Query processing in multistore systems: an overview

Carlyna Bondiombouy, Patrick Valduriez

Query Processing in Multistore Systems: an overview

Carlyna Bondiombouy* and Patrick Valduriez

Inria and LIRMM,
University of Montpellier,
Montpellier,France
Email: carlyna.bondiombouy@inria.fr
Email: patrick.valduriez@inria.fr
*Corresponding author

Abstract: Building cloud data-intensive applications often requires using multiple data stores (NoSQL, HDFS, RDBMS, etc.), each optimised for one kind of data and tasks. However, the wide diversification of data store interfaces makes it difficult to access and integrate data from multiple data stores. This important problem has motivated the design of a new generation of systems, called multistore systems, which provide integrated or transparent access to a number of cloud data stores through one or more query languages. In this paper, we give an overview of query processing in multistore systems. We start by introducing the recent cloud data management solutions and query processing in multidatabase systems. Then, we describe and analyse some representative multistore systems, based on their architecture, data model, query languages and query processing techniques. To ease comparison, we divide multistore systems based on the level of coupling with the underlying data stores, i.e., loosely-coupled, tightly-coupled and hybrid. Our analysis reveals some important trends, which we discuss. We also identify some major research issues.

Keywords: cloud data stores; multistore systems; multidatabase systems; query processing.

Reference to this paper should be made as follows: Bondiombouy, C. and Valduriez, P. (xxxx) 'Query processing in multistore systems: an overview', *Int. J. Cloud Computing*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Carlyna Bondiombouy is a Ph.D. student at University of Montpellier, France. She is working on query processing in cloud. She holds a Master degree in information security from University Cheikh Anta Diop, Dakar, Senegal.

Patrick Valduriez is a senior researcher at Inria, working on big data management. He has authored over 250 technical papers and several textbooks, among which "Principles of Distributed Database Systems". He currently serves as associate editor of several journals, including VLDBJ, DAPD and Internet and Databases. He has served as PC chair or general chair of major conferences such as SIGMOD and VLDB. He obtained the best paper award at VLDB00. He was the recipient of the 1993 IBM scientific prize in Computer Science in France and the 2014 Innovation Award from Inria – French Academy of Science – Dassault Systems. He is an ACM Fellow.

1 Introduction

A major trend in data management for the cloud is the understanding that there is 'no one size fits all' solution. Thus, there has been a blooming of different cloud data management solutions, specialised for different kinds of data and tasks and able to perform orders of magnitude better than traditional

relational DBMS (RDBMS). Examples of new data management technologies include distributed file systems (e.g., GFS and HDFS), NoSQL data stores (e.g., Dynamo, Bigtable, Hbase, Mongodb, Neo4j), and data processing frameworks (e.g., MapReduce, Spark).

This has resulted in a rich offering of services that can be used to build cloud data-intensive applications that can scale and exhibit high performance. However, this has also led to a wide diversification of data store interfaces and the loss of a common programming paradigm. Thus, this makes it very hard for a user to build applications that use multiple data stores, e.g., relational, document and graph databases

The problem of accessing heterogeneous data sources, i.e. managed by different data management systems such as RDBMS or XML DBMS, has long been studied in the context of multidatabase systems [ÖV11] (also called federated database systems, or more recently data integration systems [DHI12]). Most of the work on multidatabase query processing has been done in the context of the mediator-wrapper architecture, using a declarative, SQL-like language. The mediator-wrapper architecture allows dealing with three major properties of the data sources: distribution (i.e. located at different sites), heterogeneity (i.e. with different data models and languages) and autonomy (i.e. under local control) [ÖV11].

The state-of-the-art solutions for multidatabase query processing can be useful to transparently access multiple data stores in the cloud. However, operating in the cloud makes it quite different from accessing data sources on a wide-area network or the Internet. First, the kinds of queries are different. For instance, a web data integration query, e.g. from a price comparator, could access lots of similar web data sources, whereas a cloud query should be on a few but quite different cloud data stores and the user needs to have access rights to each data store. Second, both mediator and data source wrappers can only be installed at one or more servers that communicate with the data sources through the network. However, operating in a cloud, where data stores are typically distributed over the nodes of a computer cluster, provides more control over where the system components can be installed and thus, more opportunities to design an efficient architecture.

These differences have motivated the design of more specialized *multistore systems* [KVB+15] (also called polystores [DES+15]) that provide integrated access to a number of cloud data stores through one or more query languages. Several multistore systems are being built, with different objectives, architectures and query processing approaches, which makes it hard to compare them. To ease comparison, we divide multistore systems based on the level of coupling with the underlying data stores, i.e. loosely-coupled, tightly-coupled and hybrid.

Loosely-coupled systems are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can then be accessed through the multistore system common language as well as separately through their local language.

Tightly-coupled systems trade autonomy for performance, typically in a shared-nothing cluster, so that data stores can only be accessed through the multistore system, directly through their local language.

Hybrid systems tightly-couple some data stores, typically an RDBMS, and loosely-couple some others, typically HDFS through a data processing framework like MapReduce or Spark.

In this paper, we give an overview of query processing in multistore systems. The objective is not to give an exhaustive survey of all systems and techniques, but to focus on the main solutions and trends, based on the study of nine representative systems (3 for each class). The rest of the paper is organized as follows. In Section 2, we introduce cloud data management, including distributed file systems, NoSQL systems and data processing frameworks. In Section 3, we review the main query processing techniques for multidatabase systems, based on the mediator-wrapper architecture. Finally, in Section 4, we analyze the three kinds of multistore systems, based on their architecture, data model, query languages and query processing techniques. Section 5 concludes and discusses open issues.

2 Cloud Data Management

A cloud architecture typically consists of multiple sites, i.e. data centers at different geographic locations, each one providing computing and storage resources as well as various services such as application (AaaS), infrastructure (IaaS), platform (PaaS), etc. To provide reliability and availability, there is always some form of data replication between sites.

For managing data at a cloud site, we could rely on RDBMS technology, all of which have a distributed and parallel version. However, RDBMSs have been lately criticized for their "one size fits all" approach [SAD+10]. Although they have been able to integrate support for all kinds of data (e.g. multimedia objects, XML documents) and new functions, this has resulted in a loss of performance, simplicity and flexibility for applications with specific, tight performance requirements. Therefore, it has been argued that more specialized DBMS engines are needed. For instance, column-oriented DBMSs [AMH08], which store column data together rather than rows in traditional row-oriented RDBMSs, have been shown to perform more than an order of magnitude better on Online Analytical Processing (OLAP) workloads. Similarly, Data Stream Management Systems (DSMSs) are specifically architected to deal efficiently with data streams, which RDBMSs cannot even support [NPP13].

The "one size does not fit all" argument generally applies to cloud data management as well. However, internal clouds used by enterprise information systems, in particular for Online Transaction Processing (OLTP), may use traditional RDBMS technology. On the other hand, for OLAP workloads and web-based applications on the cloud, RDBMSs provide both too much (e.g. transactions, complex query language, lots of tuning parameters), and too little (e.g. specific optimizations for OLAP, flexible programming model, flexible schema, scalability) [Ram09].

Some important characteristics of cloud data have been considered for designing data management solutions. Cloud data can be very large, unstructured or semi structured, and typically append-only (with rare updates). And cloud users and application developers may be in high numbers, but not DBMS experts. Therefore, current cloud data management solutions have traded ACID (Atomicity, Consistency, Isolation, Durability) transactional properties for scalability, performance, simplicity and flexibility.

The preferred approach of cloud providers is to exploit a shared-nothing cluster [ÖV11], i.e. a set of loosely connected computer servers with a very fast, extensible interconnect (e.g. Infiniband). When using commodity servers with internal direct-attached storage, this approach provides scalability with excellent performance-cost ratio. Compared to traditional DBMSs, cloud data management uses a different software stack with the following layers: distributed storage, database management and distributed processing. In the rest of this section, we introduce this software stack and present the different layers in more details.

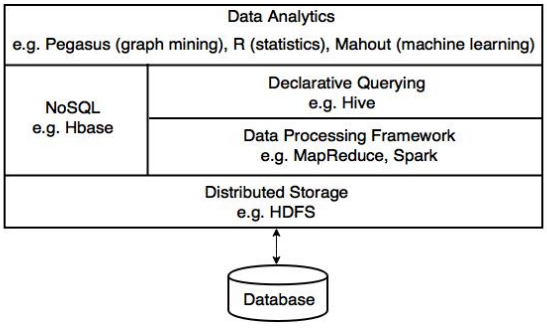
Cloud data management (see Figure 1) relies on a distributed storage layer, whereby data is typically stored in files or objects distributed over the nodes of a shared-nothing cluster. This is one major difference with the software stack of current DBMSs that relies on block storage. Interestingly, the software stack of the first DBMSs was not very different from that used now in the cloud. The history of DBMSs is interesting to understand the evolution of this software stack. The very first DBMSs, based on the hierarchical or network models, were built as extensions of a file system, such as COBOL, with inter-file links. And the first RDBMSs too were built on top of a file system. For instance, the famous Ingres RDBMS [SKWH76] was implemented atop the Unix file system. But using a general-purpose file system was making data access quite inefficient, as the DBMS could have no control over data clustering on disk or cache management in main memory. The main criticism for this file-based approach was the lack of operating system support for database management (at that time) [Sto81]. As a result, the architecture of RDBMSs evolved from file-based to block-based, using a raw disk interface provided by the operating system. A block-based interface provides direct, efficient access to disk blocks (the unit of storage allocation on disks). Today all RDBMSs are block-based, and thus have full control over disk management.

The evolution towards parallel DBMSs kept the same approach, in particular, to ease the transition from centralized systems. Parallel DBMSs use either a shared-nothing or shared-disk architecture. With

shared-nothing, each node (e.g. a server in a cluster) has exclusive access to its local disk through internal direct-attached storage. Thus, big relational tables need be partitioned across multiple disks to favor parallel processing. With shared-disk, the disks are shared among all nodes through a storage area network, which eases parallel processing. However, since the same disk block can be accessed in concurrent mode by multiple cluster nodes, a distributed lock manager [ÖV11] is necessary to avoid write conflicts and provide cache coherency. In either architecture, a node can access blocks either directly through direct-attached storage (shared-nothing) or via the storage area network (shared-disk).

In the context of cloud data management, we can identify two main reasons why the old DBMS software stack strikes back. First, distributed storage can be made fault-tolerant and scalable (e.g. HDFS), which makes it easier to build the upper data management layers atop (see Figure 1). Second, in addition to the NoSQL layer (e.g. Hbase over HDFS), data stored in distributed files can be accessed directly by a data processing framework (e.g. MapReduce or Spark), which makes it easier for programmers to express parallel processing code. The distributed processing layer can then be used for declarative (SQL-like) querying, e.g. with a framework like Hive over MapReduce. Finally, at the top layer, tools such as Pegasus (graph mining), R (statistics) and Mahout (machine learning) can be used to build more complex big data analytics.

Figure 1 Cloud data management software stack



2.1 Distributed Storage

The distributed storage layer of a cloud typically provides two solutions to store data, files or objects, distributed over cluster nodes. These two solutions are complementary, as they have different purposes and they can be combined.

File storage manages data within unstructured files (i.e. sequences of bytes) on top of which data can be organized as fixed-length or variable-length records. A file system organizes files in a directory hierarchy, and maintains for each file its metadata (file name, folder position, owner, length of the content, creation time, last update time, access permissions, etc.), separate from the content of the file. Thus, the file metadata must first be read for locating the file's content. Because of such metadata management, file storage is appropriate for sharing files locally within a cloud data center and when the number of files are limited (e.g. in the hundreds of thousands). To deal with big files that may contain high numbers of records, files need be partitioned and distributed, which requires a distributed file system.

