Building the Enterprise Fabric for Big Data with Vertica and Spark Integration

Jeff LeFevre, Rui Liu, Cornelio Inigo, Lupita Paz
Edward Ma, Malu Castellanos, Meichun Hsu
HPE Vertica
Sunnyvale, CA
{first.last,r.liu}@hpe.com

ABSTRACT

Enterprise customers increasingly require greater flexibility in the way they access and process their Big Data while at the same time they continue to request advanced analytics and access to diverse data sources. Yet customers also still require the robustness of enterprise class analytics for their mission-critical data. In this paper, we present our initial efforts toward a solution that satisfies the above requirements by integrating the HPE Vertica enterprise database with Apache Spark's open source big data computation engine. In particular, it enables fast, reliable transferring of data between Vertica and Spark; and deploying Machine Learning models created by Spark into Vertica for predictive analytics on Vertica data. This integration provides a fabric on which our customers get the best of both worlds: it extends Vertica's extensive SQL analytics capabilities with Spark's machine learning library (MLlib), giving Vertica users access to a wide range of ML functions; it also enables customers to leverage Spark as an advanced ETL engine for all data that require the guarantees offered by Vertica.

1. INTRODUCTION

HPE Vertica is a massively parallel database system, ideal for read-intensive analytic database applications, with full support for SQL and robust data management based on columnar database technology. It is well-designed for managing critical data with atomicity, consistency, integrity, and durability. While our enterprise customers use Vertica for processing Big Data, new systems such as Apache Spark in the Hadoop ecosystem have been gaining momentum. Spark is a compute engine with a generalized multistage in-memory computation structure, instead of the two-stage disk-based MapReduce paradigm. It enables access to data in HDFS and supports batch, streaming, machine learning (MLlib), SQL, and graph processing (GraphX). However, its data management is weaker in enterprise readiness compared to a commercial database system. For example, Spark does not yet have adequate isolation semantics for data sharing, nor does it offer data integrity guarantees. Therefore, it is not the best choice for companies to store and manage their mission-critical business data.

Since Vertica is an enterprise class analytical database, it is a good target for storing data resulting from extract-transform-load (ETL) processes performed in Spark. We also leverage Spark for machine learning using data stored in Vertica. In this paper we present our solution to integrate Vertica and Spark into an enterprise fabric that supports robust data management, analytics and machine learning. We define the following two primary challenges to accomplish this integration:

1. Transferring data quickly, reliably, and robustly between Vertica and Spark in order to support:
   - A pipeline for Spark to use Vertica as a fast analytic data source.
   - Spark as an ETL engine for Vertica.
2. Creating and then deploying Spark ML models within Vertica for in-database scoring (i.e., prediction).

Our approach integrates Vertica and Spark with data flowing in either direction, Vertica to Spark (*V2S*) or Spark to Vertica (*S2V*) as shown in Figure 1. In the V2S direction for analytics, we provide a connector to efficiently read Vertica data into Spark and then leverage Spark's ML library (MLlib [9]) to train models in Spark. In the S2V direction for ETL workloads we provide a connector that efficiently loads the transformed data into Vertica. To support the full analytics pipeline, another essential component we provide is a mechanism to deploy the trained models (*MD*) to Vertica for in-database scoring. In addition, since Spark now supports export of some models in Predictive Model Markup Language (PMML) [7, 10], it makes sense to deploy PMML models from Spark to Vertica. Such capability can also serve other PMML producers such as SAS or Distributed R (DR) [14]. To make our Vertica and Spark integration easily accessible for Spark users, our architecture shown in Figure 1 allows each component (V2S, MD, S2V) to be initiated from within Spark. The approaches presented in this paper are available as a beta release on the HPE Marketplace [5].

Contributions

- Fast, reliable data transfer between Spark and Vertica with exactly once semantics.
  - Our parallelized method for V2S eliminates all data shuffling between Vertica nodes.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
 ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2903744

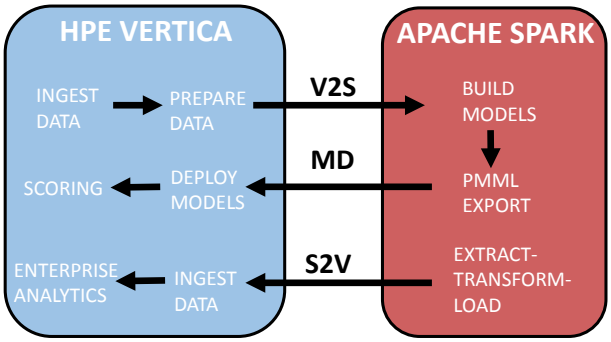

Figure 1: Overview of integration architecture and functionality.

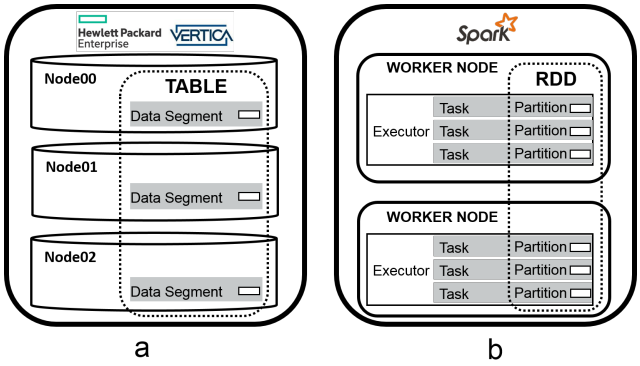

Figure 2: HPE Vertica (a) and Apache Spark (b).

  - Our parallelized method for S2V prevents partial or duplicate loads.

- Our approach tolerates Spark job failures, task failures and restarts, as well as speculative execution; and it can be applied to the general class of MapReduce systems.

- We present experimental results evaluating our methods along different dimensions such as varying parallelism, data scaling, cluster scaling, and others. We also present a comparison with other approaches showing our methods have competitive performance.

- The capability to take machine learning models built in Spark to be deployed and executed within Vertica for in-database scoring.

In the following sections we describe each of these components and some of the challenges faced. We begin with an overview of the problem in Section 2, followed by the details of our approach in Section 3. The advanced analytics pipeline component V2S is described in Section 3.1 followed by the ETL pipeline component S2V in Section 3.2, while model deployment (MD) is described in Section 3.3. Experimental results that analyze many aspects of performance are provided in Section 4. Lastly, discussion, related work and brief conclusion are presented in Sections 5, 6, and 7.

2. PROBLEM OVERVIEW

Our goal is to integrate two distinct systems, HPE Vertica and Apache Spark for the following specific purposes. First we want to support the full analytics pipeline to extend Vertica's analytics capabilities with ML models created in Spark. Second we want to transfer data quickly and reliably between HPE Vertica and Apache Spark in support of the analytics pipeline (V2S component) and for using Spark as a fast ETL engine for Vertica (S2V component).

To achieve the first goal, we need to share and recreate ML models built in Spark with Vertica. Spark's MLlib includes a variety of machine learning algorithms such as classification, clustering, and regression, and models built with MLlib must be able to execute on Vertica. A common standard format helps to facilitate this task, and here we use PMML format to export the model and develop a mechanism to deploy the model on Vertica. To achieve the second goal, we need a reliable transfer mechanism that enables fast data transfer between HPE Vertica and Apache Spark in both directions, along with data semantics guarantees required for an Enterprise solution. Our focus in this paper is mainly on the data transfer mechanisms and their semantics.

Given these goals, our primary problem is to *reliably* transfer data between two distinct systems. From the user's perspective, ideally 'reliably' should indicate standard transactional semantics that a user expects in an enterprise database. Hence we desire *exactly-once semantics*, ensuring all of the data is transferred once and only once. Thus the data should move from one system to the other in its totality and current state at the time the move was requested. While there are database constructs to achieve transactional semantics within a single database, maintaining exactly-once semantics across systems is a challenge.

2.1 System Descriptions

In this section we briefly outline the characteristics of HPE Vertica and Apache Spark that are relevant in the discussion of our work. Figure 2 provides a high-level picture of each system and the details of each are described below.

