SHC: Distributed Query Processing for Non-Relational Data Store

Weiqing Yang*, Mingjie Tang*, Yongyang Yu†, Yanbo Liang*, Bikas Saha*
*Hortonworks †Purdue University
{wyang, mtang, yliang, bikas}@hortonworks.com, yu163@purdue.edu

***Abstract*—We introduce a simple data model to process non-relational data for relational operations, and SHC (Apache Spark - Apache HBase Connector), an implementation of this model in the cluster computing framework, Spark. SHC leverages optimization techniques of relational data processing over the distributed and column-oriented key-value store (i.e., HBase). Compared to existing systems, SHC makes two major contributions. At first, SHC offers a much tighter integration between optimizations of relational data processing and non-relational data store, through a plug-in implementation that integrates with Spark SQL, a distributed in-memory computing engine for relational data. The design makes the system maintenance relatively easy, and enables users to perform complex data analytics on top of key-value store. Second, SHC leverages the Spark SQL Catalyst engine for high performance query optimizations and processing, e.g., data partitions pruning, columns pruning, predicates pushdown and data locality. SHC has been deployed and used in multiple production environments with hundreds of nodes, and provides OLAP query processing on petabytes of data efficiently.**

***Index Terms*—Query processing and optimization; distributed computing; in-memory computing;**

I. Introduction

In the era of "big data", data is growing faster than ever before and by the year 2020, about 1.7 megabytes of new information will be created every second for every human being on the planet. Meanwhile, a large number of datasets come in non-relational formats (e.g., key-value pairs). For example, user check-in activities of Airbnb, friend messages of Facebook, and logistic information of trucks are modeled as key-value pairs, and stored in distributed non-relational data stores (e.g., HBase [8]). HBase is a column-oriented key-value store, and has been widely used because of its lineage with Hadoop [7] ecosystem. HBase runs on top of HDFS [9] and is well-suited for fast read and write operations on large datasets with high throughput and low input/output latency.

With the fast growth of accumulating data, there is a pressing need to handle large non-relational data spreading across many machines that are available for single-machine analysis. Thus, the challenge today is blending the scale and performance of NoSQL with the ease of use of SQL, and efficiently executing OLAP queries over the key-value stores. As a result, various new systems seek to provide a more productive user experience by offering relational interfaces on big data. Systems like Pig [15], Hive [10], Dremel [5], and Spark SQL [24] all take advantage of declarative queries to provide richer automatic optimizations.

More recently, Apache Phoenix [14], one of SQL query engines to access a key-value store, transforms a SQL query into native HBase API calls, and pushes as many operators as possible into an HBase cluster. It indeed relieves a user's burden to query HBase data by writing primitive HBase API calls. However, Apache Phoenix usually runs into performance bottlenecks when executing expensive computation tasks like joining data across different big tables. This is attributed to the fact that Phoenix is not an efficient computing engine itself, i.e., without cost-based query plan optimization. More importantly, Apache Phoenix is not able to read data efficiently when data is not originally written by Phoenix itself.

On the other hand, in-memory data processing engines, e.g., Spark [40] and Flink [6], are extended to provide a SQL interface for relational data operations. For example, Spark SQL [24] optimizes SQL query processing via a newly developed Spark SQL optimizer called Catalyst. Catalyst is capable of optimizing a SQL query plan via both rule-based and cost-based optimization mechanisms. However, Spark SQL is not able to optimize query processing for HBase data efficiently. For example, Spak SQL considers HBase as a general data source without understanding HBase's specific data storage mechanism, and it fails to push certain operations like data partition pruning or filtering predicate into HBase directly.

In this work, the extensive engineering works found that it is nontrivial work to provide relational query processing over distributed key-value stores, and optimizations within the distributed in-memory engines (e.g., Spark SQL). We demonstrate our efforts to provide SQL query processing on distributed key-value stores with Spark [40] and Spark SQL [24]. The major contributions of this work are listed below:

- We design a data model to map non-relational data to relational data. The model allows users to process non-relational data via the developed relational operators, e.g., SQL operators and Spark Dataframe API. Meanwhile, based on the proposed data model, we provide multiple data encoding and decoding mechanisms (e.g., Avro, Phoenix type) for reading and writing data to HBase efficiently.
- We develop a plug-in system implementation to embed into the current Spark SQL processing engine, and optimize the logical query and physical query plan based on the nature of distributed key-value stores. For example, we optimize the relational scan operations based on

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00165

Features	SHC	Spark SQL	Phoenix Spark	Huawei Spark HBase
SQL	√	√	√	√
Dataframe API	√	√	√	√
In-memory	√	√	√	√
Query planner	√	√	√	√
Query optimizer	√	√	√	√
Multiple data coding	√	√	×	×
Concurrent query execution	Thread pool	User-level process	User-level process	User-level process

TABLE I: Comparison between SHC and other systems

partition punning and filter pushdown into HBase. The related operators are invoked inside the HBase cluster directly. This saves Spark's computation and memory usage by scanning HBase's byte arrays directly.

- We propose a new security token management policy, this enable users to read data from multiple data stores in a secure environment. In this way, SHC is able to join data across multiple secure data stores in the same cluster.
- We demonstrate the system in a real productive environment, since it natively supports Spark over HBase for data processing. The released version is already adopted by multiple companies and end-users.

The rest of this paper is organized as follows. Section II lists the related work. Section III introduces the background information and related optimization targets of the system. Section IV provides the data model and programing interface of the system. Section V gives the system architecture and design principles. Section VI details these system implementations. Finally, experimental studies are given in Section VII, and Section VIII concludes the work.

II. Related Work

Unstructured key-value systems are an attractive approach to scaling distributed data stores because data can be easily partitioned and little communication is needed between partitions. Amazon's Dynamo [27] demonstrates how they can be used in building flexible and robust commercial systems. Many works have focused on providing such data stores for key-value data format.

Google's BigTable [25], Apache HBase [8] and Apache Cassandra [4] provide consistency and transactions at the per-record or row level, while these systems provide weaker guarantees than traditional ACID properties. Redis [17] is an in-memory storage system providing a range of data types and an expressive API for datasets that fit entirely in memory. ASTERIX [1] has a semi-structured data model internally. Flink [6] also has a semi-structured model.

Several systems have sought to combine relational query processing with the procedural processing engines initially used for large clusters. DryadLINQ [38] compiles language-integrated queries to a distributed DAG execution engine. LINQ queries are also relational but can operate directly on objects. Some well-known relational query processing engines are Hive [10], Impala [11] from Cloudera, BigSQL [3] from IBM, Presto [16] from Facebook. Fatma et al. [28] give a good survey and performance study on SQL for big data processing.

More recently, taking advantage of the very large memory pools available in modern machines, Spark and Spark-related systems (e.g., Graphx, Spark-SQL, and DStream) [40], [39] are developed to overcome the drawbacks of MapReduce. Spark SQL [24] is a relational query processing engine with a rich collection of relational operations to efficiently implement the computation over HBase. Unlike SHC, it requires that the dataset fits entirely in memory. For instance, Spark SQL considers the whole dataset as a HadoopRDD, and this is a general data type for different data sources. Without specific optimization, Spark SQL has to process redundant data and suffers from extensive computations. In this work, we extend Spark SQL especially for HBase data store, and tailor this system specifically for query optimization. This is not studied before. Table I gives a comparison on multiple systems to access HBase data and provide relational data operation. In addition, the plug-in system design in this work is motivated from GiST [29] and SP-GiST[23], which provide pluggable system to implement user's own indexes inside RDBMS. Meanwhile, multiple data management systems [32], [31] are built based on HBase for effective cooperation.

