

Apache HAWQ: Next Step In Massively Parallel Processing

APRIL 12, 2016

ALEXEY GRISHCHENKO ([HTTPS://CONTENT.PIVOTAL.IO/AUTHORS/ALEXEY-GRISHCHENKO](https://content.pivotal.io/authors/alexey-grishchenko))



The first major version of Apache HAWQ

(incubating) (<http://hawq.incubator.apache.org/>) to benefit from the Apache Software Foundation governance will greatly enhance performance, combining the best of Massively Parallel Processing (MPP) and batch system designs, while overcoming key limitations. A newly redesigned execution engine addresses straggler-producing hardware failures, concurrency limits, needs for storing intermediate data, scale, and speed to introduce an order of magnitude of gains in overall system effectiveness.

The Pivotal HAWQ development process started more than 3 years ago with a fork of the **Greenplum database** (<http://greenplum.org/>). The main idea was running SQL queries on a Hadoop cluster over the data stored in HDFS. There were many enhancements introduced in HAWQ before the first public release of the software 3 years ago. But from the query execution standpoint, Pivotal HAWQ remained the same as the Greenplum database—it was a classical MPP execution engine.

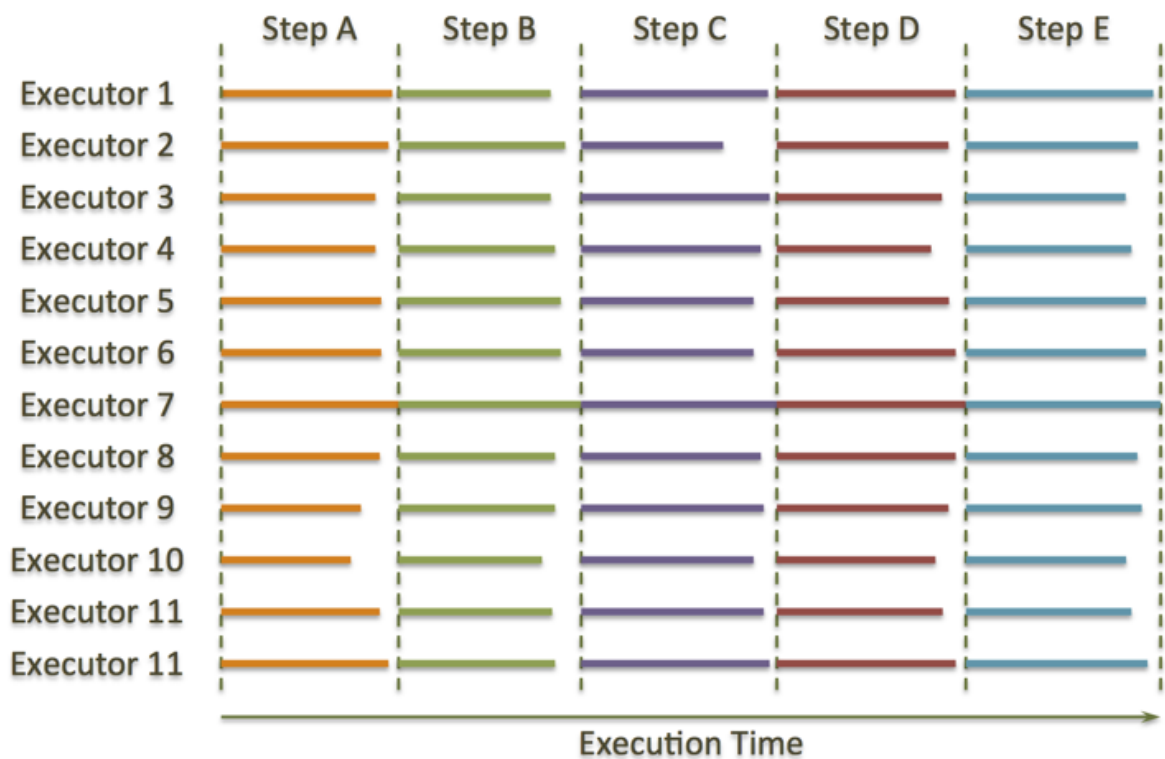
Since then the HAWQ codebase has been contributed to the ASF project and remains the core of **Pivotal HDB** (<http://pivotal.io/big-data/pivotal-hdb>), our commercial offering of Hadoop native SQL. Also, just this week, Hortonworks has announced that they will introduce an offering powered by Apache HAWQ in an expanded partnership with Pivotal (<http://hortonworks.com/press-releases/hortonworks-pivotal-expand-relationship-deliver-enterprise-ready-modern-data-platforms-data-management-analytics/>).

