netherlands eScience center

Modern Data Management Solutions

Author: Milena Ivanova

Version: 08.07.2014

Abstract:

Modern scientific disciplines create data collections with enormous size and variety from sensor measurements, advanced instruments, model simulations, and research publications. Efficient data management solutions are crucial to keep abreast in eScience and enable new scientific discoveries.

This document comprises a short overview of modern data management technology. We describe the most important functionality and limitations of widely-spread solutions with the purpose to provide a guideline for choosing of a data management solution suitable for a concrete eScience project needs.

Contents

Introduction

Modern scientific disciplines create data collections with enormous size and variety from sensor measurements, advanced instruments, model simulations, and research publications. Efficient data management solutions are crucial to keep abreast in eScience and facilitate new scientific discoveries.

The current NLeSC project portfolio is characterized with a rich variety of data sources and processing needs and there is no single solution fitting them all. In the scientific data management world in general, and in the NLeSC projects in particular, there is a variety of solutions. It is our believe that the selection of a suitable data management component of the overall architecture is an important prerequisite for the success of a eScience projects.

Many traditional solutions are entirely file-based: the data are stored in a file repository and processed by tailored software. Another common approach uses a mixed architecture: only the metadata are put in a database, while raw data are kept in file repositories and glued together via table columns with respective file locations. Users and applications need to first consult the database to locate datasets of interest that are accessed in the next stage.

In this document we focus on two categories of systems that allow for storing and managing the raw data: database management systems (DBMSs) for data with well defined structure, and NoSQL systems for unstructured/semistructured data.

File-based and Mixed Solutions

File-base solutions are very common in the scientific world, especially for the so called “long tail” of scientific applications. Files are used for both raw measurement data and sometimes for the metadata, which may describe the acquisition method, structure and semantics of the raw data. Often, the metadata are partially encoded in the file names and directory structure. This approach is easy to implement and with a small start-up cost which makes it a logical choice for small data sets.

With the growth of the file-based repositories mixed solutions became the larger scientific community’s preferred tool to standardize file structures and manage metadata. The metadata are put in a database, which facilitates browsing through the data set and discovering of the sub-sets of interest. The raw data are kept in file repositories and glued together via table columns with respective file locations. Users and applications need to first consult the database to locate data sets needed and access the raw data in the next stage. Often the gluing functionality is provided by a special software layer called middleware.

One of the issues and potential problems of the mixed solutions is maintaining data integrity, i.e. provide proper update of the metadata database each time new files are added to the repository, and ensure that the links in the file location columns do not refer to missing or moved files.

Important limitations of file-based and mixed systems are inflexibility for exploration and handling of new requirements, inefficient access, limited scalability, and lack of support for concurrent usage by multiple users. If the user wants to perform a new kind of analysis, a new application needs to be developed or an existing one adjusted, (re)compiled, and installed. If the file format does not support indexing and accessing small data units, processing may involve unnecessary access to large data volumes in order to retrieve a small interesting data item (e.g. sequencing data in BAM/SAM format). Specialized software tools are often limited to data sizes fitting in memory (e.g. cross-match of FITS astronomical catalogs in the Java-based TopCat tool). Their application to large data sets requires additional engineering effort to chunk the data and schedule piece-wise processing. In some scientific domains the solution perceived as the best is to buy a machine with big enough memory.

Database Management Systems

Characteristics

Relational database management systems (RDBMSs) are well-established technology for persistent storage, management, and retrieval of data sets. Below we highlight the main characteristics of a RDBMS: relational data model, the declarative language SQL, data independence, and support for concurrent data access and manipulation.

Data model

Relational data model represents data as tables of rows and columns. Different real-world entities are represented as different tables. An entity set can have various properties represented as table columns. Each entity instance is stored as a table row. Typically one or several of the columns form a key (called primary key) that uniquely identifies all rows in the table. The relationships between entities are

represented in an abstract way through data values. Columns in a related table, called foreign key, contain values of the primary key columns in the referred table, hereby establishing the connection. The DBMS software provides automatic support of these features, for instance uniqueness of the primary key and valid references of the foreign keys.

Query language

The Structured Query Language (SQL) is a high-level declarative language. It is used to specify **what** data are to be retrieved as opposed to **how** the system should do it. The database engine provides itself the answer of the **how** question with the help of the query optimizer. It is a very important component that, given a declarative query, creates different, but logically equivalent, executable plans, estimates their costs, and chooses the best one for execution.

