SQL: From Traditional Databases to Big Data

Yasin N. Silva
Arizona State University
ysilva@asu.edu

Isadora Almeida
Arizona State University
isilvaal@asu.edu

Michell Queiroz
Arizona State University
mfelippe@asu.edu

ABSTRACT
The Structured Query Language (SQL) is the main programing language designed to manage data stored in database systems. While SQL was initially used only with relational database management systems (RDBMS), its use has been significantly extended with the advent of new types of database systems. Specifically, SQL has been found to be a powerful query language in highly distributed and scalable systems that process Big Data, i.e., datasets with high volume, velocity and variety. While traditional relational databases represent now only a small fraction of the database systems landscape, most database courses that cover SQL consider only the use of SQL in the context of traditional relational systems. In this paper, we propose teaching SQL as a general language that can be used in a broad range of database systems from traditional RDBMSs to Big Data systems. This paper presents well-structured guidelines to introduce SQL in the context of new types of database systems including MapReduce, NoSQL and NewSQL. A key contribution of this paper is the description of an array of course resources, e.g., virtual machines, sample projects, and in-class exercises, to enable a hands-on experience with SQL across a broad set of modern database systems.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education

General Terms
Design, Experimentation

Keywords
Databases curricula; SQL; structured query language; Big Data

1. INTRODUCTION
The Structured Query Language (SQL) is the most extensively used database language. SQL is composed of a data definition language (DDL), which allows the specification of database schemas; a data manipulation language (DML), which supports operations to retrieve, store, modify and delete data; and a data control language (DCL), which enables database administrators to configure security access to databases. Among the most important reasons for SQL's wide adoption are that: (1) it is primarily a declarative language, that is, it specifies the logic of a program (*what* needs to be done) instead of the control flow (*how* to do it); (2) it is relatively simple to learn and understand because it is declarative and uses English statements; (3) it is a standard of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO); and (4) it is, to some extent, portable among different database systems. Even though SQL was initially proposed for traditional relational databases, it has also been found to be an effective language in several new types of database systems, particularly Big Data management systems (BDMSs) that process large, fast and diverse data. While BDMSs have significantly changed the database landscape and traditional RDBMs represent now only a small fraction of it, most books and database courses only present SQL in the context of RDBMs.

In this paper we propose learning SQL as a powerful language that can be used to query a wide range of database systems, from traditional relational databases to modern Big Data systems. The main contributions of this paper are:

- The description of the new database landscape and the identification of broad types of Big Data systems where SQL can be effectively used.
- The presentation of well-structured guidelines to prepare course units that focus on the study of SQL on new types of database systems including MapReduce, NoSQL and NewSQL.
- A detailed description of class resources for each unit to enable a hands-on experience with SQL across a broad set of modern database systems. These resources include virtual machines, data generators, sample programs, projects, and in-class exercises.
- Files of all the resources made available to instructors [1]. The goal is to enable instructors to use and extend these resources based on their specific needs.

The rest of the paper is organized as follows. Section 2 describes the new database landscape and the types of BDMSs where SQL can be used. Sections 3, 4, and 5 present the details of how SQL can be introduced in the context of the new types of database systems (MapReduce, NoSQL, and NewSQL). These sections describe many class resources which will be made available in [1]. Section 6 presents a discussion of the integration of the proposed units into database courses. Section 7 concludes the paper.

2. NEW DATABASE LANDSCAPE
The development and widespread use of highly distributed and scalable systems to process Big Data is considered as one of the recent key technological developments in computer science [3, 4, 5, 6]. These systems have significantly extended the database landscape and are currently used in many application scenarios, e.g., social media data mining, scientific data analysis, recommendation systems, and web service analysis. BDMSs are usually composed of clusters of commodity machines and are often dynamically scalable, i.e., nodes can be added or removed as needed. The area of BDMSs has quickly developed and there are now dozens of BDMS-related open-source and commercial products. Some of these systems have proposed their own query languages or application program interfaces but several others have recognized the benefits of using SQL and support it as a query language. The use of SQL in DBMSs is expanding and some top

database researchers believe that many more systems will move to support SQL [2]. The authors propose that SQL should be taught in the context of the following types of BDMSs in addition to traditional RDBMSs:

- **MapReduce**. It is considered to be one of the main frameworks for Big Data processing. It enables building highly distributed programs that run on failure-tolerant and scalable clusters. Apache Hadoop [7] is the most widely used MapReduce implementation. Examples of systems that support SQL to query data in Hadoop are: Hive [8] and Spark (Spark SQL) [9].
- **NoSQL**. These data stores have been designed to provide higher scalability and availability than conventional relational databases while supporting a simplified transaction and consistency model. Some examples of NoSQL data stores are: MongoDB [10], Apache HBase [11], Google's BigTable [12], and Apache Cassandra [13]. While many NoSQL systems do not support SQL natively, several systems have been proposed to enable SQL querying on these systems. Examples of such systems are: Impala [14], Presto [15] and SlamData [16].
- **NewSQL**. These systems aim to have the same levels of scalability and availability of NoSQL systems while maintaining the ACID properties (Atomicity, Consistency, Isolation and Durability), relational schema, and SQL query language of traditional relational databases. Examples of NewSQL systems are: VoltDB [17], MemSQL [18], NuoDB [19], and Clustrix [20].