The ecosystem of Spark is growing quickly, and multiple systems are developed based on Spark or Spark SQL. For example, Simba [35] and LocationSpark [34] are extending Spark to support spatial query processing. MatFast [37] and Dmac [36] are optimizing the execution plan for efficient distributed matrix computations. Spark-llap [18] is providing database column and row level access control in Spark SQL.

III. Preliminaries

A. Overview of Spark and Spark SQL

Apache Spark [40], [39] is a general purpose, widely used cluster computing engine for big data processing, with APIs in Scala, Java and Python. Since its inception, the community has built a rich ecosystem based on Spark for in-memory big data analytics, including libraries for streaming, graph processing, and machine learning. Spark provides an efficient abstraction for in-memory cluster computing called Resilient Distributed Datasets (RDDs). Each RDD is a distributed collection of Java or Python objects partitioned across a cluster. Users can manipulate RDDs through the functional programming APIs (e.g., `map`, `filter`, `reduce`), which take functions in the programming language as input and ship them to other nodes in the cluster.

Spark SQL [24] integrates relational processing with Spark's functional programming API. Spark SQL leverages the benefits of relational processing (e.g., declarative queries and optimized storage), and allows users to call other analytics libraries in Spark (e.g., MLib [12] for machine learning) through its DataFrame API. Spark SQL can perform relational operations on both external data sources (e.g., JSON, Parquet [13] and Avro [2]) and Spark's built-in distributed collections (i.e., RDDs). Meanwhile, it evaluates operations lazily to get more optimization opportunities. Spark SQL

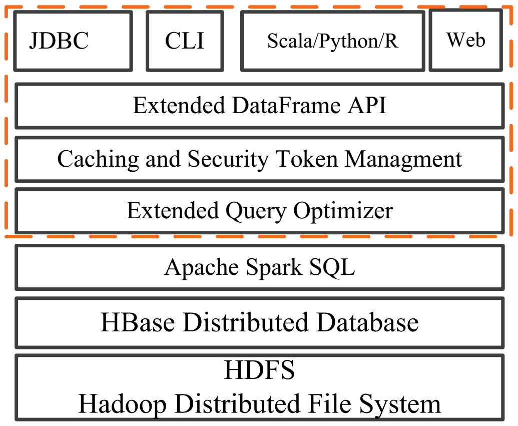

JDBC
CLI
Scala/Python/R
Web
Extended DataFrame API
Caching and Security Token Managment
Extended Query Optimizer
Apache Spark SQL
HBase Distributed Database
HDFS
Hadoop Distributed File System

Fig. 1: SHC architecture

introduces a highly extensible optimizer called Catalyst, which makes it easy to add data sources, optimization rules, and data types for different application domains such as machine learning and graph analytics.

B. Overview of HBase

Apache HBase is a high performance horizontally scalable data store engine for big data, suitable as the store of records for mission critical data. Instead of complete rows, data is stored as fragments of columns in the form of $\langle$key, value$\rangle$ pairs. HBase contains two main components: (1) an HMaster, and (2) a set of Region Servers. The HMaster is responsible for administrative operations, e.g., creating and dropping tables, assigning regions, and load balancing. Region Servers manage the actual data stores. A Region Server is responsible for region splitting and merging, and for data manipulation operations [8]. An HBase table may have one or more regions, where each region contains HBase Store Files that are the basic data storage units. In addition, each region contains a temporary Memory Store and a Write-Ahead-Log file for fault tolerance. HBase operates alongside ZooKeeper [22] that manages configuration and naming tasks.

When a client executes an HBase operation, it contacts ZooKeeper to obtain the operation address of the Region Server that hosts the data on which the client will operate. Next, the client executes the required operation by accessing the region where the desired data is stored. Although the HMaster does not play a role in data manipulation operations, it orchestrates in the background to manage and monitor administrative tasks related to tables and regions.

C. Technical Challenges and Targets of SHC

We are planning to extend Spark SQL to meet the following goals.

Generalizing Relational Data Model

Data is stored as $\langle$key, value$\rangle$ pairs in HBase, and it enables an efficient data storage in a distributed setup. However, this simple data storage format inhibits users from processing data with familiar relational operations efficiently. Therefore, we propose a new data model to map non-relational data to relational data tables, and develop associated relational data operators. Users only need to focus on the predefined relational model, and execute relational queries (i.e., SQL or Dataframe API) over the underlying distributed non-relational key-value store. Thanks to relational query optimization techniques, this would save the computation cost.

Light-weight and Efficient Plug-in Implementation

SHC provides a high-performance implementation using the current Spark SQL optimization engine. We are able to embed a user's query into the current workflow of SQL query plan with least modification in Spark SQL itself. Therefore, we are going to take advantage of Spark Data Source API(SPARK-3247) introduced from Spark-1.2.0. The Data Sources API provides a mechanism for accessing structured data through Spark SQL. Filtering and column pruning can be pushed all the way down to the data source in many cases. Spark community have made extensive efforts to optimize query processing over multiple data sources, e.g., Avro, CSV, and Dbase Table File Format. However, until now, we cannot find any optimization for HBase specifically, e.g., without unified data modeling, efficient data partitioning, column pruning, and native data encoding or decoding.

Therefore, in this work, we hope to bridges the gap between the simple HBase key-value store and complex relational SQL queries, enabling users to perform complex data analytics on top of HBase using Spark. In order to work with the current Spark data processing pipeline more tightly, an HBase Dataframe is extended to make an efficient access to data stored in HBase. Specifically, a DataFrame for Spark is equivalent to a table in a relational database, and can be manipulated in similar ways on the "native" distributed collections in Spark (RDDs). Unlike an RDD, a DataFrame keeps track of its schema and supports various relational operations that lead to more optimized executions.

An HBase DataFrame is a standard Spark DataFrame, and is able to interact with any other data sources such as Hive, ORC, Parquet, JSON, etc. We choose Spark SQL Dataframe rather Spark RDD, since we want to optimize the query processing based on Spark SQL plans. Meanwhile, HBase community has already developed a HBase connector for Spark, and it has a rich support at the RDD level. However, the design relies on the standard HadoopRDD with HBase built-in TableInputFormat, and it has some performance limitations. For example, a HadoopRDD fails to understand the schema of data and performs redundant data processing while scanning tables. On the other hand, Spark is advocating Dataset and DataFrame APIs, which provide built-in query plan optimization. End-users prefer to use DataFrames and Datasets based interfaces.

There are other alternative implementations to access HBase data via Spark DataFrame, e.g, [19]. However, their design is somewhat complicated. For example, Huawei HBase connector [19] applies very advanced and aggressive customized optimization techniques by embedding its own query optimization plan inside the standard Spark SQL Catalyst engine, shipping an RDD to HBase and performing complicated tasks inside the HBase coprocessor. This approach is able to achieve high runtime performance. But as we know, Spark evolves very quickly. It's difficult to maintain stability due to its design complexity. Therefore, we plan to develop a new system to overcome these potential bottlenecks and weakness.