2.1.1 HPE Vertica Database

Figure 2(a) provides an illustration of a Vertica database spanned over multiple nodes in a Vertica cluster. HPE Vertica is an MPP column-store [13, 15], distributed database with traditional transaction semantics (i.e., ACID). Tables in Vertica are segmented across nodes in the cluster as shown in the figure. *Segments* correspond to non-overlapping hash ranges, and each range is stored on a particular node. By default, tables are created using several columns in the segmentation (*hash*) expression, but segmentation columns may be explicitly provided, and it is possible to create unsegmented tables as well. For fault tolerance, Vertica provides replication of tables across nodes by setting its k-safety value where k is the number of failed nodes tolerated before data loss. When storing its data, Vertica uses a proprietary storage structure called *Read Optimized Storage*, or ROS, which is a highly optimized read-oriented disk storage that includes compression and indexing. Vertica also has a corresponding *Write Optimized Storage*, or WOS. The details of Vertica's ROS and WOS storage are beyond the scope of this paper but can be found in [13].

Vertica supports advanced SQL analytics, including the ability for users to write custom functions called User-Defined Extensions (UDx). In this way, Vertica's analytics functionality can be extended to support many algorithms.

2.1.2 Apache Spark

Figure 2(b) provides an illustration of Apache Spark, showing multiple nodes in a Spark cluster. Spark [17] is a compute engine and even though it provides SQL processing capabilities, it is not a database management system and does not provide shared data access nor ACID semantics. Spark's machine learning library, MLlib [9], provides parallelized versions of various ML algorithms. Spark's main abstraction is a Resilient Distributed Dataset (*RDD*), which is distributed throughout the cluster, and *worker nodes* manage local execution of tasks that operate on RDD data.

RDD. An RDD [16] is an immutable in-memory construct that is divided into *partitions* and distributed across a Spark cluster. RDDs themselves are not durable, but their data could be saved to an external storage location effectively creating a snapshot of the current RDD. However, since it is immutable any new manipulation performed on an RDD would produce a new RDD. In this way the in-memory RDD is simply a distinct copy of any data source from which it was created.

Spark Scheduler. Spark uses a batch-based task scheduler, where jobs are run asynchronously after they are submitted by the user. Jobs are composed of tasks, launched by *executors* and a job will create some number of tasks that execute independently of each other, with each operating on an associated partition of data as indicated in Figure 2(b). Tasks are coordinated by the Spark master (not shown), are stateless (ideally), and do not communicate or coordinate among themselves. Some of Spark's fault tolerance mechanisms include (a) maintaining the provenance (i.e., lineage) of an RDD such that an RDD partition's data can be recomputed any time and (b) the ability to restart failed or straggling tasks, i.e., speculative execution.

DataFrames. Recently Spark introduced DataFrames and its External Data Source API. A Spark DataFrame is effectively a wrapper around an RDD that provides a schema, where each of the RDD's rows is of Spark SQL Row type. A *DataFrame* in Spark is analogous to a database *table*, but since DataFrames are based on RDDs they are immutable. The External Data Source API was introduced in Spark to provide a standard API to save/load data to/from another data source. Developers may implement specific versions of the load/save functions for their data source. This is the interface we use in our work and have implemented load-/save functions specifically for Vertica. The API is based on enabling easy access for Spark users to external data but introduces several technical challenges, described below, due to Spark's compute architecture and especially its scheduler.

2.2 Possible Approaches and Challenges

Given the characteristics of each system, next we describe a first possible approach and the challenges that arise when introducing parallelism that results in competing design goals: performance versus consistency.

2.2.1 Vertica-centric Approach

A Vertica-centric approach would be one that enables Vertica to send the data to Spark (*push*-based) for the V2S direction, or let Vertica pull the data from Spark (*pull*-based) for the S2V direction. This Vertica-centric method can potentially have very high performance since it can rely on Vertica's query optimizer to plan and control the parallel data transfer in an efficient way because it is aware of the internal details of Vertica's data layout and cluster configuration. Although this would automatically provide parallelization and transactional semantics, this approach is not well-suited for Spark's batch-based scheduler as we describe next.

Since the Spark scheduler can only create a set of independent, stateless Spark tasks to perform the load or save actions, we do not have a direct way to coordinate the data push or pull actions with Vertica from Spark tasks. Spark tasks would need to start and then wait for some coordination to begin receiving or sending data to/from Vertica. Furthermore with a Vertica-centric approach, even if Vertica initiates the transfer by launching a Spark job, it is not clear when the tasks will be run or even *re-run* since tasks are scheduled by Spark and a submitted job can have an unbounded wait time depending upon how many other Spark jobs are currently executing in the system and on the scheduling algorithm (several are available). Alternatively, an independent third system with its own scheduler could be used to initiate and coordinate the actions. This approach to coordination is possible but adds extra complexity due to the need for a third system/process and scheduler, and in the end would still depend upon the Spark scheduler.

2.2.2 Challenges with Parallelism

To achieve performance we can potentially use many tasks, i.e., parallelism. By launching many tasks, a high degree of parallelism can be achieved but since they are independent and cannot coordinate with each other, there is no guarantee of reliable data transfer. For example, when saving data to Vertica, if a task fails or Spark itself crashes this could result in partial data transfer. If a task is restarted by Spark then it could repeat its own data transfer, potentially duplicating the data. Moreover, there is a subtlety in that even if a task only commits after it is completely done, it could still fail immediately *after* the commit and be restarted, again resulting in duplication. When loading data from Vertica, many independent tasks executing at different times or re-executing after failure can result in loading an inconsistent view of the Vertica data if a table is updated in-between tasks.

Compounding the problem further is that even if many parallel tasks connect to different Vertica nodes, it is not likely that a task will request data that is local to that node, hence in addition to the outbound network traffic sending the data out to Spark this will induce a lot of internal network traffic between Vertica nodes as they exchange data to answer each request. Hence, the technical challenges of our parallelized design include reducing internal data shuffling between Vertica nodes and how to provide a consistent view of the data while providing fast data transfer.

In this work we present and evaluate our approach that achieves parallelism and reliability with exactly-once semantics and with workload optimization. We describe this approach next in Section 3. Although this work is based upon Apache Spark, we believe that our methods can be applied to the general class of MapReduce [11] systems.

3. OUR APPROACH

Our approach for transferring data between Spark and Vertica utilizes the parallelism of the Vertica MPP DMBS as well as the parallelism inherent in Spark's computation model. Our key insights to solving the problem are that we

LOAD	SAVE

`df.read`
`  .format(Def.Source)`
`  .options(opts)`
`  .load()`
	`df.write`
	`  .format(Def.Source)`
	`  .options(opts)`
	`  .mode(mode)`
	`  .save()`

Table 1: Spark's External Data Source API example

can rely on knowledge of the data layout in Vertica, and use Vertica itself and its ACID semantics as a way to guarantee reliable, fast data transfer with exactly once semantics. The connector is optimized to load large volumes of data from Spark to Vertica and designed to be robust to Spark failure scenarios such as total Spark failure, failed tasks, duplicated tasks, or speculative tasks and still provide exactly-once semantics. Next we discuss our approach that is implemented using DataFrames and Spark's External Data Source API. Our implementation using RDD (for Spark ML methods that operate on RDDs) provides a similar functionality. First in Section 3.1 we describe our approach for transferring data from Vertica to Spark, V2S. Then in Section 3.2 we describe our approach for transferring data from Spark to Vertica, S2V.

To help clarify the following discussions, Table 1 shows the current syntax for Spark's External Data Source API to load/save data from/to an external source assuming an existing Spark DataFrame `df`. The API includes a reference to the `DefaultSource` name, which is the implementation specific source code, in this case our Vertica-specific implementation, which is called "com.vertica.spark.datasource.DefaultSource". It also includes an arbitrary list of key=value options, `opts`; these typically include user names, passwords, table names, and host IP addresses among other parameters relevant for the source-specific implementation. Additional options available with our approach are described in the following V2S and S2V sections where relevant. When saving data, the API also requires a save mode, `mode`, which can include overwritting or appending to an existing table.

