DSC-204A: Scalable Data Systems, Winter 2024

19: Streaming processing - 1

Lecturer: Hao Zhang *Scribe: Ever Wong*

1 Recap: Batch Processing

Batch processing is well-suited for tasks that are not sensitive to latency. The Map-reduce model fundamentally comprises two components: the mapper and the reducer. This structure allows users to execute tasks by customizing these two functions. To enhance performance, the Combiner and Partitioner are employed as optimization tools. It is possible to orchestrate multiple Map-reduce jobs to construct a comprehensive processing workflow, where communication between jobs is facilitated through disk I/O, leading to **low performance**.

Some pros of this batch processing using Map-reduce:

- It is very **expressive** and **scalable** functional programming.
- It is **fault tolerant** because it is easy to build redundancy in Map-reduce systems since we use disks to store data.

We aim to minimize disk writes due to their slow nature. The motivation for developing a new system stems from the fact that the Map-reduce framework necessitates writing to file systems and utilizing distributed storage, which can be inefficient.

2 Arithmatic Intensity

Arithmetic Intensity (AI) is a pivotal concept that fundamentally gauges the speed of our computations. Before delving deeper, let's briefly revisit the basics of AI.

2.1 Basics of processors

Q: How does a processor execute machine code?

A simple example of an add instruction is shown in Figure 1. Here we load from register rbx and add that value to register rax. In general, there are three types of ISA commands to manipulate register contents:

- Memory access: **load** (copy bytes from a DRAM address to register); **store** (reverse); put constant
- Arithmetic & logic on data items in registers: **add/multiply/etc.;** bitwise ops; compare, etc.; handled by **ALU**
- Control flow (branch, call, etc.); handled by **CU**

It's important to note the disparity in speed between loading a value from a register versus loading it from memory, with the latter being significantly slower. To decrease the frequency of direct main memory access and thereby reduce latency, we utilize **caches**. These are compact, local memory units that act as buffers for instructions and data, enhancing the overall efficiency of data retrieval.

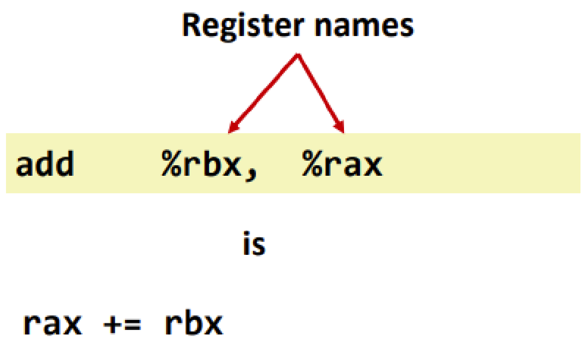


Figure 1: An add instruction

2.2 I/O slows down computation

In the add example, operations such as load and store are executed within registers, which are integrated into the processor and thus costly in terms of resources. Accessing memory is a necessity, but it's crucial to acknowledge that I/O operations can decelerate our workflow. Refer to Figure 2 for an illustration of the **memory hierarchy**.

In summary, it's imperative to take into account the impact of I/O operations in conjunction with the computations we execute, to optimize overall performance.

2.3 Measure the impact of I/O

I/O operations can impede computation speeds at every level of the memory hierarchy. In the context of Map-reduce, the practice of saving and loading results from distributed storage contributes to diminished performance.

The concept of arithmetic intensity can be employed to quantify this slowdown:

$$\text{AI} = \frac{\#\text{Compute Op}}{\#\text{I/O Op}}$$

Arithmetic Intensity (AI) is defined as the ratio of the number of **compute** operations to the number of **I/O** operations. When compute operations significantly outnumber I/O operations, AI is high, indicating an efficient system. Conversely, a low AI suggests inefficiency, as excessive time is devoted to reading or writing rather than actual computation.

