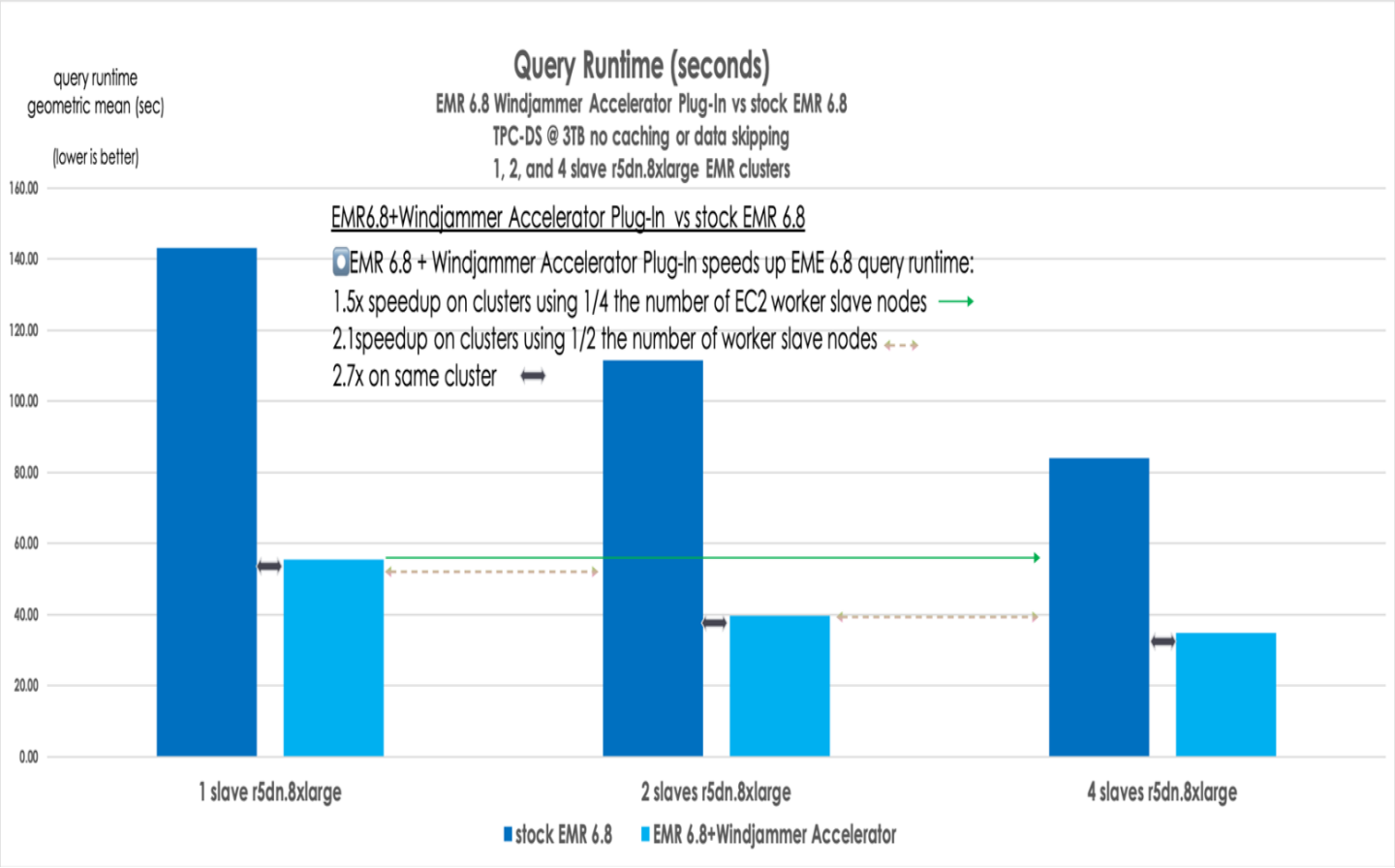
EMR 6.x + Windjammer's EMR Spark Accelerator Plug-In

White Paper
August 2023

This white paper discusses how AWS EMR 6.x Spark plus Windjammer's EMR Spark Accelerator Plug-In provides industry leadership in Spark query execution performance, price/performance, fault tolerance, and predictability with 100% transparency and compatibility.

Summary

AWS EMR provides a powerful managed cluster platform that simplifies running big data frameworks on AWS to process and analyze vast amounts of data. Windjammer EMR Spark Accelerator is a breakaway Cloud Data Lake Query Native Execution Engine that transparently plugs into AWS EMR 6.x Spark cluster deployments as a bootstrap extension and JAR. Adding the Windjammer Spark EMR Accelerator plug-in reduces the cost of EMR Spark deployment >50% while accelerating performance ~3X. Windjammer EMR Spark Accelerator also provides cloud store based checkpointing which increases query fault tolerance with rapid query fault recovery from any software, node, or cluster interruption, including loss of full cluster, without the need for an external shuffle service. Beyond disruptive cost, performance, and fault tolerance improvements, Windjammer's EMR Spark Accelerator provides predictable and reliable performance, reducing both query elapsed time and run time variance to ~1/3.



This white paper describes the mechanisms by which EMR 6.x plus Windjammer's EMR Spark Accelerator disrupts big data cloud analytics performance and cost with high availability and reports the competitive benchmark results.

Introduction

The need to process large data sets using a variety of applications (analytics, artificial intelligence (AI), and machine learning tools, …) for several purposes has led to the prevalence of Data Lakes as shared repositories of structured and unstructured data at any scale. Data lakes are typically realized using a highly available shared storage repository decoupled from compute clusters and accessed over an interconnected network, typically Ethernet, into which authoritative data is stored, such as a public cloud object store (for example, Amazon S3, Azure Data Lake Store (ADLS), or Google Cloud Object Store (GCS)) or a shared storage system that supports the Hadoop Distributed File System (HDFS) or the Network File System (NFS) protocol.

Separation of the physical computer systems responsible for performing computational work (collectively known as compute nodes) and those responsible for storing digital data (collectively known as storage nodes) is a common architecture for big data applications in large-scale deployments in enterprises and in public clouds. This deployment model enables independent provisioning, scaling, and upgrading of compute clusters and storage clusters. Compute clusters are typically created on-demand, and additions and changes may be made to the number of physical computer systems constituting nodes of the compute cluster. This flexibility is termed "**elastic scaling**". Nodes of a cluster may be transient and made available for inclusion in the cluster for a limited time, and only a short programmatic advance warning of their unavailability (for example, thirty seconds) may be given. An example of a transient node is Amazon's EC2 Spot Instance.

Since compute clusters and instances are transient, permanent and intermediate data must be stored or replicated outside the compute cluster in a shared repository.

Providing efficient and fault tolerant job and query execution on transient, elastic compute clusters accessing disaggregated data lakes presents many fundamental challenges to the present state of the art, including performance, financial cost, fault tolerance, and predictability.

Spark's Fundamental Technical Limitations

Spark has fundamental technical limitations in executing jobs and queries on Data Lakes using disaggregated transient compute nodes and cloud object stores. Spark uses the JVM for query execution, which is very CPU intensive and causes server sprawl and high operating cost. Spark was not designed for disaggregated computing, and is unable to utilize the bandwidth of cloud stores to mask their high latency. Spark fault tolerance is based on RDDs running in JVMs saving intermediate data to local storage.

The impacts of Spark's fundamental limitations are:

- Spark JVM query execution is CPU intensive and garbage collection causes performance instability: increases the number of required worker nodes, high cost of deployment, performance instability and management challenges
- Spark cannot exploit high bandwidth of today's cloud storage systems: long query run times
- Spark fault tolerance: requires complex and expensive shuffle services, and cannot recover from cluster failures (jobs and queries must be restarted from the beginning)

The challenge is to create software that that transparently and compatibly addresses the Performance, Cost, Availability, and Predictability limitations in the SPARK ecosystem

Windjammer EMR Spark Accelerator

Windjammer Technologies has invented a fundamental new architecture which solves inherent problems in executing Spark jobs and queries on cloud Data Lakes utilizing disaggregated compute and storage.