Object storage manages data as objects. An object includes its data along with a variable amount of metadata, and a unique identifier in in a flat object space. Thus, an object can be represented as a triple (oid, data, metadata), and once created, it can be directly accessed by its oid. The fact that data and metadata are bundled within objects makes it easy to move objects between distributed locations. Unlike in file systems where the type of metadata is the same for all files, objects can have variable amounts of metadata.

This allows much user flexibility to express how objects are protected, how they can be replicated, when they can be deleted, etc. Using a flat object space allows managing massive amounts e.g. billions or trillions) of unstructured data. Finally, objects can be easily accessed with a simple REST-based API with put and get commands easy to use on Internet protocols. Object stores are particularly useful to store a very high number of relatively small data objects, such as photos, mail attachments, etc. Therefore, most cloud providers leverage an object storage architecture, e.g. Amazon Web Services S3, Rackspace Files, Microsoft Azure Vault Storage and Google Cloud Storage.

Distributed file systems in the cloud can then be divided between block-based, extending a traditional file system, and object-based, leveraging an object store. Since these are complementary, there are also systems that combine both. In the rest of this section, we illustrate these three categories with representative systems.

2.1.1 Block-based Distributed File Systems

One of the most influential systems in this category is Google File System (GFS). GFS [GGL03] has been developed by Google (in C++ on top of Linux) for its internal use. It is used by many Google applications and systems, such as Bigtable and MapReduce, which we discuss next.

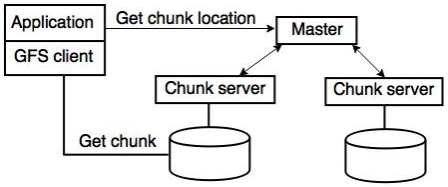
Similar to other distributed file systems, GFS aims at providing performance, scalability, fault-tolerance and availability. However, the targeted systems, shared-nothing clusters, are challenging as they are made of many (e.g. thousands of) servers built from inexpensive hardware. Thus, the probability that any server fails at a given time is high, which makes fault-tolerance difficult. GFS addresses this problem. It is also optimized for Google data-intensive applications, such as search engine or data analysis. These applications have the following characteristics. First, their files are very large, typically several gigabytes, containing many objects such as web documents. Second, workloads consist mainly of read and append operations, while random updates are rare. Read operations consist of large reads of bulk data (e.g. 1 MB) and small random reads (e.g. a few KBs). The append operations are also large and there may be many concurrent clients that append the same file. Third, because workloads consist mainly of large read and append operations, high throughput is more important than low latency.

GFS organizes files as a tree of directories and identifies them by pathnames. It provides a file system interface with traditional file operations (create, open, read, write, close, and delete file) and two additional operations: snapshot and record append. Snapshot allows creating a copy of a file or of a directory tree. Record append allows appending data (the "record") to a file by concurrent clients in an efficient way. A record is appended atomically, i.e. as a continuous byte string, at a byte location determined by GFS. This avoids the need for distributed lock management that would be necessary with the traditional write operation (which could be used to append data).

The architecture of GFS is illustrated in Figure 2. Files are divided into fixed-size partitions, called *chunks*, of large size, i.e. 64 MB. The cluster nodes consist of GFS clients that provide the GFS interface to applications, chunk servers that store chunks and a single GFS master that maintains file metadata such as namespace, access control information, and chunk placement information. Each chunk has a unique id assigned by the master at creation time and, for reliability reasons, is replicated on at least three chunk servers (in Linux files). To access chunk data, a client must first ask the master for the chunk locations, needed to answer the application file access. Then, using the information returned by the master, the client can request the chunk data to one of the replicas.

This architecture using single master is simple. And since the master is mostly used for locating chunks and does not hold chunk data, it is not a bottleneck. Furthermore, there is no data caching at either clients or chunk servers, since it would not benefit large reads. Another simplification is a relaxed consistency model for concurrent writes and record appends. Thus, the applications must deal with relaxed consistency using techniques such as checkpointing and writing self-validating records. Finally, to keep the system highly available in the face of frequent node failures, GFS relies on fast recovery and replication strategies.

Figure 2 GFS architecture



There are open source implementations of GFS, such as Hadoop Distributed File System (HDFS), a popular Java product. HDFS has been initially developed by Yahoo and is now the basis for the successful Apache Hadoop project, which together with other products (MapReduce, Hbase) has become a standard for big data processing. There are other important open source block-based distributed file systems for cluster systems, such as GlusterFS for shared-nothing and Global File System 2 (GFS2) for shared-disk, both being now developed by Red Hat for Linux.

2.1.2 Object-based Distributed File Systems

One of the first systems in this category is Lustre, an open source file system [Whi12]. Lustre was initially developed (in C) at Carnegie Mellon University in the late 1990s, and has become very popular in High Performance Computing (HPC) and scientific applications in the cloud, e.g. Intel Cloud Edition for Lustre. The architecture of the Lustre file system has three main components:

- One or more metadata servers that store namespace metadata, such as filenames, directories, access permissions, etc. Unlike block-based distributed file systems, such as GFS and HDFS, where the metadata server controls all block allocations, the Lustre metadata server is only involved when opening a file and is not involved in any file I/O operations, thus avoiding scalability bottlenecks.

- One or more object storage servers that store file data on one or more object storage targets (OSTs). An object storage server typically serves between two and eight OSTs, with each OST managing a single local disk file system.

- Clients that access and use the data. Lustre presents all clients with a unified namespace for all of the files and data, using the standard file system interface, and allows concurrent and coherent read and write access to the files in the file system.

These three components can be located at different server nodes in a shared-disk cluster, with disk storage connected to the servers using storage area network. Clients and servers are connected with the Lustre file system using a specific communication infrastructure called Lustre Networking (LNET). Lustre provides cache consistency of files' data and metadata by a distributed lock manager. Files can be partitioned using *data striping*, a technique that segments logically sequential data so that consecutive segments are stored on different disks. This is done by distributing objects across a number of object storage servers and OSTs. To provide data reliability, objects in OSTs are replicated using primary-copy replication and RAID6 disk storage technology.

When a client accesses a file, it completes a filename lookup on the metadata server and gets back the layout of the file. Then, to perform read or write operations on the file, the client interprets the layout to map the operation to one or more objects, each residing on a separate OST. The client then locks the file range being operated on and executes one or more parallel read or write operations directly to the OSTs. Thus, after the initial lookup of the file layout, unlike with block-based distributed file systems, the metadata server is not involved in file accesses, so the total bandwidth available for the clients to read and write data scales almost linearly with the number of OSTs in the file system.

Another popular open-source object-based distributed file system is XtreemFS [HCK+08]. XtreemFS is highly fault-tolerant, handling all failure modes including network splits, and highly-scalable, allowing objects to be partitioned or replicated across shared-nothing clusters and data centers.

2.1.3 Combining Block Storage and Object Storage

An important trend for data management in the cloud is to combine block storage and object storage in a single system, in order to support both large files and high numbers of objects. The first system that combined block and object storage is Ceph [WBM+06]. Ceph is an open source software storage platform, now developed by Red Hat, that combines object, block, and file storage in a shared-nothing cluster at exabyte scale. Ceph decouples data and metadata operations by eliminating file allocation tables and replacing them with data distribution functions designed for heterogeneous and dynamic clusters of unreliable object storage devices (OSDs). This allows Ceph to leverage the intelligence present in OSDs to distribute the complexity surrounding data access, update serialization, replication and reliability, failure detection, and recovery. Ceph and GlusterFS are now the two major storage platforms offered by Red Hat for shared-nothing clusters.

HDFS, on the other hand, has become the De facto standard for scalable and reliable file system management for big data. Thus, there is much incentive to add object storage capabilities to HDFS, in order to make data storage easier for cloud providers and users. In Azure HDInsight, Microsoft's Hadoop-based solution for big data management in the cloud, HDFS is integrated with Azure Blob storage, the object storage manager, to operate directly on structured or unstructured data. Blob storage containers store data as key-value pairs, and there is no directory hierarchy.

Hortonworks, a distributor of Hadoop software for big data, has recently started a new initiative called Ozone, an object store that extends HDFS beyond a file system, toward a more complete storage layer. Similar to GFS, HDFS separates metadata management from a block storage layer. Ozone uses the HDFS block storage layer to store objects identified by keys and adds a specific metadata management layer on top of block storage.

2.2 NoSQL Systems

NoSQL systems are specialized DBMSs that address the requirements of web and cloud data management. As an alternative to relational databases, they support different data models and different languages than standard SQL. They emphasize scalability, fault-tolerance and availability, sometimes at the expense of consistency. NoSQL (Not Only SQL) is an overloaded term, which leaves much room for interpretation and definition. In this paper, we consider the four main categories of NoSQL systems that are used in the cloud: key-value, wide column, document and graph. In the rest of this section, we introduce each category and illustrate with representative systems.

2.2.1 Key-Value Stores

In the key-value data model, all data is represented as key-value pairs, where the key unlikely identifies the value. Object stores, which we discussed above, can be viewed as a simple form of key-value store. However, the keys in key-value stores can be sequences of bytes of arbitrary length, not just positive integers, and the values can be text data, not just Blobs. Key-values stores are schemaless, which yields great flexibility and scalability. They typically provide a simple interface such as put (key, value), value=get (key), delete (key).

A popular key-value store is Dynamo [DHJ+07], which is used by some of Amazon's core services that need high availability. To achieve scalability and availability, Dynamo sacrifices consistency under some failure scenarios and uses a synthesis of well known peer-to-peer techniques [PAD12]. Data is partitioned and replicated across multiple cluster nodes in several data centers, which allows to handle entire data

center failures without a data outage. The consistency among replicas during updates is maintained by a quorum-like technique and an asynchronous update propagation protocol. Dynamo employs a gossip based distributed failure detection and membership protocol. To facilitate replica consistency, it makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use. Other popular key-value stores are Memcached, Riak and Redis.

An extended form of key-value store is able to store records, as sets of key-value pairs. One key, called major key or primary key, e.g. a social security number, uniquely identifies the record among a collection of records, e.g. people. The keys are usually sorted, which enables range queries as well as ordered processing of keys. Amazon SimpleDB and Oracle NoSQL Database are examples of advanced key-value stores. Many systems provide further extensions so that we can see a smooth transition to wide column store and document stores, which we discuss next.

2.2.2 Wide Column Stores

Wide column stores are advanced key-value stores, where key-value pairs can be grouped together in columns within tables. They combine some of the nice properties of relational databases, i.e. representing data as tables, with the flexibility of key-value stores, i.e. schemaless data.

Each row in a table is uniquely identified by a *row key*, which is like a mono-attribute key in a relational table. But unlike in a relational table, where columns can only contain atomic values, tables contain wide columns, called *column families*. A column family is a set of columns, each of which has a name, a value, and a timestamp (used for versioning) and within a column family, we can have different columns in each row. Thus, a column family is like a nested table within a column. Figure 3 shows a simple example of wide column table with two rows. The first column is the row key. The two other columns are column families.

Figure 3 A wide column table with two rows

Row key	Name	Email
100	Prefix : "Dr" Last : "Dobb"	email : gmail.com : " dobb@gmail.com"
101	First : "Alice" Last : "Martin"	email : gmail.com : " amartin@gmail.com" email : free.fr : " amartin@free.fr"

Wide column stores extend the key-value store interface with more declarative constructs that allow scans, exact-match and range queries over column families. They typically provide an API for these constructs to be used in a programming language. Some systems also provide an SQL-like query language, e.g. Cassandra Query Language (CQL).

At the origin of wide column stores is Google Bigtable [CDG+08], a database storage system for shared-nothing clusters. Bigtable uses GFS for storing structured data in distributed files, which provides fault-tolerance and availability. It also uses a form of dynamic data partitioning for scalability. And like GFS, it is used by popular Google applications, such as Google Earth, Google Analytics and Google+.

