High Performance Data Analytics

Lecture 1 (26/10/2022)

Definition

HPDA is the process of quickly examining extremely large data sets to find insights, this is done by using parallel processing.

Distributed Computing

Field that studies distributed systems.

Distributed systems are systems where its components are located in different physical computers, and each process works autonomously and communicates with messages through a shared network.

Every process has its own private memory, different components compute at the same time, there is a lack of a global clock and the processes are independent, allowing failure in a node not to interfere in the service.

Some of its applications are: cloud computing, IoT, wireless sensors network. And it uses some key algorithms such as consensus (reliable agreement on a decision), leader election, reliable broadcast of message and replication.

Cloud computing

On demand availability of computer systems through the internet, typically relates to distributed systems.

Challenges of Distributed systems

Programming: programming for concurrency adds new programming mistakes, as it's impossible to predict all concurrency issues, must be able to coordinate between programs and lacks global view or possibility of debugging code.

Resource sharing: system shares computing resources with all users.

Scalability: system must be elastic, and grow with increased number of demand and users whilst maintaining performance level. This requires of course the necessity to add hardware.

Fault handling: how to detect, mask and recover from failure. In distributed systems failure is the default expected behavior.

Heterogeneity: system consists of different hardware/software.

Transparency: Users don't care, and should not be aware, of how/where code/data is.

Security: Availability of services and confidentiality of data.

Parallel computing

Many calculations or execution of processes carried out at the same time, with the goal of improving performance.

Levels of parallelism:

- Bit-Level: process multiple bits concurrently.
- Instruction-Level: processes multiple instructions concurrently on a CPU.
- Data: run the same computation on different data.
- Task: run different computations concurrently.

High Performance Computing

Field providing massive computing resources for a computational task.

Challenges:

- Programming: require low level API and code optimization to achieve performance, performance-optimized code is difficult to maintain and debug, load balancing issues (using all computing resources efficiently).
- Scalability: stricter than other systems.

Computational Science

Use of computational resources to solve complex scientific research. Uses theory, simulation, big data analysis and experimentation as pillars of science.

Big Data

Big data analytics: Extracting insights from large datasets to support decisions

Big data challenges:

- Volume: the amount of data
- Velocity: the data volume per time
- Variety: the different types of data
- Veracity: the trustworthiness of data
- Value: to usefulness of data

Lecture 2 (07/11/2022)

Terminology

Raw data: collected information that is not derived from other data.

Derived Data (data product): data produced after computations.

View: presentation of conclusions to answer specific questions, based on the data product.

Semantic normalization: the process of reshaping free from data to structured data.

Data Management Plan (DMP): describes the strategy and measures for handling research data during and after a project.

Data Life Cycle: creation, distribution, use, maintenance and distribution.

Information lifecycle management: business term for the practices, tools and policies to manage the data life cycle in a cost effective way.

Data governance: control that assures the data is in accord to the business requirement and constraints.

Data provenance: the documentation of input, data transformation and the involved systems to support analysis, tracing and reproducibility.

Data-Lineage: allows to identify the source of the data, used to generate data products.

Service Level Agreement: contracts defining quality, for example: performance/reliability and responsibilities between service and user/provider.

Process Model

Process model is a model to describe processes. There are a few qualities of descriptions:

- Descriptive: describes the events that occur in a process.

- Prescriptive: describes the intended process and how it's executed. Also defines rules and guidelines for the process.

- Explanatory: provide reasons for the process, describe the requirements and establish links between processes.

Data cleaning and ingestion: importing raw data into a big data system. Wrangling data stands for the process of cleaning the data from raw to structured data. Usually follows the ETL process.

It's important to define and document data governance policies to ensure data quality, for example the format of data data, the handling of missing values.

Programming paradigms

The programming paradigm must follow a couple characteristics. It has to be imperative, declarative, data driven, and multi paradigm.

Data models

A data model describes how information is organized in the system. It is a tool to specify, access and process information.

Logical model: abstraction expressing objects and operations.

Physical model: maps logical structures onto hardware resources.

Operations

Operations define how you can interact with the data, the minimal operations consist of storing and retrieving data, furthermore users may also want to search existing data or update it.

POSIX: create, open, write and read.

CRUD: create, read, update, delete.

Amazon S3: put, get, delete.

Data modeling

The process of creating a data model for an information system.

Concurrency: The model must have atomicity, visibility and isolation.

Availability: Robust to errors and to node failures, able to scale.

Durability: modifications must be persistently stored.

Consensus: several processes agree on a single data value.

- Agreement: every correct process must agree on the same value.
- Integrity: all correct processes decide upon at most one value v. If one decides v, then v has been proposed by some process.
- Validity: if all processes propose the same value v, then all correct processes decide v.
- Termination: every correct process decides upon a value.

Relational model

Database model based on first-order predicate logic. Data is represented as tuples. The tuples with similar semantics are grouped.

Columnar model

Data is stored in rows and columns. A column is a tuple. Each row can contain different columns with complex objects.

Key-Value store

Data is stored as value and addressed by a key. The value can contain complex objects. Keys can be forged to simplify look up.

Document Model

Collection of documents, each containing semi-structured data (json/xml). Addressing to lookup documents is implementation specific. References between documents are possible.

Graph

Entities are stored as nodes and relations as edges in the graph. Properties/attributes provide additional information as key/value.

Fact based model

Store raw data as time stamped atomic fact aka log files of current status. Never delete true facts (immutable data). Make individual facts unique to prevent duplicates.

Data lake

With cheap storage cost, people promote the concept of the data lake, it combines data from any source and of any time and model, allowing for conducting future analysis and not missing any opportunity.

Attributes

- Collect everything: all time all data (raw and processed data) and decide which data is important during analysis.
- Dive anywhere: enable users across multiple business units to refine, explore and enrich data on their own terms.
- Flexible access: shared infrastructure supporting various patterns (batch, interactive, online, search)

Lecture 3 (14/11/2022)

Relational model

Database model represented as tuples, with indexes, where similar tuples are grouped together. No collection is supported in tuples. The schema specifies the structure of the table and their relationships.

Relationships: model relationships between the data, can be 1-to-1. 1-to-many or many-to-many. Relationships can be expressed as columns but are not optimal, usually relationships are expressed in different tables. This can be illustrated using a relationship diagram.

Keys: A key is the set of attributes that identify a tuple in a table, a key can only refer to one tuple. Primary keys is the name of the attribute that identify the tuple. Foreign keys are the inherited key of another table. Natural key is a key that is naturally unique. Surrogate key is an artificial key, like the numeric index for a row.

Normalization: Process of organizing tables to minimize redundancy. It reduces dependencies within and across tables, prevents inconsistency across replicated information and the most important attribute: it reduces storage space and optimize queries.