3. SQL IN MAPREDUCE SYSTEMS

MapReduce is an extensively used programming framework for processing very large datasets. Apache Hadoop is its most popular implementation. A MapReduce program divides a large dataset into independent chunks that can be processed in a parallel fashion over dynamic computer clusters. The overall processing is divided into two main phases: *map* and *reduce*, and the framework user needs to provide the code for each phase. The *map* and *reduce* functions have the following form:

map: (k1,v1) → list(k2,v2)
reduce: (k2,list(v2)) → list(k3,v3)

The input dataset is usually stored on an underlying distributed file system and different chunks of this dataset are processed in parallel by different *map* tasks. Each *map* task processes an input chunk one record or line at the time. Each *map* function call receives a key-value pair (k1,v1) and generates a list of (k2,v2) pairs. Each generated key-value pair is sent over the network to a reduce node (*shuffle* phase). The framework guarantees that all the intermediate pairs with the same key (k2) will be sent to the same *reduce* node where they will form a single group. Each *reduce* call processes a group (k2,list(v2)) and outputs a list of (k3,v3) pairs, which represent the overall output of the MapReduce job.

While MapReduce is a powerful framework to build highly distributed and scalable programs, it is also complex and difficult to learn. In fact, even simple data operations, like joining two datasets or identifying the top-K records, require relatively complex MapReduce programs. This is the case because MapReduce requires users to build a program using a procedural language that needs a detailed specification of how a processing task should be carried out. Considering this limitation, several systems have been proposed to: (1) enable the use of SQL-like languages on top of MapReduce-based systems, e.g., Apache Hive [8] and Apache Pig [22], and (2) integrate SQL with MapReduce-based computations, e.g., Spark SQL [9].

3.1 Hive: SQL Queries on Hadoop

Apache Hive [8] is a system that supports the processing and analysis of data stored in Hadoop. Hive allows projecting structure onto this data and querying the data using HiveQL, a SQL-like query language. One of the key features of Hive is that it transparently converts queries specified in HiveQL to MapReduce programs. This enables the user to focus on specifying *what* the query should retrieve instead of specifying a procedural MapReduce program in Java. Hive uses indexing structures to accelerate the execution of queries and was particularly built to support data warehouse applications, which require the analysis of primarily read-only massive datasets (data that does not change over time). While HiveQL is Hive's main query language, Hive also allows the use of custom *map* and *reduce* functions when this is a more convenient or efficient way to express a given query logic.

HiveQL supports many of the features of SQL but it does not strictly follow a full SQL standard. Hive supports multiple DDL and DML commands such as CREATE TABLE, SELECT, INSERT, UPDATE and DELETE. Moreover, starting with Hive 0.13, it is possible to support transactions with full ACID semantics at the row (record) level.

We propose the following in-class activity to introduce HiveQL.

Using Virtual Machines. Since many Big Data systems rely on distributed architectures, a computer cluster is needed to enable a direct interaction with these technologies. Many institutions, however, do not have such clusters available for teaching. A solution that the authors recommend is the use of virtual machines (VM) with all the required packages already installed and configured. In the case of Hive, the authors recommend the use of Cloudera's VM [23] which includes: (1) CentOS Linux as the operating system, (2) Hadoop, (3) Hive, and (5) Hue, a web-based application that can be used to write and run HiveQL queries.

Data Generators and Datasets. The queries presented in this section use *MStation2*, a synthetic dataset of meteorological station data prepared by the authors. The dataset has two tables: *Station* (*stationID*, *zipcode*, *latitude*, *longitude*) and *WeatherReport* (*stationID*, *temperature*, *precipitation*, *humidity*, *year*, *month*). The data generator and a sample dataset are available in [1]. The generator can be modified to produce datasets of different sizes and data distributions for additional exercises or projects.