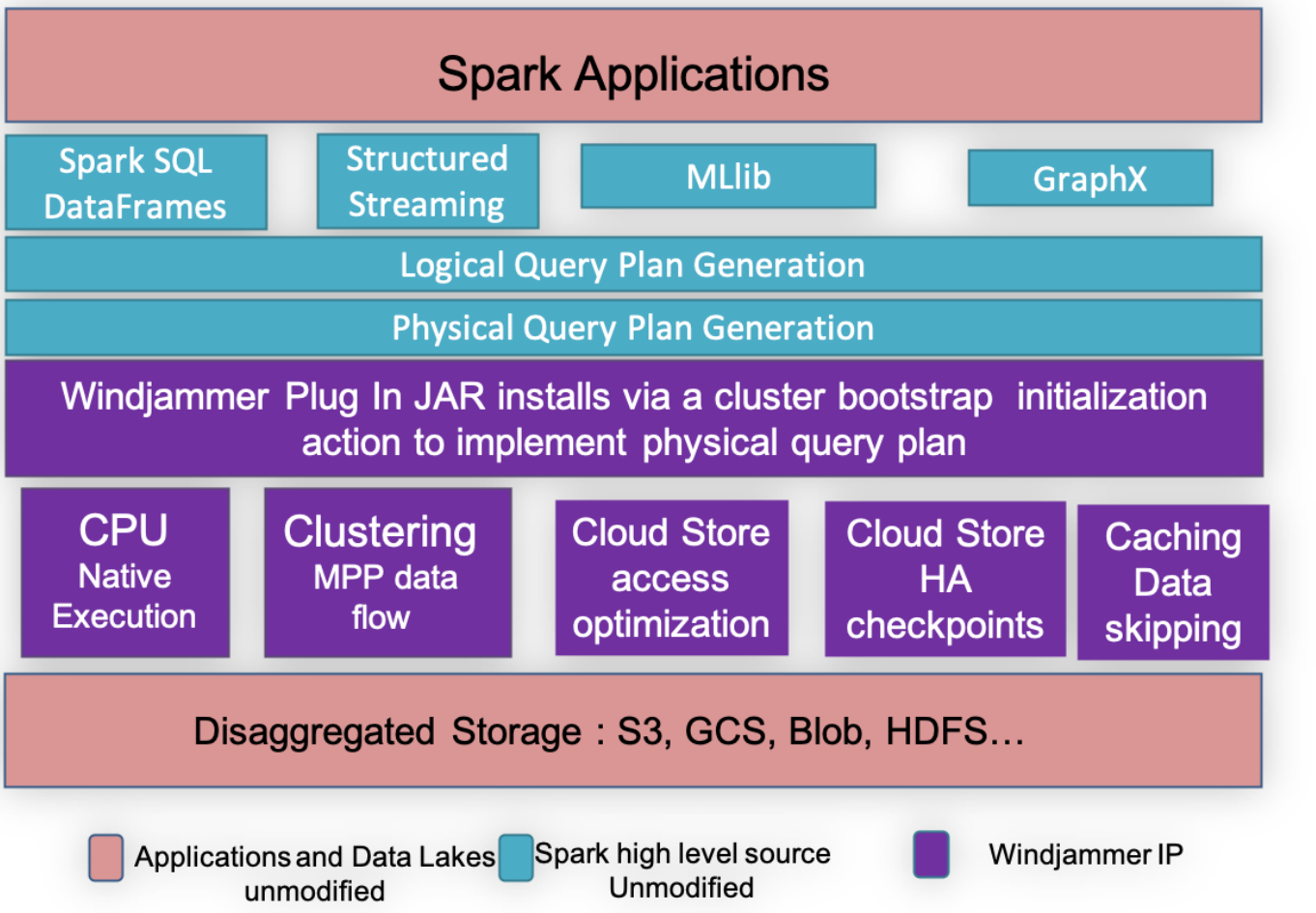
The key architectural elements of Windjammer's EMR Spark Accelerator are:

- Native query execution
  - Eliminates JVM overheads, cuts CPU time per query, eliminates server sprawl
- Massively Parallel Processing (MPP) data flow query execution across the compute cluster
  - Eliminates Map-Reduce bottlenecks and dependencies on local storage
- Precise, parallel, asynchronous prefetching provides Data Lake access optimization
  - Provides 3X utilization of each compute node's Data Lake repository bandwidth to drive MPP data flow pipelines
- Cloud store based efficient and automatic query checkpointing and fault recovery
  - Enables broad query fault tolerance (including cluster and spot instance interruptions and auto-scale events) using fully disaggregated compute nodes and cloud storage
- Transparent plug-in native execution engines for AWS EMR 6.X
  - Transparently accelerates and cost reduces workload deployments with 100% compatibility while increasing fault tolerance and predictability

Windjammer EMR Spark Accelerator Architecture

The block diagram below shows Windjammer's cloud SQL query native execution engine integrated into Spark. Windjammer's Spark Native Engine (SNE) is depicted in Purple in this diagram. in this diagram. All other elements of the software stack, including Spark applications, Spark libraries, Spark connectors, Spark query optimizer, and storage access libraries are completely unmodified.

Note that SNE plugs in below Spark physical query plan generation as a SparkSQL extension. As a result, all spark applications and APIs and Spark high level processing are not affected, providing transparency and 100% compatibility. SNE takes the physical plan and executes it natively across the nodes in the cluster, implementing a MPP dataflow computation model. The JVM is not used during query execution thereby cutting cpu overhead and eliminating garbage collection instabilities. During query execution, SNE maximizes cloud store bandwidth utilization of each compute node using precise parallel asynchronous prefetching, and SNE efficiently checkpoints to the cloud store for query fault tolerance.

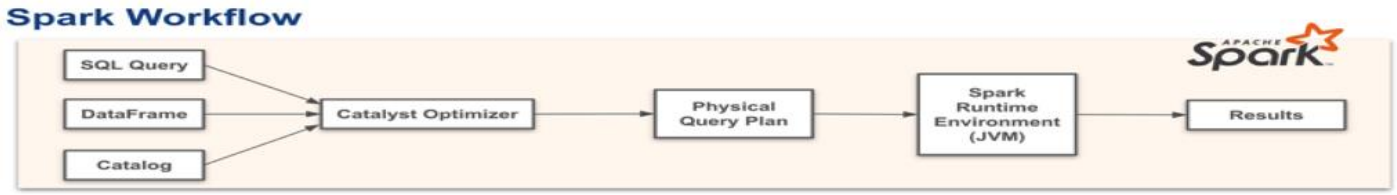


Patents:
Disaggregated Query Processing on Data Lakes Based on Pipelined, Massively Parallel, Distributed Native Query Execution on Compute Clusters Utilizing Precise, Parallel, Asynchronous Shared Storage Repository Access
U.S. Patent No. 11,327,966 B1
Published Patent Application No. 2022/0277006 A1

Windjammer EMR Spark Accelerator Design

The diagram below presents an overview of Spark's workflow.

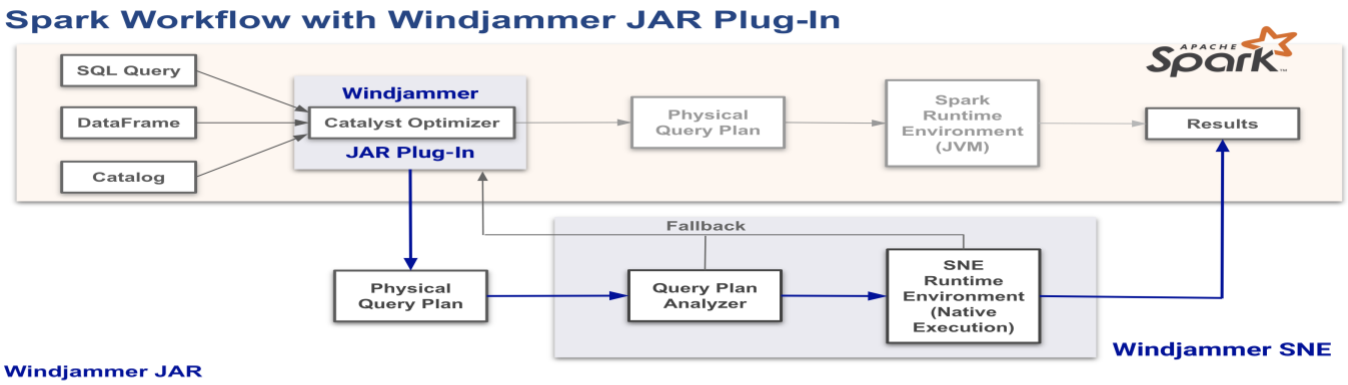
The user submits the SQL query along with a DataFrame and the locations of the datasets required by the query, which we call catalog here. Spark's Catalyst optimizer takes all this information and creates a physical query plan, which Spark's JVM-based runtime environment uses to execute the queries. The results are then returned to the user.



The diagram below shows what the workflow looks like when we add Windjammer to Spark. The Windjammer JAR is deployed through Spark's extension framework as a lightweight JAR that can be installed with a single click and that transparently plugs in as a thin wrapper around the query plan optimizer. What this JAR does is direct the physical query plan to Windjammer SNE (Spark Native Execution), where it gets analyzed by the query plan analyzer and then sent to SNE's runtime environment for query execution. The advantage of wrapping around the query optimizer is that we're able to leverage the query optimizations. After completing the query execution, SNE hands the results to Spark, where they then get passed to the user as if Spark had done the computation itself.

If the query plan analyzer determines that SNE can't successfully accelerate the query for some reason (for instance if it has UDFs which SNE is accelerating in the next release), the JAR transparently falls back and hands the plan execution to Spark's stock JVM query engine.

The whole workflow is transparent to the user and transparent to all query submission methods, it'll work without any changes to existing applications. All queries complete with exact semantics of stock Spark



Windjammer JAR

- Transparently plugs in as a thin wrapper around the query plan optimizer
- Directs physical query plan to Windjammer's Spark Native Execution (SNE) engine instead of Spark, EMR, or Dataproc
- Deployed through `spark.sql.extensions` framework
- Transparent fallback to Spark, EMR or Dataproc if Windjammer SNE is unable to accelerate
- Transparent to all submission interfaces such as `spark-submit`, `spark-shell`, `spark-sql`, notebooks, etc.