- 1NF: No collections in the table, has a primary key.
- 2NF: No redundancy of data.
- 3NF: Columns are not functional dependency to something else than a candidate key.
- 4NF: Do not store multiple relationships in one table.

Databases

Database is an organized collection of data, includes queries, schema, views, etc.

Database management system is the software that interacts with the user and the database.

Lecture 4 (21/11/2022)

Hadoop:

Framework for scalable processing of data, based on Google's MapReduce paper. Consists of the Hadoop distributed file system (HDFS) and the MapReduce execution engine that schedules tasks on HDFS. This execution exploits data locally to avoid network data transfer.

HDFS: Reliable store on commodity-of-the-shelf hardware, implemented in JAVA, it provides a single-writer, multiple-reader concurrency model.

Some of its features are hierarchical namespace, high availability and automatic recovery, replication of data, rack awareness and parallel file access.

Architectural Components

Namenode: central manager for the file system namespace, controls filenames, permissions, information about file location. For high availability there is a secondary namenode that backups data.

Datanode: provides storage for objects, directly communicates to other datanotes for object replication.

TaskTracker: accept and run map, reduce and shuffle.

JobTrack: central manager for running MapReduce jobs

MapReduce

The idea to apply a processing pipeline consisting of map and reduce operation. The map filters and convert input records into key-value tuples. The reduce receives all tuples with same key and performs an operation.

Hadoop takes care of reading input and distributing the tuples to reduce. The types of keys, value and format depends on the configuration.

Phase of MapReduce:

1. Distribute code (JAR files)
2. Determine files to read, blocks and file splits, assign mappers to slits and slots.
3. Map: invoke local map functions.
4. Combine: perform local reduction by the key.
5. Shuffle: sort by the key, exchange data.
6. Partition: partition key space among reduces (usually done via hashing).
7. Reduce: invoke reducers.
8. Write output: each reducer writes to its own file.

Parallel access to files: A MapReduce job processes all files in a directory, providing parallelism on the file level, each file is read independently. MapReduce jobs process records that are grouped in input splits, each split is processed by one mapper. Therefore files formats that are not splittable must be avoided.

Steps of MapReduce tasks on Hadoop

1. Client submits a job to the JobTracker.
2. Jobtracker identifies the location of data via the NameNode.
3. JobTracker locates TaskTracker nodes with free slots close to the data.
4. JobTracker starts tasks on the TaskTracker nodes.
5. Monitoring of TaskTracker nodes: if heartbeat signals are missed, work is rescheduled on another TaskTracker. Tasktracker will notify the JobTracker when a task fails.
6. The JobTracker constantly updates its status. Clients can also query this information.

Hadoop 2: the return

The goal is to achieve real time and interactive processing of events. It introduces YARN (Yet Another Resource Negotiator), it's still fully compatible to hadoop1, it supports classical mapReduce and TEZ, DAG of tasks.

System architecture

Yarn modularizes JobTracker functionality, bringing resource management and job scheduling/execution inclusive monitoring.

Applications are executed in containers with the ResourceManagers component as a global daemon. The per-node NodeManager manages and monitors local resources.

TEZ Execution Engine

Tez is Hindi for "speed". Allow modeling and execution of data processing logic.

Lecture 5 (28/11/2022)

General data model for dataflow languages

Data represented in tuples: t = (x1, ... , xn) where xi is of a given type. Input/output are lists of tuples.

Typical operators for Dataflow processing:

- Operations in individual tuples: map/foreach transform or process data of individual tuples or groups, filter by comparing key to a value.

- Operations that require the complete input data: group tuples by key, sort by key, join multiple relations, split tuples of a relation based on condition.

Programming Paradigm for Data Flow

Focus on data movement and transformation.

The models program as directed graphs of data flowing between operations, with input/output illustrated as a node and the node being an operation.

Operation is run once all inputs become valid. One operation can work on a single data element or on the complete data. Parallelism is inherently supported by data flow languages.

Dataflow works best with stateless programs. Whilst stateful dataflow graphs support mutable states.

Programming example:

read("file.csv").filter("word" == "big data").reduce(count)

The goal is to visualize the processing pipeline of data-flows with a schema.

Apache Pig

Apache Pig is an infrastructure for executing big data programs, offering language and compiler without need for server. Data is stored on HDFS and uses MapReduce or TEZ execution engine.

Has a high level scripting language called Pig Latin with a batch mode and an interactive shell.

Data model:

- Tuple: ordered set of named fields

- Bag: collection of tuples

- Relation: is a baglike table

Scripting language Pig Latin

Dataflow oriented imperative programming language.

Datatypes like basic types, tuples, bags and maps.

Statement: operator with a named relation as input and output.

For computation, additional arithmetic operators are provided.

[See functions and operators on slide]

Accessing and manipulating data with Pig

The pig shell is convenient for interactive usage, it checks schema and certain language errors.

Invoke code in other languages via user-defined functions.

Pig Latin can be embedded into python, javascript and java.

[See commands and examples in slide]

Architecture

File format: support for Avro, csv, RCFile, sequence file, binary, json, hive's table. Internally BinStorage formats are used for intermediate files. The schema can be part of the file to be loaded or explicitly given.

Performance advice and parallelism

- Lazy evaluation applies several optimizations automatically. It rearranges work and merge operations if possible.

- Flexible number of reducers for the parallelism.

- Use TEZ instead of MapReduce.

- Use schemas for numeric data.

- Choose the key for the hadoop partitioner.

- Intermediate relations can be compressed via properties.

In case of small input files, aggregate them before using Pig.

Summary

- Data flow programming paradigm is easily parallelizable.
- Pipe diagrams visualize data flow programs.
- Pig provides a data flow oriented programming infrastructure. Input/output to HDFS, utilizing MaoReduce and TEZ, no additional servers needed.
- Pig Latin is a domain-specific programming language. Only a few basic operations are necessary, FOREACH iteration over tuples and nested attributes, but pigLatin is complex and may introduce complex errors.
- Pig can be called from Python to script complex workflows.
- User defined functions can be integrated into pigLatin.

Lecture 5.2 (28/11/2022)

Hive overview

Hive is data warehouse functionality on top of Hadoop/HDFS. Uses compute engines like MapReduce, TEZ and spark; storage formats like text, ORC, HVASE, Rcfile and Avro. Manages metadata in RDBMS.

Access via SQL-like query language HiveQL. Similar to SQL-92 but several features are missing. Limited transactions, subquery and views.

The query latency is between 10s of seconds to minutes, in newer versions it is sub subseconds.

<u>Features:</u>

Basic data indexing.

User defined functions to manipulate data and support data-mining.

Interactive shell: hive.

Hive web interface with simple GUI data access and query execution.

WebHCat API (restful API).

Data types:

Primitive types (int, float, strings, dates, boolean).