In a Bigtable row, a row key is an arbitrary string (of up to 64KB in the original system). A column family is a unit of access control and compression. A column family is defined as a set of columns, each identified by a *column key*. The syntax for naming column keys is **family:qualifier**, e.g. “email:gmail.com” in Figure 3. The qualifier, e.g. “gmail.com”, is like a relational attribute value, but used as a name as part of the column key to represent a single data item. This allows the equivalent of multi-valued attributes within a relational table, but with the capability of naming attribute values. In addition, the data identified by a column key within a row can have multiple versions, each identified by a timestamp (a 64 bit integer).

Bigtable provides a basic API for defining and manipulating tables, within a programming language such as C++. The API offers various operators to write and update values, and to iterate over subsets of

data, produced by a scan operator. There are various ways to restrict the rows, columns and timestamps produced by a scan, as in a relational select operator. However, there are no complex operators such as join or union, which need to be programmed using the scan operator. Transactional atomicity is supported for single row updates only.

To store a table in GFS, Bigtable uses range partitioning on the row key. Each table is divided into partitions, each corresponding to a row range. Partitioning is dynamic, starting with one partition (the entire table range) that is subsequently split into multiple partitions as the table grows. To locate the (user) partitions in GFS, Bigtable uses a metadata table, which is itself partitioned in metadata tablets, with a single root tablet stored at a master server, similar to GFS's master. In addition to exploiting GFS for scalability and availability, Bigtable uses various techniques to optimize data access and minimize the number of disk accesses, such as compression of column families as in column stores, grouping of column families with high locality of access and aggressive caching of metadata information by clients.

Bigtable builds on other Google technologies such as GFS and Chubby Lock Service. In May 2015, a public version of Bigtable was launched as Google Cloud Bigtable. There are popular open source implementations of Bigtable, such as: Hadoop Hbase that runs on top of HDFS; Cassandra that combines ideas from Bigtable and DynamoDB; and Accumulo.

2.2.3 Document Stores

Document stores are advanced key-value stores, where keys are mapped into values of document type, such as JSON, YAML or XML. Documents are typically grouped into collections, which play a role similar to relational tables. However, documents are different than relational tuples. Documents are self-describing, storing data and metadata (e.g. markups in XML) altogether and can be different from one another within a collection. Furthermore, the document structures are hierarchical, using nested constructs, e.g. nested objects and arrays in JSON. In addition to the simple key-value interface to retrieve documents, document stores offer an API or query language that retrieve documents based on their contents. Document stores make it easier to deal with change and optional values, and to map into program objects. This makes them attractive for modern web applications, which are subject to continual change, and where speed of deployment is important.

The most popular NoSQL document store is MongoDB [PHM10], an open source software written in C++. MongoDB provides schema flexibility, high availability, fault-tolerance and scalability in shared-nothing cluster. It stores data as documents in BSON (Binary JSON), an extension of JSON to include additional types such as int, long, and floating point. BSON documents contain one or more fields, and each field contains a value of a specific data type, including arrays, binary data and sub-documents.

MongoDB provides a rich query language to update and retrieve BSON data as functions expressed in JSON. The query language can be used with APIs in various programming languages. It allows key-value queries, range queries, geospatial queries, text search queries, and aggregation queries. Queries can also include user-defined JavaScript functions.

To provide efficient access to data, MongoDB includes support for many types of secondary indexes that can be declared on any field in the document, including fields within arrays. These indexes are used by the query optimizer. To scale out in shared-nothing clusters of commodity servers, MongoDB supports different kinds of data partitioning: range-based (as in Bigtable), hash-based and location-aware (whereby the user specifies key-ranges and associated nodes). High-availability is provided through primary-copy replication, with asynchronous update propagation. Applications can optionally read from secondary replicas, where data is eventually consistent [1]. MongoDB supports ACID transactions at the document level. One or more fields in a document may be written in a single transaction, including updates to multiple sub-documents and elements of an array. MongoDB makes extensive use of main memory to speed up database operations and native compression, using its storage engine (WiredTiger). It also supports pluggable storage engines, e.g. HDFS, or in-memory, for dealing with unique application demands.

Other popular document stores are CouchDB, Couchbase, RavenDB and Elasticsearch. High-level query languages can also be used on top of document stores. For instance, the Zorba query processor supports two different query languages, the standard XQuery for XML and JSONiq for JSON, which can be used to seamlessly process data stored in different data stores such as: Couchbase, Oracle NoSQL Database and SQLlite.

2.2.4 Graph Databases

Graph databases represent and store data directly as graphs which allows easy expression and fast processing of graph-like queries, e.g. computing the shortest path between two nodes in the graph. This is much more efficient than with a relational database where graph data need be stored as separated tables and graph-like queries require repeated, expensive join operations. Graph databases have become popular with data-intensive web-based applications such as social networks and recommender systems.

Graph databases represent data as nodes, edges and properties. Nodes represent entities such as people or cities. Edges are lines that connect any two nodes and represent the relationship between the two. Edges can be undirected, in which case the relationship is symmetric, or directed, in which case the relationship is asymmetric. Properties provide information to nodes, e.g. a person's name and address, or edges, e.g. the name of the relationship such as "friend". The data graph is typically stored using a specific storage manager that places data on disk so that the time needed for graph-specific access patterns is minimized. This is typically accomplished by storing nodes as close as possible to their edges and their neighbor nodes, in the same or adjacent disk pages.

Graph databases can provide a flexible schema, as in object databases where objects are defined by classes, by specifying node and edge types with their properties. This facilitates the definition of indexes to provide fast access to nodes, based on some property value, e.g. a city's name. Graph queries can be expressed using graph operators through a specific API or a declarative query language, e.g. the Pixy language that works on any graph database compatible with its API.

A popular graph database is Neo4j [Bru14], a commercially supported open-source software. It is a robust, scalable and high-performance graph database, with full ACID transactions. It supports directed graphs, where everything is stored in the form of either a directed edge, a node or an attribute. Each node and edge can have any number of attributes. Neo4j enforces that all operations that modify data occur within a transaction, guaranteeing data consistency. This robustness extends from single server embedded graphs to shared-nothing clusters. A single server instance can handle a graph of billions of nodes and relationships. When data throughput is insufficient, the graph database can be distributed and replicated among multiple servers in a high availability configuration. However, graph partitioning among multiple servers is not supported (although there are some projects working on it). Neo4j supports a declarative query language called Cypher, which aims at avoiding the need to write traversals in code. It also provides REST protocols and a Java API. As of version 2.0, indexing was added to Cypher with the introduction of schemas.

Other popular graph databases are Infinite graph, Titan, GraphBase, Trinity and Sparksee.

2.3 Data Processing Frameworks

Most unstructured data in the cloud gets stored in distributed files such as HDFS and needs to be analyzed using user programs. However, to make application programs scalable and efficient requires exploiting parallel processing. But parallel programming of complex applications is hard. In the context of HPC, parallel programming libraries such as OpenMP for shared-memory or Message Passing Interface (MPI) for shared-nothing are used extensively to develop scientific applications. However, these libraries are relatively low-level and require careful programming. In the context of the cloud, data processing frameworks have become quite popular to make it easier for programmers to express parallel processing code. They typically support the simple key-value data model and support operators that are automatically

parallelized. All the programmer has to do is to provide code for these operators. The most popular data processing frameworks, MapReduce, Spark and now Flink, differ in the functionality they offer in terms of operators, as well as in terms of implementation, for instance, disk-based versus in-memory. However, they all target scalability and fault-tolerance in shared-nothing clusters.

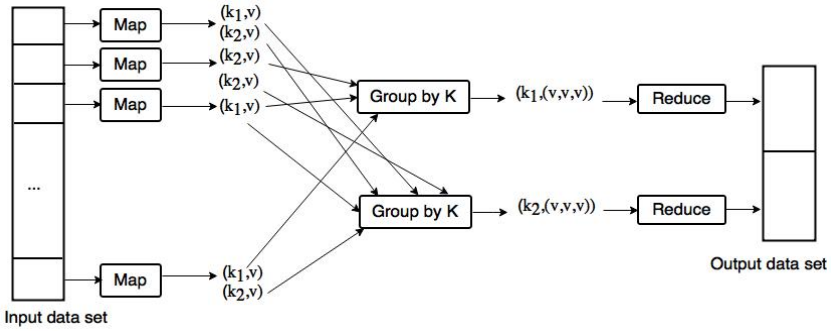
MapReduce [DG04] is a popular framework for processing and generating large datasets. It was initially developed by Google in C++ as a proprietary product to process large amounts of unstructured or semi-structured data, such as web documents and logs of web page requests, on large shared-nothing clusters of commodity nodes and produce various kinds of data such as inverted indices or URL access frequencies. Different implementations of MapReduce are now available such as Amazon MapReduce (as a cloud service) or Hadoop MapReduce (as a Java open source software).

MapReduce enables programmers to express in a simple, functional style their computations on large data sets and hides the details of parallel data processing, load balancing and fault-tolerance. The programming model includes only two operations, *map* and *reduce*, which we can find in many functional programming languages such as Lisp and ML. The map operation is applied to each record in the input data set to compute one or more intermediate (key, value) pairs. The reduce operation is applied to all the values that share the same unique key in order to compute a combined result. Since they work on independent inputs, map and reduce can be automatically processed in parallel, on different data partitions using many cluster nodes.

Figure 4 gives an overview of MapReduce execution in a cluster. There is one master node (not shown in the figure) in the cluster that assigns map and reduce tasks to cluster nodes, i.e. map and reduce nodes. The input data set is first automatically split into a number of partitions, each being processed by a different map node that applies the map operation to each input record to compute intermediate (key,value) pairs. The intermediate result is divided into n partitions, using a partitioning function applied to the key (e.g. hash(key) mod n).

Map nodes periodically write to disk their intermediate data into n regions by applying the partitioning function and indicate the region locations to the master. Reduce nodes are assigned by the master to work on one or more partitions. Each reduce node first reads the partitions from the corresponding regions on the map nodes, disks, and groups the values by intermediate key, using sorting. Then, for each unique key and group of values, it calls the user reduce operation to compute a final result that is written in the output data set.

Figure 4 Overview of MapReduce execution



Fault-tolerance is important as there may be many nodes executing map and reduce operations. Input and output data are stored in GFS that already provides high fault-tolerance. Furthermore, all intermediate data are written to disk, which helps checkpointing map operations and thus provides tolerance to soft failures. However, if one map node or reduce node fails during execution (hard failure), the task can be scheduled by the master onto other nodes. It may also be necessary to re-execute completed map tasks,

since the input data on the failed node disk is inaccessible. Overall, fault-tolerance is fine-grained and well suited for large jobs.

The often cited advantages of MapReduce are its ability to express various (even complicated) map and reduce functions, and its extreme scalability and fault-tolerance. However, it has been criticized for its relatively low-performance due to the extensive use of disk accesses, in particular compared with parallel DBMSs [SAD+10]. Furthermore, the two functions map and reduce are well-suited for OLAP-like queries with data selection and aggregation but not appropriate for interactive analysis or graph processing.

Spark is an Apache open-source data processing framework in Java originally developed at UC Berkeley [ZCF+10]. It extends the MapReduce model for two important classes of analytics applications: iterative processing (machine learning, graph processing) and interactive data mining (with R, Excel or Python). Compared with MapReduce, it improves the ease of use with the Scala language (a functional extension of Java) and a rich set of operators (map, reduce, filter, join, sortByKey, aggregateByKey, etc.). Spark provides an important abstraction, called Resilient Distributed Dataset (RDD), which is a collection of elements partitioned across cluster nodes. RDDs can be created from disk-based resident data in files or intermediate data produced by transformations with Scala programs. They can also be made memory-resident for efficient reuse across parallel operations.