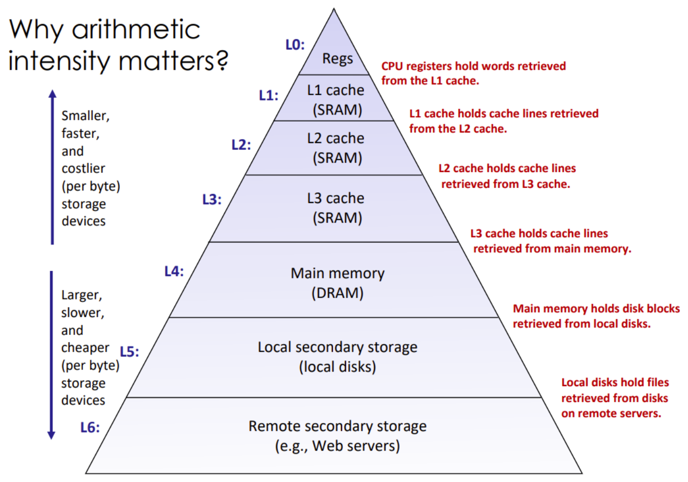

Why arithmetic intensity matters?
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers.

Figure 2: Memory hierarchy

2.4 Examples

Here are two programs: 1 and 2 , and we want to know which is better by computing the arithmetic intensity of them.

```
void add(int n, float* A, float* B, float* C){
  for (int i=0; i<n; i++)
    C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
  for (int i=0; i<n; i++)
    C[i] = A[i] * B[i];
}

float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```

Figure 3: Program 1

```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;
// assume arrays are allocated here
// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);