In this article, I am sharing the concept behind the new design of Apache HAWQ.

MPP Design

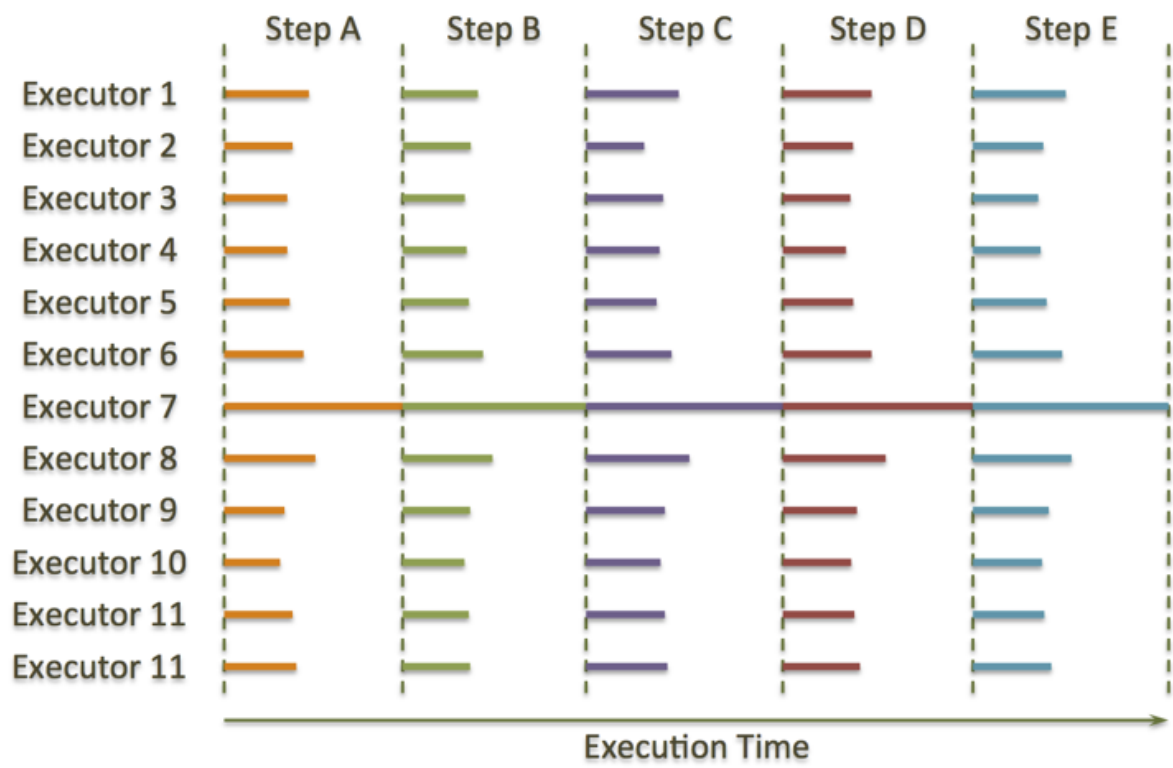
The original idea of all MPP solutions is the elimination of shared resources. Each executor has separate CPU, memory and disk resources. There is no option for one executor to access resources of the other one, except by a managed data exchange through the network. This concept of resource isolation works perfectly fine and enables high scalability for MPP solutions.

The second main concept of MPP is parallelism. Each executor runs exactly the same data processing logic, working on its private chunk of data from its local storage. There are some synchronization points between execution steps, and this synchronization is mostly implemented for data exchange (like shuffle step in **Apache Spark** (<http://spark.apache.org/>) and **MapReduce** (<https://en.wikipedia.org/wiki/MapReduce>)). Here is what a typical MPP query timeline looks like—each vertical line is a synchronization point. For example, synchronization is required for shuffling the data across the cluster to perform joins and aggregations, and the tasks itself might represent data aggregation, table joins, data sorting and other calculations executed separately by each cluster node.