Flink is the latest Apache open-source data processing framework. Based on the Stratosphere prototype [EST+13], it differs from Spark by its in-memory runtime engine which can be used for real time data streams as well as batch data processing. It runs on HDFS and supports APIs for Java and Scala.

2.3.1 Concluding Remarks

The software stack for data management in the cloud, with three main layers (distributed storage, database management and distributed processing) has led to a rich ecosystem with many different solutions and technologies, which are still evolving. Although HDFS has established itself as the standard solution for storing unstructured data, we should expect evolutions of distributed file systems that combine block storage and object storage in a single system. For data management, most NoSQL data stores, except graph databases, rely on (or extend) the key-value data model, which remains the best option for data whose structure needs to be flexible. There is also a rapid evolution of data processing frameworks on top of distributed file systems. For example, the popular MapReduce framework is now challenged by more recent systems such as Spark and Flink. Multistore systems should be able to cope with this evolution.

3 Multidatabase Query Processing

A multidatabase system provides transparent access to a collection of multiple, heterogeneous data sources distributed over a computer network [ÖV11]. In addition to be heterogeneous and distributed, the data sources can be autonomous, i.e. controlled and managed independently (e.g. by a different database administrator) of the multidatabase system.

Since the data sources already exist, one is faced with the problem of providing integrated access to heterogeneous data. This requires data integration, which consists in defining a global schema for the multidatabase over the existing data and mappings between the global schema and the local data source schemas. Once data integration is done, the global schema can be used to express queries over multiple data sources as if it were a single (global) database.

Most of the work on multidatabase query processing has been done in the context of the mediator-wrapper architecture. This architecture and related techniques can be used for loosely-coupled multistore systems, which is why we introduce them. In the rest of this section, we describe the mediator-wrapper and multidatabase query processing architectures, and the query processing techniques.

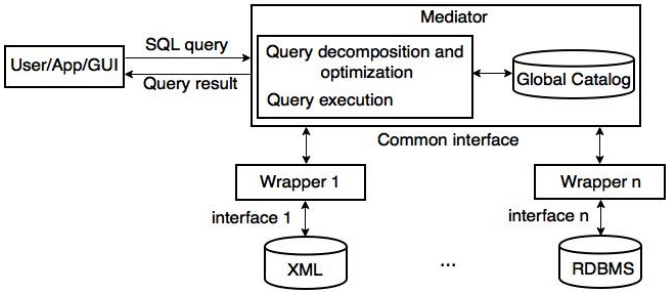
3.1 Mediator-Wrapper Architecture

In this architecture (see Figure 5), there is a clear separation of concerns: the mediator deals with data source distribution while the wrappers deal with data source heterogeneity and autonomy. This is achieved by using a common language between mediator and wrappers, and the translation to the data source language is done by the wrappers.

Each data source has an associated wrapper that exports information about the source schema, data and query processing capabilities. To deal with the heterogeneous nature of data sources, wrappers transform queries received from the mediator, expressed in a common query language, to the particular query language of the source. A wrapper supports the functionality of translating queries appropriate to the particular server, and reformatting answers (data) appropriate to the mediator. One of the major practical uses of wrappers has been to allow an SQL-based DBMS to access non SQL databases.

The mediator centralizes the information provided by the the wrappers in a unified view of all available data (stored in a global catalog). This unified view can be of two fundamental types [Len02]: local-as-view (LAV) and global-as-view (GAV). In LAV, the global schema definition exists, and each data source schema is treated as a view definition over it. In GAV on the other hand, the global schema is defined as a set of views over the data source schemas. These views indicate how the elements of the global schema can be derived, when needed, from the elements of the data source schemas. The main functionality of the mediator is to provide uniform access to multiple data sources and perform query decomposition and processing using the wrappers to access the data sources.

Figure 5 Mediator-Wrapper architecture



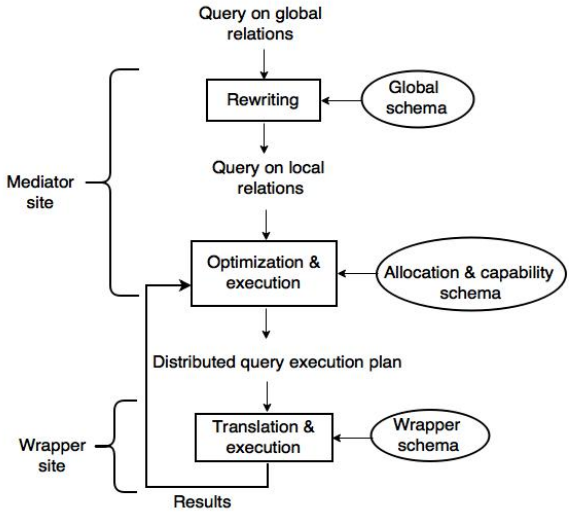
3.2 Multidatabase Query Processing Architecture

We assume the input is a query on relations expressed on a global schema in a declarative language, e.g. SQL. This query is posed on global relations, meaning that data distribution and heterogeneity are hidden. Three main layers are involved in multidatabase query processing.

The first two layers map the input query into an optimized query execution plan (QEP). They perform the functions of query rewriting, query optimization and some query execution. The first two layers are performed by the mediator and use meta-information stored in the global catalog (global schema, data source location, cost information, etc.). Query rewriting rewrites the input query into a query on local relations, using the global schema. Thus, the global schema provides the view definitions (i.e. GAV or LAV mappings between the global relations and the local relations stored in the data sources) and the query is rewritten using the views.

The second layer performs distributed query optimization and (some) execution by considering the location of the relations and the different query processing capabilities of the data sources exported by the wrappers. The distributed QEP produced by this layer groups within subqueries the operations that

Figure 6 Generic layering scheme for multidatabase query processing (modified after [ÖV11]).



can be performed by the data sources and wrappers. As in centralized DBMSs, query optimization can be static or dynamic. However, the lack of homogeneity in multidatabase systems (e.g. some data sources may have unexpected long delays in answering) make dynamic query optimization important. In the case of dynamic optimization, there may be subsequent calls to this layer after execution by the next layer. This is illustrated by the arrow showing results coming from the next layer. Finally, this layer integrates the results coming from the different wrappers to provide a unified answer to the users query. This requires the capability of executing some operations on data coming from the wrappers. Since the wrappers may provide very limited execution capabilities, e.g. in the case of very simple data sources, the mediator must provide the full execution capabilities to support the mediator interface.

The third layer performs query translation and execution using the wrappers. Then it returns the results to the mediator which can perform result integration from different wrappers and subsequent execution. Each wrapper maintains a wrapper schema that includes the local schema and mapping information to facilitate the translation of the input subquery (a subset of the QEP) expressed in a common language into the language of the data source. After the subquery is translated, it is executed by the data source and the local result is translated back in the common format.

3.3 Multidatabase Query Processing Techniques

The three main problems of query processing in multidatabase systems are: heterogeneous cost modeling, heterogeneous query optimization, to deal with different capabilities of data sources' DBMSs and adaptive query processing, to deal with strong variations in the environment (failures, unpredictable delays, etc.).

3.3.1 Heterogeneous Cost Modeling

Heterogeneous cost modeling refers to cost function definition, and the associated problem of obtaining cost-related information from the data sources. Such information is important to estimate the costs of executing subqueries at the data sources, which in turn are used to estimate the costs of alternative QEPs generated by the multidatabase query optimizer. There are three alternative approaches for determining the cost of executing queries in a multidatabase system: black-box, customized and dynamic.

The black-box approach treats the data sources as a black box, running some test queries on them, and from these determines the necessary cost information. It is based on running probing queries on data sources to determine cost information. Probing queries can, in fact, be used to gather a number of cost information factors. For example, probing queries can be issued to retrieve data from data sources to construct and update the multidatabase catalog. Statistical probing queries can be issued that, for example, count the number of tuples of a relation. Finally, performance measuring probing queries can be issued to measure the elapsed time for determining cost function coefficients.

The customized approach uses previous knowledge about the data sources, as well as their external characteristics, to subjectively determine the cost information. The basis for this approach is that the query processors of the data sources are too different to be represented by a unique cost model. It also assumes that the ability to accurately estimate the cost of local subqueries will improve global query optimization. The approach provides a framework to integrate the data sources cost model into the mediator query optimizer. The solution is to extend the wrapper interface such that the mediator gets some specific cost information from each wrapper. The wrapper developer is free to provide a cost model, partially or entirely.

The above approaches assume that the execution environment is stable over time. However, on the Internet for instance, the execution environment factors are frequently changing. The dynamic approach consists in monitoring the run-time behavior of data sources and dynamically collecting the cost information Three classes of environmental factors can be identified based on their dynamicity. The first class for frequently changing factors (every second to every minute) includes CPU load, I/O throughput, and available memory. The second class for slowly changing factors (every hour to every day) includes DBMS configuration parameters, physical data organization on disks, and database schema. The third class for almost stable factors (every month to every year) includes DBMS type, database location, and CPU speed. To face dynamic environments where network contention, data storage or available memory change over time, a solution is to extend the sampling method and consider user queries as new samples.

3.3.2 Heterogeneous Query Optimization

In addition to heterogeneous cost modeling, multidatabase query optimization must deal with the issue of the heterogeneous computing capabilities of data sources. For instance, one data source may support only simple select operations while another may support complex queries involving join and aggregate. Thus, depending on how the wrappers export such capabilities, query processing at the mediator level can be more or less complex. There are two main approaches to deal with this issue depending on the kind of interface between mediator and wrapper: query-based and operator-based.

Query-based Approach

In the query-based approach, the wrappers support the same query capability, e.g. a subset of SQL, which is translated to the capability of the data source. This approach typically relies on a standard DBMS interface such as Open Database Connectivity (ODBC) or its many variations (e.g. JDBC). Thus, since the data sources appear homogeneous to the mediator, query processing techniques designed for homogeneous distributed DBMS can be reused. However, if the data sources have limited capabilities, the additional capabilities must be implemented in the wrappers, e.g. join queries may need to be handled at the mediator, if the data source does not support join.

Since the data sources appear homogeneous to the mediator, a solution is to use a traditional distributed query optimization algorithm with a heterogeneous cost model. However, extensions are needed to convert the distributed execution plan into subqueries to be executed by the data sources and subqueries to be executed by the mediator. The hybrid two-step optimization technique is useful in this case: in a first step, a static plan is produced by a centralized cost-based query optimizer; in a second step, at startup time, an execution plan is produced by carrying out site selection and allocating the subqueries to the sites.

Operator-based Approach

In the operator-based approach, the wrappers export the capabilities of the data sources through compositions of relational operators. Thus, there is more flexibility in defining the level of functionality between the mediator and the wrapper. In particular, the different capabilities of the data sources can be made available to the mediator.

Expressing the capabilities of the data sources through relational operators allows tighter integration of query processing between mediator and wrappers. In particular, the mediator-wrapper communication can be in terms of sub plans. We illustrate the operator-based approach with planning functions proposed in the Garlic project [HKWY97]. In this approach, the capabilities of the data sources are expressed by the wrappers as planning functions that can be directly called by a centralized query optimizer. It extends a traditional query optimizer with operators to create temporary relations and retrieve locally stored data. It also creates the PushDown operator that pushes a portion of the work to the data sources where it will be executed.

The execution plans are represented, as usual, with operator trees, but the operator nodes are annotated with additional information that specifies the source(s) of the operand(s), whether the results are materialized, and so on. The Garlic operator trees are then translated into operators that can be directly executed by the execution engine. Planning functions are considered by the optimizer as enumeration rules. They are called by the optimizer to construct sub plans using two main functions: accessPlan to access a relation, and joinPlan to join two relations using access plans. There is also a join rule for *bind join*. A bind join is a nested loop join in which intermediate results (e.g. values for the join predicate) are passed from the outer relation to the wrapper for the inner relation, which uses these results to filter the data it returns. If the intermediate results are small and indexes are available at data sources, bindings can significantly reduce the amount of work done by a data source. Furthermore, bindings can reduce communication cost.