The SQL language contains data definition language (DDL) and data manipulation language (DML). The DDL is used to define the **database schema**: the structure of the data. It allows for creation of tables, indices, and abstract views. It allows the database designer to define various constraints for data consistency (e.g. the value of age should not be negative). The DML offers operations to insert, update, delete, and query data. The queries can include clauses for filtering of data rows based on logical predicates, joining of multiple tables, and aggregations over entire table or groups of rows.

Data independence

The relational model and its language provide an abstraction from the physical storage level. For example, a table can be physically re-organized by sorting it on a given column, or a new index can be created for it without any change of the applications that use it.

Transactions

Multiple applications and users can concurrently access and update the data in a safe way. The transactions in a DBMS, i.e. operations modifying data, have the so called ACID guarantee: **A**tomicity, **C**onsistency, **I**solation, and **D**urability.

- Atomicity: when a transaction is composed of several individual operations they are executed as an atomic operation: all or none.
- Consistency: Each transaction leaves the system in a consistent state.
- Isolation: multiple transactions execute without interfering with each other, as if each of them is the only transaction in the system. This is achieved through transaction serialization.
- Durability: once a transaction is completed, its result is safely stored and recoverable if the system crashes.

Row-stores and column-stores

There are two principle ways to linearize a two-dimensional table into an one-dimensional storage structure, which gives two major classes of RDBMSs. Row-stores keep data organized row by row. They are very efficient if all columns of a row have to be retrieved and are preferred for workloads of small horizontal transactions that operate on one or a couple of rows, e.g. withdrawal from a bank account. Column-stores organize data column by column. They are efficient for aggregations over entire data sets and are preferred for analytical workloads, such as total amount of withdrawals at branch X in month Y.

Most database vendors focused for many years on the development of row-store solutions. We observe in the last decade an expansion of product catalogs with column-store alternatives targeting the market of analytical applications and business intelligence. Some hybrid solutions are also available.

RDBMS on the market

The market of RDBMS is very mature. The most popular commercial systems are Oracle, IBM DB2, and MS SQL Server. Some successful new products are Teradata, Greenplum, Vertica, etc. The most mature open source systems are PostgreSQL, MonetDB, and MySQL.

When not to use RDBMS

There are a number of situations when relational DBMSs experience limitations in performance or functionality and are probably not the best match:

- Application data does not fit well the relational data model and cannot be easily represented as tables. Examples of those are high-dimensional arrays in many scientific and engineering domains, text sources and multimedia documents.
- Data are not well structured or the schema is often changed. The strict relational schema can be too rigid for such applications.
- The application requires stable and predictable performance. The abstract relational model and SQL allow for specifying arbitrary complex queries whose execution time can be unpredictable. If strict service level agreements are required for an application, predictable behavior and performance of the system might be more important than the best possible execution strategy.
- Table joins are expensive if the data is distributed across multiple computers. This can limit the horizontal scalability.
- Data refinery. If a raw data set needs to undergo a one-time expensive processing but is not queried on a regular basis, the overhead to create schema and put the data in a RDBMS might be unjustifiable and even prohibitive.

NoSQL Systems

The NoSQL systems provide an alternative to the SQL-based relational database systems. Starting from a complete rejection of SQL-based systems, the name is currently interpreted as Not-Only- SQL. The http://nosql-database.org/ site lists more than 150 systems in this category. Active development started in 2009 and was inspired by a couple of research papers on systems, such as Google's BigTable [ref] and Amazon's Dynamo [ref].

Characteristics

The main driving force behind the NoSQL movement is the requirements for performance, horizontal scalability, and availability of Internet-scale web applications. Such applications need to serve hundreds of millions of users and need scalability and availability that are not easy or cost-affordable to achieve with RDBMSs. At the same time they can often tolerate some inconsistent or imprecise results and don't need the strict ACID properties provided by the RDBMSs.

Thus, the NoSQL systems move away from many of the features traditionally offered by RDBMSs. They have a simpler data model and query language and relax some of the ACID guarantees. Those relaxations make it easier to safely partition a database across multiple machines, that is, they enable a better horizontal scalability. However, operations typically encapsulated inside the RDBMS are now left to the application designer.

