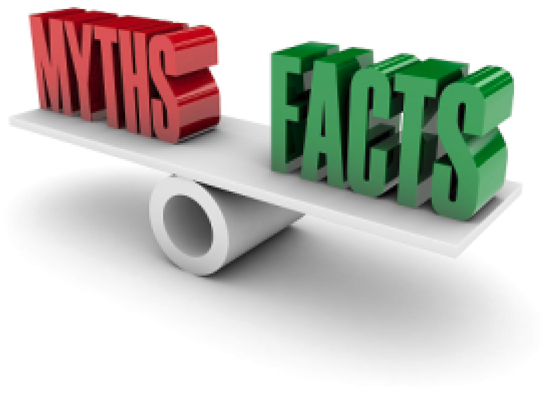
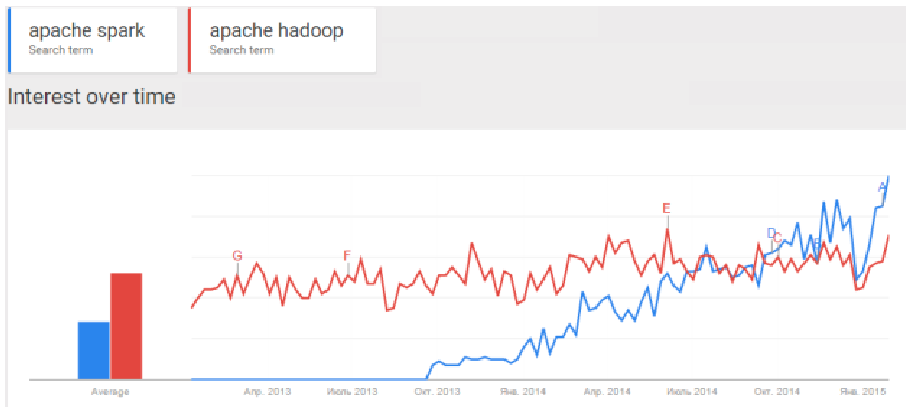
Distributed Systems Architecture

brought to you by Alexey Grishchenko

Spark Misconceptions



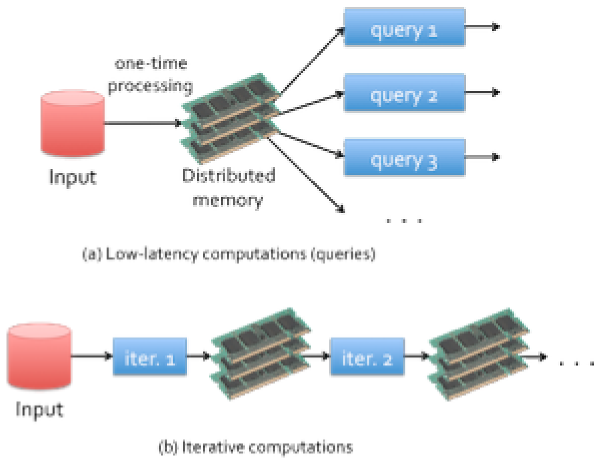
At the moment there is a huge buzz in the media about the Apache Spark framework and little by little it becomes next big thing in a field of "Big Data". The simplest thing we can do to prove this is to look at the google trends diagram:



I have shown here both Hadoop and Spark for the last 2 years. So Spark is becoming more and more popular among the end customers, and they are looking over the internet for more information about Spark. Given this big hype around the technology, it is surrounded by many myths and misconceptions and many people treat it as a silver bullet that would solve their problems with Hadoop giving them 100x better performance.