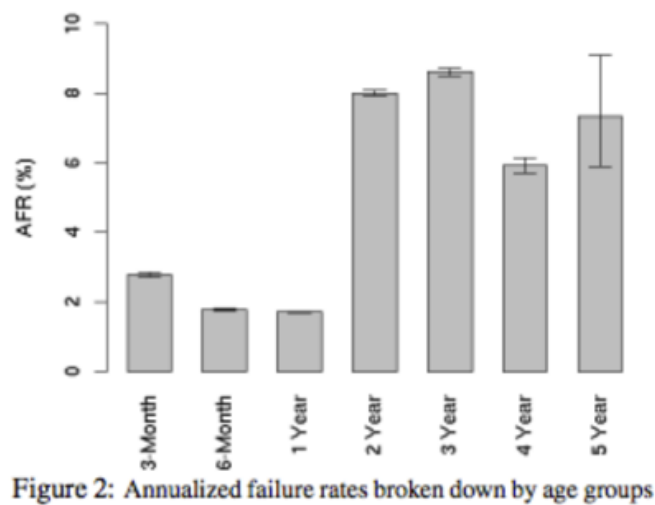
MPP Design Problems

But, this design leads to the main problem of all the MPP solutions—stragglers. If one node is constantly performing slower than the others, the whole engine performance is limited by the performance of this problematic node, regardless the cluster size. Here is an example of how the degraded node (Executor 7 in my example) can bring down the cluster performance:



Most of the time, all the executors are idle except one. This is because they are waiting for the synchronization with Executor 7, which is the source of our problems. For example, this can happen in an MPP system when RAID performance on one of the nodes degrades because of the failed disk, when CPU performance degrades because of hardware or OS-level problems, and so on. All MPP systems face this problem.

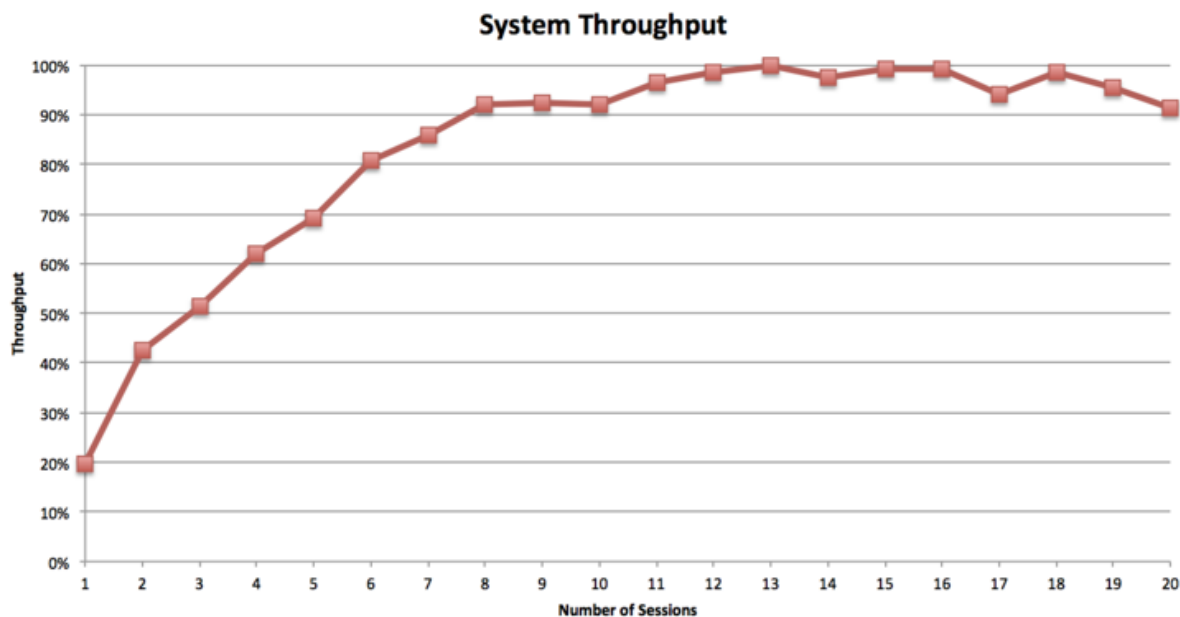
If you take a look at the research of Google regarding disk failure rates (https://www.usenix.org/legacy/events/fast07/tech/full_papers/pinheiro/pinheiro_old.pdf) at scale, you can see that the observed AFR (annualized failure rate) is, at best, 2% for the new drives that passed an initial 3 months of life:



With 1000 disks in a cluster, you would observe 20 failures a year or approximately bi-weekly failure. With 2000 disks, you would have weekly failure, and, with 4000 drives, you would have two failures a week. After 2 years of exploitation, you can multiply these numbers by 4, which means two failures a week for a 1000-disk cluster.