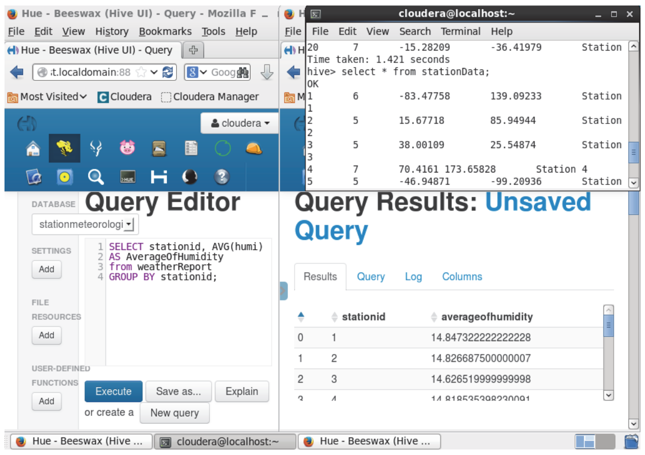
Loading and Querying the Data. The commands for this activity are listed in Fig. 1. To start interacting with Hive, students can open a terminal window using Cloudera's VM and start the Hive console using command C1. Students can then create the database *MStation* (C2) and use the DDL commands C3 and C4 to create tables *stationData* and *weatherReport*, respectively. Students can then load the data generated by the MStation2 data generator into both tables using commands C5 and C6. Both, the Hive console and the Hue application, can be used next to run SQL queries. Students can use the available tables to write queries such as (Q1-Q4 in Fig. 1):

- Q1: For each zipcode, compute the average precipitation level.
- Q2: Group the data by station and output the average humidity level at each station.
- Q3: Identify the minimum and maximum temperatures reported by each station considering years greater than 2000.
- Q4: Compute the tenth highest temperatures ever reported. List the temperature, zip code, month and year.

Fig. 2 shows the result of queries using the Hive console and Hue.

```
C1: sudo hive
C2: CREATE database MStation;
C3: CREATE TABLE stationData (stationed int, zipcode int,
latitude double, longitude double, stationname string);
C4: CREATE TABLE weatherReport (stationid int, temp double,
humi double, precip double, year int, month string);
C5: LOAD DATA LOCAL INPATH '/home/cloudera/datastation/
stationData.txt' into table stationData;
C6: LOAD DATA LOCAL INPATH '/home/cloudera/datastation/
weatherReport.txt' into table weatherReport;
Q1: SELECT S.zipcode, AVG(W.precip) FROM stationData S JOIN
weatherReport W ON S.stationid = W.stationid GROUP BY
S.zipcode;
Q2: SELECT stationid, AVG(humi) FROM weatherReport GROUP BY
stationid;
Q3: SELECT stationid, MIN(temp), MAX(temp) FROM weatherReport
WHERE year > 2000 GROUP BY stationid;
Q4: SELECT S.zipcode, W.temp, W.month, W.year FROM
stationData S JOIN weatherReport W ON S.stationid =
W.stationid ORDER BY W.temp DESC LIMIT 10;
```

Figure 1. Hive Commands and Queries



```
20      7       -15.28209       -36.41979       Station
Time taken: 1.421 seconds
hive> select * from stationData;
OK
1       6       -83.47758       139.09233       Station
1
2       5       15.67718        85.94944        Station
2
3       5       38.00109        25.54874        Station
3
4       7       70.4161 173.65828       Station 4
5       5       -46.94871       -99.20936       Station
```

```
SELECT stationid, AVG(humi)
AS AverageOfHumidity
from weatherReport
GROUP BY stationid;
```

	stationid	averageofhumidity
0	1	14.847322222222228
1	2	14.826687500000007
2	3	14.626519999999998
3	4	14.818535398230091