In this article I would cover the main misconceptions about this technology to set a specific level of expectations for the technology guys looking forward to applying this framework in their system. I would say that the main sources of misconceptions are rumors and oversimplifications introduced by some specialists on the market. Spark documentation is clear enough to disprove them all, but it requires much reading. So, the main misconceptions I would cover are:

1. Spark is an in-memory technology
2. Spark performs 10x-100x faster than Hadoop
3. Spark introduces completely new approach for data processing on the market

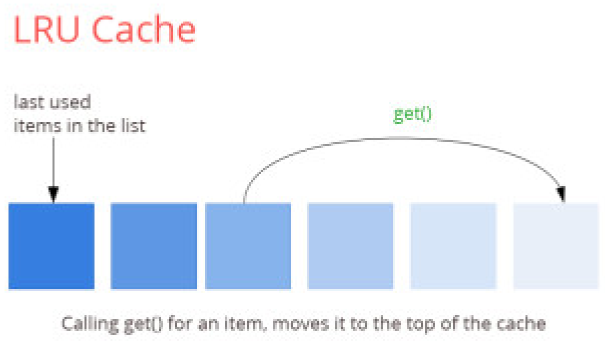


First and the most popular misconception about Spark is that "***Spark is in-memory technology***". Hell no, and none of the Spark developers officially states this! These are the rumors based on the misunderstanding of the Spark computation processes.

But let's start from the beginning. What kind of technology do we call in-memory? In my opinion it is the technology that allows you to **persist** the data in RAM and effectively process it. What do we see in Spark? It has no option for in-memory data persistence, it has pluggable connectors for different persistent storage systems like HDFS, Tachyon, HBase, Cassandra and so on, but it does not have native persistence code, neither for in-memory nor for on-disk storage. Everything it can do is to **cache** the data, which is not the "persistence". Cached data can be easily dropped and recomputed later based on the other data available in the source persistent store available through connector.

Next, some of the guys complain that even given the information above, Spark processes data in memory. Of course it does, because you just don't have an option to process the data in any other way. All the operations in the OS APIs allow you to just load the data from block devices into memory and unload it back to the block devices. You cannot compute something directly on the HDDs without loading their data into memory, so all the processing in the modern systems is basically in-memory processing.

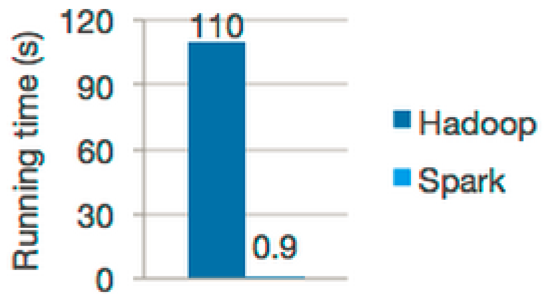
Given the fact that Spark allows you to use in-memory cache with the LRU eviction rules, you might still assume that it is in-memory technology, at least when the data you are processing fits in memory. But let's turn to the RDBMSs market and take 2 examples out of there – Oracle and PostgreSQL. How do you think they process the data? They use shared memory segment as a pool for the table pages, all the data reads and data writes are served through this pool. And this pool also has LRU eviction rules to evict the non-dirty table pages from it (and force the checkpoint process if there are too many dirty pages). So in general, modern databases are also efficiently utilizing in-memory LRU cache for their needs. Why don't we call Oracle or PostgreSQL in-memory solutions? And what about Linux IO, did you know that all the IO operations are passing the OS IO cache which is the same LRU cache?


LRU Cache
last used items in the list
get()
Calling get() for an item, moves it to the top of the cache

And even more. Do you think that Spark processes all the transformations in memory? You would be disappointed, but the heart of Spark, "shuffle", writes data to disks. If you have a "group by" statement in your SparkSQL query or you are just transforming RDD to PairRDD and calling on it some aggregation by key, you are forcing Spark to distribute data among the partitions based on the hash value of the key. The "shuffle" process consists of two phases, usually referred as "map" and "reduce". "Map" just calculates hash values of your key (or other partitioning function if you set it manually) and outputs the data to **N** separate files on the local filesystem, where **N** is the number of partitions on the "reduce" side. "Reduce" side polls the "map" side for the data and merges it in new partitions. So if you have an RDD of **M** partitions and you transform it to pair RDD with **N** partitions, there would be **M*N** files created on the local filesystems in your cluster, holding all the data of the specific RDD. There are some optimizations available to reduce amount of files. Also there are some work undergo to pre-sort them and then "merge" on "reduce" side, but this does not change the fact that each time you need to "shuffle" you data you are putting it to the HDDs.