void fused(int n, float* A, float* B, float* C, float* D,
  float* E) {
  for (int i=0; i<n; i++)
    E[i] = D[i] + (A[i] + B[i]) * C[i];
}
// compute E = D + (A + B) * C
fused(n, A, B,C, D, E);
```

Figure 4: Program 2 (fusion)

In Program 1, we have two functions add and mul. In each function, there is a for loop. At each step we perform four operations:

1. Read A[i]
2. Read B[i]
3. Add or multiply A[i] with B[i]
4. Store the result in C[i]

We perform 2 loads, 1 store per math op, thus AI=1/3. The overall AI is also 1/3.

In Program 2, we fuse the add and multiply operations together, and we save the times to access the array since we perform computations **in place**. Now we have 4 loads, 1 store per 3 math ops, which makes AI = 3/5. Note that AI becomes larger, so computation fusion makes our program run faster!

There are numerous approaches to implementing the fusion technique across various operators and multidimensional arrays. Modern compilers, such as LLVM, are designed to scrutinize your code and align it with predefined fusion templates. This process of code fusion is a widespread optimization strategy.

To summarize why Map-reduce is slow in one sentence: it has **low arithmetic intensity** due to **disk I/O**. This low AI simply means that the CPUs will be idle because the time is spent on reading and writing.

3 PageRank

PageRank is basically to rank the importance of each web page (or person) from the entire internet. We want to know which person is very prominent and which web pages are more popular. We can construct a graph to represent this, where each node has a floating point weight as shown in Figure 5.

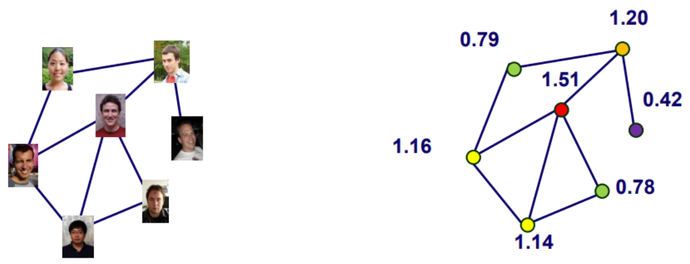


Figure 5: PageRank

The computation includes three parts:

- Initally we assign weight 1.0 to each page (node).
- We do updates iteratively by selecting an arbitrary node and try to update it using the weights from its neighbors, e.g., $R_1 \leftarrow 0.1 + 0.9 \cdot (\frac{1}{2}R_2 + \frac{1}{4}R_3 + \frac{1}{3}R_5)$.
- A theory proves that this process will converge. All results will be unique, regardless of selection ordering. The converged graph is called **power graph**.

We can use MapReduce to compute PageRank. We use Map function to generate values to pass along each edge such as $(1, \frac{1}{2}R_2), (1, \frac{1}{4}R_3)$ for node 1. We use Reduce function to combine edge values to get new rank, like performing $R_1 \leftarrow 0.1 + 0.9 \cdot (\frac{1}{2}R_2 + \frac{1}{4}R_3 + \frac{1}{3}R_5)$ to update R_1.

However, this will have low arithmetic intensity and we cannot optimize this with fusion because it is computed in a distributed cluster and we cannot do things in place. Spark will help us handle this kind of problems.

4 Spark

Spark supports in-memory, fault-tolerant distributed computing.

4.1 Goals

Spark was proposed by a former UC Grad student called Matei Zaharia who has won the Netflix Prize.

It is a programming model for cluster-scale computations where there is **significant reuse** of intermediate datasets. These include iterative machine learning, graph algorithms and interactive data mining. We keep the intermediates in memory rather than write them to persistent distributed file system.

While memory offers faster access times compared to storage, it is not as capacious, limiting our ability to retain everything in memory. Additionally, relying solely on memory for storing intermediate results poses a risk; in the event of a failure, these results could be lost. Therefore, it's crucial to devise a strategy for implementing fault tolerance, particularly for large-scale distributed in-memory computations.

4.2 Three necessary conditions

- Memory: large (cheap) enough
- Network: fast (cheap) enough
- fault tolerance: at least as good as map-reduce

In 2010s, the bandwidth (speed) of some hardware devices are shown in Figure 6.

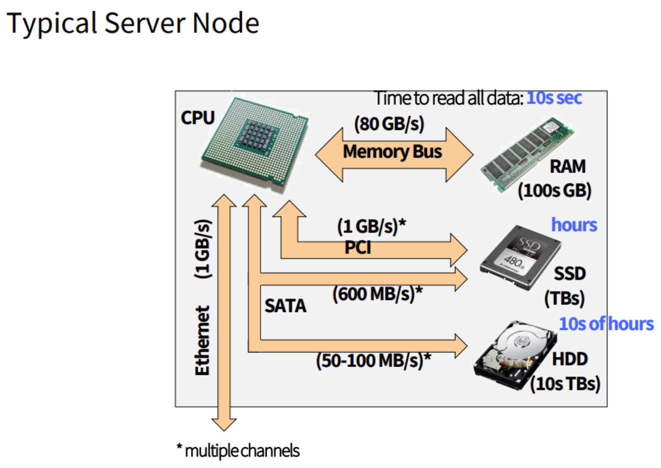


Figure 6: Typical server node

From Figure 7, we can also see that the memory capacity grows sub-linearly. It increases almost 30% per year, which is a crazy number.

SSD becomes cheaper than HDD after some crosspoint while SSD is also faster than HDD. Therefore, the transition from HDD to SSD accelerates and for example, already most instances in AWS have SSDs. We can predict that there will be SSD only clusters and SSD will eventually replace HDD.

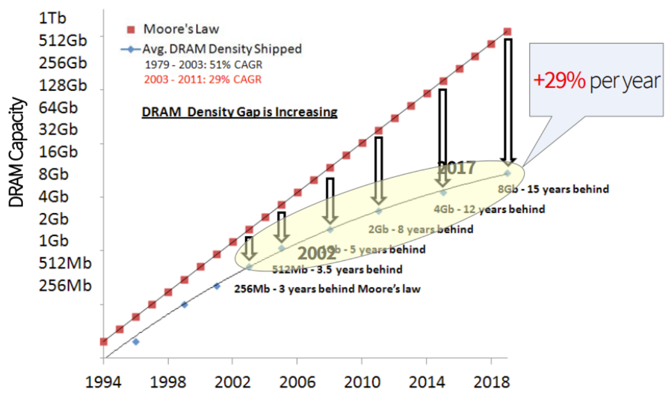


Figure 7: Memory capacity

Ethernet Bandwidth also grows likw 33-40% per year and we put all the numbers together in Figure 8.

This growth means that memory hierarchy has shifted one layer up. A layer up from DRAM is high-bandwidth memory (HBM), and the layer on top of that is SRAM (a smaller but faster memory). In addition, HDD is virtually dead. We have unlimited space of SSD. Today's RAM space is yesterday's SSD space. Today's SSD space is yesterday's HDD space. Ethernet may become faster than PCI/SATA bandwidth. In the near future, maybe one more layer will be shifted up. Nvidia is dealing with HBM while a new company called Groq is dealing with SRAM and they can do Llama-2 inference faster than Nvidia. We can use our knowledge to predict the trend of the future.

4.3 Fault tolerance for in-memory calculations

Due to the growth of hardware performance, we now only have one condition to meet: fault tolerance. We can do the following:

- Replicate all computations
  - Expensive solution: decreases peak throughput
- Checkpoint and rollback
  - Periodically save state of program to persistent storage
  - Restart from last checkpoint on node failure
- Maintain log of updates (commands and data)
  - High overhead for maintaining logs

4.4 Resilient distributed dataset (RDD)

Example query: "What type of mobile phone are all the visitors using?"

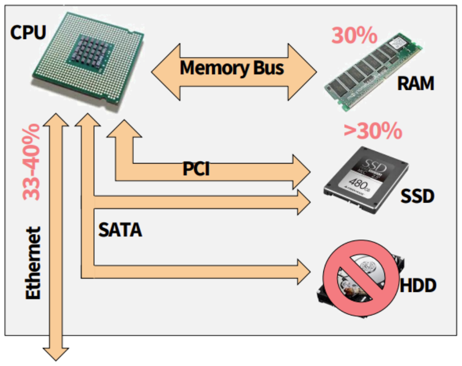
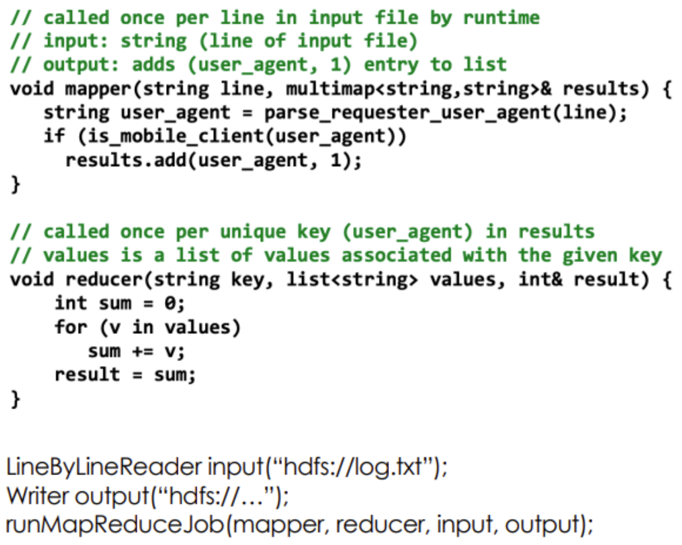


Figure 8: Growth of different device

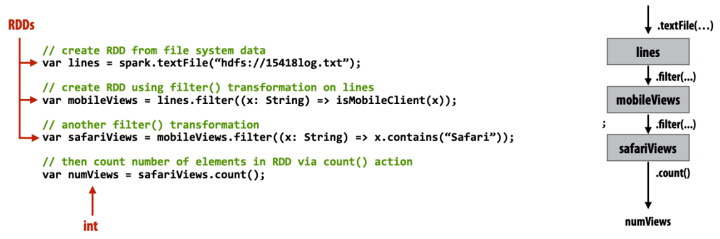