Bags (arrays) and dictionaries.

Derived data types, like structs, can be defined by users.

Data organization

Table: like in relational databases with a schema, the Hive data definition language (DDL) manages tables, the data is stored in files on HDFS.

Partitions: table key determining the mapping to directories, this reduces the amount of data to be accessed in filters.

Bucket/clusters: data of partitions are mapped into files. Hash value of a column determines the partition.

Managing schemas with HCatalog

Schemas are stored in Hive's Metastore.

HCatalog is a layer providing a relational data abstraction. Tables include metadata that can be accessed regardless of file format. Exposes schemas and allows data access by other apps.

Can send Java messages upon integration of new data.

Provides the REST API WebHCat.

Command line interface: hcat.

Execution of Hive Queries with Hadoop

1. User submits a query to a user interface
2. The driver receives the sql query.
3. The compiler parses the sql query.
    a. Case 1: it changes the data definition or is very simple.
    b. Case 2: it requires to actually process/ingest data (typical case)
4. Information about the data types and structure of the table is queried from the metastore.
5. The query plan generator translates the query into an execution plan.
6. The execution engine runs the DAG and handles dependencies.
7. The result is stored on HDFS and read by the UI.

Beyond MapReduce – LLAP: Live Long and Process

Features identified to be necessary to improve performance: asynchronous spindle-aware IO, prefetching caching of column chunks, multi threaded JIT-friendly operator pipelines.

Strategy: Optional long living daemon can be run on worker nodes, the demon receives task and query fragments. Compress cached data. LLAP is integrated into YARN and can use Slider for a fast deployment. Locality becomes the responsibility of the caller.

LLAP benefits and Architecture

Significantly reduce start up time.

Caching of compressed data reduces memory footprint.

Scalable architecture and other frameworks such as Pig can use the daemon.

Together with Apache Ranger, it enables column level access control.

File formats: Overview

Hive supports typical HDFS file types. SequenceFile, CSV, JSON, AVRO, can access data stored in HBASE via HbaseStorageHandler.

A format is not necessarily well suited for queries, a query may require a full scan of all content.

Requirements of Hive lead to ORC file format.

ORC: Optimized Row Columnar

Line-weight index stored within the file.

Compression based on data type.

Concurrent reads of the same file.

Split files without scanning for markers.

Support for adding/removal of fields.

Partial support of table updates.

Partial ACID support (if requested by user).

ORC files

Stripe: group of row data.

Postscript: contains file metadata.

Index data (per stripe and row group).

Compression of blocks.

Tool to output ORC files.

Bloom Filters

Its goal is to identify if an element is a member of a set with n elements.

Space efficient probabilistic data structure: allows false positives but not false negatives, fixed bit array to test on arbitrary sized n.

Allocate array B of m bits, l indicates a key is present. Provide a k hash function hi(key) 1,...,m

Summary

Hive provides an SQL interface to files in Hadoop.

- Idea: avoid time consuming data injection
- Create schemas on demand based on need.
- Uses MapReduce or TEZ for data processing.
- Processing may require full scan of files.

Original Hive was a HDFS client + metadata service. LLAP is server based.

HCatalog offers a relational abstraction to many file formats.

- Create a schema once, use it everywhere.
- Data can be organized in partitions and buckets/clusters.
- ORC files are optimized for Hive.
- Import of data not necessary.

HiveQL provides an extended SQL syntax.

- Schema enables complex data types.
- Processing of JSON possible.
- MapReduce jobs can be embedded.

Lecture 6 (05/12/2022)

Overview of Hbase

Column oriented key-value database for structured data. Based on Google's BigTable, consists of simple data and consistency model.

Scalable for billions of rows with millions of columns. Sharding of tables: distribute keys automatically among servers. Stretches across data centers.

Custom query language. With real time queries, compression, im-memory execution, Bloom filters and block cache to speed up queries.

Uses HDFS and supports MapReduce.

Uses ZooKeeper for configuration, notification and synchronization.

Has an interactive shell.

Data model

Namespace: logical grouping of tables for quota, security.

Table: consists of rows and a name.

Row: consists of a row key and many columns with values. Key/values are binary (converted from any data type), but the hbase shell stores all data as strings.

Column: consists of a column family and a qualifier.

Column family: string with printable characters, grouped by requirement.

Cell: combination of row, column.

Versioning: upon update, update timestamp. Can keep multiple versions.

Consistency

Row keys cannot be changed.

Strong consistency of reading and writing.

Mutations are typically atomic. Columns of multiple column families of one row can be changed atomically. Order of concurrent mutations not defined. Successful operations are made durable.

Mutation of multiple rows are not atomic, they need more than one API call.

The tuple (row, column, version) specifies the cell. Normally the version is the timestamp, but can be changed. The last mutation to a cell defines the content. Any order of versions can be written.

Get and scan return recent versions but maybe not the newest.

- A get may return an old version but between subsequent gets the version never decreases.

- Any row returned must be consistent.
- A scan returns all mutations completed before started and may contain later changes.
- Content read is guaranteed to be durable.

Deletes masks newer puts until compaction is done.

Co-Processors

Run custom code on Region Server

Coprocessor concept allow to compute functions based on column values

Similar to database triggers.

Hooks are executed on the region servers implemented in observers.

Can be used for secondary indexing, complex filtering and access control.

Scope for the execution:

- All tables (system coprocessors)
- On a table (table coprocessors)

Observer intercepts method invocation and allows manipulation.

- RegionObserver: intercepts data access routines on RegionServer/table.
- WALObserver: intercepts write-ahead log, one per RegionServer.
- MasterObserver: intercepts schema operations.

Currently must be implemented in Java.

Can be loaded from the hbase shell.

Zookeeper overview

Centralized service for coordination providing.

- Configuration information.
- Distributed synchronization.
- Group management.

Simple: uses a hierarchical namespace for coordination

Strictly ordered access semantics.

Distributed and reliable using replication.

Scalable: a client can connect to any server.

Hierarchical namespace

Similar to a file system but kept in main memory (and durable).

ZNodes represent both file and directory.

Nodes:

- Contain metadata: version numbers, ACL changes, timestamps.
- Additional application data is read together with stats().
- Watch can be set on a node: triggered once and a znode changes.
- Ephemeral nodes: are automatically removed once the session that created them terminates.

Consistency guarantees

Atomicity: no partial results.

Single system image: same data regardless to the server connected.

Reliability: an update is persisted.

Timeless: a client's view can lacj behind only a certain time.

Optional: sequential consistency

- Updates are applied in the order they are performed.
- Znodes need to be marked as sequentia, if this is needed.

<u>Reliability</u>

Quorum: reliable as long as ceil(N/2) nodes are available.

Uses Paxos Consensus protocols with atomic message transfer.

