Cloud Computing and Big Data Analytics: What Is New from Databases Perspective?

Rajeev Gupta, Himanshu Gupta, and Mukesh Mohania

IBM Research, India
{grajeev,higupta8,mkmukesh}@in.ibm.com

Abstract. Many industries, such as telecom, health care, retail, pharmaceutical, financial services, etc., generate large amounts of data. Gaining critical business insights by querying and analyzing such massive amounts of data is becoming the need of the hour. The warehouses and solutions built around them are unable to provide reasonable response times in handling expanding data volumes. One can either perform analytics on big volume once in days or one can perform transactions on small amounts of data in seconds. With the new requirements, one needs to ensure the real-time or near real-time response for huge amount of data. In this paper we outline challenges in analyzing big data for both *data at rest* as well as *data in motion*. For big *data at rest* we describe two kinds of systems: (1) NoSQL systems for interactive data serving environments; and (2) systems for large scale analytics based on MapReduce paradigm, such as Hadoop, The NoSQL systems are designed to have a simpler key-value based data model having in-built *sharding*, hence, these work seamlessly in a distributed cloud based environment. In contrast, one can use Hadoop based systems to run long running decision support and analytical queries consuming and possible producing bulk data. For processing *data in motion*, we present use-cases and illustrative algorithms of data stream management system (DSMS). We also illustrate applications which can use these two kinds of systems to quickly process massive amount of data.

1 Introduction

Recent financial crisis has changed the way businesses think about their finances. Organizations are actively seeking simpler, lower cost and faster to market alternatives about everything. Clouds are cheap and allow businesses to off-load computing tasks while saving IT costs and resources. In cloud computing applications, data, platform, and other resources are provided to users as services delivered over the network. The cloud computing enables self-service with no or little vendor intervention. It provides a utility model of resources where businesses only pay for their usage. As these resources are shared across a large number of users, cost of computing is much lower compared to dedicated resource provisioning.

Many industries, such as telecom, health care, retail, pharmaceutical, financial services, etc., generate large amounts of data. For instance, in 2010, Facebook had 21 Peta Bytes of internal warehouse data with 12 TB new data added every day and 800

TB compressed data scanned daily [12]. These data have: large *volume*, an Indian Telecom company generates more than 1 Terabyte of call detail records (CDRs) daily; high *velocity*, twitter needs to handle 4.5 Terabytes of video uploads in real-time per day; wide *variety*, structured data (e.g., call detail records in a telecom company), semi-structured data (e.g., graph data), unstructured data (e.g., product reviews on twitter), which needs to be integrated together; and data to be integrated have different *veracity*, data needs to cleaned before it can be integrated. Gaining critical business insights by querying and analyzing such massive amounts of data is becoming the need of the hour.

Traditionally, data warehouses have been used to manage the large amount of data. The warehouses and solutions built around them are unable to provide reasonable response times in handling expanding data volumes. One can either perform analytics on big volume once in days or one can perform transactions on small amounts of data in seconds. With the new requirements, one needs to ensure the real-time or near real-time response for huge amount of data. The 4V's of big data – volume, velocity, variety and veracity—makes the data management and analytics challenging for the traditional data warehouses. Big data can be defined as data that *exceeds the processing capacity of conventional database systems*. It implies that the data count is too large, and/or data values change too fast, and/or it does not follow the rules of conventional database management systems (e.g., consistency). One requires new expertise in the areas of data management and systems management who understands how to model the data and prepare them for analysis, and understand the problem deeply enough to perform the analytics. As data is massive and/or fast changing we need comparatively many more CPU and memory resources, which are provided by distributed processors and storage in cloud settings. The aim of this paper is to outline the concepts and issues involved in new age data management, with suggestions for further readings to augment the contents of this paper. Here is the outline of this paper: in the next section, we describe the factors which are important for enterprises to have cloud based data analytics solutions. As shown in Figure 1, big data processing involves interactive processing and decision support processing of *data-at-rest* and real-time processing of *data-in-motion*, which are covered in Section 3, 4, and 5, respectively. For each data processing application, one may need to write custom code to carry out the required processing. To avoid writing custom code for data processing applications, various SQL like query languages have been developed. We discuss these languages in Section 6. Section 7 summarizes some enterprise applications which illustrate the issues one needs to consider for designing big data applications. Specifically, we consider applications in telecom, financial services, and sensor domains. We conclude by outlining various research challenges in data management over cloud in Section 8.

Figure 1 presents various components of big data processing story. This figure also mentions the section numbers corresponding to various components in this paper.

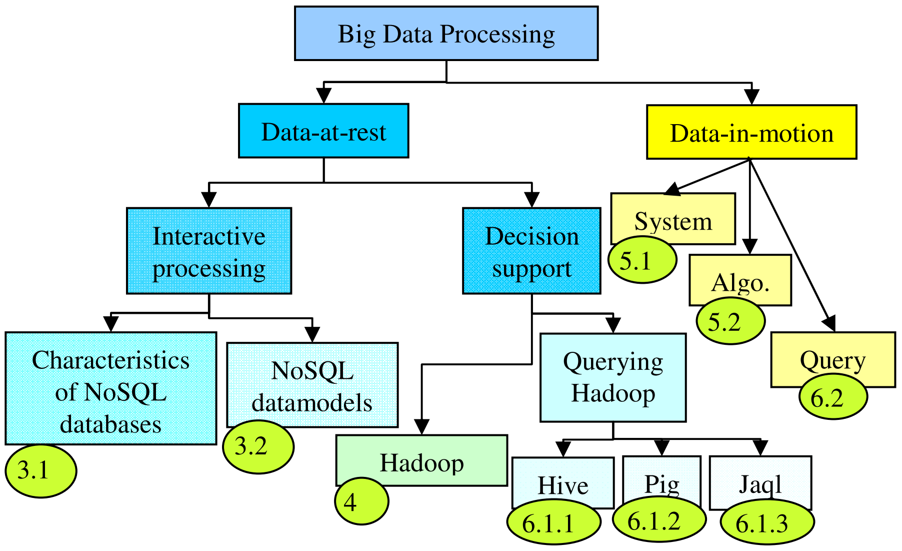

Fig. 1. Outline of the paper

2 Cloud Data Management

Cloud computing can be used for performing massive scale data analytics in a cost effective and scalable manner. In this section we discuss the interplay of cloud computing and data management, specifically, what are the benefits of cloud computing for data management; and the factors one should consider while moving from a dedicated data management infrastructure to a cloud based infrastructure.