Using planning functions for heterogeneous query optimization has several advantages. First, planning functions provide a flexible way to express precisely the capabilities of data sources. In particular, they can be used to model non relational data sources such as web sites. Second, since these rules are declarative, they make wrapper development easier. Finally, this approach can be easily incorporated in an existing, centralized query optimizer.

The operator-based approach has also been used in DISCO, a multidatabase system designed to access data sources over the web [TRV98]. DISCO uses the GAV approach and an object data model to represent both mediator and data source schemas and data types. This allows easy introduction of new data sources with no type mismatch or simple type mismatch. The data source capabilities are defined as a subset of an algebraic machine (with the usual operators such as scan, join and union) that can be partially or entirely supported by the wrappers or the mediator. This gives much flexibility for the wrapper implementers to decide where to support data source capabilities (in the wrapper or in the mediator).

3.3.3 Adaptive Query Processing

Multidatabase query processing, as discussed so far, follows essentially the principles of traditional query processing whereby an optimal QEP is produced for a query based on a cost model, and then this QEP is executed. The underlying assumption is that the multidatabase query optimizer has sufficient knowledge about query runtime conditions in order to produce an efficient QEP and the runtime conditions remain stable during execution. This is a fair assumption for multidatabase queries with few data sources running in a controlled environment. However, this assumption is inappropriate for changing environments with large numbers of data sources and unpredictable runtime conditions as on the Web.

Adaptive query processing is a form of dynamic query processing, with a feedback loop between the execution environment and the query optimizer in order to react to unforeseen variations of runtime conditions. A query processing system is defined as adaptive if it receives information from the execution environment and determines its behavior according to that information in an iterative manner [AH00]. In

the context of multidatabase systems, the execution environment includes the mediator, wrappers and data sources. In particular, wrappers should be able to collect information regarding execution within the data sources.

Adaptive query processing adds to the traditional query processing process the following activities: monitoring, assessing and reacting. These activities are logically implemented in the query processing system by sensors, assessment components, and reaction components, respectively. These components may be embedded into control operators of the QEP, e.g. an Exchange operator. Monitoring involves measuring some environment parameters within a time window, and reporting them to the assessment component. The latter analyzes the reports and considers thresholds to arrive at an adaptive reaction plan. Finally, the reaction plan is communicated to the reaction component that applies the reactions to query execution.

4 Multistore Systems

Multistore systems provide integrated access to a number of cloud data stores such as NoSQL, RDBMS or HDFS, sometimes through a data processing framework such as Spark. They typically support only read-only queries, as supporting distributed transactions across data stores is a hard problem. We can divide multistore systems based on the level of coupling with the underlying data stores: loosely-coupled, tightly-coupled and hybrid. In this section, we introduce for each class a set of representative systems, with their architecture and query processing. We end the section with a comparative analysis.

In presenting these systems, we strive to use the same terminology we used so far in this paper. However, it is not easy as we often need to map the specific terminology used in the original papers and ours. When necessary, to help the reader familiar with some systems, we make precise this terminology mapping.

4.1 Loosely-Coupled Multistore Systems

Loosely-coupled multistore systems are reminiscent of multidatabase systems in that they can deal with autonomous data stores, which can be accessed through the multistore system common interface as well as separately through their local API. They follow the mediator-wrapper architecture with several data stores (e.g. NoSQL and RDBMS) as depicted in Figure 7. Each data store is autonomous, i.e. locally controlled, and can be accessed by other applications. Like web data integration systems that use the mediator-wrapper architecture, the number of data stores can be very high.

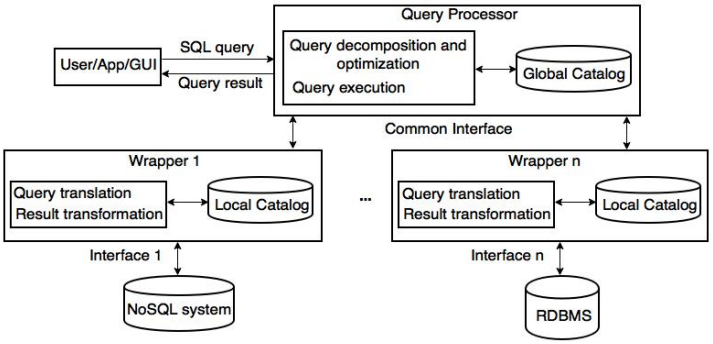
There are two main modules: one query processor and one wrapper per data store. The query processor has a catalog of data stores, and each wrapper has a local catalog of its data store. After the catalogs and wrappers have been built, the query processor can start processing input queries from the users, by interacting with wrappers. The typical query processing is as follows:

1. Analyze the input query and translate it into subqueries (one per data store), each expressed in a common language, and an integration subquery.

2. Send the subqueries to the relevant wrappers, which trigger execution at the corresponding data stores and translate the results into the common language format.

3. Integrate the results from the wrappers (which may involve executing operators such union and join), and return the results to the user. We describe below three loosely-coupled multistore systems: BigIntegrator, Forward and Qox.

Figure 7 Loosely-coupled multistore systems



BigIntegrator

BigIntegrator [ZR11] supports SQL-like queries that combines data in Bigtable data stores in the cloud and data in relational data stores. Bigtable is accessed through the Google Query Language (GQL), which has very limited query expressions, e.g. no join and only basic select predicates. To capture GQL's limited capabilities, BigIntegrator provides a novel query processing mechanism based on plugins, called absorber and finalizer, which enable to pre and post-process those operations that cannot be processed by Bigtable. For instance, a "LIKE" select predicate on a Bigtable or a join of two Bigtables will be processed through operations in BigIntegrator's query processor.

BigIntegrator uses the LAV approach for defining the global schema of the Bigtable and relational data sources as flat relational tables. Each Bigtable or relational data source can contain several collections, each represented as a source table of the form "table-name_source-name", where table-name is the name of the table in the global schema and source-name is the name of the data source. For instance, "Employees_A" represents an Employees table at source A, i.e. a local view of Employees. The source tables are referenced as tables in the SQL queries.

Figure 8 BigIntegrator

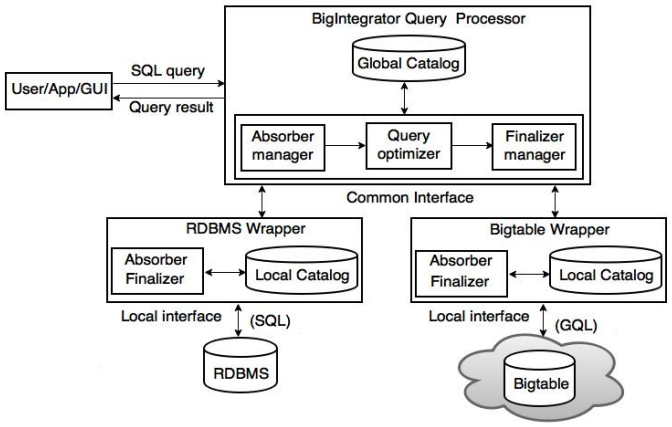


Figure 8 illustrates the architecture of BigIntegrator with two data sources, one relational database and one Bigtable data store. Each wrapper has an importer module and absorber and finalizer plug-ins. The importer creates the source tables and stores them in the local catalog. The absorber extracts a subquery, called access filter, from a user query that selects data from a particular source table, based on

the capabilities of the source. The finalizer translates each access filter (produced by the absorber) into an operator called interface function, specific for each kind of source. The interface function is used to send a query to the data source (i.e. a GQL or SQL query).

Query processing is performed in three steps, using an absorber manager, a query optimizer and a finalizer manager. The absorber manager takes the (parsed) user query and, for each source table referenced in the query, calls the corresponding absorber of its wrapper. In order to replace the source table with an access filter, the absorber collects from the query the source tables and the possible other predicates, based on the capabilities of the data source. The query optimizer reorders the access filters and other predicates to produce an algebra expression that contains calls to both access filters and other relational operators. It also performs traditional transformations such as select push down and bind join The finalizer manager takes the algebra expression and, for each access filter operator in the algebra expression, calls the corresponding finalizer of its wrapper. The finalizer transforms the access filters into interface function calls.

Finally, query execution is performed by the query processor that interprets the algebra expression, by calling the interface functions to access the different data sources and executing the subsequent relational operations, using in-memory techniques.

Forward

The Forward multistore system, or so-called Forward middleware in [OPV14], supports SQL++, an SQL-like language designed to unify the data model and query language capabilities of NoSQL and relational databases. SQL++ has a powerful, semi-structured data model that extends both the JSON and relational data models. FORWARD also provides a rich web development framework [FOPZ14], which exploits its JSON compatibility to integrate visualization components (e.g. Google Maps).

The design of SQL++ is based on the observation that the concepts are similar across both data models, e.g. a JSON array is similar to an SQL table with order, and an SQL tuple to a JSON object literal. Thus, an SQL++ collection is an array or a bag, which may contain duplicate elements. An array is ordered (similar to a JSON array) and each element is accessible by its ordinal position while a bag is unordered (similar to a SQL table). Furthermore, SQL++ extends the relational model with arbitrary composition of complex values and element heterogeneity. As in nested data models, a complex value can be either a tuple or collection. Nested collections can be accessed by nesting SELECT expressions in the SQL FROM clause or composed using the GROUP BY operator. They can also be unnested using the FLATTEN operator. And unlike an SQL table that requires all tuples to have the same attributes, an SQL++ collection may also contain heterogeneous elements comprising a mix of tuples, scalars, and nested collections.

Forward uses the GAV approach, where each data source (SQL or NoSQL) appears to the user as an SQL++ virtual view, defined over SQL++ collections. Thus, the user can issue SQL++ queries involving multiple virtual views. The Forward architecture is that of Figure 7, with a query processor and one wrapper per data source. The query processor performs SQL++ query decomposition, by exploiting the underlying data store capabilities as much as possible. However, given an SQL++ query that is not directly supported by the underlying data source, Forward will decompose it into one or more native queries that are supported and combine the native query results in order to compensate for the semantics or capabilities gap between SQL++ and the underlying data source. Although not described in the original paper [OPV14] cost-based optimization of SQL++ queries is possible, by reusing techniques from multidatabase systems when dealing with flat collections. However, it would be much harder considering the nesting and element heterogeneity capabilities of SQL++.

QoX

QoX [SWCD12] is a special kind of loosely-coupled multistore system, where queries are analytical data-driven workflows (or data flows) that integrate data from relational databases, and various execution

engines such as MapReduce or Extract-Transform-Load (ETL) tools. A typical data flow may combine unstructured data (e.g. tweets) with structured data and use both generic data flow operations like filtering, join, aggregation and user-defined functions like sentiment analysis and product identification. In a previous work [SWCD09], the authors proposed a novel approach to ETL design that incorporates a suite of quality metrics, termed QoX, at all stages of the design process. The QoX Optimizer deals with the QoX performance metrics, with the objective of optimizing the execution of dataflows that integrate both the back-end ETL integration pipeline and the front-end query operations into a single analytics pipeline.

The QoX Optimizer uses xLM, a proprietary XML-based language to represent data flows, typically created with some ETL tool. xLM allows capturing the flow structure, with nodes showing operations and data stores and edges interconnecting these nodes, and important operation properties such as operation type, schema, statistics, and parameters. Using appropriate wrappers to translate xLM to a tool-specific XML format and vice versa, the QoX Optimizer may connect to external ETL engines and import or export dataflows to and from these engines.