Hbase

<u>Distribution of data</u>

Uses HDFS as backend to store data, it utilizes replication and place servers close to data

Lecture 7 (12/12/2022)

In memory Computation

Processing of data stored in memory, it has optimally near-interactive response regardless of volume.

Advantage: No slow I/O necessary, thus having fast response time.

Disadvantage: Data must fit in the memory of distributed databases. Additional persistence is usually required. Fault tolerance is mandatory

Big data approaches: Apache Spark and Apache Flink.

Spark: overview

As an in-memory processing and storage engine, it loads data from HDFS Cassandra and HBase. Resource management via YARN, mesos, Spark or Amazon EC2. It can also use Hadoop or work standalone.

Supports task scheduling and monitoring.

Has multiple APIs. For Java, python, R, Scala.

Iteractive shells, and exceution on local or cluster mode.

Data model

Distributed memory model: Resislient Distributed Dataset (RDDs).

- Typically a list or a map.
- Is immutable.
- High level APIs provide different

Shared variables offer shared memory access.

Durability of data: RDDs live until the SparkContext is terminated.

Fault tolerance is provided by re-computing data.

Resilient Distributed Datasets (RDD)

Creation of a RDD is either by parallelizing an existing collection, or by referecing a dataset on distributed storage or HDFS.

RDDs can be transformed into new RDDS (for example by filtering one RDD).

- RDD can be redistributed (shuffled)
- RDD is computed if needed, but can be cached in memory or stored
- Computation runs in parallel on the partitions
- Rdd knows its data lineage (how it was computed)

Fault-tolerant collection of elements are split into choosable number of partitions and distributed. Dereived RDDs can be re-computed by usying the recorded lineage.

Shared variables

Broadcast variables (for read-only): transfer to all executor. This should not be modified alter.

Accumulators (reduce variables): counters that can be incremented.

Architecture: execution of applications

Driver program: process runs main(), creates and uses SparkContext.

Task: a unit of work processed by one executor.

Job: a spark action triggering computation starts a job.

Stage: collection of tasks executing the same code. It works independently on partitions without data shuffling.

Executor process: provides slots to run tasks, and isolates apps, therefore data cannot be shared between apps.

Cluster manager: allocate cluster resources and run executor.

Data processing

Driver controls data flow and computation. Executor processes are spawned on nodes, those store and manage RDDs and perform the computation on their local partition. In local mode only one executor is created.

Execution of code:

- Closure is computed: variables/methods needed for execution
- The driver serializes the closure together with the task (code). Broadcast variables are not requested to be packed with the task.
- The driver sends the closure to executors.
- Tasks on the executor runs the closure, which manipulates the data.

Persistence

The data lineage of a RDD is stored.

Actions trigger computation, so no intermediate results are kept.

The methods cache() and persist() enable preserving of results: The first time a RDD is computed, it is kept for further use, each executor keeps its local data, cache()

keeps data in memory and persist() allows it to choose the storage level.

Spark manages the memory cache automatically. LRU cache, old RDDs are evicted to secondary storage or deleted. If a RDD is not in cache, re-computation might be triggered.

Storage levels:

- MEMORY_ONLY: keep java objects in memory, or re-compute them.
- MEMORY_AND_DISK: keep java objects in memory or store them on disk.
- MEMORY_ONLY_SER: keep serialized java objects in memory.
- DISK_ONLY: store the data only on secondary storage.

Parallelism

Spark runs one task for each partition of the RDD. The recommended amount of partitions is 2-4 for each CPU.

The RDD default value is set it can be changed manually.

The number of partitions is inherited from the parent RDD.

Shuffle operations contain the argument numTasks which define the number of partitions for the new RDD.

Some actions contain numTask, which define the number of reducers, by default 8 parallel tasks for groupByKey() and reduceByKey().

Analyze the data partitioning using glom().

Computation

Uses lazy execution: apply operations only when results are needed. Intermediate RDDs can be re-computed multiple times, and users can persist RDDs in memory or disk for later use.

Many operations apply user defined functions or lambda expressions.

Code and closure are serialized on the driver and sent to executors.

RDD partitions are processed in parallel (data parallelism).

Operation types:

- Transformations: create a new RDD locally by applying operations.
- Actions: return values to the driver program.
- Shuffle operations: re-distribute data across executors.

Transformations:

map(func): pass each element through func

filter(func): include those elements for which func returns true

flatMap(func): similar to map, but func returns a list of elements

mapPartitions(func): like map but runs on each partition independently

sample(withReplacement, fraction, seed): pick a random fraction of data

union(otherDataset): combine two datasets

intersection(otherDataset): set that contains only elements in both sets

distinct([numTasks]): returns unique elements

cartesian(otherDataset): returns all pairs of elements

pipe(command, [envVars]): pipe all partitions through a program

Key/Values RDD additionally support:

- groupByKey([numTasks]): combines values of identical keys in a list
- reduceByKey(func, [numTasks]): aggregate all values of each key
- aggregateByKey(zeroValue, seqOp, combOp, [numTasks]): aggregates multiple values for keys (e.g., min, max, sum), uses neutral element for initializer
- sortByKey([ascending], [numTasks]): order the dataset
- join(otherDataset, [numTasks]): pairs (K,V) elements with (K,U), returns (K, (V,U))
- cogroup(other Dataset, [numTasks]): returns (K, iterable, iterable)

Actions

reduce(func): aggregates elements using: func(x, y) ⇒z

Func should be commutative and associative

count(): number of RDD elements

countByKey(): for K/V, returns hashmap with count for each key

foreach(func): run the function on each element of the dataset. Useful to update an accumulator or interact with storage

collect(): returns the complete dataset to the driver

first(): first element of the dataset

take(n): array with the first n elements

takeSample(withReplacement, num, [seed]): return random array

takeOrdered(n, [comparator]): first elements according to an order

saveAsTextFile(path): convert elements to string and write to a file

saveAsSequenceFile(path): ...

saveAsObjectFile(path): uses Java serialization

Shuffle

Repartitions the RDD across the executors.

Costly operation.

May be triggered implicitly by operations or can be enforced.

Requires network communication.

The number of partitions can be set. Optionally a partition function as well.

Operations:

- repartition(numPartitions): reshuffle the data randomly into partitions
- coalesce(numPartitions): decrease the number of partitions
- repartitionAndSortWithinPartitions(partitioner): repartition according to the partitioner, then sort each partition locally.

Typical mistakes

Using local variables in distributed memory.

Object serialization may be unexpected and slow.

Writing to STDOUT on executors.

Yarn with Spark

Two alternative deployment modes: cluster and client.

Interactive shells/driver requires client moe.

Spark dynamically allocates the number of executors based on the load.

Batch Applications

Submit batch application via spark-submit.

Support JARs (Scala or java).

Supports Python code.

To query results check output.