Efficient Data Encoding and Decoding

To store data in HBase, at first, one should convert various data types (e.g., Avro, Phoenix, and Scala type) into HBase's byte arrays. Therefore, SHC provides different data encoders, and users can choose a candidate based on their needs. More importantly, SHC is able to scan data in HBase's byte array format natively. Therefore, Spark does not need to read the whole HBase data store into internal memory. This would save extensive computation and space.

Secure Access to Multiple Data Stores

Due to static token acquisition of Spark, Spark only supports use cases when a user accesses a single secure HBase cluster. It does not offer the ability to dynamically talk to a new secure service after the application has been launched without a restart. For example, there are multiple services in a data center. Two HBase clusters are storing the input of streaming data (e.g., user actives), while another cluster stores the static user information (e.g., users' profiles) in Hive tables. When a data scientist wants to analyze user's shopping habits by querying both the HBase and Hive data storage, the Spark application needs to access multiple secure data storage servers simultaneously. Therefore, SHC develops a new credentials manager to provide security token propagation when accessing multiple services.

Overall, SHC is built on Spark SQL [24] and is optimized specially for large scale relational query processing and data analytics over distributed non-relational data stores (i.e., HBase). SHC inherits and extends SQL and DataFrame API, so that users can easily specify different queries and analytics to interact with the underlying data. Figure 1 shows the overall architecture of SHC. SHC follows a similar architecture as that of Spark SQL, but introduces new features and components across the system stack. In particular, new modules different from Spark SQL are highlighted by the orange box in Figure 1. At first, we add more logical and physical plan optimization based on Spark SQL query optimizer, this is specifically designed for under-hood storage, in this work, we can choose HBase storage. Next, a security token manager is developed for users to access multiple data storage. Finally, user can access SHCthrough provided JDBC, command line in Scala, Python and R language.

IV. Data Model and Query Language API

In this section, we describe how SHC models non-relational data, and the corresponding data encoding or decoding mechanism to store different data types in HBase. Finally, we present extended relational operation APIs.

A. Data Mapping

HBase Model

HBase is a column-oriented database, and is the implementation of Google's Big Table storage architecture. It can manage structured and semi-structured data and has some built-in features such as scalability, versioning, compression and garbage collection. HBase is not a relational database and provides four coordinates definition (i.e., Row Key, Column families, Column Qualifiers, and Versions) for accessing data. At first, each row has a unique row key. Next, the data inside a row is organized into column families; each row has the same set of column families, but across rows, the same column families do not need the same column qualifiers. Note that HBase stores column families in their own data files. Thus, they need to be defined beforehand, and it is difficult to make changes to column families. Furthermore, column families define actual columns, which are called column qualifiers. Naturally, each column can have a configurable number of versions, and one can access the data for a specific version of a column qualifier.

Therefore, an individual row is accessible through its row key and is composed of one or more column families. Each column family has one or more column qualifiers (called "column") and each column can have one or more versions. To access an individual piece of data, one needs to know its row key, column family, column qualifier, and version.

Relational Data Mapping in SHC

We model HBase data as a relational table, and for each table, a data catalog is defined beforehand. Each catalog includes the row key, and the columns of data type with predefined column families. The catalog defines the mapping between the HBase columns and the mapped relational table schema. A running example is given below to illustrate the HBase schema mapping.

Given users' access activity logs to a website, data scientists need to store user's associated actives, e.g., users' profiles, visited pages, and corresponding time into HBase. Therefore, we build a mapping from an HBase table to a SHC's relational table via the following catalog. The predefined data catalog enables users to execute relational operations on the HBase table without knowing details of the underlying data store.

```
def catalog = s """{
"table":{"namespace":"default",
    "name":"actives",
"tableCoder":"PrimitiveType",
    "Version":"2.0" },
"rowkey":"key",
"columns":{
"col0":{"cf":"rowkey",
    "col":"key","type":"string"},
"user-id":{"cf":"cf1", "col":"col1",
    "type":"tinyint"},
"visit-pages":{"cf":"cf2", "col":"col2",
    "type":"string"},
"stay-time":{"cf":"cf3", "col":"col3",
    "type":"double"},
"time":{"cf":"cf4", "col":"col4",
    "type":"time"}
}
}""".stripMargin
```

Code 1: Catalog definition in SHC

Take Code 1 as an illustration, we build a table called `actives` and the input data encoding format is `PrimitiveType`. The version number of input table is recorded as `Version: 2.0`. In addition, there are two critical parts of this catalog. First part is the row key definition of the input table, and the other part is the columns mapping between table column of relational table and the column family and column qualifier in HBase. For example, code 1 defines a schema for an HBase table with name as `actives`, row key as `key` and a number of columns (`col1 - col4`). Note that the row key has to be defined in details as a column (`col0`), which has a specific column family (i.e.,`rowkey`).

B. Data Encoding and Decoding

There are no specific data types, e.g., `String`, `Int`, or `Long` in HBase, and all data is stored as byte arrays. Therefore, when a data instance is inserted, it is converted into a byte array using HBase provided `Put` and `Result` interfaces. HBase implicitly converts the data to byte arrays through a serialization framework, stores it into the cell of HBase, and produces the byte arrays. Thus, HBase implicitly converts data to an equivalent representation while putting and getting the value. Thus, SHC come to process existing HBase tables as long as they are storied as specific data types, e.g., Java primitive types, Avro records, Phoenix data types as listed below. Meanwhile, thanks to the plug-in implementation of SHC, we also support customized data encoders, i.e., users can define their own data types based on certain application requirements.

1) Java primitive types: `PrimitiveType` encoder is a customized serialization using a simple native encoding mechanism. It leverages HBase's native encoding/decoding, and does extra work to resolve the order inconsistency between Java primitive types and the byte array of HBase. SHC can use it to support Java primitive types like `int`, `String`, and so on.

During the data scanning, HBase is not aware of the data type except for byte arrays, and the order inconsistency between Java primitive types and byte array. Therefore, we have to preprocess the filtering condition before setting the filter in the `Scan` operation of HBase specifically, this would help avoid any data loss. Inside the Region Server, records not satisfying the query condition are filtered out. The details to scan the primitive bytes array are introduced in Section VI.