Below is an example of a physical query plan. It illustrates the kind of details that are present in a query plan, although usually the query plans in practice are much longer than this. Different query operations are on separate lines and arranged in a tree structure

Spark Physical Query Plan

Query plan is a text file with various query operations arranged in a tree

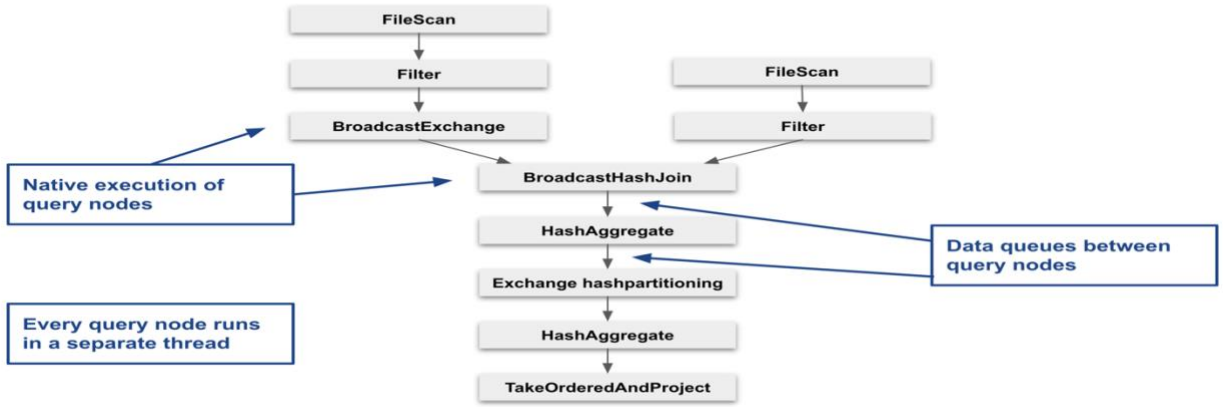
```
== Physical Plan ==
TakeOrderedAndProject(limit=100, orderBy=[`d_year#14` ASC NULLS FIRST,`sum_agg#6` DESC NULLS LAST,`item_sk#5` ASC NULLS FIRST],
output=[`d_year#14`,`item_sk#5`,`sum_agg#7`])
+- *HashAggregate(keys=[`d_year#14`, `ss_item_sk#38`], functions=[sum(UnscaledValue(`ss_ext_sales_price#51`))], output=[`d_year#14`,
`ss_item_sk#38`, `sum_agg#7`], aggrAttr=[`sum(UnscaledValue(`ss_ext_sales_price#51`))#81L`], resultExpr=[`d_year#14`, ``ss_item_sk#38` AS
item_sk#5`, `MakeDecimal(`sum(UnscaledValue(`ss_ext_sales_price#51`))#81L`,17,2) AS sum_agg#6`])
  +- Exchange hashpartitioning(`d_year#14`, `ss_item_sk#38`, 200) [id=#154]
     +- *HashAggregate(keys=[`d_year#14`, `ss_item_sk#38`], functions=[partial_sum(UnscaledValue(`ss_ext_sales_price#51`))],
output=[`d_year#14`, `ss_item_sk#38`, `sum#90L`], aggrAttr=[`sum#89L`], resultExpr=[`d_year#14`, `ss_item_sk#38`, `sum#90L`])
        +- *Project [`d_year#14`, `ss_item_sk#38`, `ss_ext_sales_price#51`]
           +- *BroadcastHashJoin [`d_date_sk#8`], [`ss_sold_date_sk#36`], Inner, BuildLeft
              :- BroadcastExchange BroadcastPartitioning(HashedRelationBroadcastMode(List(cast(input[0, int, true] as bigint)),false))
              :  +- *Project [`d_date_sk#8`, `d_year#14`]
              :     +- *Filter ((isnotnull(`d_moy#16`) && (`d_moy#16` = 11)) && isnotnull(`d_date_sk#8`))
              :        +- FileScan parquet [`d_date_sk#8`,`d_year#14`,`d_moy#16`] Batched: true, DataFilters: [isnotnull(d_moy#16),
(d_moy#16 = 11), isnotnull(d_date_sk#8)], Format: Parquet, Location: InMemoryFileIndex[...], PartitionFilters: [], PartitionSchema:
struct<>, PushedFilters: [IsNotNull(d_moy), EqualTo(d_moy,11), IsNotNull(d_date_sk)], ReadSchema: struct<d_date_sk:int,d_year:int,d_moy:int>
              +- *Filter (isnotnull(`ss_sold_date_sk#36`) && isnotnull(`ss_item_sk#38`))
                 +- FileScan parquet [`ss_sold_date_sk#36`,`ss_item_sk#38`,`ss_ext_sales_price#51`] Batched: true, DataFilters:
[isnotnull(ss_sold_date_sk#36), isnotnull(ss_item_sk#38)], Format: Parquet, Location: InMemoryFileIndex[...], PartitionFilters: [],
PartitionSchema: struct<>, PushedFilters: [IsNotNull(ss_sold_date_sk), IsNotNull(ss_item_sk)], ReadSchema:
struct<ss_sold_date_sk:int,ss_item_sk:int,ss_ext_sales_price:decimal(7,2)>
+-----------------------------------------------------------+
```

The diagram below shows how SNE's query plan analyzer takes this query plan and changes it into a graph structure that corresponds to the tree structure in the query plan, where the different query operations constitute different nodes in the graph. Each class of query operation has been implemented in C/C++ and is compiled natively.

The query nodes in the graph are connected with data queues that send processed data from upstream nodes to downstream nodes. In SNE's runtime environment, each of the nodes in the graph is run in a separate thread, and each of these threads may in turn spawn additional threads during runtime if it helps in processing the data more efficiently.

SNE Query Graph

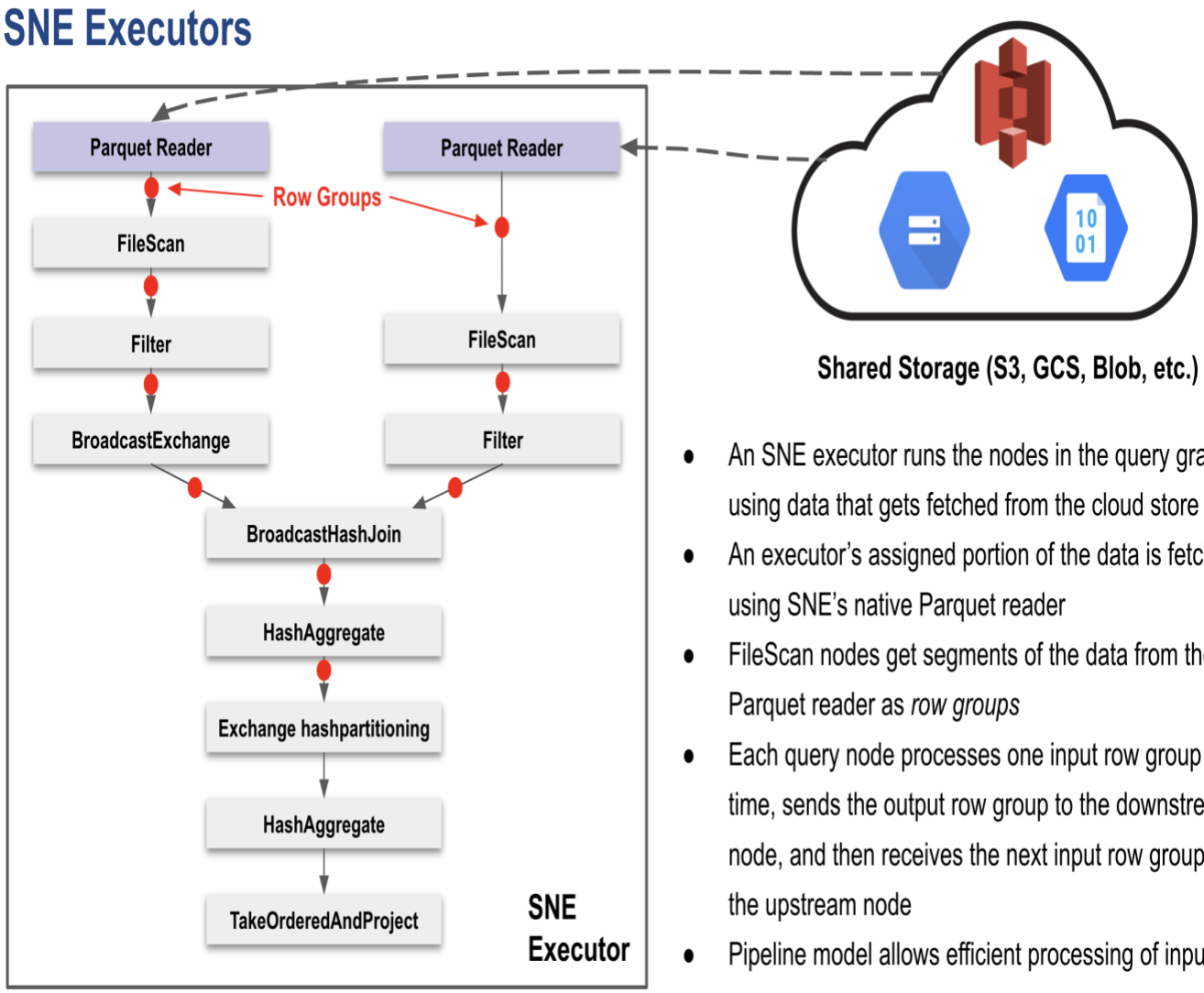
- SNE's *query plan analyzer* converts a query plan into a query graph
- SNE's *runtime environment* runs natively-compiled code for the query operations represented by the query nodes



Inside SNE, a query graph is executed in an SNE executor. Typically, SNE has many executors in a node and in a cluster, and each executor is responsible for processing a part of the data set. The data sets can be on-premises or stored on the cloud and the data is obtained in a suitable format for processing by the FileScan query nodes, which connect to a file reader.

In this example, FileScan nodes are connected to instances of SNE's native Parquet reader that fetch the data required by their connected FileScan node from the cloud store. The Parquet readers fetch only the data that's actually required, and the data is then sent to the downstream node in chunks that we call row groups.

Each query node processes one input row group at a time and sends the processed data to its downstream node in an output row group. After that it receives the next input row group until all the data has been processed. In this way, SNE uses a pipeline model that allows for the efficient processing of the data.