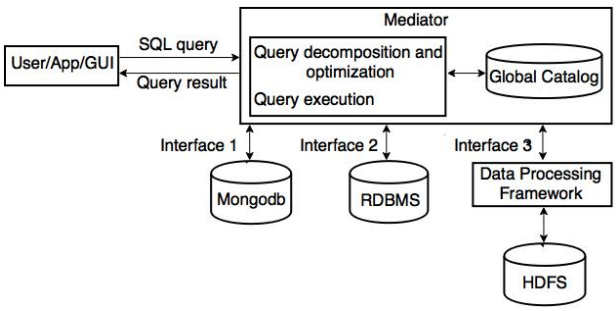
Given a data flow for multiple data stores and execution engines, the QoX Optimizer evaluates alternative execution plans, estimates their costs, and generates a physical plan (executable code). The search space of equivalent execution plans is defined by flow transformations that model data shipping (moving the data to where the operation will be executed), function shipping (moving the operation to where the data is), and operation decomposition (into smaller operations). The cost of each operation is estimated based on statistics (e.g. cardinalities, selectivities). Finally, the QoX Optimizer produces SQL code for relational database engines, Pig and Hive code for MapReduce engines, and creates Unix shell scripts as the necessary glue code for orchestrating different subflows running on different engines. This approach could be extended to access NoSQL engines as well, provided SQL-like interfaces and wrappers.

4.2 Tightly-Coupled Multistore Systems

Tightly-coupled multistore systems aim at efficient querying of structured and unstructured data for (big) data analytics. They may also have a specific objective, such as self-tuning or integration of HDFS and RDBMS data. However, they all trade autonomy for performance, typically in a shared-nothing cluster, so that data stores can only be accessed through the multistore system, directly through their local API.

Like loosely-coupled systems, they provide a single language for querying of structured and unstructured data. However, the query processor directly uses the data store local interfaces (see Figure 9), or in the case of HDFS, it interfaces a data processing framework such as MapReduce or Spark. Thus, during query execution, the query processor directly accesses the data stores. This allows efficient data movement across data stores. However, the number of data stores that can be interfaced is typically very limited.

Figure 9 Tightly-coupled multistore systems

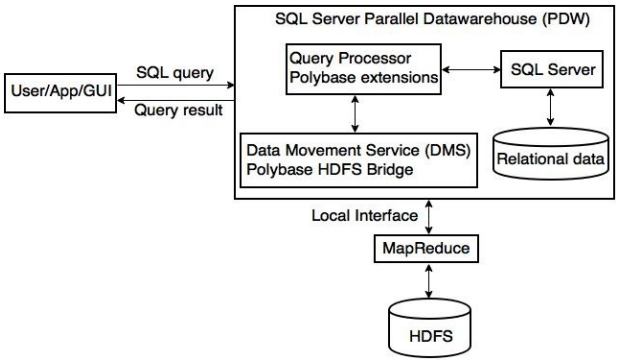


In the rest of this section, we describe three representative tightly-coupled multistore systems: Polybase, HadoopDB and Estocada. Two other interesting systems are Odyssey and JEN. Odyssey [HST+13] is a multistore system that can work with different analytic engines, such as parallel OLAP system or Hadoop. It enables storing and querying data within HDFS and RDBMS, using opportunistic materialized views, based on MISO [LSH+14]. MISO is a method for tuning the physical design of a multistore system (Hive/HDFS and RDBMS), i.e. deciding in which data store the data should reside, in order to improve the performance of big data query processing. The intermediate results of query execution are treated as opportunistic materialized views, which can then be placed in the underlying stores to optimize the evaluation of subsequent queries. JEN [YZÖ+15] is a component on top of HDFS to provide tight-coupling with a parallel RDBMS. It allows joining data from two data stores, HDFS and RDBMS, with parallel join algorithms, in particular, an efficient zigzag join algorithm, and techniques to minimize data movement. As the data size grows, executing the join on the HDFS side appears to be more efficient.

Polybase

Polybase [DHN+13] is a feature of the SQL Server Parallel Data Warehouse (PDW) product, which allows users to query unstructured (HDFS) data stored in a Hadoop cluster using SQL and integrate them with relational data in PDW. The HDFS data can be referenced in Polybase as external tables, which make the correspondence with the HDFS file on the Hadoop cluster, and thus be manipulated together with PDW native tables using SQL queries Polybase leverages the capabilities of PDW, a shared-nothing parallel DBMS. Using the PDW query optimizer, SQL operators on HDFS data are translated into MapReduce jobs to be executed directly on the Hadoop cluster. Furthermore, the HDFS data can be imported/exported to/from PDW in parallel, using the same PDW service that allows shuffling PDW data among compute nodes.

Figure 10 Polybase architecture



The architecture of Polybase, which is integrated within PDW, is shown in Figure 10. Polybase takes advantage of PDW's Data Movement Service (DMS), which is responsible for shuffling intermediate data across PDW nodes, e.g. to repartition tuples, so that any matching tuples of an equi-join be collocated at the same computing node that performs the join. DMS is extended with an HDFS Bridge component, which is responsible for all communications with HDFS. The HDFS Bridge enables DMS instances to also exchange data with HDFS in parallel (by directly accessing HDFS splits).

Polybase relies on the PDW cost-based query optimizer to determine when it is advantageous to push SQL operations on HDFS data to the Hadoop cluster for execution. Thus, it requires detailed statistics on external tables, which are obtained by exploring statistically significant samples of HDFS tables. The query optimizer enumerates the equivalent QEPs and select the one with least cost. The search space is obtained

by considering the different decompositions of the query into two parts: one to be executed as MapReduce jobs at the Hadoop cluster and the other as regular relational operators at the PDW side. MapReduce jobs can be used to perform select and project operations on external tables, as well as joins of two external tables. However, no bind join optimization is supported. The data produced by the MapReduce jobs can then be exported to PDW to be joined with relational data, using parallel hash-based join algorithms.

One strong limitation of pushing operations on HDFS data as MapReduce jobs is that even simple lookup queries have long latencies. A solution proposed for Polybase [VTP+14] is to exploit an index built on the external HDFS data using a B+-tree that is stored inside PDW. This method leverages the robust and efficient indexing code in PDW without forcing a dramatic increase in the space that is required to store or cache the entire (large) HDFS data inside PDW. Thus, the index can be used as a pre-filter by the query optimizer to reduce the amount of work that is carried out as MapReduce jobs. To keep the index synchronized with the data that is stored in HDFS, an incremental approach is used which records that the index is out-of-date, and lazily rebuilds it. Queries posed against the index before the rebuild process is completed can be answered using a method that carefully executes parts of the query using the index in PDW, and the remaining part of the query is executed as a MapReduce job on just the changed data in HDFS.

HadoopDB

The objective of HadoopDB [ABA+09] is to provide the best of both parallel DBMS (high-performance data analysis over structured data) and MapReduce-based systems (scalability, fault-tolerance, and flexibility to handle unstructured data) with an SQL-like language (HiveQL). To do so, HadoopDB tightly couples the Hadoop framework, including MapReduce and HDFS, with multiple single-node RDBMS (e.g. PostgreSQL or MySQL) deployed across a cluster, as in a shared-nothing parallel DBMS.

HadoopDB extends the Hadoop architecture with four components: database connector, catalog, data loader, and SQL-MapReduce-SQL (SMS) planner. The database connector provides the wrappers to the underlying RDBMS, using JDBC drivers. The catalog maintains information about the databases as an XML file in HDFS, and is used for query processing. The data loader is responsible for (re)partitioning (key, value) data collections using hashing on a key and loading the single-node databases with the partitions (or chunks). The SMS planner extends Hive, an Hadoop component that transforms HiveQL into MapReduce jobs that connect to tables stored as files in HDFS. This architecture yields a cost-effective parallel RDBMS, where data is partitioned both in RDBMS tables and in HDFS files, and the partitions can collocated at cluster nodes for efficient parallel processing.

Query processing is simple, relying on the SMS planner for translation and optimization, and MapReduce for execution. The optimization consists in pushing as much work as possible into the single node databases, and repartitioning data collections whenever needed. The SMS planner decomposes a HiveQL query to a QEP of relational operators. Then the operators are translated to MapReduce jobs, while the leaf nodes are again transformed into SQL to query the underlying RDBMS instances. In MapReduce, repartitioning should take place before the reduce phase. However, if the optimizer detects that an input table is partitioned on a column used as aggregation key for Reduce, it will simplify the QEP by turning it to a single Map-only job, leaving all the aggregation to be done by the RDBMS nodes. Similarly, repartitioning is avoided for equi-joins as well, if both sides of the join are partitioned on the join key.

Estocada

Estocada [BBD+15] is a self-tuning multistore system with the goal of optimizing the performance of applications that must deal with data in multiple data models, including relational, key-value, document and graph. To obtain the best possible performance from the available data stores, Estocada automatically distributes and partitions the data across the different data stores, which are entirely under its control and hence not autonomous. Hence, it is a tighly-coupled multistore system.

Data distribution is dynamic and decided based on a combination of heuristics and cost-based decisions, taking into account data access patterns as they become available. Each data collection is stored as a set of partitions, whose content may overlap, and each partition may be stored in any of the underlying data stores. Thus, it may happen that a partition is stored in a data store that has a different data model than its native one. To make Estocada applications independent of the data stores, each data partition is internally described as a materialized view over one or several data collections. Thus, query processing involves view-based query rewriting.

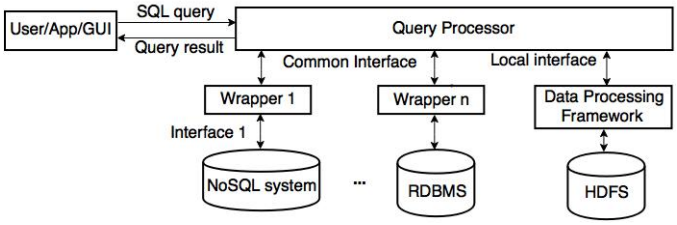
Estocada support two kinds of requests, for storing data and querying, with three main modules: storage advisor, catalog, query processor and execution engine. These components can directly access the data stores through their local interface. The query processor deals with single model queries only, each expressed in the query language of the corresponding data source. However, to integrate various data sources, one would need a common data model and language on top of Estocada. The storage advisor is responsible for partitioning data collections and delegating the storage of partitions to the data stores. For self-tuning the applications, it may also recommend repartitioning or moving data from one data store to the other, based on access patterns. Each partition is defined as a materialized view expressed as a query over the collection in its native language. The catalog keeps track of information about partitions, including some cost information about data access operations by means of binding patterns which are specific to the data stores.

Using the catalog, the query processor transforms a query on a data collection into a logical QEP on possibly multiple data stores (if there are partitions of the collection in different stores). This is done by rewriting the initial query using the materialized views associated with the data collection, and selecting the best rewriting, based on the estimated execution costs. The execution engine translates the logical QEP into a physical QEP which can be directly executed by dividing the work between the data stores and Estocada's runtime engine, which provides its own operators (select, join, aggregate, etc.).

4.3 Hybrid systems

Hybrid systems try to combine the advantages of loosely-coupled systems, e.g. accessing many different data stores, and tightly-coupled systems, e.g. accessing some data stores directly through their local interface for efficient access. Therefore, the architecture (see Figure 11) follows the mediator-wrapper architecture, while the query processor can also directly access some data stores, e.g. HDFS through MapReduce or Spark.

Figure 11 Hybrid architecture



We describe below the three hybrid multistore systems: Spark SQL, CloudMdsQL and BigDAWG.

Spark SQL

Spark SQL [AXL+15] is a recent module in Apache Spark that integrates relational data processing with Spark's functional programming API. It supports SQL-like queries that can integrate HDFS data accessed

through Spark and external data sources (e.g. relational databases) accessed through a wrapper. Thus, it is a hybrid multistore system with tight-coupling of Spark/HDFS and loose-coupling of external data sources.