2) Avro: SHC supports the Avro format natively, as it's a common practice to store structured data in HBase. Users can persist Avro records in HBase directly. SHC fully supports all Avro schemas. Internally, an Avro schema is converted to a native Spark Catalyst data type automatically. Code 2 shows how SHC inserts Avro records into HBase.

```
def catalog = s"""{
"table":{"namespace":"default",
    "name":"Avrotable"},
"rowkey":"key",
"columns":{
"col0":{"cf":"rowkey", "col":"key",
    "type":"string"},
"col1":{"cf":"cf1", "col":"col1",
    "type":"binary"}
}
}""".stripMargin

sc.parallelize(avros).toDF.write.options(
    Map(HBaseTableCatalog.tableCatalog ->
        catalog,
        HBaseTableCatalog.newTable ->
        "5"))
        .format("org.apache.spark.sql.
    execution.datasources.hbase")
    .save()
```

Code 2: Avro record insertion in SHC

At first, a catalog is defined, and this is the schema for the HBase table named `Avrotable`, where row key as `key` and one column `col1`. The row key has to be defined in details as a column (`col0`), which has a specific cf (`rowkey`). Next, a collection named `avros` is prepared by end-users to insert into HBase. The call to function `sc.parallelize(avros)` distributes data to create a RDD. Then, function `toDF` returns a DataFrame, and writes the associated DataFrame to the external storage system (e.g. HBase). Specifically, given a DataFrame with a specified schema catalog, function `save` creates an HBase table with five regions and saves the corresponding Avro records. The number of regions is defined by parameter `HBaseTableCatalog.newTable`.

```
def avroCatalog = s"""{
          "table":{"namespace":"default",
              "name":"avrotable"},
          "rowkey":"key",
          "columns":{
           "col0":{"cf":"rowkey",
               "col":"key",
               "type":"string"},
           "col1":{"cf":"cf1",
               "col":"col1",
               "avro":"avroSchema"}
          }
        }""".stripMargin

 def withCatalog(cat: String): DataFrame =
     {
    sqlContext
        .read
        .options(Map(HBaseTableCatalog.
        tableCatalog -> avroCatalog))
        .format("org.apache.spark.sql.
        execution.datasources.hbase")
        .load()
 }
 val df = withCatalog(catalog)

 val result = df.filter($"col0" <=
     "row120").select("col0", "col1")
```

Code 3: OLAP query with Dataframe API in SHC

3) Phoenix type: SHC can be used to write data to an HBase cluster for further downstream processing. It supports

Avro serialization for input and output data, and by default it invokes a customized serialization using a simple native encoding mechanism. When reading input data, SHC pushes down filters to HBase for efficient scans of the data. Given the popularity of Phoenix data in HBase, we support Phoenix data as an input to HBase in addition to Avro data types. Also, defaulting to the simple native binary encoding seems susceptible to future changes and is risky for users who write data from SHC into HBase. For example, with SHC evolving, backwards compatibility needs to be properly handled. By default, SHC supports native Phoenix type. This means SHC can write data in Phoenix type or read existing data written by Phoenix. This enables SHC to work with Apache Phoenix data, and helps users to work with exist Phoenix tables in HBase directly.

C. Programming APIs

SHC supports full-fledged Spark SQL and DataFrame API. Meanwhile, SHC is a plug-in implementation for Spark to access HBase as an external data source. This means SHC can work with any programing language that is supported by Spark, e.g., Scala, Java, and Python. We illustrate how to read written data and execute OLAP queries via Dataframe and SQL API in Code 3 and Code 4.

Dataframe API

At first, function `read` ingests saved HBase data as a DataFrame, where the input options are added for the underlying data source, e.g., `HBaseTableCatalog.tableCatalog` as `avroCatalog`. Therefore, function `withCatalog` fetches underlying saved HBase records as a Dataframe, and the corresponding APIs for Spark Dataframe are invoked. Finally, users select certain columns from the Dataframe for the underlying HBase data.

SQL

Further more, users can manipulate an HBase DataFrame from Spark SQL. For example, users can query data based on SQL in Code 4. Function `createOrReplaceTempView` registers DataFrame `df` as a temporary table with name `avrotable`. `sqlContext.sql` function allows one to execute SQL queries over the registered tables.

```
df.createOrReplaceTempView("avrotable")
val c = sqlContext.sql("select count(1)
    from avrotable")
```

Code 4: SQL query in SHC

```
val df_time = sqlContext.read
    .options(Map(HBaseTableCatalog.
    tableCatalog -> writeCatalog,
    HBaseSparkConf.TIMESTAMP ->
        tsSpecified.toString))
    .format("org.apache.spark.sql.
            execution.datasources.hbase")
    .load()

val df_range = sqlContext.read.options(Map(
HBaseTableCatalog.tableCatalog ->
    writeCatalog,
    HBaseSparkConf.MIN_TIMESTAMP -> "0",
    HBaseSparkConf.MAX_TIMESTAMP ->
    oldMs.toString))
.format("org.apache.spark.sql.execution.
        datasources.hbase")
.load()
```

Code 5: Query based on timestamps in SHC

Query with Different Timestamps of HBase

HBase is a key-value store with data associated with timestamps and versions. Therefore, it is necessary to support query data based on the timestamp and version of saved record in SHC. In this work, four parameters related to timestamps could be set in the query context, and SHC would only retrieve qualified data efficiently. The four parameters are “TIMESTAMP”, “MIN_TIMESTAMP”, “MAX_TIMESTAMP”, and “MAX_VERSIONS” respectively. Users can query records with different timestamps or time ranges with “MIN_TIMESTAMP” and “MAX_TIMESTAMP”. In the meantime, use can use concrete values instead of `tsSpecified` and `oldMs` in Code 5. The first DataFrame `df_time` retrieves data whose timestamp is `tsSpecified`, while DataFrame `df_range` gets data whose time is from `0` to `oldMs`.

V. System Architecture

In this section, we describe the components of SHC, and the multiple layers of aggregation that allow it to scale across data centers. SHC consists of a storage layer, a data processing layer, and a query caching layer. We choose HBase as the default data store. Details of HBase (e.g., Region Server, data consistency policy, and fault tolerance) are not covered here. We mainly introduce the data processing and caching layers.

A. Data Processing Layer

We at first assume Spark and HBase are deployed in the same cluster, where Spark runs along with HBase. From Figure 1, Spark SQL runs on top of HBase. Spark executors are co-located with Region Servers of HBase. In this way, a Spark executor is able to fetch data from HBase based on data locality. When a Spark executor is co-located with an HBase Region Server, data locality is achieved by identifying the Region Server's location. Spark makes best effort to co-locate the task with the Region Server. Each executor performs `Scan/BulkGet` on the part of the data co-located on the same host. `Scan` and `Get` are the two means offered by HBase to read HBase data. `BulkGet` is a batch operation of `Get`.

HBaseTableScanRDD. In this work, we propose `HBaseTableScanRDD` to scan the underlying HBase data, and it is a standard RDD. We re-implement `getPartitions`, `getPreferredLocations` and `compute` functions. With this customized RDD, most of critical techniques introduced in this work can be applied, e.g., partition pruning, column pruning, predicate pushdown, and data locality. Details are presented in the next section.