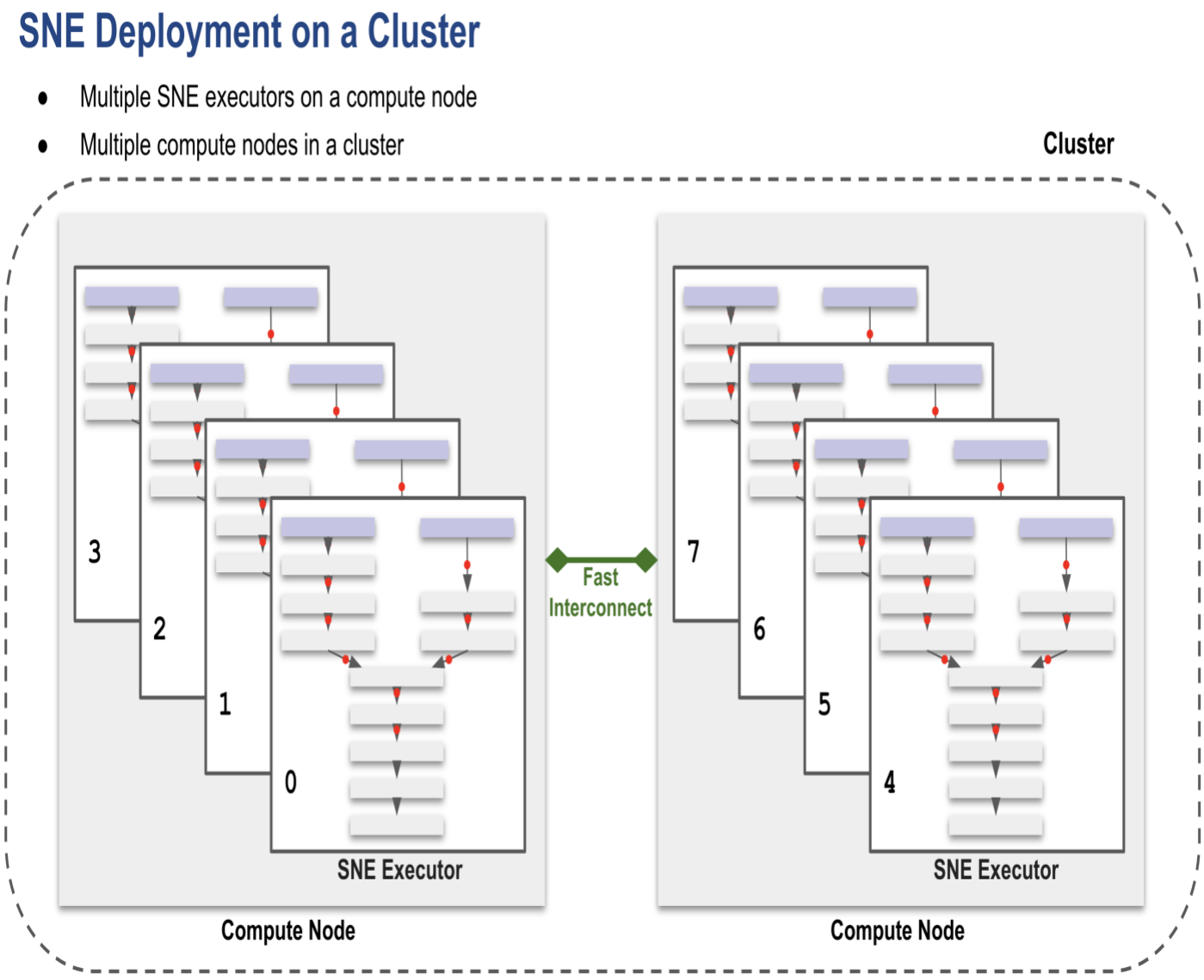


- An SNE executor runs the nodes in the query graph using data that gets fetched from the cloud store
- An executor's assigned portion of the data is fetched using SNE's native Parquet reader
- FileScan nodes get segments of the data from the Parquet reader as *row groups*
- Each query node processes one input row group at a time, sends the output row group to the downstream node, and then receives the next input row group from the upstream node
- Pipeline model allows efficient processing of input data

The prior diagram showed a single SNE executor, but in practice there will be multiple executors running across multiple compute nodes, which are individual machines in a cluster.

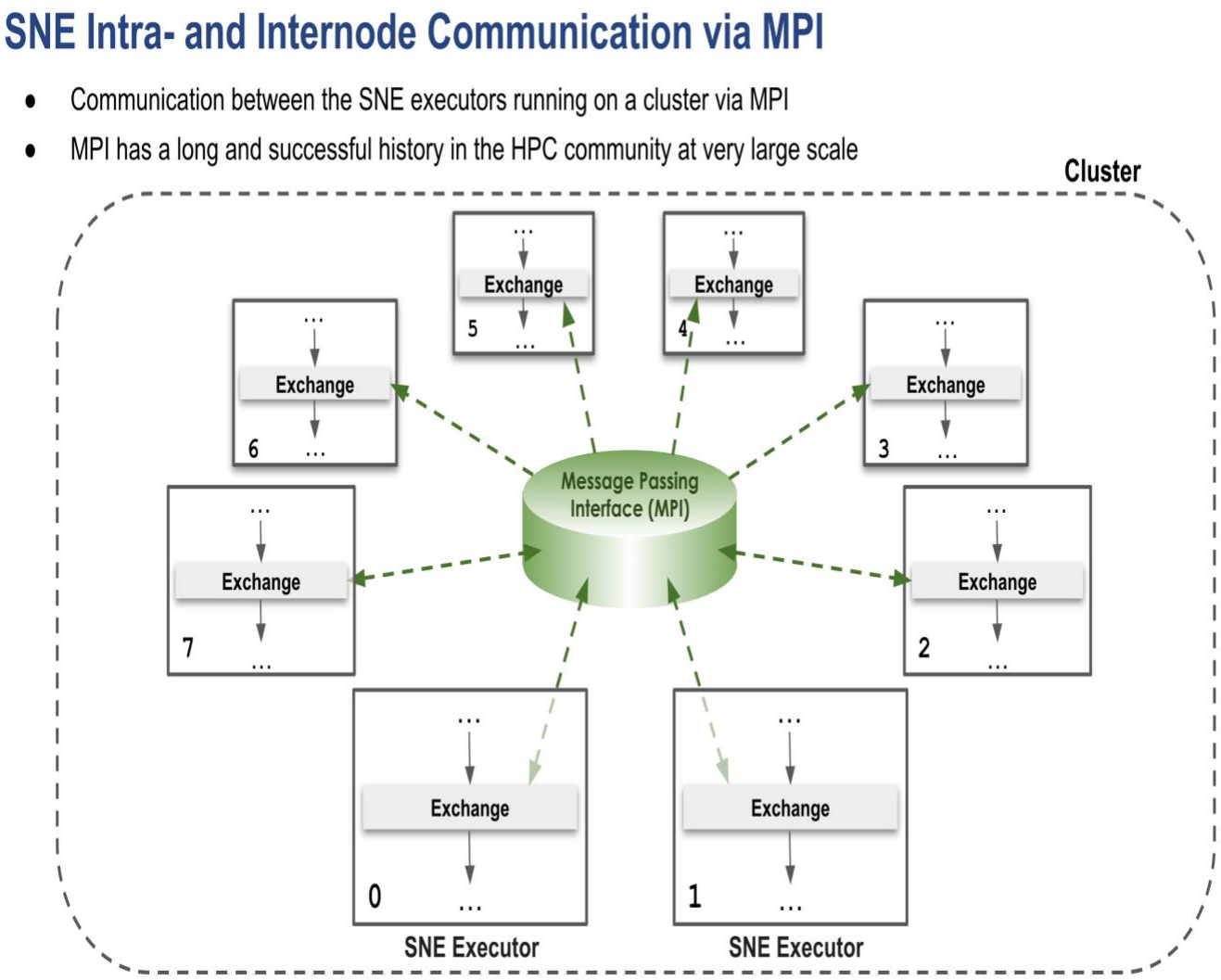
In the example on this slide below we have two compute nodes in the cluster with four executors each. In practice there will be more compute nodes and more SNE executors running on each of them to fully utilize the cluster compute resources available for the query execution.

The compute nodes are connected by typically interconnected with high speed ethernet for efficient data transfer.



Communication between SNE executors is done using the Message Passing Interface, or MPI, which has a long and successful history in the high-performance computing community at very large scale. This is important especially for the Exchange nodes in the query plan, where the data in the incoming row groups is exchanged between the different executors based on some partitioning scheme.