2.1 Benefits of Cloud Computing

As mentioned earlier, a large volume of data is generated by many applications which cannot be managed by traditional relational database management systems. As organizations use larger and larger data warehouses for ever increasing data processing needs, the performance requirements continue to outpace the capabilities of the traditional approaches. The cloud based approach offers a means for meeting the performance and scalability requirements of the enterprise data management providing agility to the data management infrastructure. As with other cloud environments, data management in the cloud benefits end-users by offering a pay-as-you-go (or utility based) model and adaptable resource requirements that free up enterprises from the need to purchase additional hardware and to go through the extensive procurement process frequently. The data management, integration, and analytics can be offloaded to public and/or private clouds. By using public-cloud, enterprises can get processing power and infrastructure as needed, whereas with private-cloud enterprises can improve the utilization of existing infrastructure. By using cloud computing, enterprises can effectively handle the wide ranging database requirements with minimum effort, thus allowing them to focus on their core work rather than getting bogged down with the infrastructure. Despite all these benefits, decision to move from dedicated infrastructure to the cloud based data processing depends on several logistics and operational factors such as security, privacy, availability, etc.; which are discussed next.

2.2 Moving Data Management to Cloud

A data management system has various stages of data lifecycle such as data ingestion, ETL (extract-transform-load), data processing, data archival, and deletion. Before moving one or more stages of data lifecycle to the cloud, one has to consider the following factors:

1. **Availability Guarantees:** Each cloud computing provider can ensure a certain amount of availability guarantees. Transactional data processing requires quick real-time answers whereas for data warehouses long running queries are used to generate reports. Hence, one may not want to put its transactional data over cloud but may be ready to put the analytics infrastructure over the cloud.
2. **Reliability of Cloud Services:** Before offloading data management to cloud, enterprises want to ensure that the cloud provides required level of reliability for the data services. By creating multiple copies of application components the cloud can deliver the service with the required reliability of service.
3. **Security:** Data that is bound by strict privacy regulations, such as medical information covered by the Health Insurance Portability and Accountability Act (HIPAA), will require that users log in to be routed to their secure database server.
4. **Maintainability:** Database administration is a highly skilled activity which involves deciding how data should be organized, which indices and views should be maintained, etc. One needs to carefully evaluate whether all these maintenance operations can be performed over the cloud data.

Cloud has given enterprises the opportunity to fundamentally shift the way data is created, processed and shared. This approach has been shown to be superior in sustaining the performance and growth requirements of analytical applications and, combined with cloud computing, offers significant advantages [19]. In the next three sections, we present various data processing technologies which are used with cloud computing. We start with NoSQL in the next section.

3 NoSQL

The term NoSQL was first used in 1998 for a relational database that does not use SQL. It encompasses a number of techniques for processing massive data in distributed manner. Currently it is used for all the alternative data management technologies which are used for solving the problems for which relational databases are a bad fit. It enables efficient capture, storage, search, sharing, analytics, and visualization of the massive scale data. Main reason of using NoSQL databases is the scalability issues with relational databases. In general, relational databases are not designed for distributed horizontal scaling. There are two technologies which databases can employ for meeting scalability requirement: *replication* and *sharding*. In the *replication* technology, relational databases can be scaled using a master-slave architecture where reads can be performed at any of the replicated slave; whereas writes are performed at the master. Each write operation results in writes at all the slaves, imposing a limit to scalability. Further, even reads may need to be performed at master as previous write may not have been replicated at all the slave nodes. Although the situation can be improved by using multiple masters, but this can result in conflicts among masters whose resolution is very costly [28]. Partitioning (*sharding*) can also be used for scaling writes in relational databases, but applications are required to be made partition aware. Further, once partitioned, relations need to be handled at the application layer, defeating the very purpose of relational databases. NoSQL databases overcome these problems of horizontal scaling.

3.1 Characteristics of NoSQL Databases

An important difference between traditional databases and NoSQL databases is that the NoSQL databases do not support updates and deletes. There are various reasons for this. Many of the applications do not need update and delete operations; rather, different versions of the same data are maintained. For example, in Telecom domains, older call detail records (CDRs) are required for auditing and data mining. In enterprise human resource databases, employee's records are maintained even if an employee may have left the organization. These, updates and deletes are handled using insertion with version control. For example, Bigtable[33] associates a timestamp with every data item. Further, one needs customized techniques for implementing efficient joins over massive scale data in distributed settings. Thus, joins are avoided in NoSQL databases.

Relational databases provide ACID (Atomicity, consistency, integrity, and durability) properties which may be more than necessary for various data management applications and use cases. Atomicity of over more than one record is not required in most of the applications. Thus, single key atomicity is provided by NoSQL databases. In traditional databases, strong consistency is supported by using conflict resolution at write time (using read and write locks) which leads to scalability problems. As per [26], databases cannot ensure three CAP properties simultaneously: Consistency, Availability, and Partition tolerance (i.e., an individual operation should complete even if individual components are not available). Among consistency and availability, the later is given more importance by various NoSQL databases, e.g., giving service to a customer is more important. Consistency can be ensured using *eventual consistency* [27] where reconciliation happens asynchronously to have eventually consistent database. Similarly, most applications do not need *serialization isolation* level (i.e., to ensure that operations are deemed to be performed one after the other). *Read committed* (i.e., lock on writes till the end of transaction) with single key atomicity is sufficient. Durability is ensured in traditional relational databases as well as NoSQL databases. But traditional databases provide that by using expansive hardware whereas NoSQL databases provide that with cluster of disks with replication and other fault tolerance mechanisms. An alternative of ACID for distributed NoSQL databases is BASE (Basic Availability, Soft-state, and Eventual consistent). By not following the ACID properties strictly, query processing is made faster for massive scale data. As per [26], one does not have any choice between the two (one has to go for BASE) if one needs to scale up for processing massive amount of data. Next we present various data models used for NoSQL along with their commercial implementations.