Build self contained sparks applications.

Web UI

Sophisticated analysis of performance issues.

Monitoring features like: running and previous jobs, details for job execution, storage usage, environment variables, details about executor.

Started automatically when a spark shell is run.

Create web.pages in YARN UI

Spark history server keeps the data of previous jobs.

Spark 2.0: Data Structure

RDD provides low level access and transformations. No Structure on data imposed – basically just some bags of tuples.

Dataframes extending RDDs: imposes a schema on tuples but tuples remain untyped. Acts like a relational database. Additional higher level semantics and operators, like aggregation. These operators extract better performance.

Datasets (spark 2): offer strongly-typed and untyped API. Converts tuples individually into classes with efficient encoders. Compile time checks for data types.

Higher Level Abstraction

Spark SQL: deal with relational tables. Access JDBC, hive tables or temporary tables. Limitation: has no update statement, the insert statement is only for Parquet files. Spark SQL engine offers catalyst optimizer for dataset and data frames.

GraphX: graph the processing.

Spark streaming: discretized streams accept data from sources (TCP stream, file, kafka, flime, kinesis, twitter).

Mllib: machine learning library that provides efficient algorithms for several ML tasks.

Lecture 8 (19/12/2022)

Stream Processing

The stream processing paradigm is the same as dataflow programming.

The programming implements operations functions and defines data dependencies.

Uniform streaming: An operation is executed on all elements individually.

By default no view of complete data at any time.

Advantages: Pipelining of operations and massive parallelism is possible. Data is in memory. Data dependencies of kernel are known and can be dealt at compile time.

Restrictions of the programming model: Windowing, the sliding window contain multiple elements. The stateless vs stateful paradigm.

Storm Overview

Real time stream computation system for high velocity data, able to process a million records per second per node.

Implemented in Clojure. User APIs are provided for java.

Utilizes Yarn to schedule computation.

Fast, scalable, fault tolerant, reliable, easy to operate.

General use cases: online processing of large data volume, speed layer in the lambda architecture, data ingestion into the HDFS ecosystem, parallelization of complex functions

Also support for some languages via streamparse.

Several high level concepts are provided.

Data model

Tuple: ordered list of named elements. Contain fields, and possible to have dynamic types.

Stream: a sequence of tuples.

Spout: a source of streams for a computation. (EX: kafka messages, tweets, real time data).

Bolts: processors for input streams producing output streams. (Perform filtering, aggregation, join data, talk to databases, etc).

Topology: the graph of the calculation represented as a network. (The parallelism is statically defined).

Partitions and Stream Groupings

Multiple instances of spots/bolts each process a partition. Stream grouping defines how to transfer tuples between partitions.

Selection of groupings:

- Shuffle: send a tuple to a random task.
- Field: send tuples which share the values of a subset of fields to the same task.
- All: replicate/broadcast tuple across all tasks of the target bolt.
- Local: prefer local tasks if available, otherwise use shuffle.
- Direct: producer decides which consumer task receives the tuple.

Use Cases

Several companies still use Strom, for ex: twitter, yahoo, spotify.

Architecture Components

Nimbus node: Storm master node, upload computation jobs, distribute code across the cluster, monitor computation and reallocates workers (upon node failure, tuples and jobs are re-assigned).

Worker nodes run Supervisor daemon which start/stop workers. Its processes execute nodes in the topology.

Zookeeper is used to coordinate the Storm cluster. Performs the communication between nimbus and

supervisors, stores which services to run on which nodes, establishes the initial communication between services.

Architecture supporting tools

Kyro serialization framework: supports serialization of standard Java objects.

Apache Thrift for cross-language support: creates RPC client and serves for interlanguage communication.

Topologies are Thrift struct and Nimbus offer Thrift service: allows to define and submit them using any language.

Execution Model

Multiple topologies can be executed concurrently. Usually sharing the nodes, but with the isolation scheduler exclusive node use is possible.

Worker process: runs in its own JVM, belongs to one topology, and spawns and runs executors threads.

Executors: always attached to a single thread. Runs one or more tasks of the same bolt/spout, the tasks are executed sequentially. By default one task per thread, and the assignment of tasks to executors can change to adapt the parallelism by using storm “rebalance command”.

Task: the execution of one bolt/spout.

Processing of tuples

A tuple emitted by a spot may create derived tables with dependencies. If the processing of a tuple fails, storm reruns the nodes dependent on that tuple.

Consistency

At least once processing semantics. One tuple may be executed multiple times on bolts, and if a error occur a tuple is restarted from its spout.

Restarts tuple if a timeout/failure occurs.

Processing Strategy

Track tuple processing using a random 64 bit message ID. Explicit record all spout tuple Ids a tuple is derived from.

Acker tasks track the tuple DAG implicity for each tuple. Spout informs Acker tasks of the new tuple. Acker notifies all spouts if a derived tuple is completed. Hashing maps spout tuple ID to acker tasks.

Acker uses 20 bytes per tuple to track the state of the tuple tree, by using an XOR of all derived tuple Ids and all acked tuple Ids the ack value is 0 only when the processing of the tuple is completed.

Programming requirements

Fault tolerance strategy requires developers to acknowledge successful processing of each tuple and prevent early retransmissions of the tuple from the spout. And to anchor products derived tuple to link to its origin.

Failure and handling

Task node fault: Tuple Ids at the root of tuple tree time out, starts a new task and replay of tuples begins, requiring transactional behavior of spouts

Acker task fault: After timeout, all pending tuples managed by acker are restarted.

Spout task fault: Source of the spout needs to provide tuples again.

> If reliable processing is not needed, we can set
>
> config topology acker to 0 and immediately back all tuples on each spout.

Exactly once semantics

Semantics guarantees each tuple is executed exactly once. Operations depending on this semantics are: updates of stateful computation, global counters, database updates.

Strategies to achieve exactly once semantic:

- Provide idempotent operations
- Execute tuples strongly ordered to avoid replicated execution
- Use Storm’s transactional topology

Performance aspects

Processing of individual tuples: introduces overhead but provides low latency.

Batch stream processing: group multiple tuples into batches, increases throughput but increases latency, allows to perform batch local aggregations.

Micro batches: are a typical compromise (eg: 10 tuples).

Programming and execution

Java is the primary interface, but also supports ruby, python and fancy.

Integration with Hive, HDFS, Hbase, database via JDBC, Solr, spouts for consuming data from kafka.

Has a web interface for checking processes.

Debugging

Support local and distributed mode, in local mode simulates worker nodes with threads, using debug mode will output component messages.

HDFS integration

HdfsBolt can write tuples into CSV or sequenceFiles. Has file rotation policy including action and conditions, so it can move or delete old files after certain conditions are met. Synchronization policy defines when the file is synchronized to HDFS.