In fact, at a certain scale, your MPP system would always have a node with a degraded disk array, which would lead to degraded performance for this node, limiting the performance of the whole cluster as mentioned before. **This is the main reason why there are almost no MPP installations in the world with more than 50 servers in a single cluster.**

There is one more important difference between MPP solutions and batch solutions like MapReduce, and it is concurrency. 注意这个定义 **Concurrency is the number of queries that can be efficiently executed in parallel.** MPP is perfectly symmetric—when execution is running, each node of the cluster is performing exactly the same tasks in parallel. It implies that the concurrency level of an MPP solution is completely unrelated to the amount of nodes in the cluster. For example, clusters of 4 nodes and 400 nodes would support the same level of concurrency, and their degradation would start at almost the same point. Here is an example of what I'm talking about:



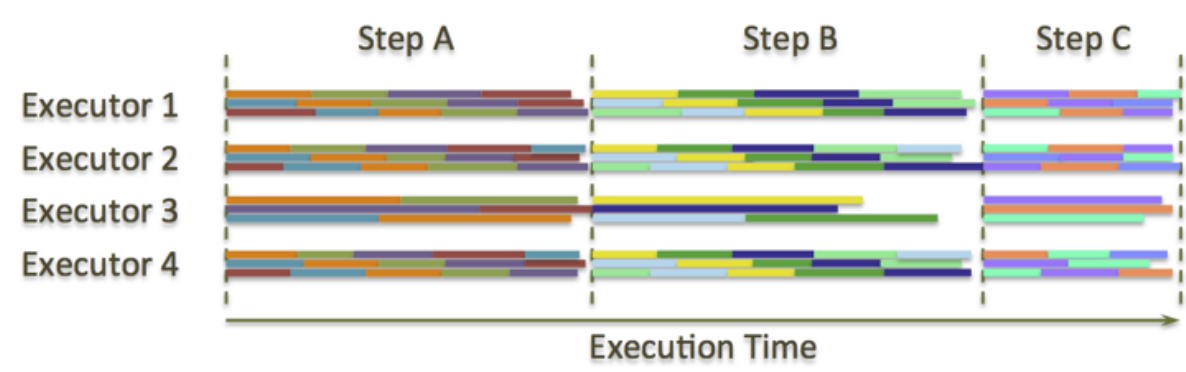
As you might see, 10-18 parallel sessions (queries) deliver the maximum throughput. If you built the graph further to 20+ sessions, it would show you a slow degradation to somewhat 70% of possible throughput and even lower. To clarify, the throughput is defined as a number of queries (of the same kind) executed in a fixed time range (that is a long enough interval for a representative result). A similar observation was made by the Yahoo team investigating Impala concurrency limitations (<http://hortonworks.com/blog/impala-vs-hive-performance-benchmark/>). FYI, Impala is an MPP engine on top of Hadoop. **Ultimately, low concurrency is the tradeoff MPP solutions have to pay for low query latency and high data processing speed.**

Batch Processing Design

mpp使每个机器上跑一样的程序，于是瓶颈在于每台机器所能达到的并行数量；由于无法通过加node来增加并行度，但从单个query上看又能获得较低的latency，所以是一个tradeoff。

To address this problem, a new class of solutions has emerged—starting with the MapReduce paper publication (<http://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf>) and its further evolution. This principle is implemented in such systems as Apache Hadoop