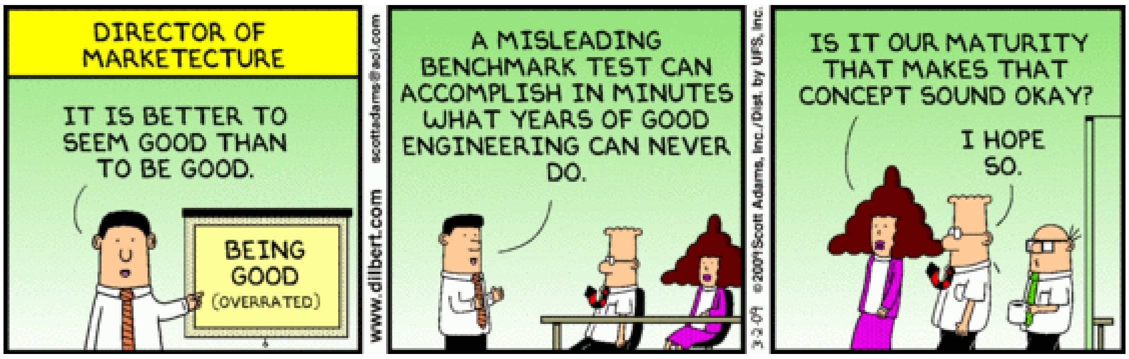
So finally, Spark is not an in-memory technology. It is the technology that allows you to efficiently utilize in-memory LRU cache with possible on-disk eviction on memory full condition. It does not have built-in persistence functionality (neither in-memory, nor on-disk). And it puts all the dataset data on the local filesystems during the "shuffle" process.

Next misconception is that "***Spark performs 10x-100x faster than Hadoop***". Let's refer to one of the early presentations on this topic: http://laser.inf.ethz.ch/2013/material/joseph/LASER-Joseph-6.pdf. It states as a goal of Spark to support **iterative** jobs, typical for machine learning. If you refer to the Spark main page on Apache website, you would again see an example of where the Spark shines:



Logistic regression in Hadoop and Spark

And again, this example is about the machine learning algorithm called "Logistic Regression". What is the essential part of the most machine learning algorithms? They are ***repeatedly iterating*** over the same dataset many times. And here is where Spark in-memory cache with LRU eviction really shines! When you iteratively scan over the same dataset many times in a row, you need to read it only the first time you want to access it, after that you are just reading it from the memory. And it is really great. But unfortunately, I think they are running these tests in a bit tricky way – running on Hadoop they don't utilize HDFS cache capabilities (http://hadoop.apache.org/docs/r2.3.0/hadoop-project-dist/hadoop-hdfs/CentralizedCacheManagement.html). Of course they are not obligated to, but I think with this option the difference in performance would be reduced to approximately 3x-4x (because of more efficient implementation, no intermediate data put on HDDs, faster task startup times).


DIRECTOR OF MARKETECTURE
IT IS BETTER TO SEEM GOOD THAN TO BE GOOD.
BEING GOOD (OVERRATED)
A MISLEADING BENCHMARK TEST CAN ACCOMPLISH IN MINUTES WHAT YEARS OF GOOD ENGINEERING CAN NEVER DO.
IS IT OUR MATURITY THAT MAKES THAT CONCEPT SOUND OKAY?
I HOPE SO.
www.dilbert.com scottadams@aol.com
3-2-09 ©2009 Scott Adams, Inc./Dist. by UFS, Inc.

The long history of benchmarking in the enterprise space has taught me one thing: never trust the benchmarks. For any 2 systems competing with each other you would find a dozen of examples where SystemA is faster than SystemB and a dozen of examples where SystemB is faster than SystemA. What you can trust (of course, with a bit of scepsis) are the independent benchmarking frameworks like TPC-H – they are independent and are trying to prepare the benchmark that would cover most of the cases showing the real performance of the solutions.