Figure 2. Executing SQL queries in Hive

3.2 Spark: Integrating SQL and MapReduce

Apache Spark [9] is a highly distributed data processing framework. In contrast to Hadoop's two-stage disk-based MapReduce approach, Spark works by loading the data into a cluster's memory and querying it using a wider set of processing primitives (which also include operations similar to *Map* and *Reduce*). Spark's performance has been reported to be up to one hundred times faster than Hadoop's for some applications [9]. For distributed storage, Spark can work with Hadoop Distributed File System (HDFS), OpenStack, Amazon S3, etc. Spark uses the notion of Resilient Distributed Datasets (RDD). An RDD is an immutable collection of objects that are partitioned and distributed across multiple physical nodes of a cluster, which allows the data to be processed in parallel. Once an RDD has been instantiated, the user can apply two types of operations: transformations and actions. Transformations return new RDDs while actions generate outputs. Spark SQL is one of the key tools supported by Spark. Spark SQL is the Spark module for structured data processing, and uses DataFrames (equivalent to relational database tables) as its main programming abstraction. The data in a DataFrame is organized into named columns. DataFrames can be created from already existing RDDs, Hive tables, or JSON files. Spark SQL supports many of the features of SQL and, therefore, it is a powerful tool to learn SQL while working with big datasets.

```
C1: wget http://real-chart.finance.yahoo.com/table.csv?
s=AAPL&d=6&e=4&f=2015&g=d&a=11&b=12&c=1980&ignore=.csv
C2: mv table.csv?s=AAPL table.csv
C3: hadoop fs -put ./table.csv /data/
C4: spark-shell --master yarn-client --driver-memory 512m --
executor-memory 512m
C5: import org.apache.spark.sql._
C6: val base_data = sc.textFile("hdfs://sandbox.hortonworks.
com:8020/data/table.csv")
C7: val attributes = base_data.first
C8: val data = apple_stocks.filter(_(0) != attributes(0))
C9: case class AppleStockRecord(date: String, open: Float,
high: Float, low: Float, close: Float, volume: Integer,
adjClose: Float)
C10: val applestock = data.map(_.split(",")).map(row =>
AppleStockRecord(row(0), row(1).trim.toFloat,
row(2).trim.toFloat, row(3).trim.toFloat, row(4).trim.toFloat,
row(5).trim.toInt, row(6).trim.toFloat,
row(0).trim.substring(0,4).toInt)).toDF()
C11: applestock.registerTempTable("applestock")
C12: applestock.show
C13: applestock.count
C14: output.map(t => "Record: " + t.toString).collect().
foreach(println)
Q1: val output = sql("SELECT * FROM applestock WHERE close >=
open")
Q2: val output = sql("SELECT MAX(close-open) FROM applestock")
Q3: val output = sql("SELECT date, high FROM applestock ORDER
BY high DESC LIMIT 10")
Q4: val output = sql("SELECT year, AVG(Volume) FROM applestock
WHERE year > 1999 GROUP BY year")
```

Figure 3. Spark Commands and SQL Queries

We describe next an in-class activity with Spark SQL. Hortonworks Sandbox [24] is an excellent virtualization environment to provide students with hands-on experience in Spark SQL. This virtual machine comes with Apache Spark fully installed and enables the execution of commands from a terminal. The dataset to be used in this activity contains the historical prices of the Apple stock (AAPL, 1980-2015) and was obtained from the Yahoo! Finance website [25]. The dataset can be downloaded as a comma-separated values (CSV) file. Each record contains the stock values of a single date and includes the following attributes: *date*, *open price*, *high price* (highest stock value), *low price* (lowest stock value), *close price*, *volume*, and *adjClose* (close price adjusted for dividends and splits). An additional attribute *year* (extracted from date) is added to enable de specification of some interesting queries.

Fig. 3 shows the sequence of commands to load and query the data using the Hortonworks VM. The student downloads first the dataset (C1), renames it to *table.csv* (C2), copies the file into Hadoop's HDFS (C3), and starts the Spark shell (C4). Then, C5 is executed to import a library that is needed to use Spark SQL. Command C6 creates the RDD *base_data* using the dataset stored in the HDFS. C7 and C8 create a new RDD (*data*) which contains all the records in *base_data* with the exception of the record with the headers. C9 creates the class *AppleStockRecord* which specifies the attributes and data types of the Apple stock dataset. C10 parses the records of *data*, creates an instance of *AppleStockRecord* for each record, and populates the RDD *applestock* with the created instances. C11 registers *applestock* as a DataFrame. This DataFrame can now be used as a relational table. C12 and C13 show the top 20 records and the number of records of *applestock*, respectively. Students can now be assigned to write SQL queries such as (Q1-Q4 in Fig. 3):

- Q1: Report the records where the close price is greater or equal than the open price.
- Q2: Identify the record with the largest difference between the close and the open prices.
- Q3: Report the ten records with the highest high prices.
- Q4: Report the average stock volume per year, considering only years greater than 1999.

A key feature of Spark is that the output of SQL queries can be used as the input of MapReduce-like computations. This is, in fact, done in C14 to print the results of the SQL queries.

4. SQL IN NOSQL SYSTEMS

NoSQL (Not only SQL) is a broad class of data stores that aims to provide higher scalability and availability than traditional relational databases. NoSQL systems have the following core properties: (1) they usually use a distributed and fault-tolerant architecture where more computers can be easily added, (2) data is partitioned among different computers, (3) data is replicated to provide fault tolerance, and (4) they often support a simplified transaction/consistency model, for instance *eventual consistency* (given a sufficiently long period of time in which no changes are made, all the previous updates are expected to eventually propagate through the system). There are different types of NoSQL such as: (1) Key-Value stores, e.g., Cassandra and Riak, which store data in a schema-less way using key-value pairs; (2) Document stores, e.g., MongoDB and CouchDB, which store documents in a given format (XML, JSON, binary, etc.); and (3) Tabular data stores, e.g., HBase and BigTable, which store tabular data that can have many attributes and records.