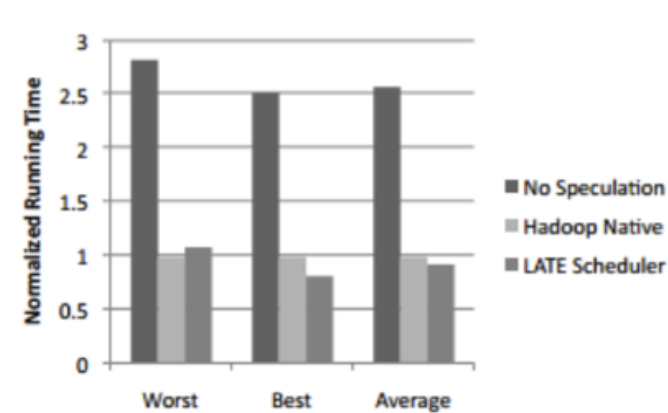
MapReduce (https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html), Apache Spark (<http://spark.apache.org/>) and some others. The main idea is a single execution “step” between two synchronization points is split into a number of separate “tasks”, and the number of tasks has completely no relation to the number of executors. For example, in MapReduce over HDFS the number of tasks is equal to the number of input splits, which is usually the same as the number of HDFS blocks in input files. Between the synchronization points, these tasks are assigned to executors in arbitrary order based on executor availability, in contrast with MPP where each processing task is bounded to specific executor holding required data shard. Synchronization points for MapReduce includes job start, shuffle and job end. For Apache Spark it is job start, shuffle, cached dataset and job end. Here is how the processing works—taking Apache Spark as an example, each bar represents a separate “task”, and each executor can process 3 tasks in parallel:



You can see that executor 3 is degraded—it is executing all the tasks almost 2 times slower. But it is not a problem—it just gets fewer tasks to execute. If the problem is getting worse, **speculative execution** (https://en.wikipedia.org/wiki/Speculative_execution) will help—the tasks of the slow node would be restarted on other ones.

This approach became possible because of shared storage. For processing a chunk of data, you don’t need to have this chunk of data stored on your specific machine. Instead, you can fetch it from the remote machine. Of course, remote processing is always more expensive compared to the local one because the data has to move—so the engine tries, as much as it can, to process the data locally. But in the case of degraded machines and batch processes, speculative execution helps with the degraded nodes, which is completely impossible in MPP solutions.

Here is a good study of speculative execution in the cloud (https://www.usenix.org/legacy/event/osdi08/tech/full_papers/zaharia/zaharia.pdf):



This picture is related to **WordCount** (https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Example:_WordCount_v1.0) performance. As you might see, speculative execution can speed execution time almost 2.5x in a cloud environment, and cloud environments are well-known for their straggler problems (<http://blog.scalyr.com/2012/10/a-systematic-look-at-ec2-io/>). A combination of shared storage and more granular scheduling allows batch processing systems to scale better than MPP solutions—to thousands of nodes and tens of thousands of HDDs.

Batch Processing Design Problems

But, everything comes at a cost. With MPP, you don't need to put intermediate data on the HDDs because a single executor processes a single task and can just stream its data to the next execution task directly. This is called “pipelining”, and it delivers greatly improved performance. When you have a number of non-related tasks that can be executed serially by a single executor, like in batch processing, you have no option but to **store the intermediate results on the local drives** (<http://0x0fff.com/spark-architecture-shuffle/>) before the next execution step would be started to consume your data. This is what makes these systems slower.

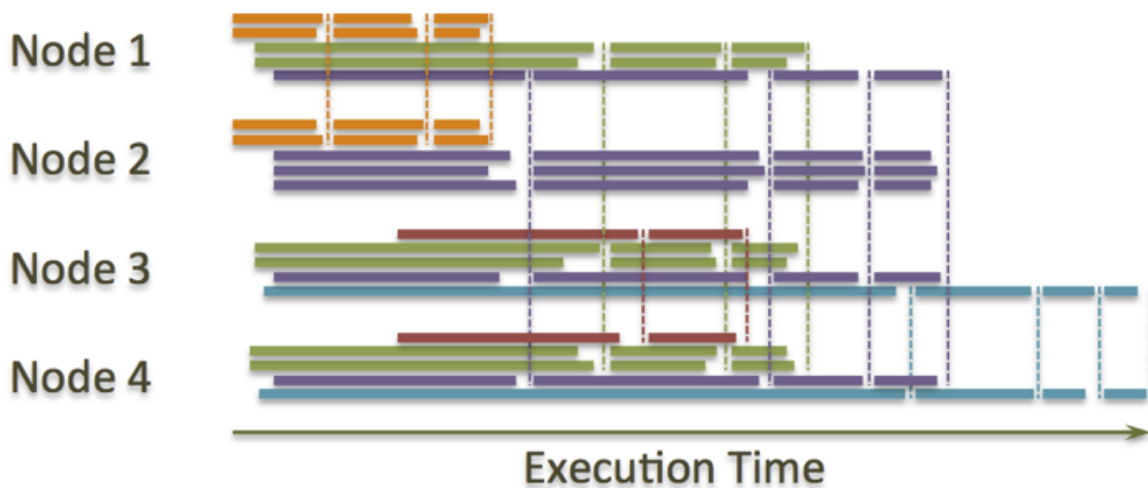
In my experience, comparing performance of a modern MPP system to a solution like Apache Spark—on the same hardware cluster—**would show that Apache Spark is generally slower and can be 3x-5x slower.** The reasonable limit of 50 machines in a MPP cluster would perform on the level of 250-node cluster of Apache Spark, but Spark can scale beyond 250 nodes, which is not possible for MPP.