Spark SQL uses a nested data model that includes tables and DataFrames. It supports all major SQL data types, as well as user-defined types and complex data types (structs, arrays, maps and unions), which can be nested together. A DataFrame is a distributed collection of rows with the same schema, like a relational table. It can be constructed from a table in an external data source or from an existing Spark RDD of native Java or Python objects. Once constructed, DataFrames can be manipulated with various relational operators, such as WHERE and GROUPBY, which take expressions in procedural Spark code.

Figure 12 Spark SQL architecture

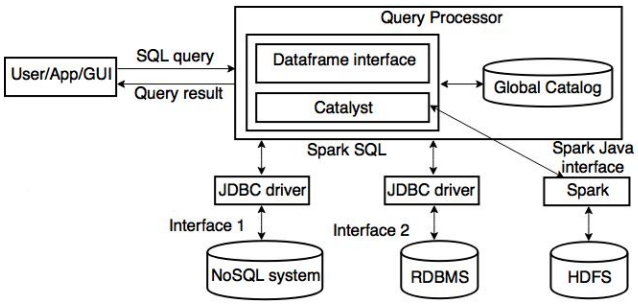


Figure 12 shows the architecture of Spark SQL, which runs as a library on top of Spark, The query processor directly accesses the Spark engine through the Spark Java interface, while it accesses external data sources (e.g. an RDBMS or a key-value store) through the Spark SQL common interface supported by wrappers (JDBC drivers). The query processor includes two main components: the DataFrame API and the Catalyst query optimizer. The DataFrame API offers tight integration between relational and procedural processing, allowing relational operations to be performed on both external data sources and Spark's RDDs. It is integrated into Spark's supported programming languages (Java, Scala, Python) and supports easy inline definition of user-defined functions, without the complicated registration process typically found in other database systems. Thus, the DataFrame API lets developers seamlessly mix relational and procedural programming, e.g. to perform advanced analytics (which is cumbersome to express in SQL) on large data collections (accessed though relational operations).

Catalyst is an extensible query optimizer that supports both rule-based and cost-based optimization. The motivation for an extensible design is to make it easy to add new optimization techniques, e.g. to support new features of Spark SQL, as well as to enable developers to extend the optimizer to deal with external data sources, e.g. by adding data source specific rules to push down select predicates. Although extensible query optimizers have been proposed in the past, they have typically required a complex language to specify rules, and a specific compiler to translate the rules into executable code. In contrast, Catalyst uses standard features of the Scala functional programming language, such as pattern-matching, to make it easy for developers to specify rules, which can be complied to Java code.

Catalyst provides a general transformation framework for representing query trees and applying rules to manipulate them. This framework is used in four phases: (1) query analysis, (2) logical optimization, (3) physical optimization, and (4) code generation. Query analysis resolves name references using a catalog (with schema information) and produces a logical plan. Logical optimization applies standard rule-based optimizations to the logical plan, such as predicate pushdown, null propagation, and Boolean expression simplification. Physical optimization takes a logical plan and enumerates a search space of equivalent physical plans, using physical operators implemented in the Spark execution engine or in the external data sources. It then selects a plan using a simple cost model, in particular, to select the join algorithms. Code generation relies on the Scala language, in particular, to ease the construction of abstract syntax trees

(ASTs) in the Scala language. ASTs can then be fed to the Scala compiler at runtime to generate Java bytecode to be directly executed by compute nodes.

To speed up query execution, Spark SQL exploits in-memory caching of hot data using a columnar storage (i.e. storing data collections as sections of columns of data rather than as rows of data). Compared with Spark's native cache, which simply stores data as Java native objects, this columnar cache can reduce memory footprint by an order of magnitude by applying columnar compression schemes (e.g. dictionary encoding and run-length encoding). Caching is particularly useful for interactive queries and for the iterative algorithms common in machine learning.

CloudMdsQL

The CloudMdsQL multistore system [KVB$^+$15, KBV$^+$16] supports a powerful functional SQL-like language, designed for querying multiple heterogeneous data sources (e.g. relational and NoSQL) A CloudMdsQL query may contain nested subqueries, and each subquery addresses directly a particular data store and may contain embedded invocations to the data store native query interface. Thus, the major innovation is that a CloudMdsQL query can exploit the full power of local data stores, by simply allowing some local data store native queries (e.g. a breadth-first search query against a graph database) to be called as functions, and at the same time be optimized based on a simple cost model, CloudMdsQL has been extended [BKLV15] to address distributed processing frameworks such as Apache Spark by enabling the ad-hoc usage of user defined map/filter/reduce operators as subqueries.

The CloudMdsQL language is SQL-based with the extended capabilities for embedding subqueries expressed in terms of each data store's native query interface. The common data model is table-based, with support of rich datatypes that can capture a wide range of the underlying data store datatypes, such as arrays and JSON objects, in order to handle non-flat and nested data, with basic operators over such composite datatypes. CloudMdsQL allows named table expressions to be defined as Python functions, which is useful for querying data stores that have only API-based query interface. A CloudMdsQL query is executed in the context of an ad-hoc schema, formed by all named table expressions within the query. This approach fills the gap produced by the lack of a global schema and allows the query compiler to perform semantic analysis of the query.

The design of the CloudMdsQL query engine [KBV$^+$16] takes advantage of the fact that it operates in a cloud platform, with full control over where the system components can be installed. The architecture of the query engine is fully distributed, so that query engine nodes can directly communicate with each other, by exchanging code (query plans) and data. This distributed architecture yields important optimization opportunities, e.g. minimizing data transfers by moving the smallest intermediate data for subsequent processing by one particular node. Each query engine node consists of two parts – master and worker – and is collocated at each data store node in a computer cluster. Each master or worker has a communication processor that supports send and receive operators to exchange data and commands between nodes. A master takes as input a query and produces, using a query planner and calatog (with metadata and cost information on data sources) a query plan, which it sends to one chosen query engine node for execution. Each worker acts as a lightweight runtime database processor atop a data store and is composed of three generic modules (i.e. same code library) - query execution controller, operator engine, and table storage - and one wrapper module that is specific to a data store.

The query planner performs cost-based optimization. To compare alternative rewritings of a query, the optimizer uses a simple catalog, which provides basic information about data store collections such as cardinalities, attribute selectivities and indexes, and a simple cost model. Such information can be exposed by the wrappers in the form of cost functions or database statistics. The query language also provides a possibility for the user to define cost and selectivity functions whenever they cannot be derived from the catalog, mostly in the case of using native subqueries. The search space of alternative plans is obtained using traditional transformations, e.g. by pushing down select predicates, using bind join, performing join ordering, or planning intermediate data shipping.

BigDAWG

Like multidatabase systems, all the multistore systems we have seen so far provide transparent access across multiple data stores with the same data model and language. The BigDAWG (Big Data Analytics Working Group) multistore system (called polystore) [DES+15] takes a different path, with the goal of unifying querying over a variety of data models and languages, Thus, there is no common data model and language. A key user abstraction in BigDAWG is an island of information, which is a collection of data stores accessed with a single query language. And there can be a variety of islands, including relational (RDBMS), Array DBMS, NoSQL and Data Stream Management System (DSMS). Within an island, there is loose-coupling of the data stores, which need to provide a wrapper (called shim) to map the island language to their native one. When a query accesses more than one data store, objects may have to be copied between local databases, using a CAST operation, which provides a form of tight-coupling. This is why BigDAWG can be viewed as a hybrid multistore system.

The architecture of BigDAWG is highly distributed, with a thin layer that interfaces the tools (e.g. visualization) and applications, with the islands of information. Since there is no common data model and language, there is no common query processor either. Instead, each island has its specific query processor. Query processing within an island is similar to that in multidatabase systems: most of the processing is pushed to the data stores and the query processor only integrates the results. The query optimizer does not use a cost model, but heuristics and some knowledge of the high performance of some data stores. For simple queries, e.g. select-project-join, the optimizer will use function shipping, in order to minimize data movement and network traffic among data stores. For complex queries, e.g. analytics, the optimizer may consider data shipping, to to move the data to a data store that provides a high-performance implementation.

A query submitted to an island may involve multiple islands. In this case, the query must be expressed as multiple subqueries, each in a specific island language. To specify the island for which a subquery is intended, the user encloses the subquery in a SCOPE specification. Thus, a multi-island query will have multiple scopes to indicate the expected behavior of its subqueries. Furthermore, the user may insert CAST operations to move intermediate datasets between islands in an efficient way. Thus, the multi-island query processing is dictated by the way the subqueries, SCOPE and CAST operations are specified by the user.

4.4 Comparative Analysis

The multistore systems we presented above share some similarities, but do have important differences. The objective of this section is to compare these systems along important dimensions and identify the major trends. We divide the dimensions between functionality and implementation techniques.

Table 1 compares the functionality of multistore systems along four dimensions: objective, data model, query language, and data stores that are supported. Although all multistore systems share the same overall goal of querying multiple data stores, there are many different paths toward this goal, depending on the functional objective to be achieved. And this objective has important impact on the design choices. The major trend that dominates is the ability to integrate relational data (stored in RDBMS) with other kinds of data in different data stores, such as HDFS (Polybase, HadoopDB, SparkSQL, JEN) or NoSQL (BigTable only for BigIntegrator, document stores for Forward). Thus, an important difference lies in the kind of data stores that are supported. For instance, Estocada, BigDAWG and CLoudMdsQL can support a wide variety of data stores while Polybase and JEN target the integration of RDBMS with HDFS only. We can also note the growing importance of accessing HDFS within Hadoop, in particular, with MapReduce or Spark, which corresponds to major use cases in structured/unstructured data integration.

Another trend is the emergence of self-tuning multistore systems, such as Estocada and Odyssey, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational/ SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data stores

Table 1 Functionality of multistore systems.

Mutistore system	*Objective*	*Data model*	*Query language*	*Data stores*
Loosely-coupled				
BigIntegrator	Querying relational and cloud data	Relational	SQL-like	BigTable,RDBMS
Forward	Unifying relational and NoSQL	JSON-based	SQL++	RDBMS, NoSQL
QoX	Analytic data flows	Graph	XML-based	RDBMS, MapReduce, ETL
Tightly-coupled				
Polybase	Querying Hadoop from RDBMS	Relational	SQL	HDFS, RDBMS
HadoopDB	Querying RDBMS from Hadoop	Relational	SQL-like (HiveQL)	HDFS, RDBMS
Estocada	Self-tuning	No common model	Native query languages	RDBMS, NoSQL
Hybrid				
SparkSQL	SQL on top of Spark	Nested	SQL-like	HDFS, RDBMS
BigDAWG	Unifying relational and NoSQL	No common model	Island query languages, with CAST and SCOPE operators	RDBMS, NoSQL, Array DBMS, DSMSs
CloudMdsQL	Querying relational and NoSQL	JSON-based	SQL-like with native subqueries	RDBMS, NoSQL HDFS

to be directly accessed with their native (or island) languages. CloudMdsQL also allows native queries, but as subqueries within an SQL-like language.

Table 2 Implementation techniques of multistore systems.