While originally most of the NoSQL systems did not adhere to the relational database model and did not support SQL for data manipulation, one of the key recent developments in this area has been the recognition that a declarative, highly expressive and standardized query language like SQL can be an effective way to query NoSQL systems [2]. This is reflected in a number of systems that enable SQL querying on common NoSQL data stores, for instance Impala [14], SlamData [16] and Presto [15] support SQL queries on HBase, MongoDB, and Cassandra, respectively.

4.1 Impala: SQL to Query HBase Tables

Impala [14] is an open-source query engine that allows running SQL queries on top of HBase tables (Impala also supports HDFS as storage sub-system). HBase [11] is an open-source NoSQL database that runs on top of HDFS and provides a fault-tolerant way of storing and processing large amounts of sparse tabular data. HBase provides efficient random reads and writes on top of HDFS (HDFS does not directly support random writes). Impala has been built leveraging and extending key components of Hive, e.g., SQL syntax, metadata and schema. While both system support SQL, Hive is best suited for long-running batch processing and Impala provides a better support for dynamic analytic queries.

We propose the following in class activity to introduce SQL in Impala. For this activity students can use the Cloudera VM [14] which has Impala, HBase and Hive already installed. The list of commands is presented in Fig. 4. The dataset used in this activity contains information about occupations in the U.S. The dataset can be found in the VM under the *dataset* folder (*sample_007.csv*). It has four attributes: *code* (occupation ID), *description*, *total_emp* (number of employees), and *salary* (combined income). We renamed the data file as *occupations.csv*. Students should first open a terminal window in the VM and start the HBase shell (C1). Next, the HBase table *OccupationsData* is created using command C2.

```
C1: hbase shell
C2: create 'OccupationsData', 'code', 'description',
'total_emp', 'salary'
C3: exit
C4: hive
C5: CREATE EXTERNAL TABLE Occupations (key STRING, description
STRING, total_emp int, salary int) ROW FORMAT DELIMITED FIELDS
TERMINATED BY ',' STORED BY 'org.apache.hadoop.hive.hbase.HBa
seStorageHandler' WITH SERDEPROPERTIES ("hbase.columns.mappin
g" = ":key, description:description,total_emp:total_emp,salary
:salary") TBLPROPERTIES("hbase.table.name"="OccupationsData");
C6: hbase.org.apache.hadoop.hbase.mapreduce.ImportTsv '-
Dimporttsv.separator=,' -Dimporttsv.columns=HBASE_ROW_KEY,
details:code,details:description,details:total_emp,details:sal
ary OccupationsData /home/cloudera/occupations.csv;
C7: impala-shell
C8: invalidate metadata [[default.]Occupations]
Q1: SELECT description, salary FROM Occupations ORDER BY
salary DESC LIMIT 10;
Q2: SELECT description, salary/total_emp AS PerPersonSalary
FROM Occupations ORDER BY PerPersonSalary DESC LIMIT 10;
Q3: SELECT SUM(salary)/SUM(total_emp) AS AvgSalary FROM
occupationsIncomeHBase WHERE description LIKE '%computer%';
Q4: SELECT code, description, salary FROM Occupations WHERE
Salary IN (SELECT MAX(salary) FROM Occupations);
```

Figure 4. Impala Commands and SQL Queries

At this point, students should switch into Hive (C3 and C4), create an external table (*Occupations*) that is linked to *OccupationsData* (C5), and load the data (C6). After this, students can start the Impala shell (C7) where they can write and execute SQL queries. Because Impala and Hive share the same metadata database, once the table is created in Hive, it can be queried in Impala. Command C8 is needed to make Impala aware of the recently created table. Students can now write an execute queries such as (Q1-Q4 in Fig. 4):

Q1: Show the occupations with the 10 highest combined salaries

Q2: Show the occupations with the 10 highest per-person salaries.

Q3: Show the average per-person income of the occupations in the Computer field.

Q4: Show the occupation that has the highest combined income.

4.2 SlamData: SQL on MongoDB

SlamData [16] is a system that allows running SQL statements on MongoDB [10], an extensively used document-oriented and open-source NoSQL database. MongoDB does not natively support SQL or multi-object transactions and works with documents as basic units of data. Documents use a JSON-like structure (BSON) and are equivalent to rows in relational databases. A set of documents forms a collection (table). Each document is composed of field and value pairs similar to JSON objects. The values of fields can be other documents, arrays, and arrays of documents. These embedded documents and arrays reduce the need for joins.

We propose the following in-class exercise to introduce the use of SQL in the context of MongoDB and SlamData. The commands for this exercise using MS Windows are listed in Fig. 5. The dataset is the 2012 U.S. Income Tax Statistics per ZIP code reported by the IRS (CSV file) [26]. We use the following attributes: *STATE*, *ZIPCODE* (99999 is the aggregation of all ZIP codes with less than 100 returns, 0 represents the whole state), *N1* (number of returns, which approximates the number of households), *N2* (number of exemptions, which approximates the population), *A00100* (AGI), and *A00200* (salaries and wages, in thousands of dollars).