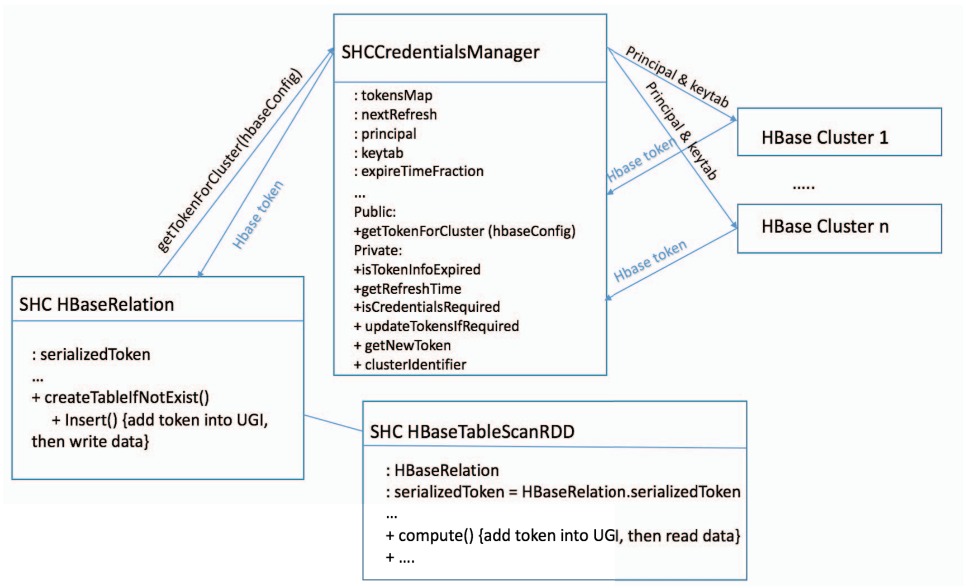


Fig. 2: Illustration of SHC's credential manager

The design makes the maintenance very easy, while achieving a good tradeoff between performance and simplicity.

B. Caching Layer

SHC's caching layer consists of two major components, namely connection caching and security token (a.k.a credential) management, respectively.

1) Connection caching: When accessing HBase data, multiple connections are created to inquire metadata information from an HBase Region Server. Specifically, the call to `ConnectionFactory.createConnection` is invoked each time when visiting HBase tables and regions. These frequent connections would bring expensive network communication overhead, since we observe ZooKeeper connections are being established for each request from the executor logs. Note that the connection creation is a heavy-weight operation and connection implementations have to be thread-safe. Hence, for long-lived processes, it would be very useful to keep a pool to cache connections. With this feature, we decrease the number of connections created drastically, and greatly improve its performance in the process.

More specifically, SHC's in-memory cache contains connection objects, its corresponding reference count, and timestamp when the reference count becomes zero. We fill the cache on demand and evict items using a lazy deletion policy. There is a housekeeping thread performing the lazy deletion policy. It periodically checks if the reference count of a connection object in the cache becomes zero, and the duration of being zero has exceeded the value of the configurable property `SparkHBaseConf.connectionCloseDelay` (10 mins by default). If so, the connection object will be closed and evicted from the cache. Servers understand the connection contents information (e.g., the unique key to an HBase connection instance), and use them to decide which one should be reused.

2) Credential management for secure service: In Hadoop, when security (also called "secure mode") is enabled, a combination of Kerberos and Delegation Tokens is used for authentication between back-end components. Kerberos [33] is a mutually distributed authentication mechanism facilitating authentication between users and services or between services. It serves as a gatekeeper for access to services. Kerberos authentication requires Kerberos credentials from the participating parties and for clients that don't have Kerberos credentials delegation token authentication is used. Once the authentication is successful, the requests are processed by the service. The application platforms must interact with Kerberos secured back-end components in secure mode. They also need to support delegation tokens where Kerberos authentication is not possible.

Spark on YARN [21] relies on Hadoop as an execution environment (i.e., Yarn), as well as the provided permanent storage (i.e., HDFS). It authenticates with Hadoop components via the Kerberos protocol, and generates a Hadoop-specific authentication secret, known as the delegation token. This is utilized to guarantee that when Kerberos credentials expire, the job can continue to communicate with other components. Therefore, Spark on YARN [21] relies on tokens to obtain access to the protected Hadoop services - HDFS, HBase, Hive, etc. At the launch time, Spark obtains tokens from all relevant services - and propagates them to ensure access to the required resources. For long running jobs, like streaming applications, where the duration can be longer than the expire time for a token, Spark supports periodic renewal of tokens using principal and keytab.

The existing Spark design fail to talk to a new secure service dynamically. For example, after the application has been launched, it cannot access the new secure service. As introduced before, users, who have access to a Hive cluster, hope to visit multiple HBase clusters to join HBase data. However, the existing environment runs into two major issues.

Issues: (1) *Static token acquisition* - Spark acquires tokens in a static manner from the default service addresses defined in the HBase/Hive configurations in the classpath. For HDFS, there is a separate configuration that lists NameNodes from which to acquire tokens. This prevents acquisition of new tokens from newly discovered services. SPARK-14743 makes current hard-coded token acquisition into a pluggable mechanism with the current implementations as plugins. Thus now we can plugin a new Custom Token Manager (like `SHCCredentialsManager` introduced later) that can be used to acquire tokens on the fly and also refresh them periodically. (2) The token management code is hard-coded in Spark core, and the query meta data is understood by the DataSource API and code. There is no clear/clean API for the DataSource layer to communicate with Spark core token layer about either new tokens or new source for token acquisition.

In this work, we build a token management layer inside SHC, and it provides fast token delivery in the security environment setup. The key part of this layer is a newly developed credentials manager namely `SHCCredentialsManager` and is illustrated in Figure 2. `SHCCredentialsManager` is in charge of token fetching, renewing and updating in the SHC. Therefore, `SHCCredentialsManager` exposes an API interface called `getTokenForCluster` for a corresponding client to obtain associated HBase tokens. Note that the input of function `getTokenForCluster`, `hbaseConf: Configuration`, and `hbaseConf` is used to identify one HBase cluster.

In secure HBase clusters, before any read and write operation on HBase tables, SHC at first asks `SHCCredentialsManager` for a valid token, and adds it to the current `UserGroupInformation`. `UserGroupInformation` is a concept in Hadoop and corresponding ecosystem security, which is beyond our discussion here.

`SHCCredentialsManager` maintains a token cacher for coming requests. Therefore, we build a hash map to maintain each HBase cluster and corresponding token. When SHC asks for a token of an HBase cluster with corresponding HBase configuration (i.e., `hbaseConf`), `SHCCredentialsManager` checks the token cache first. If this token exists, `SHCCredentialsManager` will return the token from the token cache directly. Otherwise, `SHCCredentialsManager` fetches a new token from the corresponding HBase cluster by using related principal and keytab from the request. As introduced before, the principal and keytab are used to identify each user and associated authorization to access a certain server. Thus, HBase returns the corresponding token to the cache manger based on the given principal and keytab. Then, the cache manager saves this token to cache. In addition, `SHCCredentialsManager` has a token update executor.