3.2 NoSQL Data Models

An important reason of popularity of NoSQL databases is their flexible data model. They can support various types of data models and most of these are not strict. In relational databases, data is modeled using relations and one needs to define schema before one starts using the database. But NoSQL databases can support key-value pairs, hierarchical data, geo-spatial data, graph data, etc., using a simple model. Further, in new data management applications, there is a need to frequently keep modifying schema, e.g., a new service may require an additional column or complex changes in data-model. In relational databases, schema modification is time consuming and hard. One needs to lock a whole table for modifying any index structure. We describe three data-models, namely, *key-value stores*, *document stores*, and *column families*.

- **Key-Value Stores:** In a key-value store, read and write operations to a data item are uniquely identified by its key. Thus, no primitive operation spans multiple data items (i.e., it is not easy to support range queries). Amazon's Dynamo [29] is an example of key-value store. In Dynamo, values are opaque to the system and they are used to store objects of size less than 1 MB. Dynamo provides incremental scalability; hence, keys are partitioned dynamically using a hash function to distribute the data over a set of machines or nodes. Each node is aware of keys handled by its peers allowing any node to forward a key's read or write operation to the correct set of nodes. Both read and write operations are performed on a number of nodes to handle data durability, and availability. Updates are propagated to all the replicas asynchronously (eventual consistency). Kai [25] is open source implementation of key-value store.
- **Document Stores:** In document stores, value associated with a key is a document which is not opaque to the database; hence, it can be queried. Unlike relational databases, in a document store, each document can have a different schema. Amazon's SimpleDB[30], MongoDB[32] and Apache's CouchDB [31] are some examples of NoSQL databases using this model. In CouchDB, each document is serialized in JSON (Java Script Object Notation) format, and has a unique document identifier (*docId*). These documents can be accessed using web-browser and queried using JavaScripts. As this database does not support any delete or update; in each read operation multiple versions are read and the most recent one is returned as the result. CouchDB supports real time document transformation and change notifications. The transformations can be done using the user provided *map* and *reduce* JavaScript functions (explained later in the chapter).

- **Column family stores:** Google's BigTable [33] pioneered this data model. BigTable is a sparse, distributed, durable, multi-dimensional sorted map (i.e., sequence of nested key-value pairs). Data is organized into tables. Each record is identified by a row key. Each row has a number of columns. Intersection of each row and column contains time-stamped data. Columns are organized into column-families or related columns. It supports transactions under a single row key. Multiple-row transactions are not supported. Data is partitioned by sorting row-keys lexicographically. BigTable can serve data from disk as well as memory. Data is organized into tablets of size 100-200 MB by default with each tablet characterized by its start-key and end-key. A tablet server manages 10s of tablets. Meta-data tables are maintained to locate tablet-server for a particular key. These metadata tables are also split into tablets. A *chubby file* is the root of this hierarchical meta-data, i.e., this file points to a root metadata tablet. This root tablet points to other metadata tablets which in turn points to user application data tablets. HBase [18] is an open source implementation of BigTable.

Table 1. Comparison of NoSQL databases with traditional relational databases

Product/feature	Dynamo	CouchDB	BigTable	Traditional Databases
Data Model	Key value rows	Documents	Column store	Rows/Relational
Transactional access	Single tuple	Multiple documents	Single and range	Range, complex
Data partition	Random	Random	Ordered by key	Not applicable
Consistency	Eventual	Eventual	Atomic	Transactional
Version control	Versions	Document version	Timestamp	Not applicable
Replication	Quorum for read and write	Incremental replication	File system	Not applicable

Table 1 provides comparison of these different types of NoSQL databases with traditional relational databases. Next we present Hadoop technology which can be used for decision support processing in a warehouse like setting.

4 Hadoop MapReduce

Google introduced MapReduce [6] framework in 2004 for processing massive amount of data over highly distributed cluster of machines. It is a generic framework to write massive scale data applications. This framework involves writing two user defined generic functions: *map* and *reduce*. In the *map* step, a master node takes the input data and the processing problem, divides it into smaller data chunks and sub-problems; and distributes them to worker nodes. A worker node processes one or more chunks using the sub-problem assigned to it. Specifically, each *map* process, takes a set of {*key*, *value*} pairs as input and generates one or more intermediate {*key*, *value*} pairs for each input key. In the *reduce* step, intermediate key-value pairs are processed to produce the output of the input problem. Each *reduce* instance takes a *key* and an array of *values* as input and produces output after processing the array of *values*:

$\mathtt{Map}(k_1,v_1) \bullet \mathtt{list}(k_2,v_2)$
$\mathtt{Reduce}(k_2,\ \mathtt{list}\ (v_2)) \bullet \mathtt{list}(v_3)$

Figure 2 shows an example MapReduce implementation for a scenario where one wants to find the list of customers having total transaction value more than $1000.

```
void map(String rowId, String row):
  // rowId: row name
  // row: a transaction recode
customerId= extract customer-id from row
transactionValue= extract transaction value from row
EmitIntermediate(customerId, transactionValue);

void reduce(String customerId, Iterator partialValues):
  // customerId: Id to identify a customer
  // partialValues: a list of transaction values
  int sum = 0;
  for each pv in partialValues:
    sum += pv;
  if(pv > 1000)
    Emit(cutsomerId, sum);
```