The first step is to download [10] and install MongoDB (we installed it in *C:\mongodb*). A required data directory is created in C1. MongoDB is started in C2 and a connection through the mongo.exe shell is stablished in C3. C4 can be used to list the available databases (initially, there should be one called *local*). At this point, SlamData should be downloaded [16] and installed. To run queries, students need to mount a MongoDB cluster in SlamData and host it somewhere. We will do this using the free tier of MongoLab [27]. After creating a MongoLab account, students should create a new MongoDB deployment (select *single-node* and *standard line*) and provide the database name (*project*). After the deployment is created, students can set the database user name and password by clicking on the new deployment. Then, students will import the CSV file into the MongoDB deployment. This can be done using one of the pre-filled commands provided by MongoLab under the Tools/Import-Export tab. The CSV section in this tab provides a command formatted as C5. Students need to run this command on a MS Command Prompt window. Next, students will open the GUI-based SlamData application and mount the MongoDB database. To do this, students will click on the Mount icon in SlamData and use a connection URI (formatted as C6) that can be obtained in the recently created MongoDB deployment in MongoLab. Students can now use the SlamData application to write SQL queries such as (Q1-Q4 in Fig. 5):

- Q1: Identify the number of households in your ZIP code.
- Q2: Estimate the population in each ZIP code of New York.
- Q3: Identify the zip codes with the 10 largest total gross income in the state of NY.
- Q4: Compute the average wages and salaries amount per state.

```
C1: md \data\db

C2: C:\mongodb\bin\mongod.exe

C3: C:\mongodb\bin\mongo.exe

C4: show dbs

C5: C:\mongodb\bin\mongoimport.exe -h <host:port> -d <your
database name> -c <collection> -u <dbuser> -p <dbpassword> --
type csv --file <path to .csv file> –headerline

C6: mongodb://<dbuser>:<dbpassword>@<host:port>/<dbname>

Q1: SELECT N1 FROM "/project/taxes" WHERE ZIPCODE=85304

Q2: SELECT N2, ZIPCODE FROM "/project/taxes" WHERE STATE='NY'

Q3: SELECT ZIPCODE, A00100 FROM "/project/taxes" WHERE
STATE='NY' AND ZIPCODE!=0 ORDER BY (A00100) DESC LIMIT 10

Q4: SELECT DISTINCT STATE, AVG(A00200) FROM "/project/taxes"
WHERE ZIPCODE!=0 AND ZIPCODE!=99999 GROUP BY STATE
```

Figure 5. SlamData Commands and SQL Queries

5. SQL IN NEWSQL SYSTEMS

NewSQL is a class of database systems that not only offers the same levels of scalability and availability of NoSQL systems but also preserves the ACID guarantees, relational data model, and SQL query language of traditional relational databases. NewSQL systems are divided into three groups: (1) New architectures, e.g., VoltDB and NuoDB, these are new systems designed to work on a distributed cluster of shared-nothing nodes that can dynamically scale; (2) Optimized SQL Engines, e.g., MySQL Cluster and Infobright, these systems support the same programming interface of traditional systems like MySQL but have better scalability; and (3) Transparent sharding, e.g., dbShards and ScaleBase, these systems provide a middleware layer to automatically fragment and distribute databases over multiple nodes.

Most NewSQL databases can be used to learn SQL considering the application scenarios commonly used to teach traditional RDBMs, e.g., university, employee, or inventory databases. A better approach, however, is to consider application scenarios that require the capabilities of NewSQL systems, e.g., the need of processing large or fast data in the stock market, social media networks and online games. Moreover, educators should consider applications that take advantage of the features of the NewSQL system used in class, e.g., an application that requires processing a high number of short transactions per second would be a good fit for VoltDB.

5.1 Learning SQL with VoltDB

VoltDB [17] is an in-memory and ACID-compliant NewSQL database system that uses a shared-nothing architecture as well as partitioning and replication mechanisms to achieve high transaction throughputs. The system is open-source and was designed by well-known database researchers. VoltDB aims to give solutions to real-time analytic problems, extracting insights from fast flowing data. VoltDB can be installed on Linux and Mac OS X computers and provides client libraries for Java, Python, PHP, C++, C#, etc.

VoltDB can be used to learn many aspects of SQL using hands-on class activities. For instance it can be used to learn the SQL data definition and data manipulation languages. Using a VM is also an effective way to enable a hands-on interaction with NewSQL systems like VoltDB. In this case, instructors can use the VM available at VoltDB's web site [17]. The VM includes the installed VoltDB software and a set of sample applications and tools for application development. The sample applications are useful resources to learn about VoltDB. They include: (1) a system that simulates a telephone-based voting process, (2) a memcache-like cache implementation using VoltDB, (3) a Key-Value store backed by VoltDB, and (4) a system that uses a flexible schema and JSON.