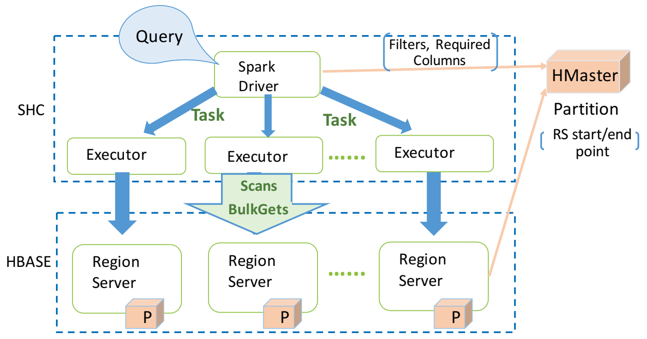


Fig. 3: System framework of SHC

It periodically refreshes cached tokens. If a token is going to expire, `SHCCredentialsManager` would renew it. Meanwhile, the token update policy is configurable. For example, `expireTimeFraction`, `refreshTimeFraction`, and `refreshDurationMins`, decide the frequency of refreshing, updating, and the best time for token renewal. Finally, `SHCCredentialsManager` also supports token serialization and deserialization when propagating tokens. Figure 2 illustrates the whole procedure.

```
spark.hbase.connector.security.
credentials.enabled = true
spark.yarn.principal =
    ambari-qa-c1@EXAMPLE.COM
spark.yarn.keytab =
    smokeuser.headless.keytab
```

Code 6: Configure credential manager in SHC

Usage of Credential Manager in SHC

SHC supports a single secure HBase cluster as well as multiple secure HBase clusters. This function is disabled by default, but users can set texttspark.hbase.connector.security.credentials.enabled" to true to enable it. Besides, users should configure related principal and keytab as Code 6 before running their applications.

VI. System Implementation

Previous sections describe how SHC servers are aggregated to handle large volumes of data and multiple users. This section discusses important optimizations for performance, and fault tolerance mechanism.

A. *Query Optimization*

Query workfollow.

As Figure 3 shows, when a query arrives at SHC, the driver of Spark at first compiles the query, and figures out the `Filters` and `Required Columns` from Spark Catalyst engine. Based on the region's metadata such as region's start point and end point, we can construct the data partitions for correspond RDD. In Spark, one task corresponds to one partition. SHC makes one partition corresponds to one Region Server. The driver aggregates `Scans` and `Gets` based on the region's metadata, and generates one task per Region Server.

If a Region Server does not hold the required data, no task is assigned to it. Actually this is the partition pruning. Note that the driver is not involved in the actual job execution except for scheduling tasks. This avoids the driver from being the bottleneck. Then, the specific tasks are sent to the preferred executors. The executors will perform the tasks on the local Region Server. This means tasks are performed in parallel on the executors to achieve better data locality and concurrency. At a high-level, SHC treats both `Scan` and `Get` in a similar way, and both actions are performed on the executors. A task may consist of multiple `Scans/bulkGets`, and the required data in a task is retrieved from only one Region Server. This Region Server will be the preferred location for the task.

*1) **Partition and Column Pruning**:* By extracting the row key from the predicates, we split the `Scan/BulkGet` into multiple non-overlapping ranges. Only the Region Servers that have the requested data will perform `Scan/BulkGet`. This means those non-overlapping ranges can fall into the start point and end point of the Region Servers. Currently, the partition pruning is performed on the first dimension of the row keys. For example, if a row key is a composite key `"key1:key2:key3"`, the partition pruning will be based on the first dimension `"key1"` only. It is very complicated to execute partition pruning on all dimensions of composite keys. SHC will implement this optimization in future releases. Note that the `WHERE` conditions need to be defined carefully. Otherwise, the partition pruning may not take effect. For example, `WHERE rowkey1 > "abc" OR column = "xyz"` (where `rowkey1` is the first dimension of the row key, and column is a regular HBase column) will result in a full scan, as we have to cover all the ranges because of the `OR` semantic.

In addition, based on the predefined catalog for mapped relational tables, SHC only retrieves required columns from Region Servers. This reduces network overhead and avoids redundant processing in Spark.

*2) **Data Locality**:* Data locality is the process of moving the computation close to where the actual data resides on the node, instead of moving large data to computation. This minimizes network congestion and increases the overall throughput of the system. In terms of HBase data store, the data locality would be specially tailored. In this work, we move computation tasks to associated Region Servers which hold the required data as close as possible.

SHC can get the host of the Region Server from the HBase meta data, then it tells Spark task scheduler to place tasks close to the Region Server. Thus, we just need to set the preferred location to the host of the Region Server. To set the preferred location of each partition, `HBaseTableScanRDD` in SHC overrides `getPreferredLocations(partion: HBaseScanPartition)` function by returning the Region Server's hostname. Note that each RDD partition constructed by SHC contains the meta data of the corresponding Region Server.

*3) **Selective Predicate (Filter) Pushdown**:* Predicate push-down is a well-known technique to improve query processing performance in RDBMS [26]. Spark SQL Catalyst generates a complete sequence of Catalyst expression trees to use in predicate pushdown.

Taking advantage of Spark SQL catalyst, SHC implements the provided data source APIs of Spark SQL. When a Spark application runs with SHC, filters can be performed twice (in two layers). The first layer is accomplished by SHC by pushing the filters down to HBase for execution. The second layer is done by Spark. After SHC fetches the results from HBase into Spark Catalyst engine, Spark will apply those filters again once it gets the fetched data from HBase for guaranteeing correctness. If we does not want to perform the repeated filtering, we have to notify Spark explicitly (refer to SHC's `unhandledFilters` API for notification). Pushing down predicates to HBase makes SHC only fetch part of original data from HBase table into Spark, which reduces network overhead.

To improve performance, a new rule-based optimization mechanism is applied here, i.e., certain predicates that HBase has a good strategy to optimize are pushed down to HBase for execution. The rest of filters are invoked by Spark itself for the sake of complete data. Take *"NOT IN"* predicate as an example. SHC does not push down the *"NOT IN"* predicate to HBase for execution, but it will be executed by Spark in the second filtering layer. Take *"SELECT * FROM tableA WHERE x NOT IN (a, b, c)"* as a concrete example. SHC asks HBase to return all records from *tableA* directly instead of performing any predicate to filter out *"a", "b", "c"*. Scanning the whole table to check if any column cell equals to *"a"/"b"/"c"* is expensive in HBase, especially when *tableA* is huge. Filtering out a few points *"a", "b", "c"* is not worth spending much effort on building heavy HBase "Not IN" filter. Filter "pushdown" is the general approach used by most database systems for filtering the source of the data as much as possible, based on the assumption that dealing with less data will almost always be faster. That isn't always true, especially if the filters are expensive to evaluate.

Meanwhile, using Spark's *UnhandledFilters* API is an effective optimization. This API tells Spark about filters which have not been implemented in SHC as opposed to returning all the filters. Without implementing this API, Spark will re-apply all the filters once data is returned from HBase. This guarantees we can get correct results even if HBase is not guaranteed to return the accurate results. Sometimes Re-applying all the filters again in Spark doesn't change any data, but it can be expensive if the filters are complicated.

*4) **Operators Fusion**:* One relational query usually involves multiple predicates as shown in Code 7.

```
select * from users where Users.a > x and
    Users.a < y and Users.b = x.
```

Code 7: SQL query with multiple predicates

This query includes a range scanning and getting operation, e.g., Users.b=x. At first, both operations are pushed down into HBase Region Servers, and are converted to HBase's internal `Scan` and `Get` calls. The qualified data tuples are returned as `Iterator[Row]` for processing in Spark SQL catalyst. Without optimization, either `Scan` or `Get` operation is related to one task. Thus, the number of tasks will be more than the number of the Region Servers. We need to send some tasks to other executors, or wait for the completion of the first batch of tasks, the second batch of tasks, and so on. This would bring extensive wait time for idle tasks. Therefore, multiple `Scan` and `Get` operations are packed into one task, and the task is sent to the HBase Region Server.