In general, Spark is faster than MapReduce because of:

1. Faster task startup time. Spark forks the thread, MR brings up a new JVM
2. Faster shuffles. Spark puts the data on HDDs only once during shuffles, MR do it 2 times
3. Faster workflows. Typical MR workflow is a series of MR jobs, each of which persists data to HDFS between iterations. Spark supports DAGs and pipelining, which allows it to execute complex workflows without intermediate data materialization (unless you need to "shuffle" it)
4. Caching. It is doubtful because at the moment HDFS can also utilize the cache, but in general Spark cache is quite good, especially its SparkSQL part that caches the data in optimized column-oriented form

All of these gives Spark good performance boost compared to Hadoop, which can really be up to 100x for short-running jobs, but for real production workloads it won't exceed 2.5x – 3x at most.

And the latest myth, the one that is quite rare: "***Spark introduces completely new approach for data processing on the market***". In fact, nothing revolutionary new is introduced by Spark. They are good in implementing the idea of efficient LRU cache and data processing pipelining, but they are not alone. If you would be open-minded thinking about this problem, you would notice that in general they are implementing almost the same concepts that were earlier introduced by MPP databases: query execution pipelining, no intermediate data materialization, LRU cache for the table pages. As you see, in general Spark pillars are the same technologies existed on the market before Spark. But of course, the big step forward is that Spark implemented them in open source and provided them for the free use of the broad international community, where most of the companies were not ready to pay for the enterprise MPP technologies while still lacking the similar level of performance.

In the end, I would like to recommend you not to trust everything you hear from the media. Trust the subject matter experts, they are usually the best persons to ask.

Share this:

76

This entry was posted in Hadoop, Spark and tagged benchmarking, hadoop, in-memory, marketing, rdbms, Spark on February 18, 2015 [https://0x0fff.com/spark-misconceptions/] .

17 thoughts on "Spark Misconceptions"

Preeti Khurana
March 19, 2015 at 9:44 am

Very nice. Looking forward to more tech write ups on spark.

Seemanto Barua
April 15, 2015 at 8:19 pm

what about 'lineage'
and the fault tolerance that it allows for. Is that possible in hdfs cache ?

0x0FFF Post author
April 16, 2015 at 7:10 am

Let's start with fault tolerance. "Cache" concept assumes that data is persisted somewhere else. If it is not this way, this is not the "cache", but yet another persistent store. In case of HDFS, caching data does not mean removing it from HDFS, so you have full fault tolerance – if the cache entry is evicted, it can be easily read from HDFS (one of N copies of your data there). So HDFS cache is fully fault-tolerant

About lineage – Hadoop and MapReduce concept does not include multi-step computation model, so the data is persisted after each MapReduce job. Within the map or reduce task it is again lineaged – when you lose one map task you would recompute it on another node with the same input split (you don't have to restart everything), if the reducer fails it would copy map outputs once again on another node and execute there.

So again, Spark is a great engine for distributed compute, but it is good because of 2 things: caching and pipelining, and both of them are not new concepts. But I agree that Spark provides unique blend of them

---

Norbert Gergely
April 22, 2015 at 12:00 pm

And one more thing… Spark introduces the reactive pattern and async calls as a standard implementing way MR jobs. This I would also pose as an improvement.

Manoj Donga
June 10, 2015 at 1:41 pm

I could find out 10x faster when tried with repeatedly iterative data like group by queries and conditions on aggregated data. I am comparing with Hive ,which is MR in backend.