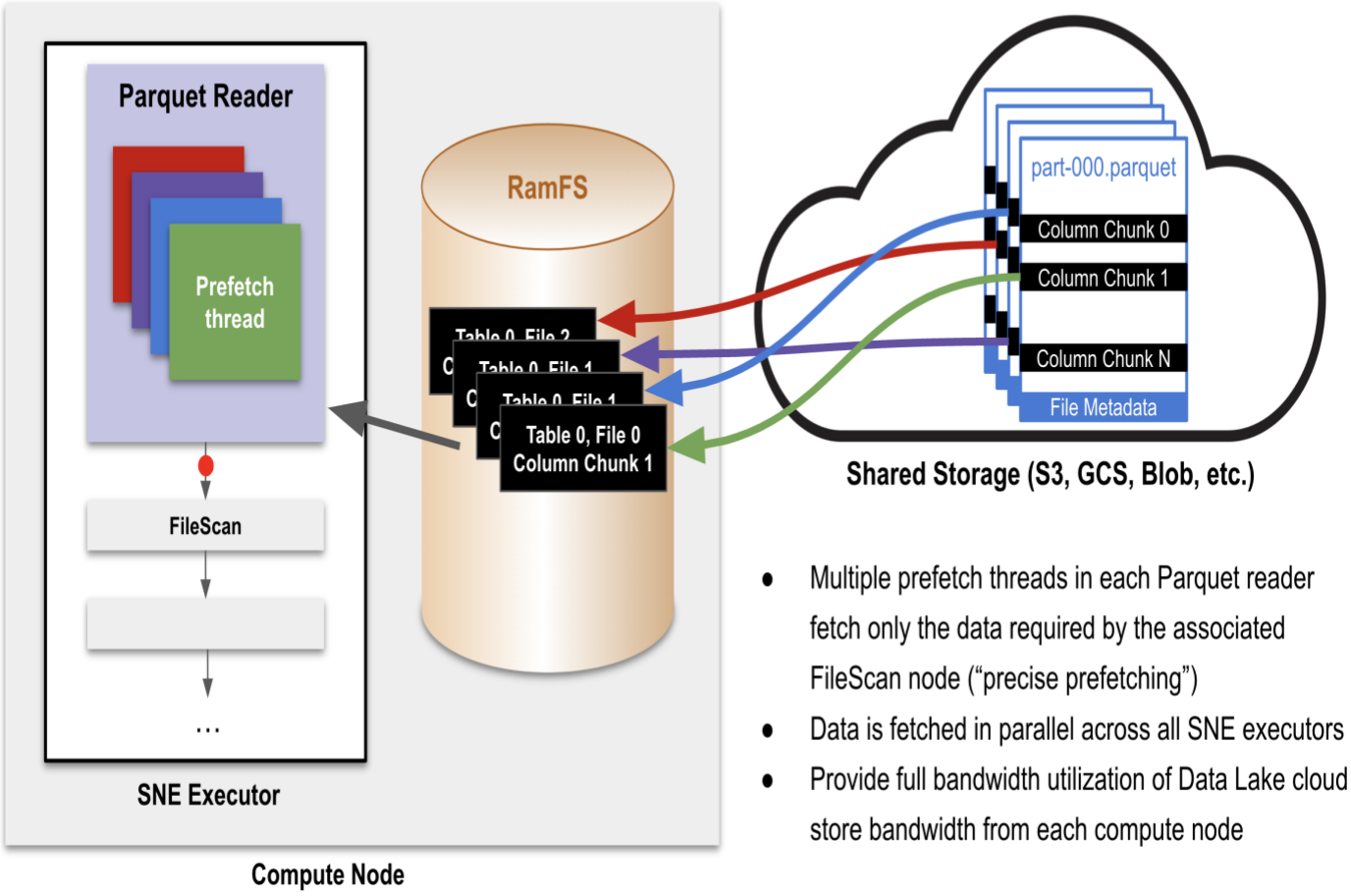
In the example shown in the diagram below, all eight executors running on the cluster are exchanging row group data with each other, but this is done agnostically with respect to the compute nodes because MPI abstracts that away, so that SNE need not worry about which executors are running on which compute nodes. This mechanism also eliminates the need for an external shuffle service.



One of the ways SNE accelerates query execution is by optimizing the cloud store bandwidth utilization when reading data sets from the cloud store. This is done by precise, parallel, asynchronous prefetching, as depicted in the diagram below.

From the information for FileScan nodes in the query plan a table list is created which tells the Parquet reader which columns in the Parquet files are needed by a particular FileScan node. Each reader has multiple prefetch threads that are each in charge of retrieving a certain chunk of data from the cloud store and putting that data in the system's RamFS, from where we can make immediate use of the data. The reader loads data from the RamFS and sequentially puts it into row groups, which it then forwards to the connected FileScan node, from where it then gets processed by the downstream query nodes.

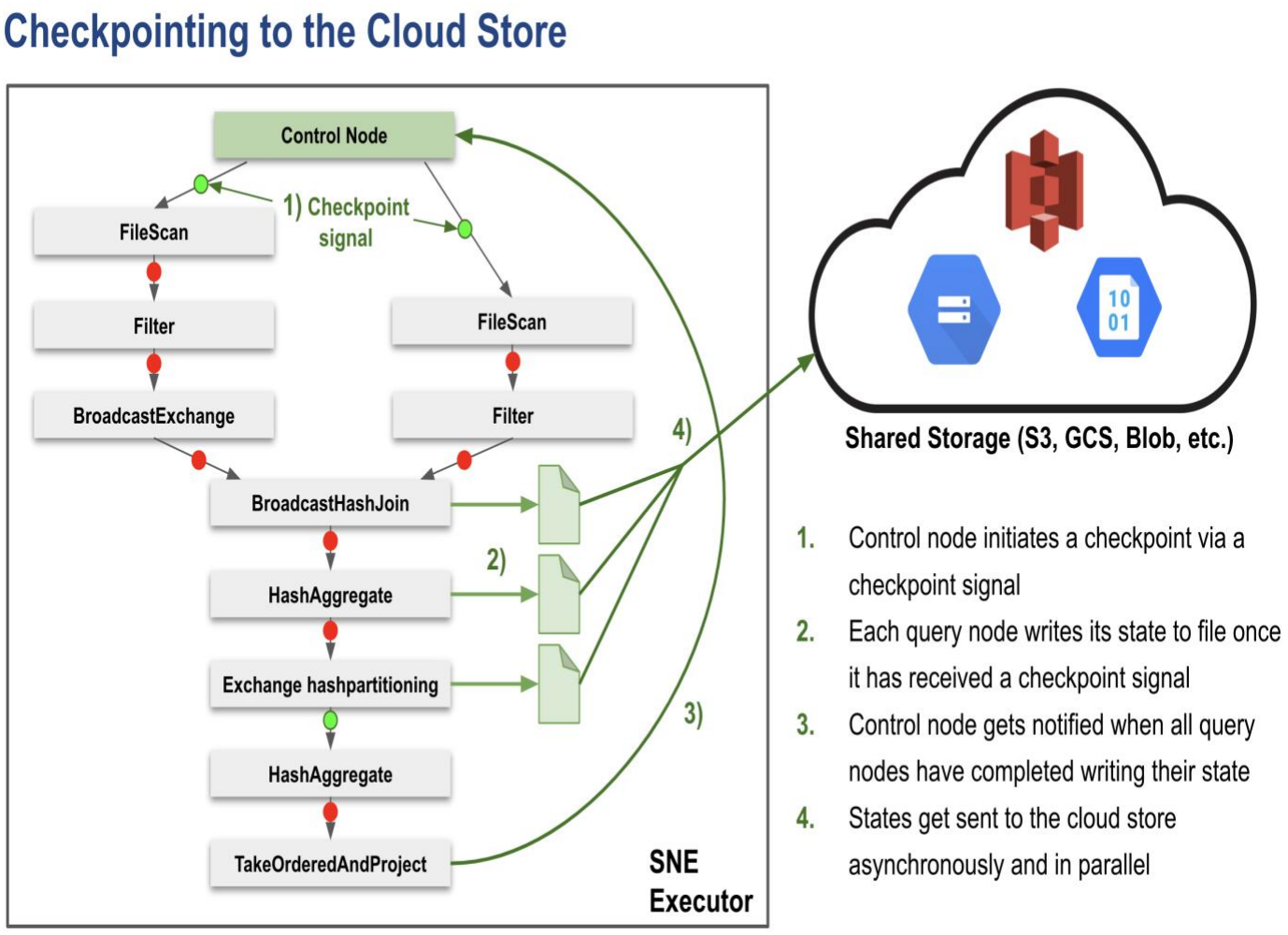
SNE's Optimized Cloud Store Bandwidth Utilization and Efficiency



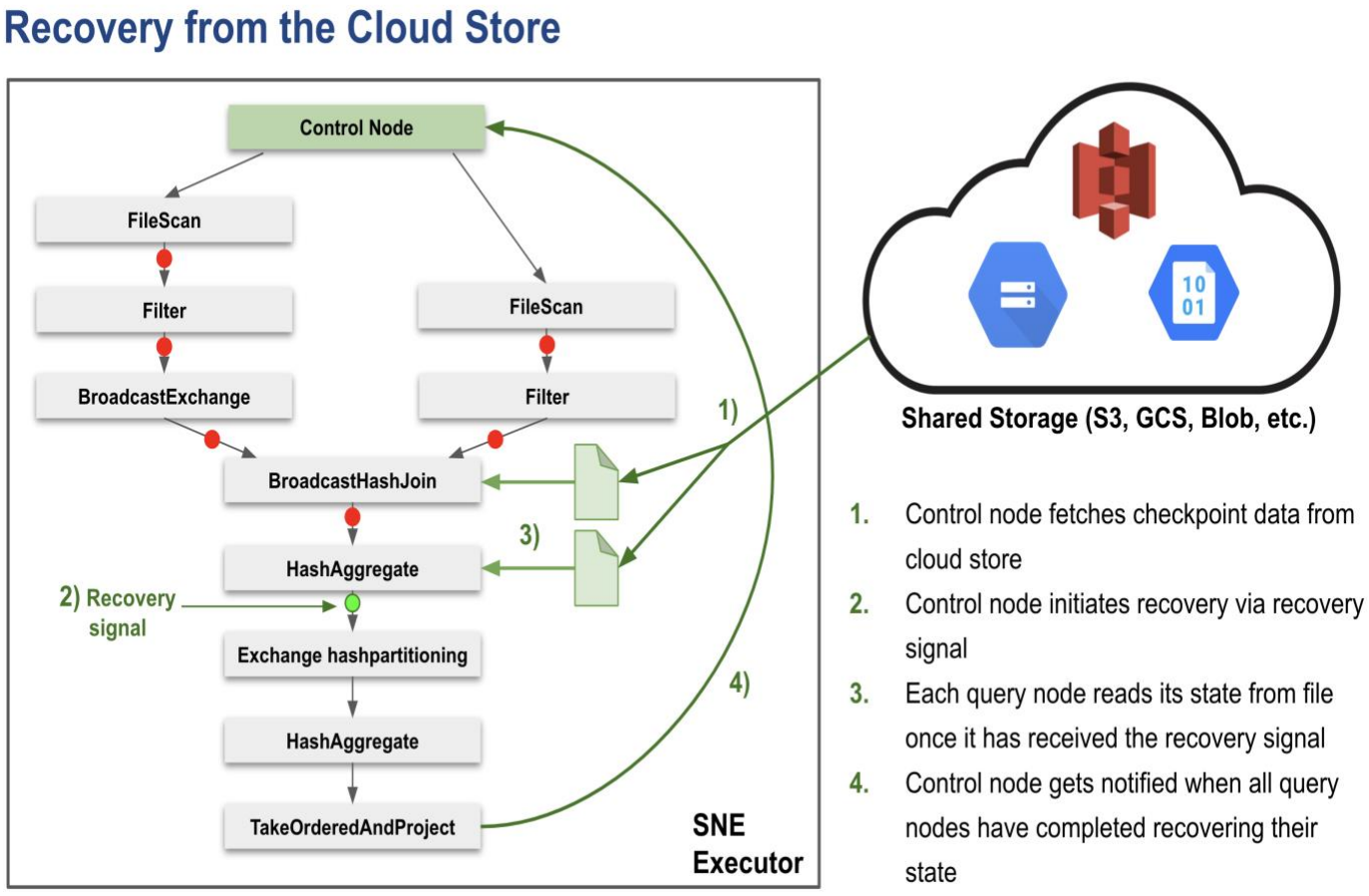
- Multiple prefetch threads in each Parquet reader fetch only the data required by the associated FileScan node ("precise prefetching")
- Data is fetched in parallel across all SNE executors
- Provide full bandwidth utilization of Data Lake cloud store bandwidth from each compute node