Fig. 2. Example *MapReduce* code

4.1 Hadoop

Hadoop [4] is the most popular open source implementation of MapReduce framework [6]. It is used for writing applications processing vast amount of data in parallel on large clusters of machines in a fault-tolerant manner. Machines can be added and removed from the clusters as and when required. In Hadoop, data is stored on Hadoop Distributed File System (HDFS) which is a massively distributed file system designed to run on cheap commodity hardware. In HDFS, each file is chopped up into a number of blocks with each block, typically, having a size of 64MB. As depicted in Figure 3, these blocks are parsed by user-defined code into {*key*, *value*} pairs to be read by *map* functions. The *map* functions are executed on distributed machines to generate output {*key*, *value*} pairs which are written on their respective local disks. The whole *key* space is partitioned and allocated to a number of reducers. Each *reduce* function uses HTTP GET method to pull {*key*, *value*} pairs corresponding to its allocated key space. For each key, a reduce instance processes the *key* and array of its associated *values* to get the desired output. HDFS follows master-slave architecture. An HDFS cluster, typically, has a single master, also called name node, and a number of slave nodes. The name node manages the file system name space, divides the file into blocks, and replicates them to different machines. Slaves, also called data nodes, manage the storage corresponding to that node. Fault tolerance is achieved by replicating data blocks over a number of nodes. The master node monitors progress of data processing slave nodes and, if a slave node fails or it is slow, reassigns the corresponding data-block processing to another slave node. In Hadoop, applications can be written as a series of MapReduce tasks also. Authors of [9] provide various data models one can use for efficient processing of data using MapReduce, such as universal model [11], column store [10], etc. By avoiding costly joins and disk reads, a combination of universal data and column store proves to be the most efficient data model.

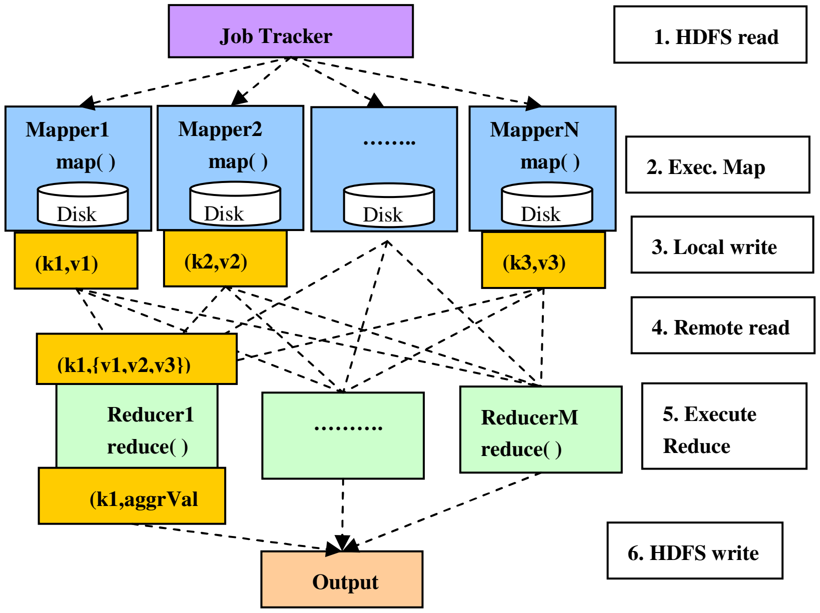

Fig. 3. MapReduce Framework

5 Data Stream Management System (DSMS)

We next turn our attention towards the second significant component of the Big Data story: analysis of the data in motion. In a conventional database, a query executes once and returns a set of results. In contrast in a streaming scenario, a query keeps getting continuously executed over a stream of data. Thus, rather than gathering large quantities of data, storing it on disk, and then analyzing it, data stream management systems (DSMSs) allow user to analyze the data-in-motion. This analysis is done in real-time thereby allowing users to trigger important events to enable enterprises to perform actions *just-in-time* yielding better results for the businesses. There are various enterprise class stream processing systems available. These systems provide better scalability and higher throughphput compared to complex event processing (CEP) systems. Distributed stream processing systems process input streams by transforming the tuples in a distributed manner using a system of computing elements and produce output streams. IBM InfoSphere Streams [36], S4 [35] and Storm [34] are some of the examples of such systems. These systems are particularly geared towards clusters of commodity hardware. Thus, one can use cloud infrastructure to perform various stream operations. These systems can be thought of as a series of connected operators. *Source operators* represent sources of data tuples. Intermediate operators perform various operations such as filtering, window aggregation, join, etc. Finally, output is fed to the *sink operators*. We describe these operators in details in the next section.

5.1 Various Stream Processing Systems

We describe three steam processing systems in this section: IBM's InfoSphere Streams, Twitter's Storm, and Yahoo's S4. InfoSphere Streams is a component based distributed stream processing platform, build to support higher data rates and various input data types. It also provides scheduling, load balancing, and high availability to ensure needs for scalability. Streams offers three methods for end-users to operator on streaming data: 1) Stream processing application declarative engine (SPADE) provides a language and runtime to create applications without understanding lower-level operations; 2) User queries can be expressed as per their information needs and interests, which are automatically converted into set of application components; 3) User can develop applications through an integrated development environment (IDE).

Storm provides with a general framework for performing streaming computations, much like Hadoop provides programmers with a general framework for performing batch operations. Operator graph defines how a problem should be processed and how data is entered and processed through the system by means of data streams. *Spouts* are entities that handle the insertion of data tuples into the topology and *bolts* are entities that perform computation. The *spouts* and *bolts* are connected by streams to form a directed graph. Parts of the computation can be parallelized by setting up a parallelism number for each *spout* and *bolt* in the job specification. *Bolts* can also send tuples to external systems, e.g., distributed databases or notification services. Storm supports the major programming languages to encode *spouts* and *bolts*. Storm supports acknowledge based guaranteed communication and generalized stream which takes any kind of objects and primitives (e.g., using thrift). Storm provides fault tolerance just like Hadoop in the face of failures of one or more operators.

In S4 terminology, each stream is described as a sequence of events having pairs of keys and attributes. Basic computational units in S4 are processing elements (PEs). Each instance of a PE is uniquely identified by the functionality of the PE, types of events that the PE consumes, the keyed attributes in those events, and the value of the keyed attributes in the events. Each PE consumes exactly those events which correspond to the value on which it is keyed. Processing nodes (PNs) are logical host for PEs. S4 routes every event to PNs based on a hash function of the values of all known keyed attributes in that event. Its programming paradigm includes writing generic, reusable, and configurable PEs which can be used across various applications. In the event of failures or higher rate events, it degrades performance by eliminating events as explained in the next section.