HBase integration

HBaseBolt: allows to write columns and update counters.

HbaseLookupBolt: Query tuples from HBase based on input.

Hive integration

HiveBolt writes tuples to Hive in batches. Requires buckted/clustered table in ORC format. Once committed it is immediately visible on Hive.

Distributed RPC

Distributed Remote procedure call.

Goal is to reliable exectution and parallelization of functions.

Helper Classes exist to setup topologies with lenar execution.

Processing of DRPCs:

- Client sends the fucntion name and arguments to DRPC server
- DRPC server creates a request ID
- The topology registered for the function receives tuples in a DRPCSpout
- The topology computes a result
- Its last bolt returns request id + output to DRPC server
- DRPC server sends result to the clinet
- Client casts output and return from blocked function

Trident

High level bstraction for real time computing: low latency queries, construct data flow topologes by invoking functions, has similaritirs to spark and pig.

Provides exactly once semantics.

Allows stateful stream processing: use Memcached to save intermediate states and backends for HDFS, Hive, HBase are available

Perfomant: executes tuples in micro batches, partial aggregation before sending tuples.

Repliable: an incrementing transaction id is assigned to each batch, update of states is ordered by a batch ID

Sparking streaming

Streaming support in spark.

Data model: continuous stream of RDDs.

Fault tolerance: checkpointing of states.

Not all data can be accessed at a given time: only data from one interval or a sliding window.

Not all transformation and operations available.

Workflow: build pipeline and start it.

Can read streams from multiple sources.

Apache Flink

One of the latest apache tools, since 2015. Supports Scala and Java, and has a rapidly growing ecosystem. Share similarities to Storm and Spark

Features: one concept for batch processing/streaming, interactive computation, optimization of jobs, exactly once semantics, event time semantics.

Lecture 9 (09/01/2023)

Components for High-Performance Data analytics

Required components: serves, storage, processing capabilities and user interfaces.

Storage: NoSQL databases are non relational, distributed and scale-out (for example HDFS, mongoDB, etc) and data warehouses with schema are useful for well known repeated analysis.

Processing capabilities: performance is important, but interactive processing is difficult, new available technology offers batch processing and real time processing.

Basic Considerations for High Performance Analytics

Analysis requires efficient processing of data.

New data is continuously coming, high velocity data.

Storage and data management techniques are needed.

Distributed algorithms

Fundamental style and abstraction level for computer programming:

- Imperative
- Declarative
- Data driven (describe patterns and transformation)
- Multi-paradigm support several at the same time

The goal: productivity of the users and performance upon execution. Having a tool supporting development, deployment and testing.

Performance depends on single core efficiency but also parallelism.

Parallelism is an important aspect for processing.

Semantics of a service:

Consistency: behavior of simultaneously executed operation. Should have atomicity, visibility and isolation.

Availability: readiness to serve operations. Should be robust to system errors, be able to scale well and function with partial network failure.

Durability: modifications should be stored on persistent storage. Should have consistency, meaning every operation leaves the system in a correct state.

Wishlist for Distributed software

A distributed software should have the following characteristics: high availability, fault tolerance, be scalable, extensible (easy to introduce new features and data), have high user productivity, be ready for the cloud, be debuggable, have a high performance and high efficiency.

Consistency limitation in distributed system

Communication is essential but fault is common.

CAP theorem: it is impossible to achieve the 3 at the same time (consistency, availability and partition tolerance).

Architectural patterns for DS

Client server approach, multilayered separating functionality, peer-to-peer, no sharing information between server, object request broker providing trasnaperency to function execution, service oriented architecture (microservices), REST.

Multitier architecture

Presentation tier: the user interface, translates tasks and results into something the user can understand.

Logic tier: coordinates the application, processes commands and makes logical decisions, is the layer that connects the frontend to the data.

Data tier: where the information is stored and retrieved.

Object Request Broker

Example: COBRA, objects appear local but are anywhere, enables communication os systems that are deployed on different platforms, OS and programming languages.

Problems and Standard algorithms

Reliable broadcast: share information across processes

Atomic commit: operation where a set of changes is a applied as a single operation

Consensus: distributed system agrees on a common decision

Leader election: choose a single process to lead the distributed system.

Mutual exclusion: establish a distributed critical selection

Non-blocking data structures: provide global concurrent modification and access

Replication: replicate data in a consistent way

Resource allocation: provision resources to task/users

Spanning tree generation

Typical problem: consensus

Definition: several processes agree for a single data value.

Consensus and consistency of distributed processes are related. Protocols such as **Paxos** ensure consistency in the entire cluster. They tolerate typical errors in distributed systems, byzantine protocols can deal with fake information.

Properties:

- Agreement: every correct process must agree on the same value.
- Integrity: all correct processes decide upon at most one value v, if one decides v, then v has been proposed by some process.
- Validity: if all processes propose the same value v, then all correct processes decide v.
- Termination: every correct process decides upon a value.

Assumptions for Paxos

Processors do not lie, they operate at arbitrary speed, may fail and may rejoin the protocol after failure.

Messages can be sent from one processor to any other and are delivered without corruption, they are sent asynchronously and may take arbitrarily long to deliver, may be lost, reordered or duplicated.

The fault tolerance with 2F+1 processors is F.

Two phase commit protocol (2PC)

Idea: one process coordinates commits and checks that all agree on the decision.

1) Prepare phase: coordinator sends messages with transaction to all participants, the participants execute transaction until commit is needed and replies yes/no based on the commit, finally the coordinator checks decision by all replies, if all are yes, then commit.
2) Commit phase: coordinator sends message to all processes with decision, processes commit or rollback the transactions, send acknowledgement, coordinator sends reply to requester.

Consistency hashing

Goal is to manage key/value data in a distributed system. Try to load balance across nodes and deal with fault tolerance.

The idea is to distribute the keys and server on a ring (0-(M-1)). Stores items multiple times by hashing key multiple times (multiple hash functions can be used).

The server with the next bigger number is responsible for data allocation. In case of server failure, the items on the server must be replicated again, and the add/removal of server will only transfer a subset of the data.

REST Architecture

REST APIs are the backbone of various distributed applications, and it is architecture style and not a protocol. The term RESTful indicates the system is conforming to REST constraints.

Constraints/Features:

- Client-server architecture
- Stateless: server does not have to keep any state information
- Cacheability: responses can be cached when labeled so
- Layered system: can utilize proxy or load-balancers
- Uniform interface: self descriptive hypermedia messages (json messages for example)
- Code on demand (optional): deliver code to extend client functionality

Rest is not bound to HTTP but uses many advantages due to HTTP, for example it can be handled by URI request, is portable and independent of client and server platform.

Utilizes semantics of HTTP request methods like GET, PUT, POST and DELETE.