During query execution, checkpointing to the cloud store is performed to allow SNE to recover to a previous state in the event of the loss of one or more compute nodes, autoscaling, or even the loss of the entire cluster crash.

As shown in the diagram below, SNE uses a control node that sends a checkpoint signal to the FileScan nodes, which in turn send it downstream. As this signal propagates through the query graph, each query node writes its state and unprocessed data to file. Once completed, these files are sent to the cloud store asynchronously and in parallel, so that query processing continues immediately after the writes to file have completed and sending the files to the cloud store happens in the background.

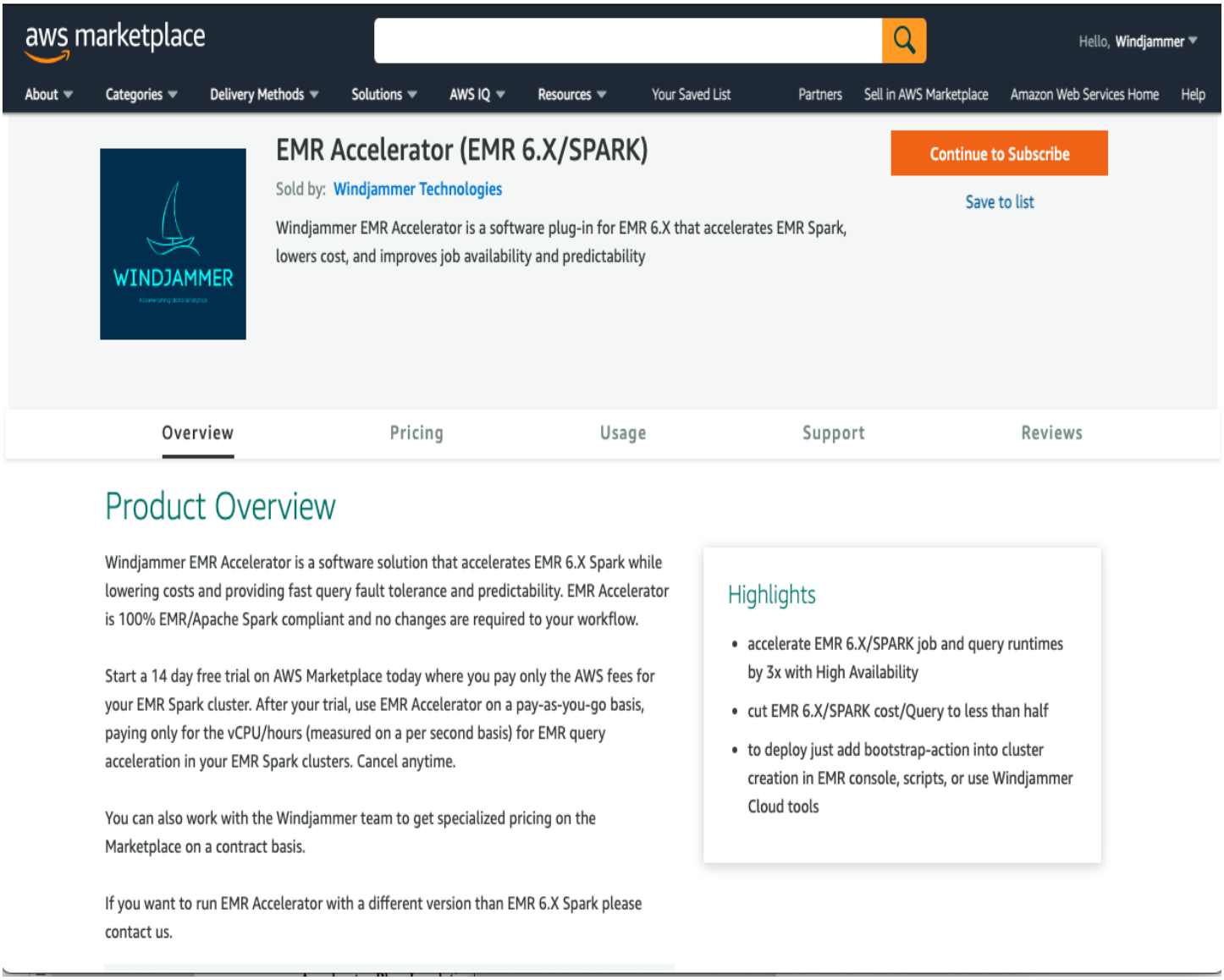


As shown in the diagram below, to recover from a checkpoint, the persisted state and unprocessed data files are fetched from the cloud store and the control node sends a recovery signal to the FileScan nodes, from where it propagates through the query graph. As each node receives the signal, it loads its state and unprocessed data from files and continues processing where it left off.



Windjammer EMR Accelerator in the AWS Marketplace

Windjammer EMR Accelerator for Spark 6.X is available in the AWS Marketplace today, click EMR 6.X Spark Accelerator to begin a free trail.

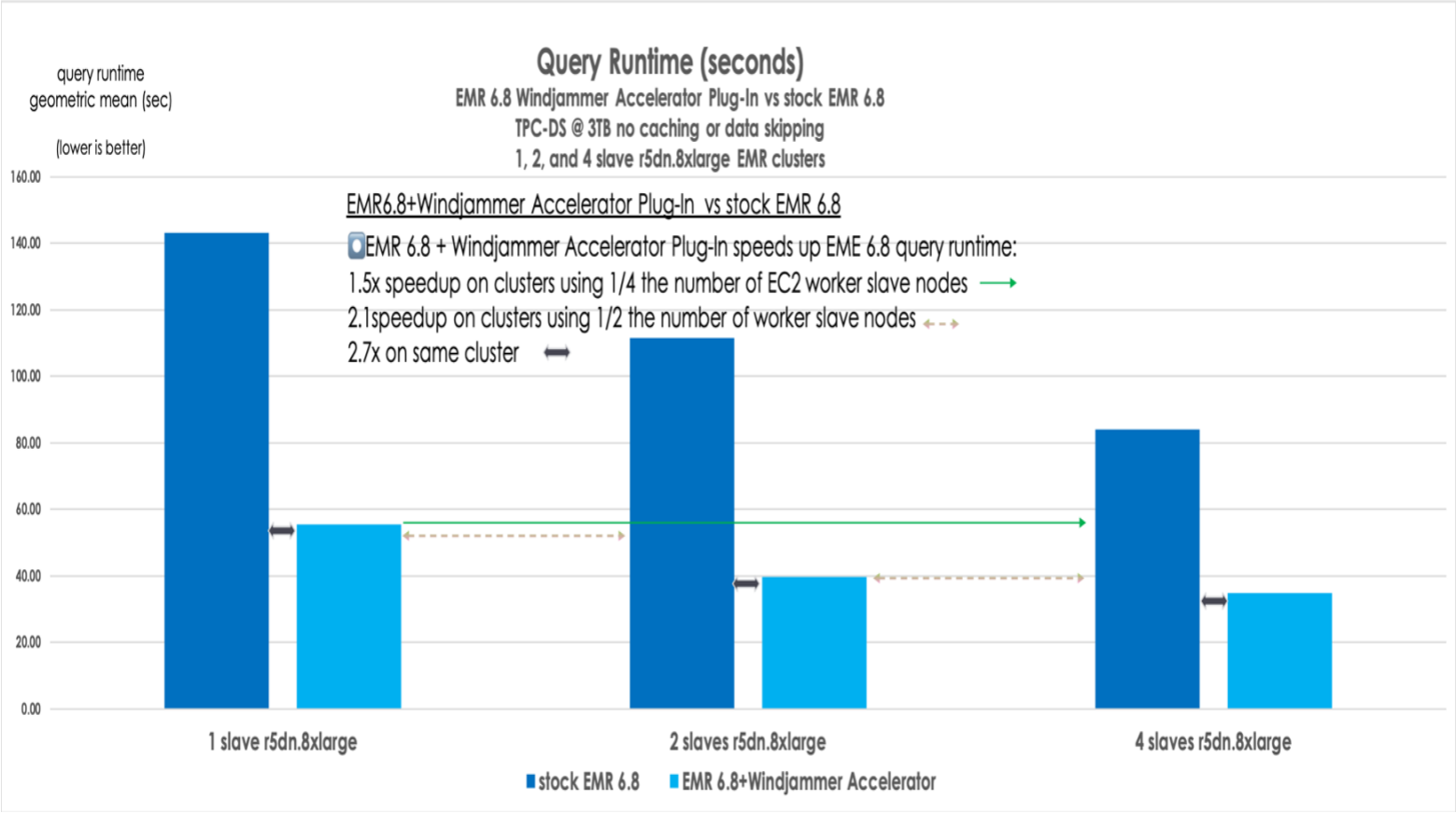


EMR 6.x + Windjammer Accelerator SNE Plug-In Performance and Competitive Benchmark Results