5.2 Example Stream Processing Algorithms

In a stream processing scenario, we need to generate answers quickly without storing all the tuples. In a typical stream processing application, some summary information of past seen tuples is maintained for processing future data. Since, the data processing is required to be done in (near) real-time, one uses in-memory storage for the summary information. With a bounded amount of memory, more often than not, it is not always possible to produce exact answers for data stream queries. In comparison, data-at-rest almost always produces exact answer (although there are some works giving on-line answers for long running Hadoop jobs). There are various works in the literature providing high quality approximation algorithms (with approximation guarantees) over data streams [38]. Various sampling techniques are proposed for matching streaming data rates with the processing rate. Random sampling can be used as simplest form of summary structure where a small sample is expected to capture the essential features of the stream. Sketching [37] is very popular technique for maintaining limited randomized summary of data for distributed processing of streams. Such sketches have been used for calculating various frequency counts (e.g., estimating number of distinct values in the stream). Histogram is a commonly used structure to capture the data distribution. Histograms can be used for query result size estimation, data mining, etc. Equi-width histograms, end-biased histograms, etc., are various types of histograms proposed in the literature. End-biased histograms can be used to answer Iceberg queries.

6 Querying Data over Cloud

In the previous section we discussed how we can process data using NoSQL databases, Hadoop, and DSMS. Various NoSQL databases are accessed using *get*(*key*) methods. For processing data in Hadoop one needs to write MapReduce programs. A MapReduce program can be written in various programming languages such as Java, Python, Ruby, etc. But this approach of writing custom MapReduce functions for each application has many problems:

1. Writing custom MapReduce jobs is difficult and time consuming and requires highly skilled developers.
2. For each MapReduce job to run optimally, one needs to configure a number of parameters, such as number of *mappers* and *reducers*, size of data block each *mapper* will process, etc. Finding suitable values of these parameters is not easy.
3. An application may require a series of MapReduce jobs. Hence, one needs to write these jobs and schedule them properly.
4. For efficiency of MapReduce jobs one has to ensure that all the reducers get a similar magnitude of data to process. If certain reducers get a disproportionate magnitude of data to process, these reducers will keep running for a long period of time while other reducers are sitting idle. This will hence in turn impact the performance of the MapReduce program.

Thus, instead various high level query languages have been developed so that one can avoid writing low level MapReduce programs. Queries written in these languages are in turn translated into equivalent MapReduce jobs by the compiler and these jobs are then consequently executed on Hadoop. Three of these languages Hive [7, 20], Jaql [13, 14], and Pig [16, 17] are the most popular languages in Hadoop Community. An analogy here would be to think of writing a MapReduce program as writing a Java program to process data in a relational database; while using one of these high level languages is like writing a script in SQL. We next briefly discuss each of these.

6.1 High Level Query Languages for Hadoop

We next briefly outline the key features of these three high level query languages to process the data stored in HDFS. Table 2 provides a comparison of these three scripting languages [3].

1. Hive: Hive [7,20] provides an easy entry point for data analysts, minimizing the effort required to migrate to the cloud based Hadoop infrastructure for distributed data storage and parallel query processing. Hive has been specially designed to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on huge volumes of data. Hive provides a subset of SQL, with features like *from* clause sub-queries, various types of *joins*, *group-by*s, *aggregation*s, "*create table as select*", etc. All these features make Hive very SQL-like. The effort required to learn and to get started with Hive is pretty small for a user proficient in SQL.

Hive structures data into well-understood database concepts like tables, columns, rows, and partitions. The schema of the table needs to be provided up-front. Just like in SQL, a user needs to first create a table with a certain schema and then only the data consistent with the schema can be uploaded to this table. A table can be partitioned on a set of attributes. Given a query, Hive compiler may choose to fetch the data only from certain partitions, and hence, partitioning helps in efficiently answering a query. It supports all the major primitive types: *integer*, *float*, *double* and *string*, as well as collection types such as *map*, *list* and *struct*. Hive also includes a system catalogue, a meta-store, that contains schemas and statistics, which are useful in data exploration, query optimization and query compilation [20].

2. Pig: Pig is a high-level scripting language developed by Yahoo to process data on Hadoop and aims at a sweet spot between SQL and MapReduce. Pig combines the best of both-worlds, the declarative style of SQL and low level procedural style of MapReduce. A Pig program is a sequence of steps, similar to a programming language, each of which carries out a single data transformation. However the transformation carried out in each step is fairly high-level e.g., filtering, aggregation etc., similar to as in SQL. Programs written in Pig are firstly parsed for syntactic and instance checking. The output from this parser is a logical plan, arranged in a directed acyclic graph, allowing logical optimizations, such as projection pushdown to be carried out. The plan is compiled by a MapReduce compiler, which is then optimized once more by a MapReduce optimizer performing tasks such as early partial aggregation. The MapReduce program is then submitted to the Hadoop job manager for execution.

Pig has a flexible, fully nested data model and allows complex, non-atomic data types such as set, map, and tuple to occur as fields of a table. A *bytearray* type is supported, to facilitate unknown data types and lazy conversion of types. Unlike Hive, stored schemas are optional. A user can supply schema information on the fly or can choose not to supply at all. The only capability required is to be able to read and parse the data. Pig also has the capability of incorporating user define functions (UDFs). A unique feature of Pig is that it provides a debugging environment. The debugging environment can generate a sample data to help a user in locating any error made in a Pig script.