Combining MPP and Batch

We can now look at the two architectures in terms of strengths and weaknesses. MPPs are much faster, but face two key pain points—stragglers and concurrency limits. With batch processing via MapReduce, we need to spend time storing intermediate results to disk, but at the same time, we can have cluster concurrency ratio scaled together with the cluster size and a cluster size that is superior to MPP solutions. How can we combine them both to have low latency and processing speed of MPP solution, mitigating the stragglers and concurrency problem of MPP solutions with batch processing-like design? I don't think you will be surprised if I tell you the answer is in the new Apache HAWQ design.

Again, how is a MPP query executed? By a number of parallel processes doing exactly the same work, with exactly the same number of processes, on each of the cluster nodes, where each of them has the data they process on local storage. But when we introduce HDFS, you are no longer tied to the local storage of the executors, which means you can get away from a fixed number of executors, and you can get away from fixed nodes where these executors must run to process their local data (i.e., you cannot process remote data in traditional MPP). Why? Because HDFS stores 3 replicas of the same block by default, which means there would be at least 3 nodes in the cluster where you can start an executor for this block to process its data locally. Also HDFS supports remote fetching of the data, which means there are at least 2 racks where you can process them with rack locality, fetching the data from remote machines with the least possible number of networks hops utilizing fast top of the rack switches.

This is why Apache HAWQ has introduced a concept of “virtual segments”—“segment” in Greenplum this is a single instance of modified PostgreSQL database residing on one of the nodes and spawning “executor” processes, one per query. If you have a small query, it can be executed by 4 executor processes or even one. If you have a big query, you might start 100 or even 1000 executors. Each query is still executed in MPP fashion with processes acting on local data (where possible) and without putting your intermediate data back on the HDDs, but virtual segments allow executors to be located anywhere. Here is how this looks like (each different color is a different query, dot lines are shuffles within a query):



This allows you to:

1. Mitigate the straggler problem of MPP systems because we can dynamically add nodes to the cluster and remove them. So, hard disk failures will not bring down performance of the whole cluster, and the system can have an order of magnitude more nodes than traditional MPPs. Now, we can temporarily remove a problematic node from the cluster, and no more executors will start on it. And, no downtime is required to remove the node.
2. A query is now executed by a dynamic number of executors, which gives you much higher concurrency, mitigating the limit in MPP systems and allowing for the greater flexibility of batch systems. Imagine the system with 50 nodes, where each can run up to 200 parallel processes. It means that you would have $50 \times 200 = 10,000$ “execution slots” in total. You can utilize them for 20 queries with 500 executors each, for 200 queries with 50 executors each, or even have a single query with 10000 executors running alone on the system. You are completely flexible here. You might have one big query with 4000 segments and 600 other queries with 10 executors each—it would still work well.
3. Fully utilize data pipelining, moving data from one executor to another in real time, between the execution stages—separate queries are still MPP and not batch. So, there is no need to put intermediate data to the local drives (whenever operations allow pipelining). This means we are getting closer to the speed of MPP systems.
4. Like MPP systems, we still keep executing on top of local data as much as possible with short-circuit HDFS reads. Each executor still tries to start its execution on the node with the highest amount of local blocks for the file it reads, which also maximizes performance.