*5) **Range Scan in HBase Byte Array**:* When multiple range search predicates (e.g., $t \in \{[a,b] \cap [c,d]\}$ or $t \in \{[a,b] \cup [c,d]\}$) are pushed down to HBase Region Server for data scanning, SHC at first converts the associated range predicates into HBase byte array types. Then, multiple predicates are merged. Take range search $t \in \{[a,b] \cap [c,d]\}$ as an example. If the range predicates satisfy $(c < b) \cap (a < c)$, then the corresponding range search predicate is merged to $t \in \{[c,b]\}$. Similarly, $t \in \{[a,b] \cup [c,d]\}$ is converted to $t \in \{[a,d]\}$. In our implementation, binary search is used to merge the lower bound and upper bound of the range search. This saves the predicate merging cost when there is a large number of predicates.

B. Fault Tolerance

SHC's fault tolerance mechanism extends those of Spark and Spark SQL. For the fault tolerance of the master node (or the driver program), SHC adopts the following mechanism. A Spark cluster can have multiple masters managed by ZooKeeper [22]. One is elected as the "leader", and the others remain in standby mode. If the current leader fails, another master will be elected. The new leader recovers the old master's state, and continues scheduling. Such a mechanism would recover all system states and the global indexes that reside in the driver program. Thus, SHC is ensured to survive after a master failure. In addition, note that all user queries (including spatial operators) will be scheduled as RDD transformations and actions in Spark. Therefore, fault tolerance for query processing is naturally guaranteed by the fault tolerance mechanism of the underlying lineage graph provided in Spark kernel and consensus model of ZooKeeper.

VII. Implementation and Experimental Results

Open source community

SHC is open sourced and extensively used in multiple leading tech companies and research groups, e.g., Paypal, Walmart, Microsoft Azure, etc. The latest version is updated to support Spark 2.2[*]. The community of SHC is very active, and we have accepted more than one hundred push requests and issues from the community. We have introduced this system in multiple talks, e.g., Spark Summit 2017 San Francisco, Spark Summit 2017 Euro, HBaseCon West 2017, and it attracts multiple companies to adopt our system as their platform to access HBase data for OLAP query processing.

Currently, SHC is hosted under the Hortonworks repo, and published as a Spark package. It is in the process of being migrated to Apache HBase trunk. During the migration, we identified some critical bugs in the HBase trunk, and they will be fixed along with the merge. The community work is tracked by the umbrella HBase JIRA HBASE-14789, including HBASE-14795 and HBASE-14796 to optimize the underlying computing architecture for `Scan` and `BulkGet`, HBASE-14801 to provide JSON user interface for ease of use, HBASE-15336 for the DataFrame writing path, HBASE-15334 for Avro support, HBASE-15333 to support Java primitive types, such as `short`, `int`, `long`, `float`, and `double`, etc., HBASE-15335 to support composite row key, and HBASE-15572 to add optional timestamp semantics. We look forward to producing a future version of the connector which makes the connector even easier to work with.

Experimental setup

All experiments are conducted on a cluster consisting of 5 nodes with following configurations, 5 machines with a 6-core Intel Xeon E5-2603 v3 1.60GHz processor and 60 GB RAM; each node is connected to a Gigabit Ethernet switch and runs Ubuntu 14.04.2 LTS with Hortonworks HDP 2.6.1 and Spark 2.2. The Spark cluster is deployed on Yarn. Our cluster configuration reserved up to 16 GB of main memory for Spark. The test cluster is managed via Ambari 2.6.1. We used TPC-DS to test the performance. TPC Benchmark DS (TPC-DS) provides a representative evaluation of performance as a general purpose decision support system [20]. For each query table in TPC-DS, we at first create the related catalog as introduced in Section IV, then execute associated TPC-DS queries. For example, TPC-DS query q39a joins four tables (i.e., warehouse, item, inventory, and date_dim). We create catalogs for each table, and write related data into HBase. Finally, the corresponding SQL query is executed. One of sample test code is available at github [†]. In this work, we mainly present the results via two represented queries of TPC-DS query, i.e., q39a and q39b, in the rest of paper.

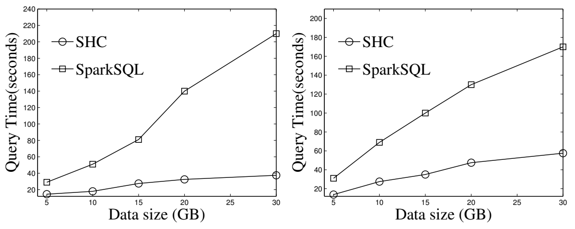
We mainly compare the performance of SHC and Spark SQL. We do not compare SHC with MapReduce2 [7], Hive [10], and Phoenix [14], since previous studies have already shown that Spark SQL outperforms the other three systems in OLAP query processing in terms of most OLAP jobs [30].

A. Comparison with Spark SQL

1) Query latency: Figure 4 measures system query latency as data size increases from 5 GB to 30 GB. SHC achieves one order of magnitude better performance on both types of queries. Spark SQL's performance drops significantly as the data size grows, since it requires scanning the whole table without predicate pushdown and data partition pruning into HBase. In contrast, SHC's performance grows smoothly as the data size increases, since SHC is able to narrow the input table down quickly to just a few data partitions. Meanwhile, the

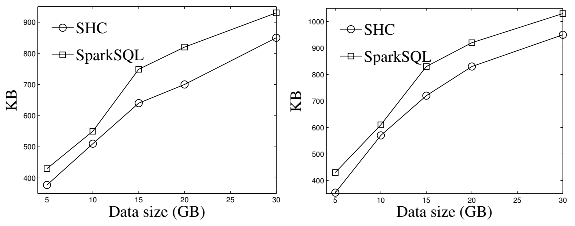
[*]https://github.com/hortonworks-spark/shc

[†]https://github.com/weiqingy/shc/blob/paper/examples/src/main/scala/org/apache/spark/sql/execution/datasources/hbase/TCPDSQ39aTestWithSHC.scala



(a) TPC-DS q39a (b) TPC-DS q39b

Fig. 4: Evaluation of query performance



(a) TPC-DS q39a (b) TPC-DS q39b

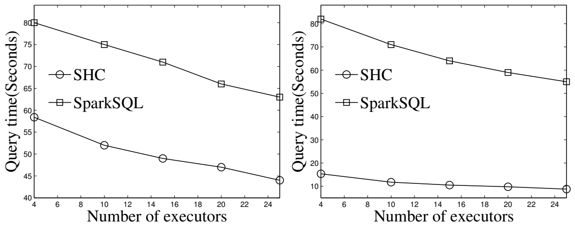
Fig. 5: Shuffle cost

other factor is the caching layer of SHC reduces the redundant connections to HBase Region Server as well.

2) Data shuffle cost: Next, we study how the increase of data size impacts data shuffle cost. Figure 5 presents the related data shuffle cost on two represented queries. We find SHC prunes more data while joining multiple tables, since SHC is able to push down the predicates into HBase data source. As introduced before, this is not optimized in Spark SQL. Therefore, SHC indeed filters out more data and improves the data shuffle cost compared to Spark SQL.