3. Jaql: Jaql is a functional data query language, designed by IBM and is built upon JavaScript Object Notation (JSON) [8] data model. Jaql is a general purpose data-flow language that manipulates semi-structured information in the form of abstract JSON values. It provides a framework for reading and writing data in custom formats, and provides support for common input/output formats like CSVs, and like Pig and Hive, provides operators such as filtering, transformations, sort, group-bys, aggregation, and join. As the JSON model provides easy migration of data to- and from- some popular scripting languages like JavaScript and Python, Jaql is extendable with operations written in many programming languages. JSON data model supports atomic values like numbers and strings. It also supports two container types: *array* and *record* of name-value pairs, where the values in a container are themselves JSON values. Databases and programming languages suffer an impedance mismatch as both their computational and data models are so different. As JSON has a much lower impedance mismatch (with respect to Java) than, XML for example, but has much richer data types than relational tables. Jaql comes with a rich array of built-in functions for processing unstructured or semi-structured data as well. For example, Jaql provides a bridge for *SystemT* [14] using which a user can convert natural language text into a structured format. Jaql also provides a user with the capability of developing *modules*, a concept similar to Java *packages*. A set of related functions can be bunched together to form a module. A Jaql script can import a module and can use the functions provided by the module.

Table 2. Comparison of Hive, Pig, and Jaql

Feature	**Hive**	**Pig**	**Jaql**
Developed by	Facebook	Yahoo	IBM
Specification	SQL like	Data flow	Data flow
Schema	Fixed schema	Optional schema	Optional schema
Turning completeness	Need extension using Java UDF	Need extension using Java UDF	Yes
Data model	Row oriented	Nested	JSON, XML
Diagnostics	Show, describe	Describe, explain commands	Explain
Java connectivity	JDBC	Custom library	Custom library

6.2 Querying Streaming Data

We describe two query languages for processing the streaming data: Continuous Query Language (CQL) [21] described by STREAM (**st**anford st**re**am dat**a** **m**anager) and Stream processing language (SPL) used with IBM InfoSphere Stream[22].

1. Continuous Query Language: CQL is an SQL based declarative language for continuously querying streaming and dynamic data, developed at Stanford University. CQL semantics is based on three classes of operator: *stream-to-relation*, *relation-to-relation*, and *relation-to-stream.* In *stream-to-relation* operators, CQL has three classes of sliding window operators: *time-based*, *tuple-based*, and *partitioned.* In the first two window operators, window size is specified using a time-interval *T* and the number of tuples *N*, respectively. The *partitioned* window operator is similar to SQL group-by which groups *N* tuples using specified attributes as keys. All *relation-to-relation* operators are derived from traditional relational queries. CQL has three relation-to-stream operators: *Istream*, *Dstream* and *Rstream.* Applying an *Istream*/ *Dstream* (insert/delete stream) operator to a relation R results in a stream of tuples inserted/deleted into/from the relation *R*. The *Rstream* (relation steam) generates a stream element $\langle s, \tau \rangle$ whenever tuple *s* is in relation *R* at time τ. Consider the following CQL statement for filtering a stream:

```
Select Istream(*)
from SpeedPosition [Range Unbounded]
where speed > 55
```

This query contains three operators: an *Unbounded* windowing operator producing a relation containing all the speed-position measurements up-to current time; relational filter operator restricting the relations with measurements having speed greater than 55; and *Istream* operator streaming new values in the relation as the continuous query result. It should be noted that there are no *stream-to-stream* operators in CQL. One can generate output streams using input streams by combination of other operators as exemplified above.

When a continuous query is specified in CQL stream management system, it is compiled into a query plan. The generated query plan is merged with existing query plans for sharing computations and storage. Each query plan runs continuously with three types of components: *operators*, *queues*, and *synopses*. Each operator reads from input queues, processes the input based on its semantics, and writes output to output queues. Synopses store the intermediate stage needed by continuous query plans. In CQL query plans synopses for an operator are not used by any other operator. For example, to perform window-join across two streams, a join operator maintains one synopsis (e.g., hash of join attributes) for each of the join inputs.

2. Stream Processing Language: This is a structured application development language to build applications over InfoSphere streams. System-S [23] is the stream processing middleware used by SPL. It supports structured as well as unstructured data stream processing. It provides a toolkit of operators using which one can implement any relational query with window extensions. It also supports extensions for application domains like signal processing, data mining, etc. Among operators, *functor* is used for performing tuple level operations such as filtering, projection, attribute creation, etc.; *aggregate* is used for grouping and summarization; *join* is used for correlating two streams; *barrier* is used for consuming tuples from multiple streams and outputting a tuple in a particular order; *punctor* is also for tuple level manipulations where conditions on the current and past tuples are evaluated for generating punctuations in the output stream; *split* is used for routing tuples to multiple output streams; and *delay* operator is used for delaying a stream based on a user-supplied time interval. Besides these System-S also has edge adaptors and user defined operators. A *source* adaptor is used for creating stream from an external source. This adaptor is capable of parsing, tuple creation, and interacting with diverse external devices. A *sink* adaptor can be used to write tuples into a file or a network. It supports three types of windowing: *tumbling* window, *sliding* window, and *punctuation*-based window. Its application toolkit can be quickly used by application developers for quickly prototyping a complex streaming application.

7 Data Management Applications over Cloud

In this section, we consider three example applications where large scale data management over cloud is used. These are specific use-case examples in telecom, finance, and sensors domains. In the telecom domain, massive amount of *call detail records* can be processed to generate near real-time network usage information. In finance domain we describe the fraud detection application. We finally describe a use-case involving massive scale spatio-temporal data processing.