The benefit for users is that the API is standardized and simple. We note that this API only represents the current implementation while our below methodologies apply generally for load and save methods.

3.1 Transferring Data from Vertica to Spark (V2S)

In this this section we describe the design and semantics for the V2S component of the HPE Vertica Connector for Apache Spark. The connector is invoked with the `LOAD` API shown in Table 1, which also supports additional processing operations (e.g., `.filter()`, `.select()`, `.count()`) which can be applied to the DataFrame during load. The connector is optimized to pushdown these operations into Vertica in order to take advantage of the computational power of Vertica to both pre-process the data and minimize the amount of data transferred over the network. The transfer of a Vertica table to Spark is atomic and will load a consistent view of the table into Spark.

The key insight with our approach is that we can utilize Vertica's data layout and its ACID semantics to enable fast data transfer to load a consistent view of the data along with exactly-once semantics. To achieve the highest possible data transfer performance, our solution connects the computation pipelines of Vertica and Spark in an optimized way, which not only utilizes parallel channels for data movement, but also ensures:

1. Data flow optimization by pushing down computation into Vertica as appropriate.

2. Data request locality optimization on Vertica nodes.

Spark's computation is organized as a chain of RDDs (or DataFrames) which forms the execution pipeline and is evaluated *lazily*. Due to Spark's batch-style scheduler, the user can only request Vertica data by launching a Spark Job and then waiting for the Job to finish. For this reason, we require a *pull*-style approach whereby the user's Spark Job directly asks for the data that it wants, which is then pulled into Spark. For each Spark partition (i.e., task), our approach is to formulate a unique query requesting a non-overlapping subset of the desired Vertica data, as well as any pre-processing required (e.g., push-downs), and the union of the result of all queries represents the full table data. In Section 3.1.1 we describe the computation pipeline and pre-processing to achieve optimization (1) above, then in Section 3.1.2 we describe our query formulation to achieve optimization (2).

3.1.1 Reducing the Amount of Data in the Pipeline

Figure 3 shows the data flow in the computation pipeline of our V2S approach. The full computation job spans both Vertica and Spark: data from a Vertica table flows through multiple Vertica query execution plans in parallel on the different cluster nodes, and is then transferred into Spark's pipelines. Vertica can pre-process the data with project, filter, and count operators as supported by Spark's External Data Source API. By pushing these operators down into Vertica, only the pre-processed results are passed through the system boundary and into Spark's computation pipeline, potentially reducing the amount of data transferred.

Spark's External Data Source API currently does not support pushdowns for joins or aggregations, however, if the user pre-defines a view in Vertica that represents the join or aggregation our connector can load the view. V2S supports loading views (and similarly unsegmented tables) by producing *synthentic* hash ranges which allow us to also use parallelism during data transfer for views. In this way, views can enable pushdown of other operations such as joins and aggregations. All pushdowns are computed inside Vertica within our compute pipeline as shown by the connected arrows inside the Vertica nodes in Figure 3. In this way our notion of computation pushdown can extend beyond the simple pushdowns enabled by the Data Source API. This reduces data flow in the pipeline and helps us to achieve the data flow optimization (1) in Section 3.1.

3.1.2 Data Locality Aware Approach

As the number of parallel connections between the two systems increases, the network bandwidth between the Vertica nodes can become a bottleneck in the computation pipeline, potentially slowing it down. This is caused by the amount of data shuffling between Vertica nodes to retrieve all of the table data. This occurs because each Spark partition (task) connects via JDBC to a Vertica node and issues a query to pull a specific portion of the table data, which is typically scattered across all Vertica nodes.

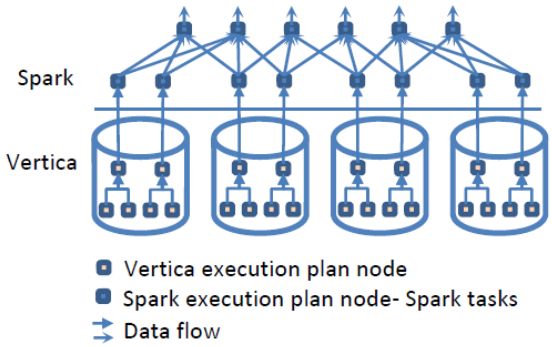

Figure 3: Connected computation pipeline.

Our novel solution to prevent unnecessary data shuffling between Vertica nodes is to have each Spark task formulate a query that only requests data that is local to the node to which it connects. Thus, each query to a Vertica node only targets the local *segment* of the data.

Vertica distributes its table data via hashing based on the SEGMENTED BY HASH(*columns*) clause or implicitly, storing segments with different contiguous hash ranges across the nodes based on hashing. This creates a hash-ring, whereby each node stores one segment of the ring. To formulate a pull query, we first look up the hash-ring segmentation boundaries, along with the node that contains each segment; this information is stored in the Vertica system catalog and can be queried. Then for each Spark task, we formulate a unique query that asks for a specific non-overlapping hash range within a segment boundary. This ensures that only one node will contain the requested data – the node which stores the segment range. The data locality aware approach has minimal data skew and respects the Vertica segment boundaries.

Figure 4 shows two examples of the Vertica table segments for a 4 node Vertica cluster and the corresponding Spark partition tasks. In both examples, the inner ring represents the Vertica hash ring, which indicates each of the four segments 0–3. The outer ring represents the Spark partitions (hence tasks), each of which will request one or more ranges of the hash ring. In Figure 4(a), there are only 2 Spark partitions, so each partition formulates a query requesting 2 segments (i.e., 2 hash ranges). In Figure 4(b), there are 8 Spark partitions, so each partition formulates a query requesting $\frac{1}{2}$ of a segment. The resulting data contained within each Spark partition is one or more contiguous hash ranges from Vertica segments. Since the desired number of Spark partitions is a user option in our implementation, the number of partitions is not restricted by the number of Vertica segments.

By ensuring the task only connects (via JDBC) to the Vertica node that stores the corresponding segment range, spurious network traffic between Vertica nodes is eliminated, effectively using the full network bandwidth between Vertica and Spark only for data transfer between the two systems, accelerating the entire computation pipeline. The data locality-aware partition queries help us to achieve the data locality optimization (2) in Section 3.1.

With many independent, parallel queries (one per task) that could access Vertica at different times, we need to ensure a consistent view of the table data. To do this, we can utilize Vertica's *epoch* feature which enables each query to request a specific version of the table data. In this way all tasks can formulate their queries to request data from the same epoch (e.g., last epoch). Although Spark tasks may fail and be restarted until they complete successfully, using epochs ensures that all tasks load a consistent view of the table. Together, our design for V2S is optimized for parallel data transfer, pushdown processing, takes advantage of data locality, is robust to Spark task failures, and preserves our desired exactly-once semantics.

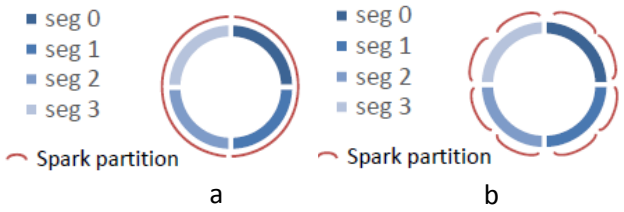

Figure 4: 4 Vertica data storage segments with (a) 2 Spark partitions and (b) 8 Spark partitions.

3.2 Transferring Data from Spark to Vertica (S2V)

In this this section we describe the design and semantics for the S2V component of the HPE Vertica Connector for Apache Spark. The connector is invoked with the `SAVE` API shown in Table 1, which supports saving to an existing table (append mode) or overwriting/creating a new table (overwrite mode). The transfer of a Spark DataFrame to Vertica is atomic and will never pollute the target table with partial loads or duplicate data, even though we optimize the performance by using many independent, parallel tasks.