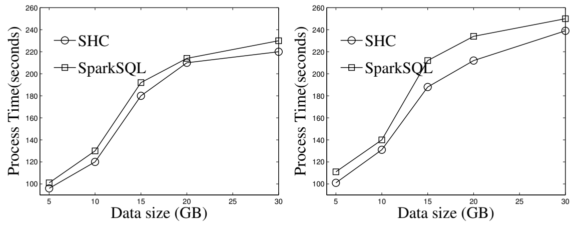
3) Effect of number of executors: Executors are worker nodes' processes in charge of running individual tasks in a given Spark job. We analyze how SHC performs by varying the number of executors. Figure 6 gives the query processing time as the number of executors increases. We observe either SHC's or Spark SQL's runtime decreases as the number of executors increases. However, such speedups slow down as the number of executors reaches a certain number. This is attributed to the parallelism of the system depends on the available resources. For Spark job running over Yarn, resource management of Yarn controls the number of executors to be used, and the allocated resource is limited for each job. This is the runtime upper bound of one Spark job.

4) Write throughput: Figure 7 shows the performance of data write operation in Spark SQL and SHC. We assume the



(a) TPC-DS q39a (b) TPC-DS q39b

Fig. 6: Effect of executor number



(a) TPC-DS q39a (b) TPC-DS q38

Fig. 7: Evaluation of write performance

data is already stored in HDFS (e.g., Hive table). The corresponding data is inserted into an HBase table. We evaluate the time to store data in corresponding queries. For example, four tables of TPC-DS query q39a, i.e., warehouse, item, inventory, and date_dim are stored in HBase for performance evaluation. From Figure 7, SHC outperforms Spark SQL more than 20% on data write, since the data encoding policy of SHC is more efficient than Spark SQL. Meanwhile, as the data size increases, the performance of Spark SQL becomes closer to that of SHC. This is because the data size becomes so large that there are less optimization opportunities for SHC's optimizer to improve the performance for write operation.

5) Effect of different data encodings: We study how these systems perform on different data encoding and decoding policies, e.g., Avro, Phoenix, and native primitive types. Table II gives performance (i.e., write, query, and memory usage) on different data encoding types. Note Spark SQL does not support Phoenix and Avro type for HBase tables internally. From the experimental studies, we find that Java primitive types save more time for query processing and data written than the other two types. Meanwhile, it also takes less memory than the others. In addition, the Phoenix type is competitive with Avro type, since both need to record the structure information of input tables.

System	Types	Query time(s)	Write time(s)	Memory usage(MB)
SHC	Native	16	220	1585
	Phoenix	21	231	1773
	Avro	99	241	1790
Spark SQL	Native	71	240	1632
	Phoenix	×	×	×
	Avro	×	×	×

TABLE II: Performance on different encoding types

VIII. Conclusions

Overall, this work presented a system to improve OLAP query processing for distributed key value storage. SHC provides a data model to operate non-relational data storage, and SHC improves query processing and demonstrate SHC's usage in real applications. Empirical studies on various real datasets demonstrate the superiority of our approaches compared with existing systems.

Future works: All the optimizations about the composite key are on its first dimension for now, and we are working on making those optimizations on all the dimensions. Also, SHC will add more data coders to support more data types.

Acknowledgment

We want to thank all SHC contributors in this work, especially Zhan Zhang, who started this project, and also thank all SHC users in helping us validate this work. We also want to thank Apache HBase and Spark community for providing their feedback and making this better. Finally, this work has leveraged the lessons from earlier Spark HBase integrations and we want to thank their developers for paving the road. Both Mingjie Tang and Yongyang Yu are the corresponding authors.

References

[1] "Asterix," http://asterix.ics.uci.edu/.
[2] "Avro," https://avro.apache.org.
[3] "Bigsql," https://www.ibm.com/us-en/marketplace/big-sql.
[4] "Cassandra," http://cassandra.apache.org/.
[5] "Dremel," https://dremel.apache.org/.
[6] "Flink," http://flink.apache.org.
[7] "Hadoop," http://hadoop.apache.org/.
[8] "Hbase," https://hbase.apache.org/.
[9] "Hdfs," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[10] "Hive," https://hive.apache.org/.
[11] "Impala," https://impala.incubator.apache.org/.
[12] "Mllib," http://spark.apache.org/mllib/.
[13] "Parquet," https://parquet.apache.org.
[14] "Phoenix," https://phoneix.apache.org/.
[15] "Pig," https://pig.apache.org/.
[16] "Presto," https://prestodb.io/.
[17] "Redis," https://redis.io/.
[18] "Spark-llap," https://github.com/hortonworks-spark/spark-llap.
[19] "Spark-sql-on-hbase," https://github.com/Huawei-Spark/Spark-SQL-on-HBase.
[20] "Tpc-ds," http://www.tpc.org/tpcds/.
[21] "Yarn," https://yarn.apache.org.
[22] "Zookeeper," https://zookeeper.apache.org.
[23] W. G. Aref and I. F. Ilyas, "Sp-gist: An extensible database index for supporting space partitioning trees," *Journal of Intelligent Information Systems*, vol. 17, no. 2, pp. 215–240, Dec 2001.
[24] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, 2015, pp. 1383–1394.
[25] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.
[26] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98. New York, NY, USA: ACM, 1998, pp. 34–43.
[27] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. ACM, 2007, pp. 205–220.
[28] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *PVLDB*, vol. 7, no. 12, pp. 1295–1306, 2014.
[29] J. M. Hellerstein, *Generalized Search Tree*. Boston, MA: Springer US, 2009, pp. 1222–1224.
[30] T. Ivanov and M. Beer, "Evaluating hive and spark SQL with bigbench," *CoRR*, vol. abs/1512.08417, 2015.
[31] K. Mershad, Q. M. Malluhi, M. Ouzzani, M. Tang, M. Gribskov, and W. Aref, "Audit: Approving and tracking updates with dependencies in collaborative databases," 09 2017.
[32] K. Mershad, Q. M. Malluhi, M. Ouzzani, M. Tang, M. Gribskov, W. Aref, and D. Prakash, "Coact: A query interface language for collaborative databases," 11 2017.
[33] B. C. Neuman and T. Ts'o, "Kerberos: An authentication service for computer networks," *Comm. Mag.*, vol. 32, no. 9, pp. 33–38, Sep. 1994.
[34] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *PVLDB*, vol. 9, no. 13, pp. 1565–1568, 2016.
[35] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. ACM, 2016, pp. 1071–1085.
[36] L. Yu, Y. Shao, and B. Cui, "Exploiting matrix dependency for efficient distributed matrix computation," in *SIGMOD'15*. New York, NY, USA: ACM, 2015, pp. 93–105.
[37] Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani, "In-memory distributed matrix computation processing and optimization," in *ICDE'17, San Diego, CA, USA, April 19-22*, 2017, pp. 1047–1058.
[38] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.
[39] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. New York, NY, USA: Association for Computing Machinery and Morgan, 2016.
[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*. San Jose, CA: USENIX, 2012, pp. 15–28.