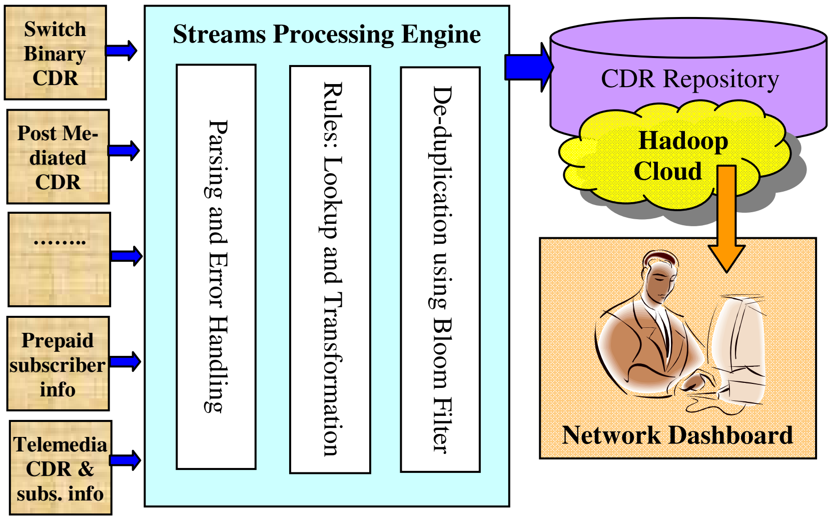
Switch Binary CDR
Post Mediated CDR
……..
Prepaid subscriber info
Telemedia CDR & subs. info
Streams Processing Engine
Parsing and Error Handling
Rules: Lookup and Transformation
De-duplication using Bloom Filter
CDR Repository
Hadoop Cloud
Network Dashboard

Fig. 4. CDR Data Processing using Big Data technologies

7.1 Dashboard for CDR Processing

Telecom operators are interested in building a dashboard that would allow the analysts and architects to understand the traffic flowing through the network along various dimensions of interest. The traffic is captured using *call detail records* (CDRs) whose volume runs into a terabyte per day. CDR is a structured stream generated by the telecom switches to summarize various aspects of individual services like voice, SMS, MMS, etc. Monitoring of CDRs flowing through cell sites helps the telecom operator decide regions where there is high network congestion. Adding new cell sites is the obvious solution for congestion reduction. However, each new site costs more than 20K USD to setup. Therefore, determining the right spot for setting up the cell site and measuring the potential traffic flowing through the site will allow the operator to measure the return on investment. Other uses of the dashboard include determining the cell site used most for each customer, identifying whether users are mostly making calls within cell site calls, and for cell sites in rural areas identifying the source of traffic i.e. local versus routed calls. Given the huge and ever growing customer base and large call volumes, solutions using traditional warehouse will not be able to keep-up with the rates required for effective operation. The need is to process the CDRs in near real-time, mediate them (i.e., collect CDRs from individual switches, stitch, validate, filter, and normalize them), and create various indices which can be exploited by dashboard among other applications. An IBM SPL based system leads to mediating 6 billion CDRs per day [24]. The dashboard creates various aggregates around combinations of the 22 attributes for helping the analysts. Furthermore, it had to be projected into future based on trends observed in the past. These CDRs can be loaded periodically over cloud data management solution. As cloud provides flexible storage, depending on traffic one can decide on the storage required. These CDRs can be processed using various mechanisms described in the chapter to get the required key performance indicators. Figure 4 shows schematic of such a system.

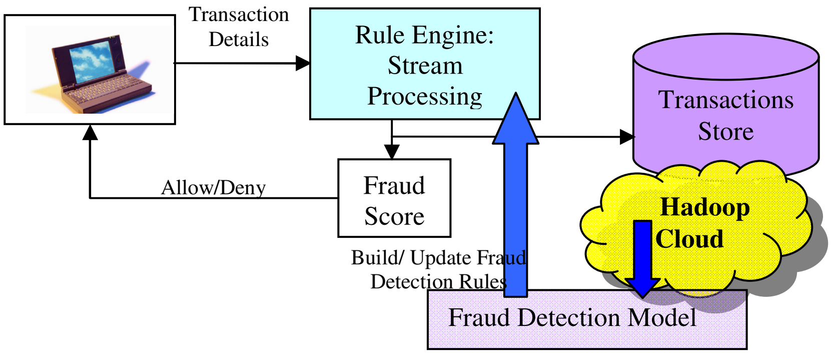

Fig. 5. Credit card fraud detection using Big Data technologies

7.2 Credit Card Fraud Detection

More than one-tenth of world's population is shopping online [2]. Credit card is the most popular mode of online payments. As the number of credit card transactions rise, the opportunities for attackers to steal credit card details and commit fraud are also increasing. As the attacker only needs to know some details about the card (card number, expiration date, etc.), the only way to detect online credit card fraud is to analyze the spending patterns and detect any inconsistency with respect to usual spending patterns. Various credit card companies compile a consumer profile for each and every card holder based on purchases s/he makes using his/her credit card over the years. These companies keep tabs on the geographical locations where the credit card transactions are made—if the area is far from the card holder's area of residence, or if two transactions from the same credit card are made in two very distant areas within a relatively short timeframe, — then the transactions are potentially fraud transactions. Various data mining algorithms are used to detect patterns within the transaction data. Detecting these patterns requires the analysis of large amount of data. For analyzing these credit card transactions one may need to create tuples of transaction for a particular credit card. Using these tuples of the transactions, one can find the distance between geographic locations of two consecutive transactions, amount of these transactions, time difference between transactions, etc. By these parameters, one can find the potential fraudulent transactions. Further data mining, based on a particular user's spending profile can be used to increase the confidence whether the transaction is indeed fraudulent.

As number of credit card transactions is huge and the kind of processing required is not a typical relational processing (hence, warehouses are not optimized to do such processing), one can use Hadoop based solution for this purpose as depicted in Figure 5. Using Hadoop one can create customer profile as well as creating matrices of consecutive transactions to decide whether a particular transaction is a fraud transaction. As one needs to find the fraud with-in some specified time, stream processing can help. By employing massive resources for analyzing potentially fraud transactions one can meet the response time guarantees.

7.3 Spatio-temporal Data Processing

Rapid urbanization has caused various city planners to rethink and redesign how we can make better use of limited resources and thereby provide a better quality of life to the residents. In recent years, the term *smarter city* has been coined to refer to a city management paradigm in which IT plays a crucial role. A smart city promises to bring greater automation, intelligent routing and transportation, better monitoring, and better city management. Smart transportation requires continuous monitoring of the vehicles over a period of time to gain patterns of behavior of traffic and road incidents. This requires generating, collecting, and analyzing massive data which is inherently spatio-temporal in nature. For monitoring and mining such massive amount of dynamic data we need big data technologies. Real time traffic data as well as weather information can be collected and processed using a stream processing system. Any traveler can get the best route for a particular destination by querying using her location and the destination. Another example is smarter environment monitoring system. Such a system is envisioned to collect weather, seismic, and pollution data. Such sensors are deployed all over a city and generate a number of readings everyday. Such data can be used to locate the source of an air pollution incident where air dispersion models can be run to provide information on probable locations of the pollution source.