```
// called once per line in input file by runtime
// input: string (line of input file)
// output: adds (user_agent, 1) entry to list
void mapper(string line, multimap<string,string>& results) {
   string user_agent = parse_requester_user_agent(line);
   if (is_mobile_client(user_agent))
     results.add(user_agent, 1);
}

// called once per unique key (user_agent) in results
// values is a list of values associated with the given key
void reducer(string key, list<string> values, int& result) {
    int sum = 0;
    for (v in values)
       sum += v;
    result = sum;
}

LineByLineReader input("hdfs://log.txt");
Writer output("hdfs://…");
runMapReduceJob(mapper, reducer, input, output);
```

Figure 9: Using MapReduce

We can use MapReduce to handle this problem like in Figure 9. The code computes the count of page views by each type of mobile phone.

In Figure 10, it is illustrated that Spark provides a suite of operators akin to those found in frameworks like PyTorch. Unlike in MapReduce, where users are expected to implement core functions, Spark abstracts these complexities, offering a more user-friendly interface. Within the Scala code presented, certain variables are designated as RDDs, which are fundamental to Spark's data handling capabilities. The diagram on the right side visualizes the dataflow graph.



```
// create RDD from file system data
var lines = spark.textFile("hdfs://15418log.txt");

// create RDD using filter() transformation on lines
var mobileViews = lines.filter((x: String) => isMobileClient(x));

// another filter() transformation
var safariViews = mobileViews.filter((x: String) => x.contains("Safari"));

// then count number of elements in RDD via count() action
var numViews = safariViews.count();
```

Figure 10: Using RDD