HTTP 1.1

Is a stateless protocol transported via TCP, has both request and response encoded in ASCII, contains standard response codes (200 OK, 404 not found, etc), include header with standardized key/values, but new key/values can be added.

HTTP 2.0+

Semantically compatible with HTTP 1.1, has data compression of headers, allows pipelining of requests and allow multiplexing of multiple requests over a single TCP connection.

HTTP 3.0 is still an internet draft, uses a UDP based transport layer instead of TCP.

Programming API access via TCP

Connect to service IP address and port via TCP, can use multiple API ro tools, like netcat, curl, python and even browser.

Curl transfers data from and to the server, is useful for scripting and testing of web servers, supports many protocols, proxies, cookies, etc (mostly used on command line).

Python API also allows for JSON retrieves.

High level performance

Goal: in the context of this lecture we assume the goal of a system is data processing. With the user perspective goal being the minimal time to solution, and for system perspective is having a cheap total cost of ownership.

Other performance aspects are productivity of users, energy and cost efficiency.

Processing Steps

1) Preparing input data. Either ingesting data into our big data environment, or in the HPC context, preparing the data for being written on a supercomputer.
2) Processing a workflow consisting of multiple steps/queries. It Is a relevant factor for productivity, having a low runtime is crucial for repeated analysis and user interaction. Multiple steps and tools can be involved in a complex workflow but for our model we consider only the execution of one job with any tool.
3) Post processing of output with external tools to produce insight. A typical strategy is having a local analysis, but the best is to have a final product returning from the workflow.

Performance factors influencing processing time

Startup phase: distribution of files/scripts, allocation of resources, starting scripts and loading dependencies, this usually has a fixed cost around the seconds.

Job execution: computing the product, costs for the computation are dependent on job complexity, software architecture and hardware and architecture used.

Cleanup phase: free resources, usually fixed cost in the order of seconds.

Big data cluster characteristics

usually commodity components, cheap interconnect, node-local storage, communication bandwidth between different racks is low.

HPC cluster characteristics

High end components, have an extra fast interconnection with global storage with dedicated servers, the network provides a high bandwidth, this allows for multiple topologies.

Hardware Strategy for software solutions

Utilize different hardware components concurrently, balance and distribute workload among all available servers, allow monitoring of components to see their utilization, if possible avoid communication.

Basic approach on assessing performance

Start with a simple model.

1) Measure time for the execution of your workload
2) Quantify the workload with some metrics (amount of tuples or data processed)
3) Compute W, the workload processed per time
4) Compute the expected performance P based on the system's hardware characteristics
5) Compare W with P, the efficiency is E = W/P
6) If E << 1, for example 0.01, you are using only 1% of the potential.

Lecture 10 (16/01/2023)

Statistical Graphics

Graphics in the field of statistics used to visualize quantitative data. The objective is to explore the content of a data set and find a structure in the data. Also used to check assumptions in the models and to communicate the results of an analysis.

Make use of scatter plots, box, histograms, probability plots, statistical maps, etc.

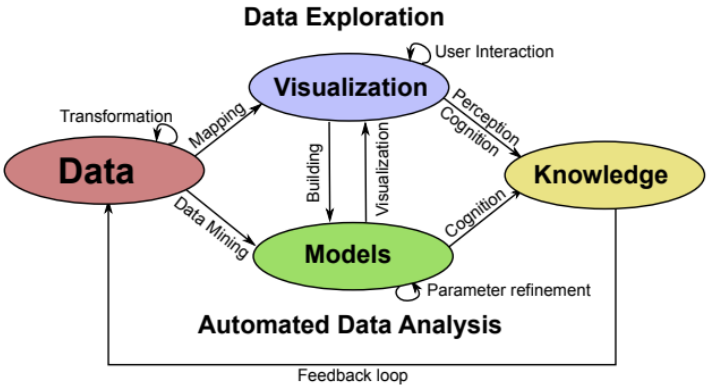
Visual Analytics

The science of analytical reasoning facilitated by interactive visual interfaces. The objective is to solve complex or time critical problems applying the scientific method, and to to present the insight and communicate it visually.

A few analytical tasks:

- understanding past situation
- monitoring events for indicators for an emergency
- supporting decision makers in times of crisis
- identifying possible alternative scenarios
- supporting decision makers in time of crisis

Visual Analytics workflow



Human-computer interaction

By comparing the capabilities of humans with computers we get the best of both worlds: pattern recognition, creative thinking and process of new info from the human brain and the accuracy and execution of algorithms of the computer.

Cognition

The mental action of acquiring knowledge and understanding through thought, experience and senses.

Communicated information and interpretation is biased for humans because of perception, and subjective knowledge.

A few categories of cognitive biases are limits of memory, too much information, not enough meaning and the need to act fast.

Visual perception the pipeline

Information is transformed several times from digital data to human. The retina and the brain interprets visual information. Efficient communication requires understanding human perception.

Optical illusions

By definition: to visually perceive images that differ from reality.

There are many different types of illusions, like: size and shapes of objects, depth perception, the moving of objects.

Design of Graphics

There are many guidelines and languages to program graphics. Important to consider the limitations of the system and the cognitive biases.

Component of Visual Mappings

Spatial substrate: mapping variables to space. Depends on the type of data (structured, unstructured) and the values (nominal, ordinal, quantitative)

Marks: visible elements (points, lines, areas, volumes)

Connection: uses points and lines to show relationships

Enclosure: boxes around elements

Retinal properties: size, orientation, color, texture, shape

Temporal encoding: animations

Guidelines

Goals of graphical displays:

- show the data
- induce the viewer to think about the substance
- avoid distorting what the data have to say
- present many numbers in a small space
- make large data sets coherent
- encourage the eye to compare different pieces of data
- reveal the data at several levels of detail
- serve clear purpose

Simple rules:

- use the right visualization for the data types
- use building blocks for graphics
- reduce information to the essential part to be communicated
- consistent use of building blocks and themes

Large scale data analytics

in-situ: analyze results while application is still computing; define the computation of data beforehand.

in-transit: analyze the data with it is on the I/O path

computational steering: interact with the application while it runs

Paraview

Tool for interactive and remote visualization of scientific data, simply requires adaptor for file formats. Generates level of detail models for interactive frame rate.

Catalyst: in-situ use case library; scripts implement analysis/visualization tasks; the user must push data via API.

Lecture 11 (23/01/2023)

Reminder HPC

HPC: field providing massive compute resources for a task

Supercomputer: aggregate power of 10.000 computer devices

Storage system: provides some kind of storage with some API

File system: provides hierarchical namespace and file interface

Parallel I/O: multiple processes can access distributed data concurrently

Supercomputer host costly storage

Compute performance grows 20 times by generation, while storage capacity/throughput grows by 6 times.

Application Data vs File

Applications work with semistructured data, like vectors, matrices, etc. While a file is just a sequence of bytes.