The key insight with our approach is that we can use Vertica itself and its ACID properties to guarantee exactly-once semantics to save data, which is achieved within a single Spark job. In essence, since Spark tasks are stateless and do not communicate with each other as is typical in MapReduce [11] style batch processing systems, we use tables in Vertica to play the role of a durable log where we record the connector's progress and final status of success or failure. While the transfer is in progress, each task utilizes these tables to inquire the state of progress of all other tasks. Using these tables, the connector can ensure that tasks do not duplicate work, and that all tasks complete successfully. Since bulk loads may include rejected rows the connector API provides user control (optional parameter) to specify a tolerance for the number of rows rejected, and a sample of the rejected rows is provided. If the number of rejected rows falls within the user's tolerance level, the save is completed and considered successful; otherwise, it fails.

To launch S2V in Spark, the user specifies the save mode and target table along with a set of other options. (e.g., host, user, etc.). When an S2V job is launched, there is first a setup phase which includes creating the following 3 temporary tables, and 1 permanent table. A *Staging Table*, which has the same schema as the target table. A *Task Status Table*, which includes one row per task that includes task/-partition id, rows inserted/failed counts, and a task `done` status. A *Last Committer Table*, which includes the task id of the the last committer. A *Final Status Table*, which includes the unique job name, failed rows percentage, and final

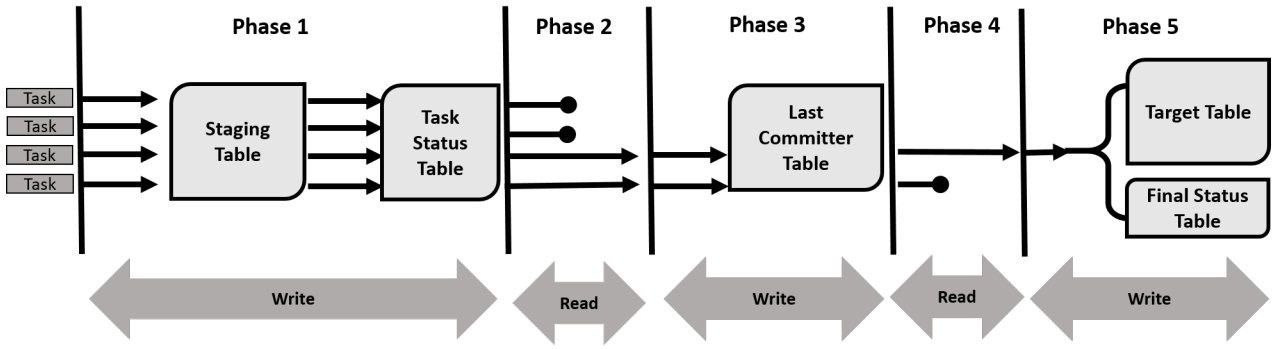

Figure 5: Phases of S2V tasks. Arrows and circles represent a single task's progression or completion.

status of each job. This table serves as a record of all S2V jobs and is not deleted upon termination. Because this table is always available, users can consult this table any time to verify the job's status, for instance in the case where there is a Spark error causing total Spark failure, which could leave the Spark user uncertain as to the job's final status.

Although the user provides only a single Vertica hostname to the API, all Vertica node IPs are looked up during setup so that tasks can connect to different Vertica nodes in order to balance the load across the Vertica cluster during data transfer. Also during this setup phase, we can repartition the DataFrame in order to obtain the desired parallelism, where the number of partitions is a user option. Typically with large data this is simply a coalesce of many partitions into fewer without any data shuffling.

Once the temporary tables have been created and the number of partitions and Vertica node IPs are known, the worker tasks can be started. We show later in Section 4.7.1 that the overhead for setup and teardown of these tables in S2V is on the order of a few seconds. Next, the challenge is to provide parallelism via independent tasks while guaranteeing that all independent tasks perform a single job of reliably saving the DataFrame to Vertica. This is achieved with the logic described in the next section.

3.2.1 Phases of tasks in S2V

Figure 5 shows the five phases of each task in a single S2V job that ensure each task completes saving its data only once, regardless of how many times an individual task is executed. In Phase 1, a task connects to one of the Vertica nodes and writes its data into the staging table and then updates its *done* status to true and commits, only if *done* is currently false otherwise it aborts the transaction. This check prevents the same task (i.e., a duplicate) from saving its data twice into the staging table. Writing to the staging table and updating the task status table are done under the same transaction in order to guarantee the record of the task saving its data is durable.

In Phase 2, the task reads the task status table to check if all tasks are done, including itself (since it could be duplicate). If any tasks are not done, then this particular task has no further work to do and it completes, as indicated by the terminal end point in phase 2 of the figure. However, if all tasks are done, then the task can advance to Phase 3. We note that several tasks can advance to Phase 3 at the same time, given they might finish in similar time frames.

In Phase 3, the task tries to update the last committer table with its task id, where the task id is empty, again committing if the id is still empty else it aborts. This is a race to update the table but ensures that *only one task* will be able to update the table with its id, which serves as a durable record of which task won the race (leader election).

In Phase 4, the task then reads back the last committer table to see if it had won the race. If it has not, it can terminate. Here we point out that each of the read phases of S2V are required since a task could have failed and been restarted at any point. By reading these tables at each phase before advancing to the next phase we prevent tasks from duplicating their previous work but ensure that they will complete all necessary work before terminating.

In Phase 5, the last committer task is the single remaining task, although due to speculative execution it could be duplicated. It verifies if the total number of failed rows is within the user's tolerance. If so, the task now commits the staging table into the target table and updates the final status table to finished, where finished is currently false; else it aborts. This check again prevents duplicate tasks from performing the same action, which ensures the final commit of the staging table data into the target table is only done once. In overwrite mode, this is simply an atomic rename of the staging table to the target table, and in append mode the task must copy the staging table data into the target table, which takes longer.

In summary, phases 1–2 ensure the reliable save of a Spark DataFrame to the staging table, which is the bulk of the work; phases 3–5 simply ensure the final atomic save of the staging table to the target table. The above process accounts for multiple task failures and restarts, speculative task execution, and also for complete Spark failure. Due to the use of a staging table, in the worst case of total Spark or Vertica failure the target table will not be affected.

3.2.2 Other Optimizations

During this entire process, S2V uses several optimizations. One is that we utilize the VerticaCopyStream Java API, which allows us to programmatically access Vertica's bulk load *COPY* utility, which is optimized for loading data quickly. Another is that we use the Avro [3] file format for each task to encode its data before sending to Vertica. The Avro format provides several benefits: it is a binary format, it does not require us to specify a delimiter (which can be difficult in the general case with unknown text data), and it also offers compression which can reduce the amount of data transferred over the network.

3.3 Model Deployment from Spark to Vertica (MD)

The goal of this component is to support in-database scoring of models generated by Spark MLlib and exported to Vertica. To this end, we have developed APIs for deploying and reading PMML models to/from Vertica. For storing PMML models in Vertica, one option is to store them into a table, but it is difficult to define a proper and generic schema for PMML models since there is a variety of model types (e.g., k-means, SVM, logistic regression, etc) and each model has a different number of features (i.e., fields). Instead, we store PMML models in an internal distributed file system (DFS) and hence are accessible to the database query engine and User-Defined Functions (UDF).

Our `DeployPMMLModel()` method stores the model itself into the DFS and also inserts its corresponding metadata (e.g., name, size, model type) into a Vertica table. Once the PMML models have been deployed, they can later be used for in-database model scoring. This requires to write a UDF that first reads the model from the DFS using our `GetPMML()` method and then uses JPMML [6] to build the corresponding model evaluator to perform scoring/prediction. To make it easier for the user, we have developed a generic model evaluator for models whose input is a numeric vector and the output is a number (e.g., logistic regression, k-means, etc).

Users can run predictions inside the database by using the scoring UDF in an SQL query. In the following example PMMLPredict() is the UDF which takes as arguments the column names corresponding to the features used by the model to do the predictions, along with the model name as a parameter.

```
SELECT PMMLPredict(
    sepal_length, sepal_width,
    petal_length, petal_width
USING PARAMETERS model_name='regression')
FROM IrisTable
```

In this way, prediction models can be trained in Spark using data from Vertica transferred with V2S, and then deployed into Vertica to do predictions on data stored in the database. This closes the loop on the full analytics pipeline with V2S and MD shown in Figure 1.