Mutistore system	*Special modules*	*Schemas*	*Query processing*	*Query optimization*
Loosely-coupled				
BigIntegrator	Importer, absorber, finalizer	LAV	Access filters	Heuristics
Forward	Query processor	GAV	Data store capabilities	Cost-based
QoX	Dataflow engine	No	Data/function shipping, operation decomposition	Cost-based
Tightly-coupled				
Polybase	HDFS bridge	GAV	Query splitting	Cost-based
HadoopDB	SMS planer, db connector	GAV	Query splitting	Heuristics
Estocada	Storage advisor	Materialized views	View-based query rewriting	Cost-based
Hybrid				
SparkSQL	Catalyst extensible optimizer	Data frames	In-memory caching using columnar storage	Cost-based
BigDAWG	Island query processors	GAV within islands	Function/data shipping	Heuristics
CloudMdsQL	Query planner	No	Bind join	Cost-based

Table 2 compares the implementation techniques of multistore systems along four dimensions: special modules, schema management, query processing, and query optimization. The first dimension captures the system modules that either refine those of the generic architecture (e.g. importer, absorber and finalizer, which refine the wrapper module, Catalyst extensible optimizer or QoX's data flow engine, which replace the query processor) or bring new functionality (e.g. Estocada's storage advisor). Most multistore systems provide some support for managing a global schema, using the GAV or LAV approaches, with some variations (e.g. BigDAWG uses GAV within (single model) islands of information). However, QoX, Estocada, SparkSQL and CloudMdsQL do not support global schemas, although they provide some way to deal with the data stores local schemas.

The query processing techniques are extensions of known techniques from distributed database systems, e.g. data/function shipping, query decomposition (based on the data stores's capabilities, bind join, select pushdown). Query optimization is also supported, with either a (simple) cost model or heuristics.

5 Conclusion

Building cloud data-intensive applications often requires using multiple data stores (NoSQL, HDFS, RDBMS, etc.), each optimized for one kind of data and tasks. In particular, many use cases exhibit the need to combine loosely structured data (e.g. log files, tweets, web pages) which are best supported by HDFS or NoSQL with more structured data in RDBMS. However, the wide diversification of data store interfaces makes it difficult to access and integrate data from multiple data stores.

Although useful, the solutions devised for multidatabase systems (also called federated database systems) or Web data integration systems need be extended in major ways to deal with the specific context of the cloud. This has motivated the design of multistore systems (also called polystores) that provide integrated or transparent access to a number of cloud data stores through one or more query languages. As NoSQL and related technologies such as Hadoop and Spark, multistore systems is a recent, important topic in data management, and we can expect much evolution in the coming years.

In this paper, we gave an overview of query processing in multistore systems, focusing on the main solutions and trends. We started by introducing cloud data management, including distributed file systems such as HDFS, NoSQL systems and data processing frameworks (such as MapReduce and Spark) and query processing in multidatabase systems. Then, we described and analyzed representative multistore systems, based on their architecture, data model, query languages and query processing techniques. To ease comparison, we divided multistore systems based on the level of coupling with the underlying data stores, i.e. loosely-coupled, tightly-coupled and hybrid.

We analyzed three multistore systems for each class: BigIntegrator, Forward and QoX (loosely-coupled); Polybase, HadoopDB and Estocada (tightly-coupled); SparkSQL, BigDAWG and CloudMdsQL (hybrid). Our comparisons reveal several important trends. The major trend that dominates is the ability to integrate relational data (stored in RDBMS) with other kinds of data in different data stores, such as HDFS or NoSQL. However, an important difference between multistore systems lies in the kind of data stores that are supported. We also note the growing importance of accessing HDFS within Hadoop, in particular, with MapReduce or Spark. Another trend is the emergence of self-tuning multistore systems, with the objective of leveraging the available data stores for performance. In terms of data model and query language, most systems provide a relational/ SQL-like abstraction. However, QoX has a more general graph abstraction to capture analytic data flows. And both Estocada and BigDAWG allow the data stores to be directly accessed with their native (or island) languages.

The query processing techniques are extensions of known techniques from distributed database systems, e.g. data/function shipping, query decomposition (based on the data stores capabilities, bind join, select pushdown). And query optimization is supported, with either a (simple) cost model or heuristics.

Since multistore systems is a relatively recent topic, there are important research issues ahead of us, which we briefly discuss.

- **Query languages.** Designing a query language for a multistore system is a major issue as there is a subtle trade-off between ease of use for the upper layers (e.g. analytics) and efficient access to the data stores. An SQL-like query language as provided by most current multistore systems will make it easier the integration with standard analytics tools, but at the expense of efficiency. An alternative towards more efficiency (at the expense of ease of use) is to allow the data stores to be directly accessed with their native (or island) languages as with Estocada and BigDAWG. A compromise is to use a functional query language that allows native subqueries as functions within SQL-like queries as in CloudMdsQL.

- **Query optimization.** This issue is directly related to the query language issue. With an SQL-like query language , it is possible to rely on a simple cost model and simple heuristics such as using bind join or select push down. However, updating the cost model or adding heuristics as new data stores are added may be difficult. An interesting solution is extensible query optimization as in the SparkSQL Catalyst. Finally, using the native data store languages directly makes the issue difficult, as native queries should be treated as a black box.

- **Distributed transactions.** Multistore systems have focused on read-only queries as it satisfies the requirements of analytics. However, as more ad more complex cloud data-intensive are built, the need for updating data across data stores will become important. Thus, the need for distributed transactions will arise. The issue is much harder as the transaction models of the data stores may be very different. In particular, most NoSQL data stores do not provide ACID transaction support.

- **Efficient data movement.** Exchanging data between data stores and the multistore system must be made efficient in order to deal with big data. Data could also be moved and copied across data stores as in Estocada and BigDAWG. To make data movement efficient will require clever data transformation techniques and the use of new memory access techniques, such as Remote Direct Memory Access.

- **Automatic load balancing.** If efficient data movement is provided across data stores, then a related issue is automatic load balancing, in order to maximize the performance of cloud data-intensive applications. This requires the development of a real-time monitoring system of resource usage integrated with all the components of the platform (the query processor and the underlying data stores).

Acknowledgements

This work was partially funded by the European Commission under the Integrated Project Coherent PaaS.

References

[ABA+09] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Rasin, and A. Silberschatz. Hadoopdb: An architectural hybrid of mapreduce and DBMS technologies for analytical workloads. *Proceedings of the Very Large Data Bases (PVLDB)*, 2(1):922–933, 2009.

[AH00] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD Int. Conf. on Data Management*, pages 261–272, 2000.

[AMH08] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *ACM SIGMOD Int. Conf. on Data Management*, pages 967–980, 2008.

[AXL+15] M. Armbrust, R. Xin, C. Lian, Y. Huai, D. Liu, J. Bradley, X. Meng, T. Kaftan, M. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *ACM SIGMOD Int. Conf. on Data Management*, pages 1383–1394, 2015.

[BBD+15] F. Bugiotti, D. Bursztyn, A. D., I. Ileana, and I. Manolescu. Invisible glue: Scalable self-tuning multi-stores. In *Int. Conf. on Innovative Data Systems Research (CIDR)*, page 7, 2015.

[BKLV15] C. Bondiombouy, B. Kolev, O. Levchenko, and P. Valduriez. Integrating big data and relational data with a functional sql-like query language. In *Int. Conf. on Database and Expert Systems Applications (DEXA)*, pages 170–185, 2015.

[Bru14] R. Van Bruggen. *Learning Neo4j*. Packt Publishing Limited, 2014.

[CDG+08] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[DES+15] J. Duggan, A. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, and S. Zdonik. The bigdawg polystore system. *ACM SIGMOD Int. Conf. on Data Management*, 44(2):11–16, 2015.

[DG04] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[DHI12] A. Doan, A. Y. Halevy, and Z. G. Ives. *Principles of Data Integration*. Morgan Kaufmann, 2012.

[DHJ+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[DHN+13] D. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, and J. Gramling. Split query processing in polybase. In *ACM SIGMOD Int. Conf. on Data Management*, pages 1255–1266, 2013.

[EST+13] S. Ewen, S. Schelter, K. Tzoumas, D. Warneke, and V. Markl. Iterative parallel data processing with stratosphere: an inside look. In *ACM SIGMOD Int. Conf. on Data Management*, pages 1053–1056, 2013.

[FOPZ14] Yupeng Fu, Kian Win Ong, Yannis Papakonstantinou, and Erick Zamora. FORWARD: data-centric uis using declarative templates that efficiently wrap third-party javascript components. *Proceedings of the Very Large Data Bases (PVLDB)*, 7(13):1649–1652, 2014.

[GGL03] S. Ghemawat, H. Gobioff, and S. Leung. The google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, 2003.

[HCK+08] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Martí, and E. Cesario. The xtreemfs architecture - a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the Very Large Data Bases (PVLDB)*, pages 276–285, 1997.

[HST+13] H. Hacigümüs, J. Sankaranarayanan, J. Tatemura, J. LeFevre, and N. Polyzotis. Odyssey: A multi-store system for evolutionary analytics. *Proceedings of the Very Large Data Bases (PVLDB)*, 6(11):1180–1181, 2013.

[KBV+16] B. Kolev, C. Bondiombouy, P. Valduriez, R. Jimenez-Peris, R. Pau, and J. Pereira. The cloudmdsql multistore system. *ACM SIGMOD/PODS (Principles of Database Systems) Conf.*, page 4, 2016.

[KVB+15] B. Kolev, P. Valduriez, C. Bondiombouy, R. Jimenez-Peris, R. Pau, and J. Pereira. Cloudmdsql: Querying heterogeneous cloud data stores with a common language. *Distributed and Parallel Databases*, page 41, 2015.

[Len02] M. Lenzerini. Data integration: A theoretical perspective. In *ACM SIGMOD/PODS (Principles of Database Systems) Conf.*, pages 233–246, 2002.

[LSH+14] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and J. Carey. MISO: souping up big data query processing with a multistore system. In *ACM SIGMOD Int. Conf. on Data Management*, pages 1591–1602, 2014.

[NPP13] A. Nayak, A. Poriya, and D. Poojary. Type of nosql databases and its comparison with relational databases. *Int. Journal of Applied Information Systems*, 5(4):16–19, 2013.

[OPV14] K. Win Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. *ACM Computing Research Repository (CoRR)*, abs/1405.3631, 2014.

[ÖV11] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011.

[PAD12] E. Pacitti, R. Akbarinia, and M. El Dick. *P2P Techniques for Decentralized Applications*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[PHM10] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB:The NoSQL Database for Cloud and Desktop Computing*. Apress, 2010.

[Ram09] R. Ramakrishnan. Data management in the cloud. In *IEEE Int. Conf. on Data Engineering*, page 5, 2009.

[SAD+10] M. Stonebraker, D. Abadi, D. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Communications of the ACM*, 53(1):64–71, 2010.

[SKWH76] M. Stonebraker, P. Kreps, W. Wong, and G. Held. The design and implementation of ingres. *ACM Trans. on Database Systems*, 1(3):198–222, 1976.

[Sto81] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, 1981.

[SWCD09] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Qox-driven ETL design: reducing the cost of ETL consulting engagements. In *ACM SIGMOD Int. Conf. on Data Management*, pages 953–960, 2009.

[SWCD12] A. Simitsis, K. Wilkinson, M. Castellanos, and U. Dayal. Optimizing analytic data flows for multiple execution engines. In *ACM SIGMOD Int. Conf. on Data Management*, pages 829–840, 2012.

[TRV98] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. Knowl. Data Eng.*, 10(5):808–823, 1998.

[VTP+14] V.Gankidi, N. Teletia, J. Patel, A. Halverson, and D. DeWitt. Indexing HDFS data in PDW: splitting the data from the index. *Proceedings of the Very Large Data Bases (PVLDB)*, 7(13):1520–1528, 2014.

[WBM+06] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[Whi12] T. White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale*. O'Reilly, 2012.

[YZÖ+15] T. Yuanyuan, T. Zou, F. Özcan, R. Gonscalves, and H. Pirahesh. Joins for hybrid warehouses: Exploiting massive parallelism and enterprise data warehouses. In *Int. Conf. on Extending Database Technology (EDBT)*, pages 373–384, 2015.

[ZCF+10] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, pages 10–10, 2010.

[ZR11] M. Zhu and T. Risch. Querying combined cloud-based and relational databases. In *Int. Conf. on Cloud and Service Computing (CSC)*, pages 330–335, 2011.