VoltDB also made available an application gallery [21] that contains many applications that can be used to develop learning activities. We propose the use of the Ad Performance application that can be downloaded from [21] and installed in VoltDB's VM. This application simulates a high velocity stream of ad events (impressions, clickthroughs and conversions) that are augmented and stored in real time. Instructors can use the following in-class activity working with this application. Students first analyze and execute the provided scripts that start the database server, create tables and views, and run a client application that creates the stream of ad events. Fig. 6 shows the statement (C1) that creates the table *event_data,* which stores all the received and processed events. During the execution of the client application, students use the provided web-based interface, shown in Fig. 7, to monitor in real-time the stream of events and the costs and rates associated with ad campaigns. Next, students can use VoltDB's Management Center to write and run SQL queries. Fig. 6 shows several queries prepared by the authors to compute various ad real-time statistics.

- Q1: For each website that shows ads, compute the number of received ad events, impressions, clicks and conversions, and the total ad cost.
- Q2: For each advertiser, get the number of processed ad events, impressions, clicks and conversions, and the total ad cost.
- Q3: For each advertiser, compute the daily number of ad events, impressions, clicks and conversions, and the daily ad cost.
- Q4: Identify the 10 ads with the largest associated cost.

This learning activity also provides a good opportunity to highlight some of the key properties of this type of NewSQL system, e.g., high transaction throughput (transactions/second) and low latency (time to process the events).

```
C1: CREATE TABLE event_data (utc_time TIMESTAMP NOT NULL,
creative_id INTEGER NOT NULL, cost DECIMAL, campaign_id
INTEGER NOT NULL, advertiser_id INTEGER NOT NULL, site_id
INTEGER NOT NULL, is_impression INTEGER NOT NULL,
is_clickthrough INTEGER NOT NULL, is_conversion INTEGER
NOT NULL);
```

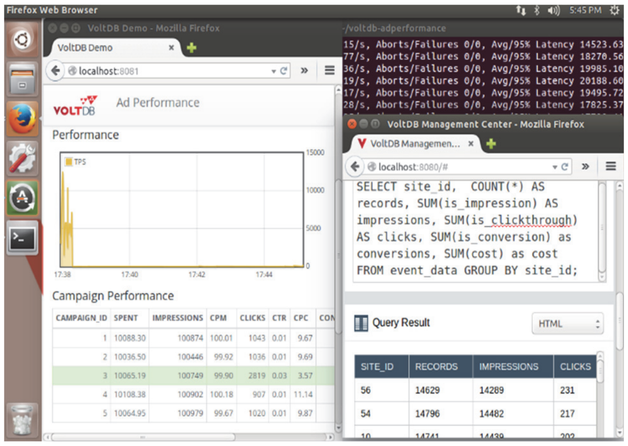
```
Q1: SELECT site_id,  COUNT(*) AS records, SUM(is_impression)
AS impressions, SUM(is_clickthrough) AS clicks,
SUM(is_conversion) AS conversions, SUM(cost) AS cost FROM
event_data GROUP BY site_id;
```

```
Q2: SELECT advertiser_id, COUNT(*) AS records,
SUM(is_impression) AS impressions, SUM(is_clickthrough) AS
clicks, SUM(is_conversion) AS conversions, SUM(cost) AS cost
FROM event_data GROUP BY advertiser_id;
```

```
Q3: SELECT advertiser_id, TRUNCATE(DAY,utc_time) AS utc_day,
COUNT(*) AS records, SUM(is_impression) AS impressions,
SUM(is_clickthrough) AS clicks,  SUM(is_conversion) AS
conversions, SUM(cost) AS spent FROM event_data GROUP BY
advertiser_id, TRUNCATE(DAY,utc_time);
```

```
Q4: SELECT creative_id AS ad, SUM(cost) AS cost FROM
event_data GROUP BY creative_id ORDER BY cost DESC LIMIT 10;
```

Figure 6. VoltDB SQL Queries



```
15/s, Aborts/Failures 0/0, Avg/95% Latency 14523.63
77/s, Aborts/Failures 0/0, Avg/95% Latency 18270.56
36/s, Aborts/Failures 0/0, Avg/95% Latency 19985.10
19/s, Aborts/Failures 0/0, Avg/95% Latency 20188.60
17/s, Aborts/Failures 0/0, Avg/95% Latency 19495.72
28/s, Aborts/Failures 0/0, Avg/95% Latency 17825.37
```

```
SELECT site_id,  COUNT(*) AS
records, SUM(is_impression) AS
impressions, SUM(is_clickthrough)
AS clicks, SUM(is_conversion) as
conversions, SUM(cost) as cost
FROM event_data GROUP BY site_id;
```