4. EXPERIMENTAL EVALUATION

In this section we evaluate our connector's functionality, performance, and scalability. We examine the connector's performance for both directions V2S and S2V. We evaluate the connector's sensitivity to parallelism, data size, data type, and data shape, as well as scalability with different cluster sizes for Vertica and Spark. Lastly, we compare V2S load (read) and S2V save (write) performance to other baselines including Spark's JDBC DefaultSource, HDFS, and Vertica's native bulk load utility.

4.1 Experimental Setup

Here we describe our software configuration, hardware configuration, datasets, and cluster configurations. For each experiment, the data originates in HDFS (co-located with Spark), and we use S2V in Overwrite mode to first save the data from Spark to Vertica then use V2S to load that same data back into Spark which ensures that the data used for S2V and V2S is exactly the same. All experimental results report the average execution time for 3 runs using a local cluster of machines.

Software Configuration. All of our experiments use HPE Vertica 7.2.1 and Apache Spark 1.5.2 configured for Hadoop 2.4+. Spark is configured with the Kryo serializer, and we assign roughly 75% of each machine's cores to Spark, leaving the remaining cores available for other background tasks including HDFS which is co-located with Spark. Spark Workers were allotted all local machine memory, and Spark executors were allotted 10GB each. We use HDFS from the Apache Hadoop 2.6 distribution, and HDFS is configured with the default block size (64MB) and replication (3×). Vertica is configured with the default settings except that replication (*k-safety*) is turned off simply for clarity of evaluation of data movement rather than database fault tolerance.

In Vertica we define a resource pool with default settings using $\frac{1}{2}$ of machine RAM (32GB in this case), and all Vertica connections use this pool. This simply helps to isolate and control the resources used by our data movement workloads from the normal database workloads. We also increased Vertica's `MAX-CLIENT-SESSIONS` parameter to 100, to allow for up to 100 connections per node during our experiments that vary the number of connections to Vertica (i.e., parallelism). The only other configuration settings explicitly modified for Vertica and Spark were temp directory paths. Other than as noted above, default settings were used; we performed no further optimizations.

Hardware Configuration. We use up to 24 machines running Red Hat Enterprise Linux Server release 6.7, each with 2 Intel Xeon 2.0GHz CPUs of 8 physical cores each with simultaneous multithreading (i.e.,hyper-threading) enabled resulting in 32 logical cores per machine. Each machine has 3 HDDs, one for the OS and applications, one for data storage, and one for temp space. Vertica table storage and HDFS block storage are located on the data storage disk. Vertica and Spark temp directories (used in part during shuffle/sort/spill) are located on the temp space disk. The machines have 64GB of RAM, and 2× 1GbE network interfaces. To separate the network traffic, we install Vertica onto one of the interfaces, and Spark onto the other. The keeps all Vertica internal traffic on one network and Spark traffic on the other, as would typically be done in a production environment with Enterprise Hardware.

Datasets. We use 2 datasets in our experiments. Dataset one (D_1) consists of 100 columns of randomly generated 8-byte float values in range [0–1), and in csv format consumes 140 GB on local disk. It contains 100 million rows (100M). Dataset two (D_2) consists of 2 columns of twitter data, the first column has a tweet_id (Long type) and the second column has tweet_text (String type), and in csv format consumes 140 GB on local disk. It contains 1.46 billion (1.46B) rows. In Vertica, string data types are represented as VARCHAR columns. In HDFS, all datasets are stored as delimited text files (CSV).

Cluster Configuration. Our primary cluster configuration uses 4 nodes for Vertica, and 8 nodes for Spark which is co-located with HDFS. We refer to this as our 4:8 cluster. This represents a 2× ratio of 2 nodes Spark/HDFS for 1 node Vertica. Empirically we observed the 2× ratio was able to sufficiently saturate network resources between the systems. All experiments use the 4:8 cluster except the cluster

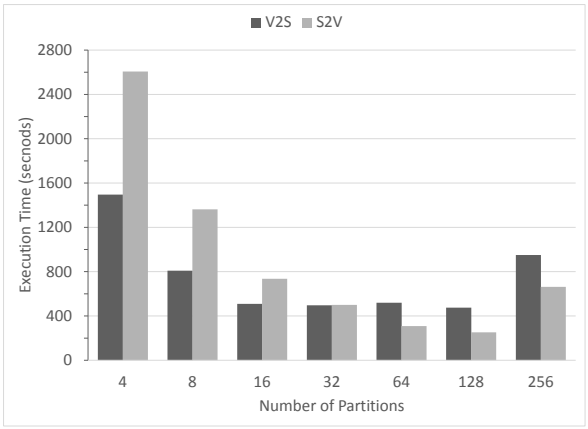

Figure 6: Varying the number of partitions.

scalability experiments. For cluster scalability experiments, we maintain the 2× ratio and vary the cluster sizes.

4.2 Performance While Varying Parallelism

In this experiment we evaluate how S2V and V2S perform as we vary the amount of parallelism. Here we use dataset D_1 with 100M rows. Figure 6 reports the execution time for V2S and S2V along the y-axis and the number of partitions along the x-axis. Given that our Vertica cluster has 4 nodes, on the far left we use 4 partitions resulting in 1 connection per Vertica node, and on the far right we use 256 partitions, resulting in 64 connections per Vertica node.

Figure 6 shows the best performance for V2S and S2V occurs somewhere in the middle ranges of partitions. For V2S, the fastest time is achieved with 128 partitions at 475 seconds. However, our experiences have shown that even though with 32 partitions performance is slightly slower at 497 seconds (<5%) using this lower value for the number of V2S partitions results in less resource usage on Vertica to achieve roughly the same performance, hence 32 is a better value in practice. For S2V, the fastest time is achieved with 128 partitions at 252 seconds.

Figure 6 clearly shows that both V2S and S2V are sensitive to the amount of parallelism. At a high level, 4 partitions is not enough parallelism (due to not enough work generated per connection) while 256 partitions is too much parallelism (due to the amount of overhead per connection). We drill down into this behavior in the following section.

4.2.1 Resources Used with Varying Parallelism

Next we examine the machine resources used during the experiments shown in Figure 6. That figure shows that both V2S and S2V performance degrades at the lower and higher ends of the parallelism spectrum tested. The Vertica resources are key, since Spark tasks are only formulating requests and sending those to Vertica for processing.

Table 2 shows the amount of CPU and network resources (outbound) used by a single Vertica node during the first 300 seconds of V2S with 4 partitions and 32 partitions during the experiments from Figure 6. In the plots shown in Table 2, max CPU usage possible is 100%, and max Network usage possible is about 125 MBps.

With 4 partitions, there is an initial burst of CPU, which then reaches steady-state at about 5% usage. Correspondingly, there is an initial saw-tooth behavior for network

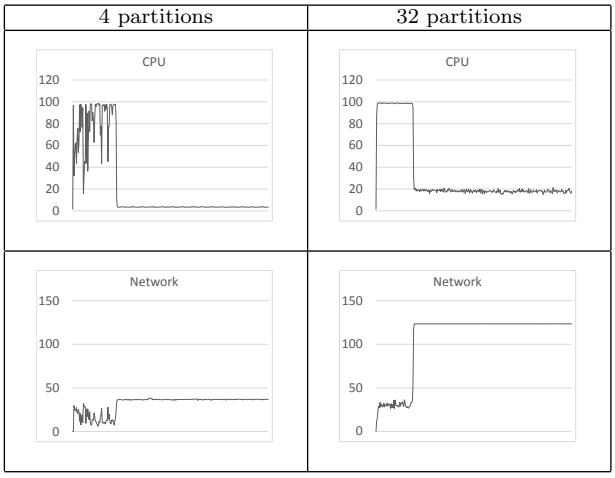

Table 2: Vertica node resource usage during V2S. The y-axis is percent CPU usage or Network MBps, the x-axis is the first 300 seconds of elapsed time.