Learning More

Apache HAWQ has introduced a completely new design, which is basically a combination of MPP and Batch, including the best from both worlds and addressing the key problems with each. Of course, there is no ideal solution for data processing—MPPs are still faster and batches still have higher concurrency and scalability. This is why choosing a specific solution for specific data processing problem is still the key, and we have a number of experts to support you. As a further reading you can take a look at **my presentation** (<http://www.slideshare.net/AGrishchenko/apache-hawq-architecture>) about Apache HAWQ, also look here (<https://blog.pivotal.io/big-data-pivotal/products/introducing-the-newly-redesigned-apache-hawq>) and here (http://events.linuxfoundation.org/sites/events/files/slides/hawq-apachecon-final-slides.pptx_.pdf). You can also subscribe to the user mailing list (<http://hawq.incubator.apache.org/#mailing-lists>).

About the Author

Biography

More Content by Alexey Grishchenko (<https://content.pivotal.io/authors/alexey-grishchenko>)

Previous

<https://content.pivotal.io/blog/how-pivotal-web-services-just-improved-the-security-of-your-platform>




How Pivotal Web Services Just Improved the Security of Your Platform

How up-to-date are your application environments?

Next

<https://content.pivotal.io/blog/case-study-refactoring-a-monolith-into-a-cloud-native-app-part-4>




Case Study: Refactoring A Monolith into a Cloud-Native App (part 4)

In this blog series, we have used the SpringTran...

Subscribe to our Newsletter

Read More




<https://pivotaltracker.com/blog/product-stack-denver-panel/>

The Product Stack

Product Stack Denver—Panel


Read More



<https://content.pivotal.io/blog/how-crunchy-data-closed-the-enterprise-postgresql-gap-all-the-way-to-cloud-native>

How Crunchy Data Closed the Enterprise PostgreSQL Gap - All the Way to Cloud-Native







Read More



<https://pivotal.io/blog/how-to-change-careers-later-in-life-into-technical-role>

How to Change Careers Later in Life into a Technical Role

Read More

LEARN (https://pivotal.io/learn)	BUILD (https://pivotal.io/build)	CONNECT (https://pivotal.io/connect)	COMPANY (https://pivotal.io/about)	<div>SpringOne Platform</div> <div> 2017 in San Francisco → (https://springoneplatform.io) 2016 Conference Highlights → (https://springoneplatform.io/2016) </div> <div>  (https://twitter.com/pivotal)  (https://www.linkedin.com/company/pivotal-software)  (https://medium.com/built-to-adapt)  (https://www.youtube.com/channel/UCpivotal)  (https://plus.google.com/105320)  (https://www.facebook.com/pivotal) </div>
Topics (https://pivotal.io/topics)	Tutorials (https://pivotal.io/platform/pcf-tutorials/getting-started-with-pivotal-cloud-foundry?utm_source=uberflipfooter&utm_medium=link&utm_campaign=gettingstartedwithpcf)	Advocates (https://pivotal.io/advocates)	About (https://pivotal.io/about)	
Products (https://pivotal.io/products)	tutorials/getting-started-with-pivotal-cloud-foundry?	#ImAPivot (https://pivotal.io/imapivot)	Locations (https://pivotal.io/locations)	
Industries (https://pivotal.io/industries)	utm_source=uberflipfooter&utm_medium=link&utm_campaign=gettingstartedwithpcf	Events (https://pivotal.io/events)	Careers (https://pivotal.io/careers)	
Resources (/resources)	PCF Dev (https://pivotal.io/pcf-dev)	Product Consultation (https://pivotal.io/office-hours)	Contact (https://pivotal.io/contact)	
Training (https://pivotal.io/training)	Documentation (http://docs.pivotal.io/)	Webinars (/webinars)	Executive Team (https://pivotal.io/team)	
Customers (https://pivotal.io/customers)	Downloads (https://network.pivotal.io/)	Blog (/)	Investors (https://pivotal.io/investors)	
	Knowledge Base (https://discuss.pivotal.io)	Podcasts (/podcasts)	Press Center (https://pivotal.io/press-center)	
	Support (https://support.pivotal.io)	Social (/social)	Partners (https://pivotal.io/partners)	
	Newsletter (https://pivotal.io/newsletter-subscription)			
	PCF Services (https://pivotal.io/platform/services)			

Subscribe to our
Newsletter

Email