In this section we present results which utilize the standard TPC-DS benchmark and at 3TB scale without caching or data skipping. We have benchmarked 1TB through 100TB and those results are consistent with the 3TB results in this presentation. In the data we present here, we have disabled caching and data skipping in all the analytic frameworks under test to normalize results across the competitive databases and highlight the inherent performance of querying on Data Lakes in cloud stores. Windjammer Accelerator native execution performance with caching and data skipping enabled is additive. When Windjammer Accelerator is installed into EMR 6.x Spark, Windjammer's benchmarking tools are included. These provide easy competitive A/B benchmarking for TPC DS queries and customer queries at any scale. In particular all the results shown in this whitepaper can be quickly reproduced.

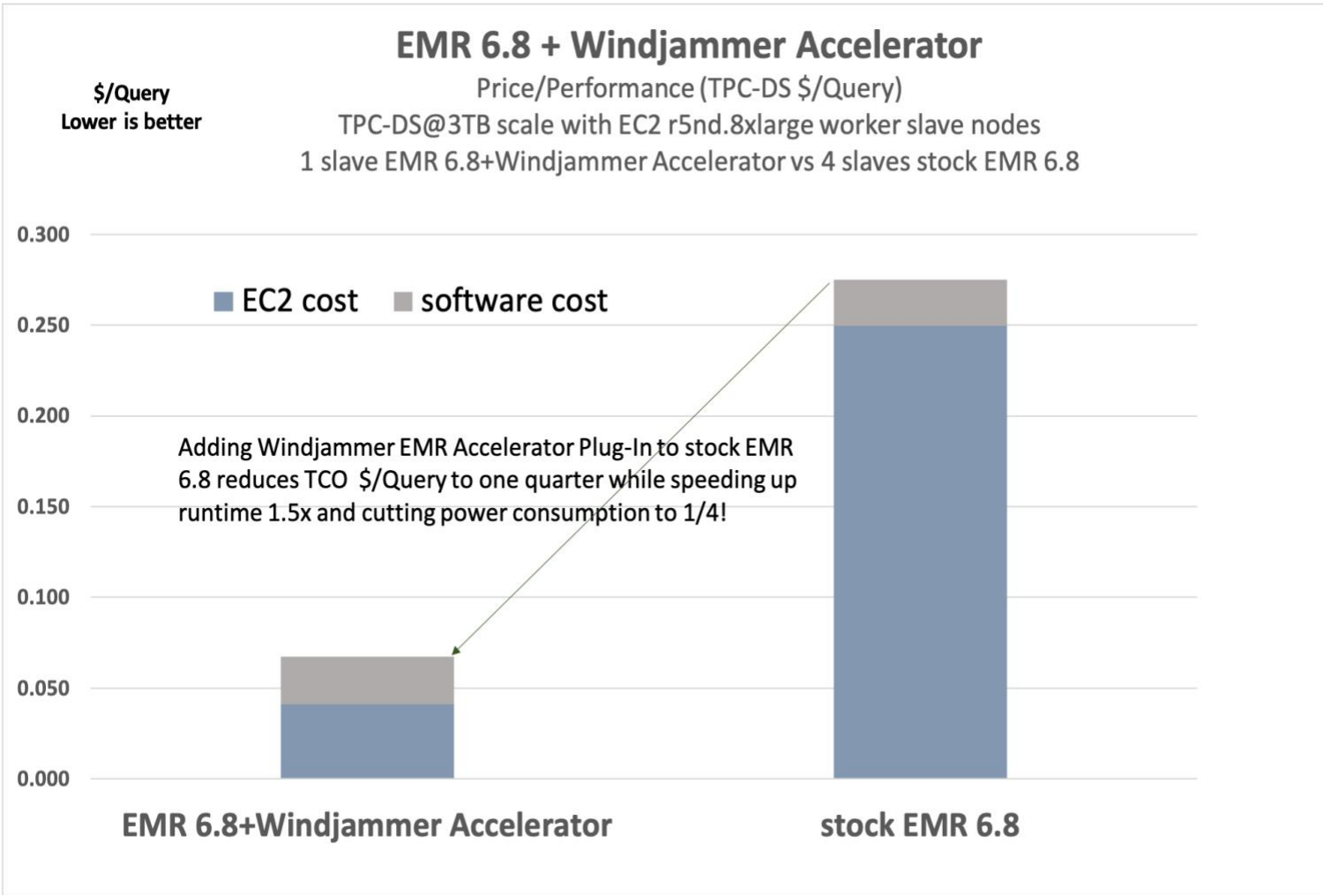
The following charts show the increased performance and the cost savings realized with Windjammer's EMR Accelerator Plug-In relative to stock EMR 6.8 running the TPC-DS benchmark at 3TB scale without caching or data skipping. The detailed TPC-DS measurement data used for these charts is posted at EMR 6.8+Windjammer Accelerator Plug-In Measurements and you can validate the data with A/B experiments in your own clusters as discussed in the Windjammer EMR Accelerator Deployment/POC Guide. Note that balanced clusters using other EC2 instance types and cluster sizes yield similar TPC-DS acceleration and cost reduction results at 3 TB, 10TB, and higher scale. All versions of EMR 6.x achieve similar benefits with Windjammer Accelerator.

The Query Runtime chart below shows both the Query Speedup and the Reduction of the Number of Required Spark Worker EC2 Slave Nodes in clusters running EMR6.8 with the Windjammer Accelerator Plug-In relative to stock EMR 6.8 Spark clusters using r5dn.8xlarge (32 vCPU, 25Gbps networking) EC2 instances with 1,2, and 4 slave nodes. EMR 6.8 clusters with the Windjammer Accelerator Plug-In relative to stock EMR 6.8 clusters provide 1.5x query runtime speedup on clusters using one quarter the number of EC2 worker slave, 2.1x speedup on clusters using half the number of nodes, and 2.7x speed-up on the same clusters.

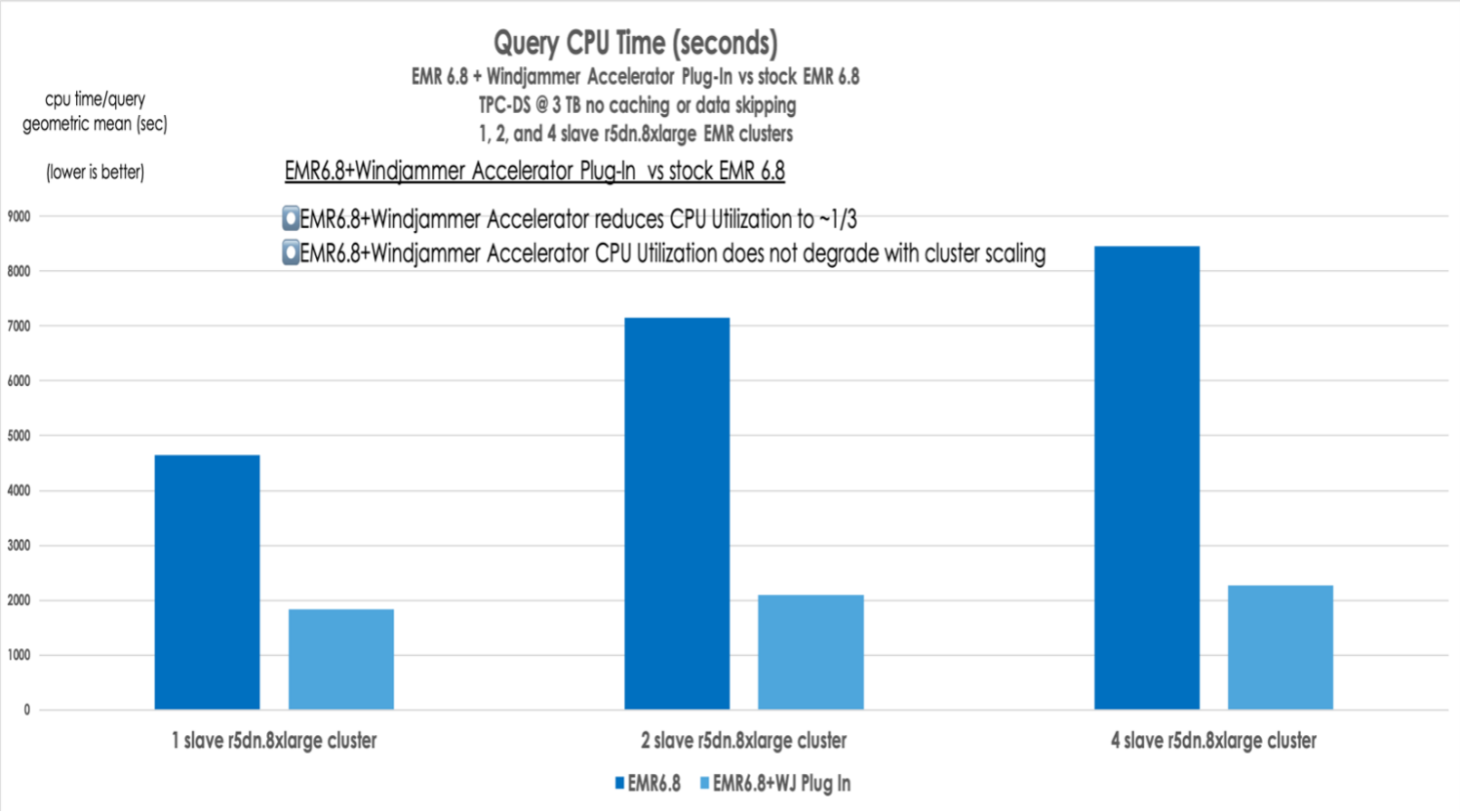


Generally, deployments of EMR6.X+Windjammer Accelerator Plug-In require much smaller cluster sizes than stock EMR6.X, resulting in both significant cost savings and performance acceleration.