MBps until it reaches steady-state at about 38 MBps. Because there are only a few requests, the network is not saturated and consequently it takes longer to complete the data transfer, as reflected in the execution time for V2S with 4 partitions in Figure 6. In contrast, with 32 partitions the CPU and network show the same initial behavior, after which the CPU reaches steady-state at about 20% usage and the network becomes fully saturated at about 120 MBps —thus the data transfer is faster as reflected in the lower execution time for V2S with 32 partitions in Figure 6. Because the network becomes saturated at this degree of parallelism, increasing the parallelism only increases the overhead without any data transfer benefit. This effect is reflected in the higher execution time for V2S with 256 partitions in Figure 6. A similar effect occurs with S2V (not shown here), although the best results are with 128 partitions instead of 32. This is because with S2V, each Spark task is alternately encoding its data into Avro format or transferring the data to Vertica so it benefits from more parallelism but S2V similarly becomes saturated with too much parallelism.

4.3 Data Scalability

Here we evaluate the performance of V2S and S2V as the data size is scaled up exponentially. We use dataset D_1 and vary the number of rows from 1 million (1M) to 1 billion (1000M). To set the parallelism for V2S and S2V, we choose the number of partitions with the best performance for each direction from Figure 6 which is 32 and 128 partitions respectively.

Figure 7 reports the total execution time in seconds (y-axis) as the number of rows is varied (x-axis), with both axes shown in log scale. The performance trend for both S2V and V2S is clearly linear, showing the connector scales well with the data size. With fewer rows, S2V is somewhat slower than V2S, most likely due to its additional overheads for batching and transactional consistency using the staging and commit tables. With more rows, the performance of each becomes similar and then S2V becomes somewhat faster than V2S. We performed additional experiments with 2 billion rows and greater, which showed these trends continued and thus are not presented here.

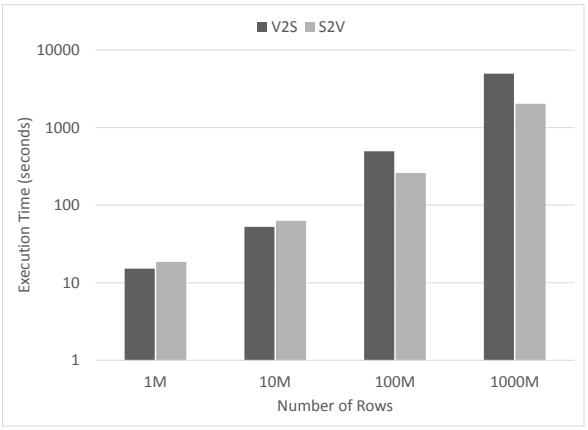

Figure 7: Varying the data size.

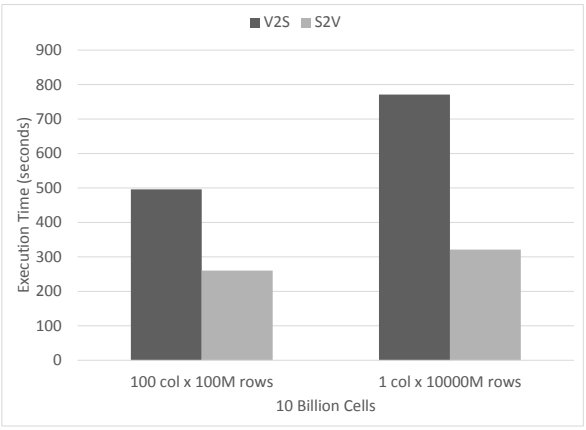

Figure 9: Varying the data dimensionality.

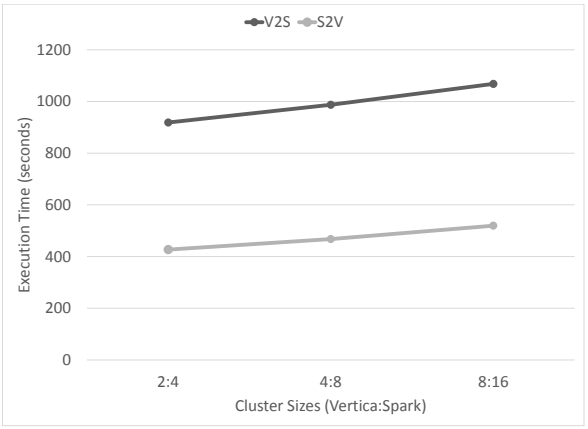

Figure 8: Varying the cluster sizes.

4.4 Cluster Scalability

In this experiment we vary the cluster sizes for our Vertica and Spark clusters while keeping the same 2× ratio. We start with 2 Vertica nodes and 4 Spark nodes denoted as 2:4, and then we double the cluster sizes to 4:8 and 8:16. We also double the data sizes with each cluster size, such that the amount of data per node is fixed. We use dataset D_1, with 100M rows for the 2:4 size, which is then doubled to 200M rows for the 4:8 cluster and then to 400M rows for the 8:16 cluster.

We base the number of partitions on the best values for V2S and S2V from Figure 6. Following these, we scale the number of partitions for V2S and S2V up and down respectively with the cluster size resulting in 16 and 64 respectively for the 2:4 cluster, 32 and 128 for the 4:8 cluster, and 64 and 256 for the 8:16 cluster.

Figure 8 reports the execution time of V2S and S2V as the cluster size and data size are scaled up in the corresponding 2× ratio. The results show that performance of V2S and S2V have a slight linear performance degradation (<10%) with each doubling of the cluster size. Overall, Figure 8 indicates that V2S and S2V scale reasonably well but we are considering optimizations as an area of future work.

	Execution Time (seconds)
V2S	378
S2V	386

Table 3: Performance with dataset D_2.

4.5 Varying the Data Dimensionality

In this experiment we examine what is the effect when the data size is the same, but the number of columns is varied. Here we use dataset D_1 which is 100 cols by 100M rows as our baseline, and this dataset has 10,000M cells. We then modify the dimensionality of this dataset to 1 col by 10,000M rows, which again has 10,000M cells. The goal is to show how performance can vary with the shape of the data.

Figure 9 reports the execution time for both variations of dataset D_1. The results show that execution time increases significantly when there is only 1 column but far more rows. This indicates there is some fixed amount of overhead per row, since even though the amount of data is the same, the results with 10,000M rows take longer to execute. For V2S, this includes the overhead for reading and encoding many more rows of data to send over JDBC, and for S2V this includes the fixed overhead to encode each row into Avro format in Spark and the overhead of parsing and unpacking each Avro row in Vertica.

4.6 Performance with Dataset D_2

In this experiment we evaluate performance with dataset D_2, which contains one column each of *Integer* and *String*. Although D_1 and D_2 are the same raw size on disk, 140 GB, D_1 has 100 million rows while D_2 has 1.46 billion rows. We again use V2S and S2V with 32 and 128 partitions, respectively. Table 3 reports the execution time in seconds for dataset D_2 using V2S and S2V. The results are interesting in that although the data sizes of D_1 and D_2 are the same, V2S is faster to load dataset D_2 (378 seconds here) than dataset D_1 (490 seconds from Figure 6). In contrast, S2V is slower to load dataset D_2 (386 seconds here) than dataset D_1 (252 seconds from Figure 6). This can be explained in part due to the performance variance of S2V and V2S seen in Figure 9, which shows the execution times change as the

number of rows and columns is varied even though data size remains the same. There are also differences in the way Vertica stores and converts different data types to/from their native formats which can affect the reading and writing time.

4.7 Comparisons with Alternative Approaches

In this section we compare V2S and S2V with several alternative approaches to load and save Spark data using dataset D_1. We first compare V2S and S2V with Spark's JDBC DefaultSource, a basic implementation of Spark's External Data Source API. The JDBC default source implements both the load and save functions. Since load supports pushdown predicates, we also test with a selection predicate. We next compare V2S and S2V with reading and writing to/from HDFS. This is a less direct comparison than JDBC DefaultSource, but serves to show how our implementation performs to shuttle data between Spark and Vertica as compared to shuttling data between Spark and HDFS, which Spark supports natively. Last we compare S2V's performance with Vertica's native bulk load utility, the `COPY` statement.

4.7.1 Comparison with JDBC Default Source

Spark's JDBC Default Source provides both load and save methods and also supports parallelism, although with parallel approaches transaction control is not centralized and thus does not provide exactly once semantics.