While RDBMS tries to make distribution transparent for the application, the NoSQL systems let the application know about and leverage on the distributed aspects. This is considered a paradigm shift in the systems' design.

In the following we highlight the main characteristics of the NoSQL systems: key-based data model, low-level query interface, flexible schema if exists at all, horizontal scalability, and relaxed consistency. Those properties are presented in different degrees in various systems and there are a number of exceptions.

Data model and query language

In the **key-based data model** the programmer selects a key to identify each data item and can, in the most cases, only retrieve an item by performing a lookup on its key. If a multiple-key retrieval is needed for a data set, this might involve some additional effort and processing at the application layer. The key lookup query pattern provides relatively uniform and predictable performance. It however leaves more complex data model logic to the application where it is intertwined with the business logic.

The key-based models can be divided in several categories depending on the type of the data item associated with the key: key-value, key-document, and column family stores.

- The key-value stores provide only **set**, **get**, and **delete** primitives for a data value. The value itself is completely opaque to the NoSQL system and it is up to the application to interpret it. Examples of this category are Amazon's Dynamo and BerkeleyDB.

- CouchDB and MongoDB are examples of key-document stores that map a key to some document that contains structured information. The documents are typically represented in Java Script Object Notation (JSON) format or alike. Mongo DB is an exception that offers a relatively high-level query language where a query document is used to specify data to be retrieved with functionality similar to SQL. It also allows indexing on various properties, which provides fast associative access to data using non-key properties.
- Column family stores were inspired by Google's BigTable. The key in this model is mapped to a row that consists of one or more column families. Each column family can contain multiple columns. The values in the columns are timestamped so that each column can hold multiple versions. This setting is particularly good for modeling historical data with timestamps. Most popular representatives of this category are Cassandra and HBase.

In addition to the key-based data models some NoSQL systems such as Neo4J use graph models. They differ substantially from the key-based models in terms of data model, physical layout, query patterns, and distribution among multiple machines. They are not covered in the current document.

In addition to the key-value lookups most NoSQL systems have bindings to Hadoop or another MapReduce framework to perform analytical queries scaling over the entire datasets.

The data structures provided by the NoSQL data models (key-value, documents) are closer to the ones of programming languages than the relational tables. They can fit better some scientific data types or simply be preferred due to familiarity with imperative programming languages.

Flexible schema

The NoSQL systems do not enforce database schema. Entities from the same set do not need to have the same properties. This can be a beneficial property when the schema needs to be modified on-the-fly with the evolution of the application. Again more work is left in the hands of the developer. For instance, if an instance lacks a property, the developer needs to recognize and code accordingly whether this is an error, or it is due to schema migration.

BASE

The NoSQL systems relax the requirements of the ACID transactions to gain better performance. The operation correctness guaranteed by ACID is in the hands of the developer. There are some guarantees known as BASE: Basically Available, Soft state, and Eventual consistency.

Multi-server durability is provided through replication. Updates are replicated in a master-slave manner. Data loss is possible with some systems (Redis), where the operation is acknowledged before confirmation of successful replication. HBase and HDFS returns control to the user only after the writes are replicated to 2 or more nodes in the cluster. Replication across data centers is usually without confirmation since it would lead to too high latency.

Horizontal scaling

Horizontal scaling is one of the main features provided by the NoSQL systems. It measures the ability of the system to scale out, that is, to increase its workload capacity by adding more machines. The ideal

goal for the horizontal scalability is the linear scalability. It means that the query capacity doubles when doubling the number of machines in the system. Traditionally it is achieved through **sharding**: splitting data and workload among multiple machines.

For systems with prevailing read workloads horizontal scalability can be achieved without data sharding by the means of replication and caching.

With data sharding each machine has to handle only the read and write requests to the shards it hosts. The distribution of the workload is facilitated by the simple key-based query model. This is in contrast to a distributed SQL database where a join operation between two tables may require substantial data transfer between nodes if the shards of the two operands are not co-located on the same server.

Since the primary access method is key-based, sharding is also typically key-based. This means that a function over the key determines on which server a key-value pair should reside. Two classes of functions are most common: hashing and range partitioning.