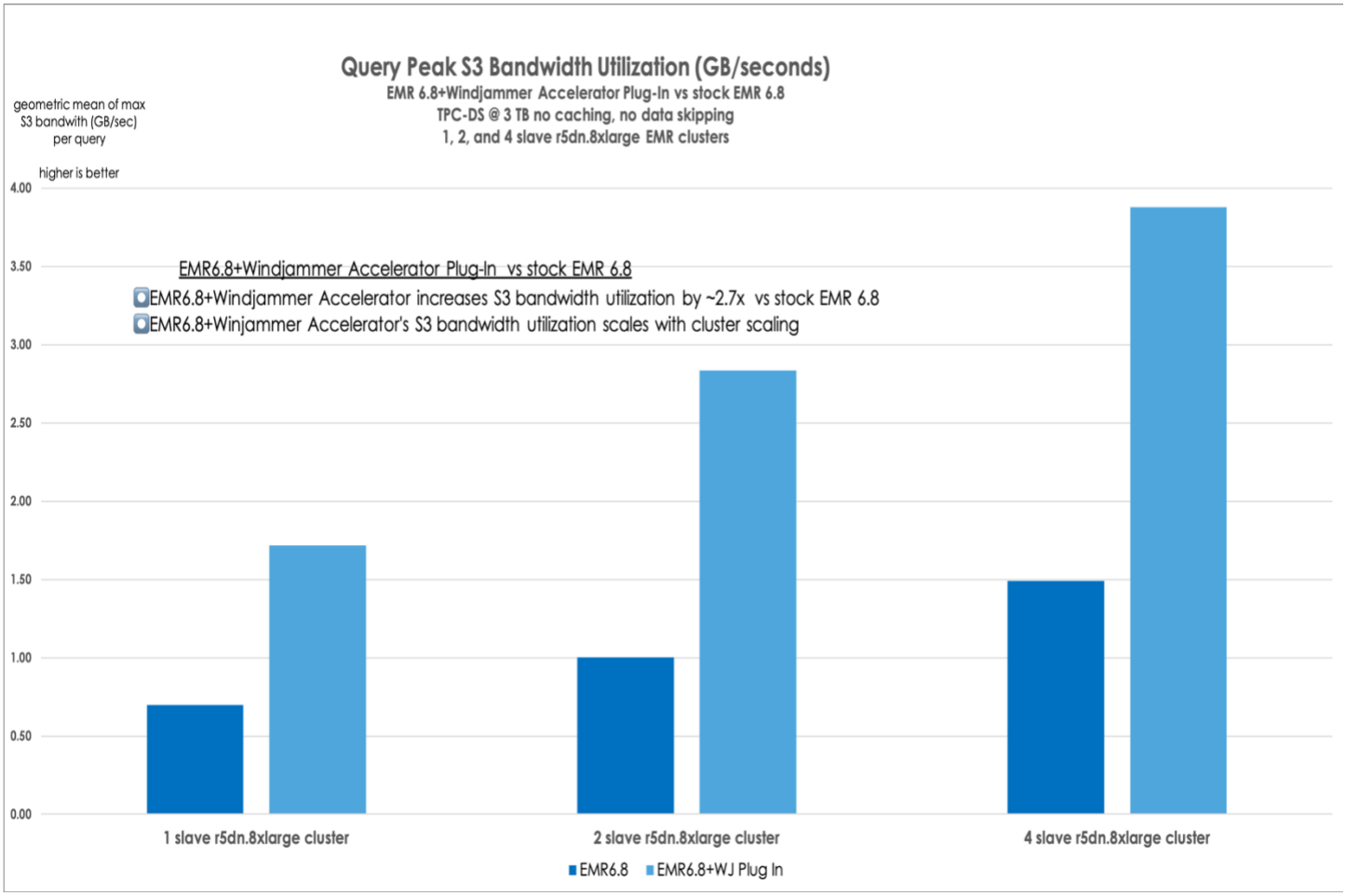
The chart below shows price/performance ($/query) of EMR 6.8+ Windjammer Accelerator running on a cluster with 1 EC2 worker slave node vs stock EMR 6.8 using 4 EC2 worker slave nodes. Adding Windjammer Accelerator slashes $/query to ¼ while accelerating performance by 1.5x. Power consumption is also slashed to ¼.



The Query CPU Utilization chart below shows the reduced CPU usage of EMR6.8 +WJ Plug-In relative to stock EMR 6.8. Clusters running EMR6.X+WJ Plug-In reduce per query CPU usage to ~1/3 of stock EMR 6.X clusters and EMR6.X+WJ Plug-In query CPU utilization remains flat as a function of cluster size whereas EMR6.x CPU overhead per query grows with cluster size. Note that deployments of EMR6.X+WindjammerAccelerator Plug-In should utilize EC2 instance types with sufficient network bandwidth (approximately 1Gbps/vCPU) to achieve balanced slave instances with high core utilization.



The Query S3 Bandwidth Utilization chart below shows the Increased S3 Bandwidth Utilization of EMR6.8+WJ-Plug-In clusters relative to stock EMR 6.8 clusters. Note that EMR6.X+Windjammer Accelerator Plug-In clusters utilizes ~3x the S3 bandwidth per query relative to stock EMR6.X clusters. Deployments of EMR6.X+Windjammer Plug-In Accelerator should utilize EC2 instance types with sufficient network bandwidth (approximately 1Gbps/vCPU) to achieved balanced EC2 instances.



Cluster EC2 Instance type and cluster sizing considerations with the Windjammer EMR Accelerator Plug-In:

As discussed above, EMR6.X+WJ Plug-In clusters use ¼ the CPU time per query, 3x the S3 bandwidth per query, and require ½ or ¼ the number of slaves while still providing acceleration relative to stock EMR6.X clusters. As a result, the selection of EC2 Instance types and cluster sizes is different than for stock EMR 6.X.

- Use EC2 instances with approximately 1Gbps/vCPU networking performance to enable balanced, high core utilization slaves. Note that EC2 instance types that are balanced for stock EMR6.X (e.g. c4.8xlarge 36 cores, 10Gb network) are unbalanced for EMR Accelerator. Due to EMR Accelerator's ¼ CPU usage and 3X network bandwidth utilization, using EC2 instances with high vCPU counts and low network bandwidth results in low core utilization due to network starvation. It is therefore important to select EC2 instances for EMR Accelerator deployments with sufficient network bandwidth (approximately 1Gps/vCPU). This includes EC2 instances with an "n" in their name.
- Local SSDs are not needed.
- Balanced EMR6.4+WJ Plug-In clusters for a given workload typically require far fewer EC2 instances than stock EMR6.X while still accelerating performance.
- EMR6.X+WJ Plug-In has perfect vertical scaling within EC2 instance families that have sufficient networking performance. For example, an EMR6.X+WJ Plug cluster with 4 slaves of EC2s instance type r5dn.4xlarge has the same performance as an EMR6.X+WJ Plug cluster with 2 slaves of EC2 instance type r5dn.8xlarge. As a result, you can use any size EC2 instance within an instance family as long as it has approximately 1Gbps/vCPU networking performance.
- You can select the number of EC2 instances in your cluster to trade off cluster cost vs query runtime

Windjammer EMR Accelerator SNE Plug-In Integration into AWS EMR Cluster Deployments

Deploying the Windjammer EMR Accelerator SNE Plug-In when provisioning clusters via the AWS EMR console or using single click install scripts is natural and trivial, utilizing bootstrap actions, metadata, and properties provided by the AWS EMR managed platform cloud services as shown below. Click this link to download Windjammer EMR Accelerator Deployment/POC Guide.

```
aws emr create-cluster
--bootstrap-actions "Path=s3://<wj-build-bucket>/bootstrap,
--steps '[{ "Name": "wj prep", "Args": [ "/tmp/wjm-prep" ], "Type": "CUSTOM_JAR", "Jar": "command-runner.jar", "ActionOnFailure":
"CONTINUE" }]'
--configurations
   [ { "Classification": "spark-defaults",
        "Properties":  {
             spark.sql.extensions="com.windjammer.spark.sql.SparkNative"
             spark.jars="/opt/wjm/jars/spark-native-extension-2.0-SNAPSHOT.jar"
       }
    }]
 …<other standard user specified emr options >…
```

Conclusions

EMR 6.x + Windjammer Spark Accelerator Plug-In delivers enterprises Spark industry leadership in:

- performance, price/performance, fault tolerance, and predictability
- with transparency and compatibility

Contact info@windjammer.io with questions and visit Windjammer.io for more information.