For load, JDBC Default Source enables parallelism by issuing a unique query per Spark partition, where each query requests a range equally divided over a `min` and `max` value of the source table. Notably, in order to obtain parallelism the current implementation requires the source table to have an integer column, and the user to know its `min` and `max` values which are provided as arguments along with the column name to JDBC Default Source. If this is not done, it will default to a single partition which results in zero parallelism. To support this requirement, for our V2S experiments below we modify dataset D_1 to add an integer column with randomly assigned values from [0-100), allowing us to easily select any percentage of rows we wish.

Figure 10 reports the execution time to load data from Vertica to Spark with V2S and Spark's JDBC Default Source. Both approaches use Spark's External Data Source API, which supports predicate pushdown. In this experiment we use a select predicate with 5% selectivity, resulting in Vertica filtering out 95% of the data before sending the result back to Spark. The results show that by pushing down the filter, Vertica does most of the work and the output is significantly reduced (to just 5M rows) and thus both approaches have simlilar performance. However, although both approaches benefit from parallelism, the results without pushdown show a performance gain of nearly 4× for V2S over JDBC Default Source. This is due to the hash-ring based query approach used by V2S whereby each connection only requests node-local data from each Vertica node. Not only does this result in better performance, it also does not induce intra-node traffic within Vertica, leading to less Vertica resource usage overall than with the JDBC Default Source approach. JDBC Default Source also issues all queries throught a single Vertica node, it does not distribute the queries evenly across all nodes, although this is likely an easy change. Moreover, with V2S there is no requirement

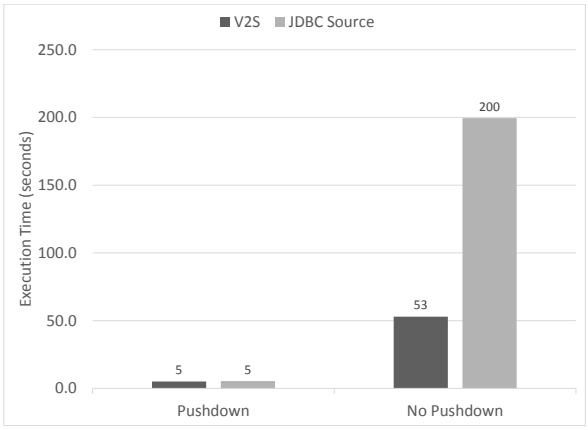

Figure 10: Load with V2S versus JDBC DefaultSource, with/without pushdown, 5% selectivity.

that the source table contains an integral column or that the user knows its `min` and `max` values. V2S's hash-ring query approach works with any existing Vertica table, and does not require the user to provide any special arguments other than the table name. For save, JDBC Default Source enables parallelism by each Spark partition issuing batches of insert statements to the target table. Notably, although each partition will commit its inserts at the end, these are independent queries thus they are not all under transaction control which can result in the target table being partially or duplicate loaded.

Figure 11 reports the execution time to save data from Spark to Vertica with S2V Spark's JDBC Default Source. In this experiment, we use dataset D_1 without any column modifications but we use only a subset of the rows. We report the execution times for 1 row, 1,000 rows (1K), 10,000 rows (10K).

The results for 1 row effectively show the overhead for both approaches; 5 seconds for S2V and 3 seconds for JDBC DefaultSource. For S2V, the overhead includes the one-time setup and teardown time of the temporary tables along with the logic to guarantee exactly-once semantics, which JDBC Default Source does not provide. Due to these overheads, S2V is slightly slower for the single row case. However, S2V's performance for larger numbers of rows is significantly better than JDBC Default Source. This is because S2V is designed for bulk loads as part of an ETL process, and it uses the `COPY` command along with batching, which is typically far faster than using a series of insert statements. We ran this experiment for larger data sizes but the trends shown in Figure 11 continued and hence results are not shown here. For example, for 1M rows S2V takes 19 seconds (as shown in Figure 7), while JDBC Default Source took longer than 3 hours, at which point we stopped the experiment.

4.7.2 Comparison with HDFS

In this section we compare our V2S and S2V approaches to load and save data between Spark and Vertica with that of reading and writing data between Spark and HDFS, for which Spark is natively integrated with HDFS source. We use our standard dataset D_1 with 100M rows. This experiment serves to show that reading and writing data between Spark and Vertica can be competitive with using HDFS. Here we use our standard 4:8 cluster (Vertica:Spark) for

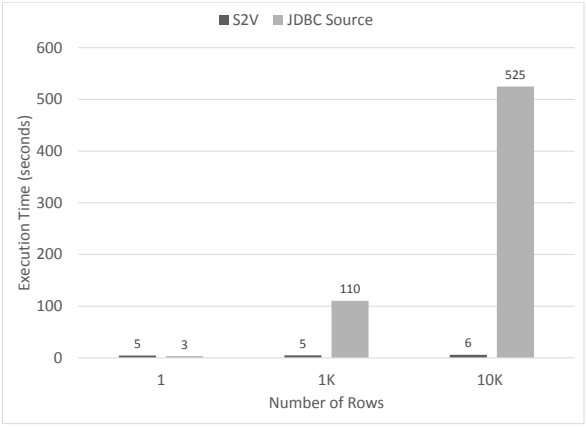

Figure 11: Save with S2V versus JDBC DefaultSource.

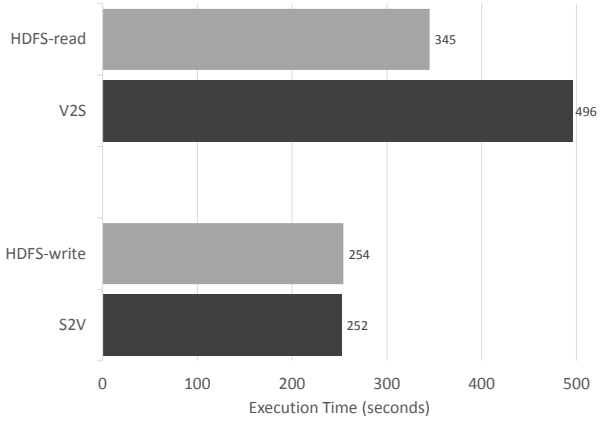

Figure 12: V2S and S2V versus HDFS read/write.

	Execution Time (seconds)
S2V	252
COPY	238

Table 4: Save with S2V versus native `COPY`.

V2S and S2V, but in addition we create another 4:8 cluster (HDFS:Spark) for the HDFS experiment in order to perform a direct comparison with our approach. Note that just like our 4 Vertica nodes, the 4 HDFS nodes are not co-located with Spark. For HDFS read and write we use Spark's native read/write methods for parquet files using DataFrames. This dataset consists of 2240 HDFS blocks, which by default results in 2240 partitions in Spark. For V2S and S2V we use the values for 32 partitions and 128 partitions respectively from Figure 6. Figure 12 reports the time to load/save data from Vertica to Spark using V2S and S2V compared with reading and writing DataFrames using HDFS. Execution time is shown along the x-axis, and along the y-axis are our three comparative methods.

For reading, the results in Figure 12 show that reading from HDFS is about 30% faster than using V2S. This is not surprising given that Spark natively supports reading from HDFS. The primary reason is that the number of partitions with HDFS read is higher since it will default to one partition per HDFS block. This results in higher parallelism (2240 partitions) than with V2S (32 partitions). With HDFS, each partition is simply an HDFS block retrieval whereas with V2S each partition results in a separate query and together all queries result in loading a consistent view of Vertica data into Spark. Moreover, each V2S query executes a hash function on each data row which is the basis of our hash-ring approach. Since HDFS is not a database and HDFS files are not updated in place, there are no issues that can cause an inconsistent view of the data as there can be with a database. For writing, the results in Figure 12 show that writing to HDFS has about the same performance as S2V. Thus, considering the case when Spark is not co-located with Vertica or HDFS, these results show that data in Spark can be saved to a Vertica cluster about as fast as saving to an HDFS cluster.