8 Discussion and Conclusion

We presented the need for processing large amount of data having high variety and veracity at high speed; different technologies for distributed processing such as NoSQL, Hadoop, Streaming data processing; Pig, Jaql, Hive, CQL, SPL for querying such data; and customer use cases. There are various advantages in moving to cloud resources from dedicated resources for data management. But some of the enterprises and governments are still skeptical about moving to cloud. More work is required for cloud security, privacy and isolation areas to alleviate these fears. As noted earlier various applications involve huge amount of data and may require real time processing, one needs tools for bulk processing of huge amount of data, real time processing of streaming data and method of interaction between these two modules. For given cloud resources one needs to associate required resources for both the modules (bulk and stream data processing) so that the whole system can provide the required response time with sufficient accuracy. More research is required for facilitating such systems.

References

1. Avizienis, A.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing (2004)
2. Srivastava, A., Kundu, A., Sural, S., Majumdar, A.: Credit Card Fraud Detection using Hidden Markov Model. IEEE Transactions on Dependable and Secure Computing (2008)
3. Stewart, R.J., Trinder, P.W., Loidl, H.-W.: Comparing High Level MapReduce Query Languages. In: Temam, O., Yew, P.-C., Zang, B. (eds.) APPT 2011. LNCS, vol. 6965, pp. 58–72. Springer, Heidelberg (2011)
4. Apache Foundation. Hadoop, `http://hadoop.apache.org/core/`
5. Awadallah, A.: Hadoop: An Industry Perspective. In: International Workshop on Massive Data Analytics Over Cloud (2010) (keynote talk)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of ACM 51(1), 107–113 (2008)
7. Hive- Hadoop wiki, `http://wiki.apache.org/hadoop/Hive`
8. JSON, `http://www.json.org`
9. Gupta, R., Gupta, H., Nambiar, U., Mohania, M.: Enabling Active Archival Over Cloud. In: Proceedings of Service Computing Conference, SCC (2012)
10. Stonebraker, M., et al.: C-STORE: A Column-oriented DBMS. In: Proceedings of Very Large Databases, VLDB (2005)
11. Vardi, M.: The Universal-Relation Data Model for Logical Independence. IEEE Software 5(2) (1988)
12. Borthakur, D., Jan, N., Sharma, J., Murthy, R., Liu, H.: Data Warehousing and Analytics Infrastructure at Facebook. In: Proceedings of ACM International Conference on Management of Data, SIGMOD (2010)
13. Jaql Project hosting, `http://code.google.com/p/jaql/`

14. Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.-C., Ozcan, F., Shekita, E.J.: Jaql: A Scripting Language for Large Scale Semi-structured Data Analysis. In: Proceedings of Very Large Databases, VLDB (2011)
15. Liveland: Hive vs. Pig, `http://www.larsgeorge.com/2009/10/hive-vs-pig.html`
16. Pig, `http://hadoop.apache.org/pig/`
17. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig-Latin: A Not-So-Foreign Language for Data Processing. In: Proceedings of ACM International Conference on Management of Data, SIGMOD (2008)
18. HBase, `http://hbase.apache.org/`
19. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Realtional Cloud: A Database-as-a-Service for the Cloud. In: Proceedings of Conference on Innovative Data Systems Research, CIDR (2011)
20. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive – A Petabyte Scake Data Warehouse Using Hadoop. In: Proceedings of International Conference on Data Engineering, ICDE (2010)
21. Arasu, A., Babu, S., Widom, J.: The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal (2005)
22. Zikopoulos, P., Eaton, C., Deroos, D., Deutsch, T., Lapis, G.: Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data. McGrawHill (2012)
23. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.: SPADE: The System S Declaratve Stream Processing Engine. In: Proceedings of ACM International Conference on Management of Data, SIGMOD (2008)
24. Bouillet, E., Kothari, R., Kumar, V., Mignet, L., et al.: Processing 6 billion CDRs/day: from research to production (experience report). In: Proceedings of International Conference on Distributed Event-Based Systems, DEBS (2012)
25. Kai, `http://sourceforge.net/apps/mediawiki/kai`
26. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P.: Cluster-Based Scalable Network Services. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP (1997)
27. Wada, H., Fekede, A., Zhao, L., Lee, K., Liu, A.: Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective. In: Proceedings of Conference on Innovative Data Systems Research, CIDR (2011)
28. Gray, J., Helland, P., O'Neil, P.E., Shasha, D.: The Dangers of Replication and a Solution. In: Proceedings of ACM International Conference on Management of Data (1996)
29. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilch, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP (2007)
30. Habeeb, M.: A Developer's Guide to Amazon SimpleDB. Pearson Education
31. Lehnardt, J., Anderson, J.C., Slater, N.: CouchDB: The Definitive Guide. O'Reilly (2010)
32. Chodorow, K., Dirolf, M.: MongoDB: The Definitive Guide. O'Reilly Media, USA (2010)
33. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: BigTable: A Distributed Storage System for Structured Data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design annd Implementation, OSDI (2006)

34. Storm: The Hadoop of Stream processing, http://fierydata.com/2012/03/29/storm-the-hadoop-of-stream-processing/
35. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed Stream Computing Platform. In: IEEE International Conference on Data Mining Workshops, ICDMW (2010)
36. Biem, A., Bouillet, E., Feng, H., et al.: IBM infosphere streams for scalable, real-time, intelligent transportation services. In: SIGMOD 2010 (2010)
37. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. In: Proceedings of the Annual Symposium on Theory of Computing, STOC (1996)
38. Babcock, B., Babu, S., Datar, M., Motvani, R., Widom, J.: Model and Issues in Data Streams Systems. ACM PODS (2002)