Applications and programmers must serialize data into a flat name space, and therefore deal with an uneasy handling of complex data types, since mapping is a performance critical task, dealing with vertical data access is impractical

The I/O stack

Parallel application: is distributed across many nodes, has a specific access pattern and may use several interfaces (file, SQL, NoSQL).

Middleware: provides high level access

POSIX: ultimately file system access

Parallel file system: Lustre, GPFS, PVFS2

File system: EXT4, XFS, NTFS

Block device: utilizes storage media to export a block api

Zoo of interfaces

Posix file: array of bytes

HDF5: container like a file system

Database: structure and arrays

NoSQL: document, key-value, graph, tuples

Namespace: hierarchical, flat, relational

Access: imperative, declarative, implicit

Concurrency: blocking vs non-blocking

Consistency semantics: visibility and durability of modifications

Application I/O types

Serial I/O: one node accesses the file, then shares with different nodes.

Parallel multi-file I/O: each node accesses their own file.

Parallel shared-file I/O: all nodes access the same file, need to be aware of concurrency.

Access Patterns

Contiguous: each piece of memory stores sequentially to the file.

Contiguous in memory, not in file: each piece of memory accesses one piece of the file but not sequentially.

Contiguous in file not in file: not the entire memory is used, but some pieces access a piece of the entire file.

Discontiguous: not the entire memory nor the entire file, each piece of memory accesses a piece of the file.

Bursty: each piece of memory sequentially reads the file, but one piece of the file at a time.

Out of core: one piece of memory accesses multiple pieces of the file.

Parallel I/O efficiency

I/O intensive science requires good performance. File systems like DKRZ offer aout 700 GiBs throughput, but I/O operations usually only achieve 10% of peak performance, due to its inefficiency.

The following characteristics can influence I/O performance:

- application access patterns and usage of storage interfaces
- communication and slow storage media
- concurrent activity
- tenable optimizations deal with characteristics of storage media
- complex interaction of the factors above

The stack of I/O hardware and software is so complex even for experts, so it requires tools for diagnosing causes, predicting performances and prescribing settings.

NetCDF

Is a high level I/O-API and ecosystem. In simple words: it's a data model, a file format and a set of APIs and libraries for various programming languages.

Together the data model, file format and APIs support the creation, access and sharing of scientific data. Allows to describe multidimensional data and include metadata which further characterizes the data,

Classic NetCDF model

NetCDF files are containers for Dimensions, Variables, and Global Attributes. A NetCDF file (dataset) has a path name and possibly some dimensions, variables and global attributes, and data values associated with the variables.

<u>Dimensions:</u>

Are used to specify variable shapes, grids and coordinate systems. A dimension has a name and a length, it can be used to represent a real physical dimension like time, latitude, or height. It can also be used to index other quantities. The same dimension can be used in multiple variables

<u>Variables:</u>

Holds multidimensional arrays of values of the same type. A variable has a name, type, shape, attributes and values.

In the classical data model, the type of a variable is the external type of its data as represented on disk.

Data:

Is stored in the form of arrays, for example:

- Temperature varying over time at a location is stored as a one dimensional array.
- Temperature over an area for a given time is stored as a two dimensional array.

Coordinate Variables:

A 1D variable with the same name as a dimension is a coordinate variable. They are associated with a dimension of one or more data variables and define a physical coordinate corresponding to that dimension.

Attributes:

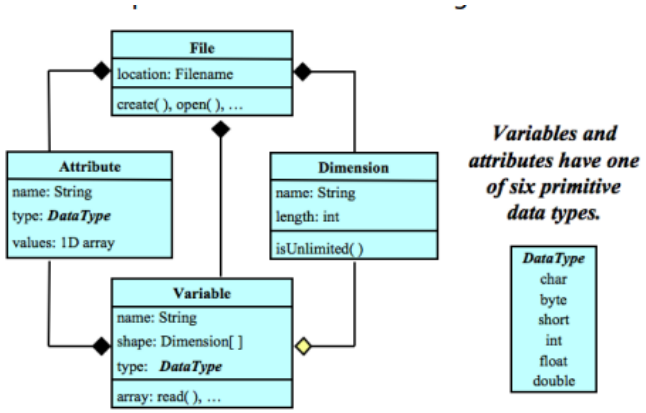
It holds metadata, contains information about properties of a variable or the whole data set. Can either be scalars or 1D array. Attributes have name, type and values, and are used to specify properties as units, standard names, special values, min and max, etc.

CDL - Common Data form Language

Notation used to describe NetCDF objects is called CDL. It provides a convenient way of describing a dataset. Utilities allow producing CDL text files from binary NetCDF datasets and vice versa.

The file contains dimension, variables, and attributes. Components are used together to capture the meaning of data and relations among data fields.

UML of Classical NetCDF model



A file has named variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One dimension may be of unlimited length.

Understanding of I/O behavior

Observation: measurement of runs on the system, can be many cases to run, this could cause bias because the measurement perturbs the behavior.

Monitoring: system/tool provided for creating observation

Theory: performance models, used to determine performance for a system.

Monitoring I/O

To understand variability better we must analyze and understand behavior. Many interesting metrics can be recorded and various tools can aid in the analysis.

Performance Variability

The issue is that measuring an operation repeatedly results in different runtime. A few reasons for that is that sometimes a certain optimization is triggered, thus shortening the I/O path.

This leads to a non-linear access performance. It is difficult to assess performance of even repeated measurements.

The goal is to predict likely reason and cause of effect by just analyzing runtime.

Benchmarking

Benefits:

- can use simple sequence of operations, easy to compare with theoretical values.
- may use a pattern like a realistic workload, provides performance estimates for workload
- Sometimes the only possibility to understand hardware capabilities, because the theoretic analysis may be infeasible.

Benchmarks are easier to code/understand/setup/run than applications and come with less restrictive license limitations

Benchmarks measure system behavior and implement simple well known behavior. Many I/O benchmarks exist covering various aspects like the APIs used, data access pattern, memory access pattern and the parallelism and concurrency.

IO-500 benchmark:

Representative: for optimized and naive workloads

Inclusive: cover various storage technologies and non-POSIX APIs

Trustworthy: representative results and prevent cheating

Cheap: easy to run and short benchmarking time (minutes)

Favors a single metric to simplify the comparison across different dimensions

Goals of the IO-500 benchmarking

Bound to performance expectations for realistic workloads. Track storage system characteristic behavior over the years, document and share the best practices.

Probing approach

Many sites run periodic regression tests to help identify performance regression with updates. Instead we run a non-invasive benchmark with a high frequency, this mimics the user-visible client behavior. Generate and analyze statistics.

Optimizations

There are many optimizations for I/O, like read-ahead or write-behind, distribution of data across servers, etc.