4.7.3 Comparison with Vertica's bulk load utility

In this experiment we compare S2V with Vertica's native bulk load utility, i.e., the `COPY` command, which is the standard way to load large amounts of data into Vertica. Here we use `COPY` as a baseline to compare S2V and gauge its performance potential. To do a parallel `COPY`, a file is split into multiple parts, evenly distributed among the Vertica nodes, and a `COPY` is issued on all of the parts.

In this experiment we use dataset D_1 to compare the loading time of S2V to parallel `COPY`. For S2V, we consider the execution times from Figure 6. For the parallel `COPY` approach, we split the original csv file into 4, 8, 16, 32, 64, and 128 parts and distribute them evenly across the 4 Vertica nodes onto their local disks We then record the total time to load the splits in parallel.

Table 4 reports the best time for S2V and the parallel `COPY` approach. The best time for `COPY` was 238 seconds, achieved with 8 file parts, 2 on each node. S2V's best time is about 6% slower than COPY; 252 seconds achieved with 128 partitions. This result shows that S2V's best time can be competitive with `COPY`, but it requires higher parallelism and consequently more Vertica resources to obtain similar performance.

5. DISCUSSION

The connector's sensitivity to the number of Spark partitions is a drawback, since it is not directly obvious what number will perform best given the user's machine configurations and cluster sizes. However, there is a clear bowl-shape to the performance as Figure 6 shows. While values at the outer ranges perform poorly, values in the middle ranges perform reasonably well. Best practices for tuning the parallelism are provided in the connector's user guide. Although memory can also be a constraint, the performance we report in this work was achieved with very modest memory sizes.

Another drawback is the connector's performance when using S2V in Append mode that requires to copy the entire staging table into the target table. While saving each task's data to the staging table is done in parallel under many independent transactions, saving the staging table into the target table must actually be a single atomic transaction which can have a higher execution time cost. We are currently exploring optimizations that can help to reduce the append time.

There are several other interesting areas of future work that we hope to investigate, in order to improve performance

and scalability. One is to pre-hash the DataFrame in S2V to match the same partitioning (*segmented by* clause) as the target Vertica table similar to our approach for V2S in Section 3.1. Each pre-hashed partition can then connect to and load its data into the corresponding Vertica node that stores that particular segment, which eliminates the network traffic between nodes in the Vertica cluster.

Another is to consider a 2-Stage approach, that is, to first copy the data to an intermediate location that both systems can access, such as HDFS or S3 in the first stage then transfer the data to the other system in the second stage. This is the approach taken by the Spark-Redshift connector [8]. One drawback of using 2 stages is that it creates an extra dependency on an intermediate storage system, and it may be slower than our single-stage approach because it requires an intermediate write of a full copy of the data. As we showed in Section 4.7.2, our approach for reading/writing directly to Vertica can be competitive with reading/writing from Spark to HDFS. However, a 2-stage approach has the potential benefits of decoupling the systems and good performance given each system's native capabilities for reading/writing the intermediate storage.

6. RELATED WORK

In this section we describe some of the related work in the area of accessing external data sources with Spark.

Spark's JDBC Default Source. Spark provides a JDBC Default Source as an implementation of its External Data Source API. The implementation offers the ability to parallelize loads and saves, enabling load or save to any database that supports JDBC. Similar to our work, it also provides pushdown capabilities via the External DataSource API. However the current JDBC Default Source approach is only "best-effort" in that it does not guarantee a consistent view of the data due to task failures. In contrast, our approach for V2S and S2V includes several mechanisms to guarantee exactly once semantics resulting in a consistent view of the data. Moreover, in Section 4.7 we show that our connector can significantly outperform JDBC Default Source.

Additionally, as noted in Section 4.7.1, JDBC Default Source has some special requirements in that it can only parallelize the load of a source table if the table has an integer column, and the user provides the column name along with the column's `min` and `max` values. Furthermore, even when transfer is parallelized all connections go through a single database host, causing that host to do much of the work. In contrast, V2S requires no special table format nor any `min` and `max` values to be provided by the user. V2S parallelism is based on Vertica's data layout, and balances the load across Vertica nodes by enabling each Spark task to connect to a specific Vertica node and request only node-local data based upon a hash segment range.

When writing data to Vertica, JDBC Default Source parallelizes the writes but uses `INSERT` statements, whereas S2V utlizes Vertica's `COPY` command which is optimized for bulk loads. Additionally since each JDBC Default Source task performs its `INSERT`s independently and in parallel, a total Spark failure or failed/restarted tasks may leave the target table in an inconsistent state.

Datastax Cassandra Connector for Spark. Cassandra [12] is a key-value store that does not support a full relational model. There is available an open source package on GitHub [4], which provides both load and save functionality for Spark RDDs and DataFrames. The Cassandra connector provides partitioning by Cassandra ranges during load/save, similar to our hash segmentation approach. The consistency semantics (strong consistency via `LOCAL_QUORUM`) offered for writes are similar to our approach but specific to Cassandra, whereas our approach applies more generally to MapReduce-class systems and any database with ACID semantics.

Databricks Redshift Connector. Redshift [1] is a data warehouse available on Amazon AWS, and there is an open-source connector package available on GitHub [8]. The connector is similar in functionality to our connector, providing parellelized read and write between Spark and Redshift, and also supports transactional semantics and pushdowns. However, this connector uses a two-stage approach for both reads and writes. That is, it requires a common shared data landing zone (in this case Amazon S3 [2]) to write temporary files in the first stage to then be consumed by the other side in the second stage. For example, with writes Spark first saves all partition data in parallel to S3, and once that completes successfully a Redshift LOAD query(s) is issued. This load query is bookended by a `BEGIN` and `END` transaction statement, and between those is a sequence of individual load statements, one per each temporary file in S3. Like our S2V approach, this achieves exactly once semantics but has some extra overhead in that it requires an intermediate data landing zone large enough for the entire dataset.

MR as ETL Engine. MapReduce type systems may often be used for ETL, as they allow for massive parallelism as well as code-specific optimizations and transformations when loading data into the target system (e.g., Vertica). Such systems can be referred to generally as the class of batch-based task scheduler systems as noted previously. Our approach for reliably transferring data to/from an ACID compliant database is general in that it can be applied to any such class of system.

Read and Write using HDFS. HDFS is not a database and does not have transactional semantics like our approach that uses Vertica, but we do provide a comparison point for reference in Section 4.7.2 that shows our approach can be competitive with reading/writing HDFS from Spark. This is a key result in that it shows one could use Vertica as a durable storage for Spark DataFrames instead of HDFS.

7. CONCLUSION

In this work we have presented our design and implementation for fast, efficient, and reliable data transfer between HPE Vertica and Apache Spark with exactly once semantics. We have evaluated its performance across several dimensions and have compared with alternative approaches, and the current results show that our methods are competitive with the alternatives tested. The HPE Vertica Connector for Apache Spark is available as a beta release on HPE Marketplace [5] since November 2015.

8. REFERENCES

[1] Amazon Redshift. https://aws.amazon.com/redshift/.

[2] Amazon Simple Storage Service. https://aws.amazon.com/s3/.

[3] Apache Avro data serialization.

[4] DataStax Cassandra Connector. https://github.com/datastax/spark-cassandra-connector.

[5] HPE Vertica Connector for Apache Spark. https://saas.hpe.com/marketplace/big-data/hpe-vertica-connector-apache-spark.

[6] JavaPMML API. https://github.com/jpmml.

[7] PMML 4.1 general structure. http://dmg.org/pmml/v4-1/GeneralStructure.html.

[8] Redshift data source for Spark. https://github.com/databricks/spark-redshift.

[9] Spark MLlib. http://spark.apache.org/mllib/.

[10] Spark PMML model export. https://spark.apache.org/docs/latest/mllib-pmml-model-export.html.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[12] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), Apr. 2010.

[13] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The Vertica analytic database: C-store 7 years later. In *VLDB*, volume 5, 2012.

[14] S. Prasad, A. Fard, V. Gupta, J. Martinez, J. LeFevre, V. Xu, M. Hsu, and I. Roy. Large-scale predictive analytics in Vertica: Fast data transfer, distributed model creation, and in-database prediction. In *SIGMOD*, 2015.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. In *VLDB*, 2005.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.