Hash-based sharding

Systems, such as Dynamo, Cassandra, and Voldemort, use a technique called consistent hashing to build distributed hash tables (DHT). With consistent hashing both data and servers are hashed to a value in a range [1,L]. The range is wrapped around itself to form a ring. Each server is then responsible for all keys with a hash value larger or equal of its own value and smaller than the hash value of the next server on the ring. If a server fails, its workload is taken over by the neighbor servers on the ring.

Data sharding in DHTs can be combined with replication. If a replication factor of R is required, the data is typically passed to the following R-1 servers on the ring.

One of the problems with hash-based sharding is creating of unbalanced shards when the number of servers is small. To alleviate this problem, several virtual nodes are created per physical machine, such that each maps to different location of the DHT ring.

One advantage of the hash-based sharding technique is its simplicity: the client can compute the hash function and send its request directly to the right server.

Range-based sharding

Systems such as BigTable, MongoDB, and HBase use range partitioning to manage data shards. The key space is split into ranges each of which is managed by one machine and possibly replicated to others.

Many systems use hierarchical splitting where ranges are divided into sub-ranges. This allows for finer control over load balancing in the system than with hash-based sharding. The metadata describing the range-to-server mapping is managed by one or several servers which incurs some overhead and potential vulnerability. The client needs to make an additional access to the system in order to first locate on which server the key of interest resides before sending a request to this server. If the metadata are managed by a single master server (BigTable), this creates a single point of failure in the system. When the metadata are secure stored on several configuration nodes (MongoDB), they need to be kept in sync by relatively complex techniques such as two-phase commits.

An additional benefit of range-partitioned shards is that they allow for efficient range scans over the keys.

Consistency

The CAP theorem published in 2002 by Brewer describes 3 properties of modern distributed systems:

- Consistency: do all replicas agree on the version of the data on the time of reading?
- Availability: Do replicas respond to read and write requests regardless of how many replicas are unavailable?
- Partition tolerance: Can the system continue to operate even if some replicas lose the ability to communicate to each other over the network?

The theorem states that a distributed system can only achieve two of these properties at the expense of the third one.

The consistency of distributed system is determined by 3 parameters: N is the total number of replicas required for each data item, W is the number of replicas that have to confirm a successful update (write) before the system acknowledges it to the client, and R- the number of replicas that have to respond with the same value upon reading before the system responds to the client. A system exemplifies **strong consistency** if R+W > N. A common choice of the parameters is W+R = N+1. It is the minimum value providing strong consistency but it still allows for temporary disagreement between replicas.

Many strong consistency systems choose for W=N and R=1, i.e. all replicas have to acknowledge an update and reading of 1 replica is sufficient to serve the client. This setting can be found in HBase that uses the Hadoop Distributed File System (HDFS) strong consistency guarantees.

If W replicas do not respond to a write request, or R replicas do not respond to a read request with a consistent value, then the system is considered unavailable for at least some time.

Systems supporting **eventual consistency**, such as Cassandra, and Dynamo, allow to specify the parameters so that R + W <= N. They allow for replicas to be out of sync. As a consequence, they need mechanisms to detect replica conflicts and a conflict resolution strategy. In many systems conflict detection is performed via so called vector clocks. Each key has an associated vector of counters with one counter for each server. Every time a server performs an update over the key value, it increases its counter. When a server receives data with a vector clock from another node, it compares the vector components and detects if it is out of sync with the recent updates (smaller local counters) or if a conflict has arisen (some counters are larger, some smaller).

There are several approaches for conflict resolution. Some systems provide all versions of the data to the client and let the application decide on the correct value. Other systems resolve the conflict based on the most recent timestamp.

Comparison of popular NoSQL systems

Data store	Data Model	Query API	Sharding
Dynamo	Key-value	Get/put	Hash-based
Voldemort	Key-value	Get/put	Hash-based
CouchDB	Document		No sharding, full replication
MongoDB	Document		Range-based
Cassandra	Column family		Hash-based
HBase	Column family		Range-based

Comparison of RDBMS and NoSQL

The table below summarizes the main differences between the two classes of systems

	RDBMS	NoSQL
Data model	Relational	Key-based
Query language	SQL	Set/Get API
Schema	Strict	Flexible
Transactions	ACID	BASE
Consistency	Strict	Eventual
Relationships	Joins	No joins

MapReduce/ Hadoop Distributed Processing

MapReduce is a technique developed by Google for distributed processing of very large data across a large cluster of machines. It includes a programming model and an associated implementation. With MapReduce model the programmer specifies the analysis in form of series of Map and Reduce tasks. The tasks are small units of work that can be executed in parallel on hundreds of cluster nodes by operating on different chunks of data. The MapReduce programs are automatically parallelized by the implementation which takes care of scheduling, moving data across nodes, fault tolerance, etc. This allows programmers without any experience of distributed and parallel systems to utilize the resources of large distributed systems.