Campaign Performance

CAMPAIGN_ID	SPENT	IMPRESSIONS	CPM	CLICKS	CTR	CPC	CON
1	10088.30	100874	100.01	1043	0.01	9.67	
2	10036.50	100446	99.92	1036	0.01	9.69	
3	10065.19	100749	99.90	2819	0.03	3.57	
4	10108.38	100902	100.18	907	0.01	11.14	
5	10064.95	100979	99.67	1020	0.01	9.87	

SITE_ID	RECORDS	IMPRESSIONS	CLICKS
56	14629	14289	231
54	14796	14482	217
10	14741	14439	202

Figure 7. Executing SQL queries in VoltDB

6. DISCUSSION

The proposed modules can be integrated into several introductory and advanced database-related courses. One approach is to adopt Big Data management systems as a replacement of the traditional relational databases currently used in most introductory database courses. We recommend using a NewSQL system, e.g., VoltDB, for this approach since it supports both the relational model and SQL. An alternative approach for introductory database courses is to complement the use of a traditional database with a Big Data system. In this case, instructors can teach database fundamentals using traditional databases and then show how the same or similar concepts can be applied to modern Big Data systems. Another approach it to integrate the study of SQL beyond traditional databases as part of a course in Big Data. This course can study the use of SQL in MapReduce, NoSQL and NewSQL systems, and compare it with other native query languages. The authors have previously integrated the study of NewSQL systems in a Big Data course and are currently integrating the study of SQL in a Big Data system as part of an undergraduate database course. We plan to report the results of these experiences in a future work.

7. CONCLUSIONS

Many application scenarios require processing very large datasets in a highly scalable and distributed fashion and different types of Big Data systems have been designed to address this challenge. Many of these systems have recognized the strengths of SQL as a query language. Most database courses, however, focus solely on traditional relational databases. In this paper we propose that SQL should be taught consider the new and broader database landscape. This exposure will enable students to get better insight from the data on a wider array of systems and applications. This paper presents a set of guidelines and a wide array of class resources to integrate the study of SQL with three core types of Big Data systems: MapReduce, NoSQL, and NewSQL.

8. REFERENCES

[1] ASU. SQL: From Traditional Databases to Big Data - course resources. http://www.public.asu.edu/~ynsilva/iBigData.

[2] Barron's. Michael Stonebraker Explains. http://blogs.barrons.com/techtraderdaily/2015/03/30/michael-stonebraker-describes-oracles-obsolescence-facebooks-enormous-challenge/.

[3] D. Kumar. Data science overtakes computer science? ACM Inroads, 3(3):18–19, Sept. 2012.

[4] A. Cron, H. L. Nguyen, and A. Parameswaran. Big data. XRDS, 19(1):7–8, Sept. 2012.

[5] A. Sattar, T. Lorenzen, and K. Nallamaddi. Incorporating nosql into a database course. ACM Inroads, 4(2):50–53, June 2013.

[6] Y. N. Silva, S. W. Dietrich, J. Reed, L. Tsosie. Integrating Big Data into the Computing Curricula. In ICDE 2015.

[7] Apache. Hadoop. http://hadoop.apache.org/.

[8] Apache. Hive. https://hive.apache.org/.

[9] Apache. Spark SQL. https://spark.apache.org/sql/.

[10] 10gen. Mongodb. http://www.mongodb.org/.

[11] Apache. Hbase. http://hbase.apache.org/.

[12] 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., 26(2):1–26, 2008.

[13] Apache. Cassandra. http://cassandra.apache.org/.

[14] Cloudera. Impala. http://impala.io/.

[15] Presto. https://prestodb.io/.

[16] SlamData. SlamData. http://slamdata.com/.

[17] VoltDB. Voltdb VM. https://voltdb.com/run-voltdb-vmware.

[18] MemSQL. MemSQL. http://www.memsql.com/.

[19] NuoDB. Nuodb. http://www.nuodb.com/.

[20] Clustrix. Clustrix. http://www.clustrix.com/.

[21] VoltDB. Application gallery. http://voltdb.com/community/applications.

[22] Apache. Pig. https://pig.apache.org/.

[23] Cloudera. Cloudera VM 4.7.0. http://www.cloudera.com/content/cloudera/en/downloads.html.

[24] Hortonworks. Hortonworks Sandbox on a VM, HDP 2.3. http://hortonworks.com/products/hortonworks-sandbox.

[25] Yahoo Finance. Historical Prices - AAPL. http://finance.yahoo.com/q/hp?s=AAPL+Historical+Prices.

[26] IRS. Tax Stats. http://www.irs.gov/uac/SOI-Tax-Stats-Individual-Income-Tax-Statistics-2012-ZIP-Code-Data-(SOI).

[27] Mongolab. Mongolab. https://mongolab.com/.