serving static web pages)
      iii. Good for shared state
   (c) Multi-Thread Cons:
      i. Bad for correctness
      ii. Have to deal with cache correctness / invalidation
9. Locks:
   (a) Locking Binary Trees requires hand-over-hand locking: Hold one lock while getting the next lock, only release the current lock when you get that lock (imagine monkey 'rings').

      This leads to increased contention on the root of the tree.
   (b) Locking graphs becomes even more of a nightmare
10. Abstractions (i.e. API):
   (a) Do NOT want abstraction to expose the underneath. This is a 'leaky' abstraction.
   (b) Performance is the ultimate leaky abstraction, as it is entirely dependent on workload. (What if workload 1 fits in RAM, but workload 2 does not?)

18.3 Scalability, but at what COST (very briefly)

We have a lot of bottlenecks in systems:

1. Netowrk

2. Disk
3. Straggler Problem (aka Garbage Collection)

However, this paper analyzes Spark SQL to see if these bottle necks are real. We know Spark aims to keep computation primarily in memory, so it would obviously not be limited by the disk. Additionally, the workload was just database queries, and Spark SQL handles that by sending bits and pieces of information throughout the network. That means it makes sense that it would barely use the network as well.

Bottom line is: no one from Spark reviewed this paper, and it was secretly a terrible paper.