The MapReduce system first reads the input file and splits it into multiple chunks. The chunks are processed by multiple map programs running in parallel. The MapReduce system then takes the results of the map programs and merges them to prepare the input for the reduce programs.

MapReduce distributed processing is useful for analysis of large volumes of unstructured data. Sources such as web-logs, sensor readings, and click-streams often have unknown or ill-defined schema. Application examples include indexing and search, text analysis, machine learning, graph analysis, and data transformations. These types of applications are usually difficult to implement using SQL and relational DBMS.

Usually MapReduce programs are written in Java, but there is also support for programs in C++, Perl, Python, Ruby, R, etc. Although the MapReduce programming model abstracts away the distributed complexity of the system, it still requires good programming skills and is difficult for end users. The need of tools for more productive analysis of unstructured data was clear and led to the development of many high-level languages and systems, such as Hive, Pig, and Tenzing, which leverage upon MapReduce programs.

Hadoop

Apache Hadoop is open-source implementation of Google's MapReduce and Google file system (GFS). The Hadoop components of interest for data management and analysis are:

- Hadoop distributed file system (HDFS)
- MapReduce programming model
- Pig: a high-level dataflow language, called Pig Latin, and compiler that creates MapReduce jobs
- Hive: a SQL-like high level language and optimizer that creates MapReduce jobs
- HBase: a distributed DBMS of the NoSQL family
- Sqoop: a tool for moving data between a relational DBMS and Hadoop

HDFS is a distributed file system that stores and replicates large files across multiple machine nodes. It uses replication to provide availability. The default replication factor is 3. The replication increases the amount of storage space needed. HDFS supports multiple readers and one writer that can only append

data at the end of a file. Existing data cannot be modified and HDFS does not provide an indexing mechanism. Therefore, Hadoop is best suited for read-only applications that need to scan an entire file, such as MapReduce programs. Applications have no knowledge or control over the data placement in the HDFS files.

The Hadoop MapReduce framework handles scheduling, running, and recovery of failed MapReduce jobs. The scheduler tries to distribute map jobs so that the required HDFS data is local to the program. During execution map and reduce programs write work data to the local file system. If a job fails, it is simply rerun by the system.

Besides Java API, MapReduce programs in any program or script language can be run using Hadoop Streaming.

Hive

Hive is an SQL-like system on top of Hadoop that provides indexing to aid lookup and range queries. It has advantages for batch processing over large append-only files and limitations with respect to ad-hoc real-time queries and row-level updates. The main advantage of Hive is scalability and fault tolerance provided by Hadoop. It improves substantially usability with respect to low-level MapReduce programs.

Coexistence of MapReduce and relational DBMS

The separation line between the two categories of systems becomes blurred in the recent years since each category is extending with features typical for the other one.

The increased popularity of MapReduce has led some relational DBMS vendors to include MapReduce functionality in their products. This brings some advantages of MapReduce, i.e. the ability to process multi-structured data using SQL, to the relational environments. Furthermore, many relational vendors recognize the fact that a lot of companies have already acquired large Hadoop clusters. To leverage those shared resources, several systems are adding capability to work on top of the HDFS storage, as opposed to using proprietary data formats on dedicated servers.

At the same time Hadoop-based systems are developing more sophisticated features commonly known from the database field. To name a few:

- column-store data formats, to optimize disk I/O when only some properties are used in the analysis;
- indexing methods, to locate (small) data units of interest without the need to scan everything;
- shared memory structures between different map-jobs, to reuse work among jobs;
- higher-level query languages, to ease specification of analysis and make it available for non-computer scientists.

Given the advantages and popularity of Hadoop for analytical processing of unstructured data, the current expectation is that most organizations will use a combination of both Hadoop and relational DBMS technologies.