0x0FFF Post author
June 10, 2015 at 8:30 pm

Hive is not only MR in backend – you can have MR, Tez or Spark as a pluggable execution engines (and maybe new ones would follow later), so it depends.
I don't think that the example with "group by" is good, but the query with a set of window functions over different windows is a good one – it requires the same data to be sorted in different orders to be executed, which implies many scans of the same data. But again, I don't say that Spark cannot be 10x and even 100x faster than traditional MR, I just say that you won't have 10x and even 2x+ improvement by just moving to Spark – only certain cases would benefit from using it, but on average moving to Spark won't magically make your system blazing fast (especially if you've already spent some time optimizing Hive on Tez, for instance)

Deva
June 26, 2015 at 6:49 am

Awesome post. Thanks

Ladle
July 2, 2015 at 7:55 am

Thanks for Awesome post 🙂

Moniruddin Ahammed
July 8, 2015 at 10:47 am

This is really a good post ..I appreciate it very much .. Thank you ..

Hoang-Mai Dinh
July 16, 2015 at 8:33 am

Thanks for your useful post. I have a question of your statement "Spark puts the data on HDDs only once during shuffles, MR do it 2 times".

As I know, this is the data flow in MR.

http://s16.postimg.org/3x34ogi8j/2015_07_16_17_29_43.png

The first "local disk" is to store spill files, the second "local disk" is to merge all spill files into a big file. The third local disk is to store partition segments.

Could you please explain again why you said "MR is 2 times and Spark is only once"? Do you know the data flow in Spark?

Thank you so much again.

0x0FFF Post author
July 23, 2015 at 2:18 pm

Thank you for the question. Here you can find the code of Hadoop 2.7.1. Merge on the map side happens only if the mapper output has increased the ***mapreduce.task.io.sort.mb*** (100MB by default) and the buffer were flushed to disk in the middle of mapper processing. So it means that in MapReduce data hits the disk "at least 2 times", but the maximum is huge as it is possible to cause multiple on-disk merges on mapper side and multiple on-disk merges on reduce side.
For more details on MapReduce you can find this article of mine: https://0x0fff.com/hadoop-mapreduce-comprehensive-description/
On Spark you can take a look at my article: https://0x0fff.com/spark-architecture/

Israel Saeta Pérez
July 4, 2016 at 10:59 am

Hi Alexey.

Regarding the disk access during shuffle, is it possible that when using sort shuffle, if all shuffled data can fit into main memory without having to spill, there is no disk I/O needed? I.e. reducers can get data directly from memory instead of disk.

0x0FFF Post author
July 4, 2016 at 11:14 am

No, it is not possible in current implementation. The data is dumped to the disk when "mapper" task finishes. But if you have enough free RAM it might stay in OS cache, consider tuning it (for example, some basic concepts are described here)

Sonia
July 18, 2016 at 6:44 pm

What about https: spark.apache.org/docs/latest/programming-guide.html#rdd-persistence ? What is the difference between that called "persistence" and the reality that you say when stating that this is not persistence?

Another question I got is, Isn't the fact of the usage of HDD for some operations what Spark calls its "graceful degradation"? I understand that only when the data being utilised is bigger than the RAM available, only then, Spark uses HDD. How do you know that Spark puts all the dataset data on the local filesystems during the "shuffle" process?

Thanks.

0x0FFF Post author
July 19, 2016 at 7:08 am

Spark is not designed to be a persistent storage engine. What they call "persistence" is effectively caching, because:
1. The data is available to your session only. You cannot share cached RDD with other sessions
2. The data lifetime is limited by your session runtime. If your session is failed or terminated, you lose your cached data
In case you have some experience with databases, this caching works more or less like temporary tables

No, in Spark all of your data hits HDDs during the shuffle step. This is not written in official guides on Spark, but you can see it in the code, and it can be confirmed by the Databricks guys when asked directly. This is how Spark works.
Moreover, no overflow happens when *"the data being utilized is bigger than the RAM available"*, because Spark splits input data into "input splits" (by Hadoop InputFormats) and processes them on task-by-task basis, i.e. processing 1PB of data on the cluster with 1TB of RAM will not necessarily cause spilling, for example calculating "count()" won't cause it

---

Pingback: Spark体系架构 | 36大数据

Pingback: Spark体系架构 | TEC.eyearth