Module 4

CLOUD PROGRAMMING

Parallel Computing and Programming Paradigms

- Consider a distributed computing system consisting of a set of networked nodes or workers.
- The system issues for running a typical parallel program in either a parallel or a distributed manner are:
- **Partitioning:** This is applicable to both computation and data as follows:
- **Computation partitioning:** This splits a given job or a program into smaller tasks. Partitioning greatly depends on correctly identifying portions of the job or program that can be performed concurrently.
- **Data partitioning :** This splits the input or intermediate data into smaller pieces. Upon identification of parallelism in the input data, it can also be divided into pieces to be processed on different workers

- **Mapping:** This assigns either the smaller parts of a program or the smaller pieces of data to underlying resources.

- This process aims to appropriately assign such parts or pieces to be run simultaneously on different workers and is usually handled by resource allocators in the system.

- **Synchronization** Because different workers may perform different tasks, synchronization and coordination among workers is necessary so that race conditions are prevented and data dependency among different workers is properly managed.

- Multiple accesses to a shared resource by different workers may raise race conditions, whereas data dependency happens when a worker needs the processed data of other workers.

- **Communication**: Because data dependency is one of the main reasons for communication among workers, communication is always triggered when the intermediate data is sent to workers.

- **Scheduling:** For a job or program, when the number of computation parts (tasks) or data pieces is more than the number of available workers, a scheduler selects a sequence of tasks or data pieces to be assigned to the workers.

- The resource allocator performs the actual mapping of the computation or data pieces to workers, while the scheduler only picks the next part from the queue of unassigned tasks based on a set of rules called the scheduling policy.

- **Motivation for Programming Paradigms:**

- Handling the whole data flow of parallel and distributed programming is very time consuming and requires specialized knowledge of programming.

- Dealing with such issues may affect the productivity of the programmer and may even result in affecting the program's time to market

- Also it may detract the programmer from concentrating on the logic of the program itself.

- Hence parallel and distributed programming paradigms or models are offered to abstract many parts of the data flow from users.

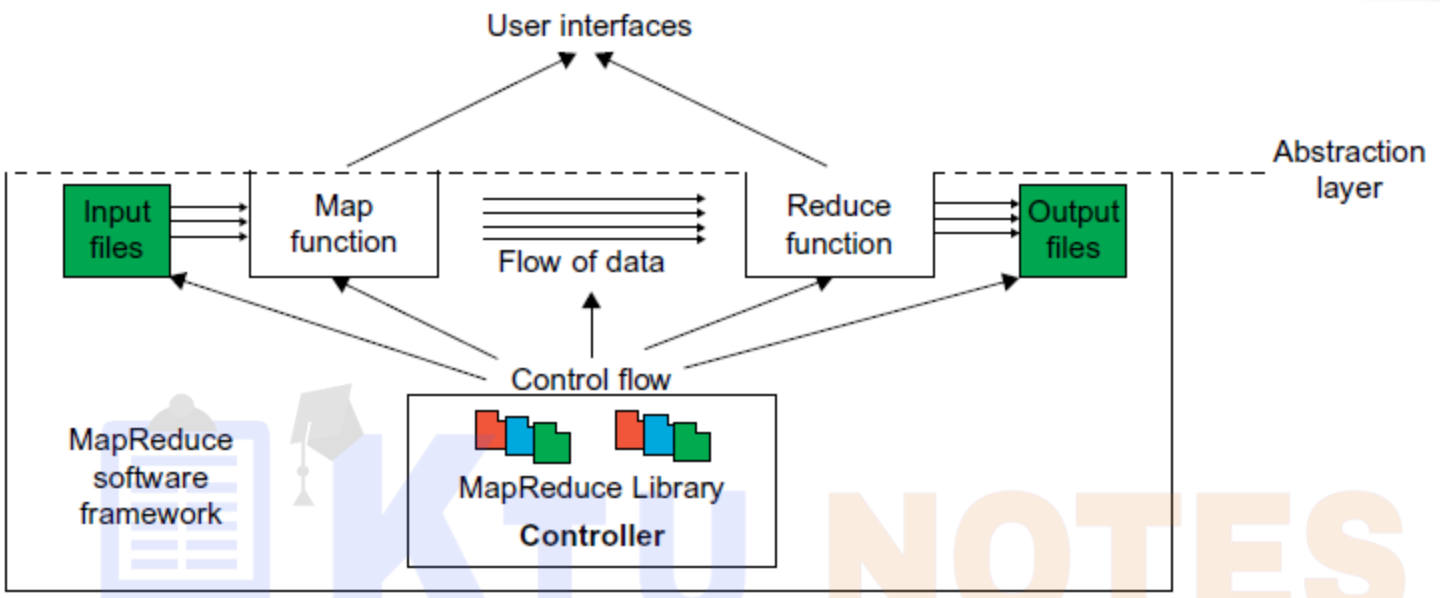
- These models aim to provide users with an abstraction layer to hide implementation details of the data flow which were needed to be coded by the users before.

- Simplicity of writing parallel programs is an important metric for parallel and distributed programming paradigms.

- Other motivations behind parallel and distributed programming models are
  - to improve productivity of programmers
  - to decrease programs' time to market
  - to leverage underlying resources more efficiently,
  - to increase system throughput, and
  - to support higher levels of abstraction

MapReduce

- A software framework which supports parallel and distributed computing on large data sets
- **Formal Definition of MapReduce**
- The MapReduce software framework provides an abstraction layer with the data flow and flow of control to users, and hides the implementation of all data flow steps such as data partitioning, mapping, synchronization, communication, and scheduling.
- Although the data flow in such frameworks is predefined, the abstraction layer provides two well-defined interfaces in the form of two functions: **Map and Reduce**
- These two main functions can be overridden by the user to achieve specific objectives.



- The figure shows the MapReduce framework with data flow and control flow.
- Input data flows through the Map and Reduce functions to generate the output result under the control flow using MapReduce software library.
- Special user interfaces are used to access the Map and Reduce resources.

- The user overrides the Map and Reduce functions first and then invokes the provided MapReduce (Spec, & Results) function from the library to start the flow of data.

- The MapReduce function, MapReduce (Spec, & Results), takes an important parameter which is a specification object, the Spec.

- This object is first initialized inside the user's program, and then the user writes code to fill it with the names of input and output files, as well as other optional tuning parameters.

- This object is also filled with the name of the Map and Reduce functions to identify these user defined functions to the MapReduce library.

- The Map and Reduce are two major subroutines. They will be called to implement the desired function performed in the main program

- The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below.

Map Function (... .)

{

... ...

}

Reduce Function (... .)

{

... ...

}

Main Function (... .)

{

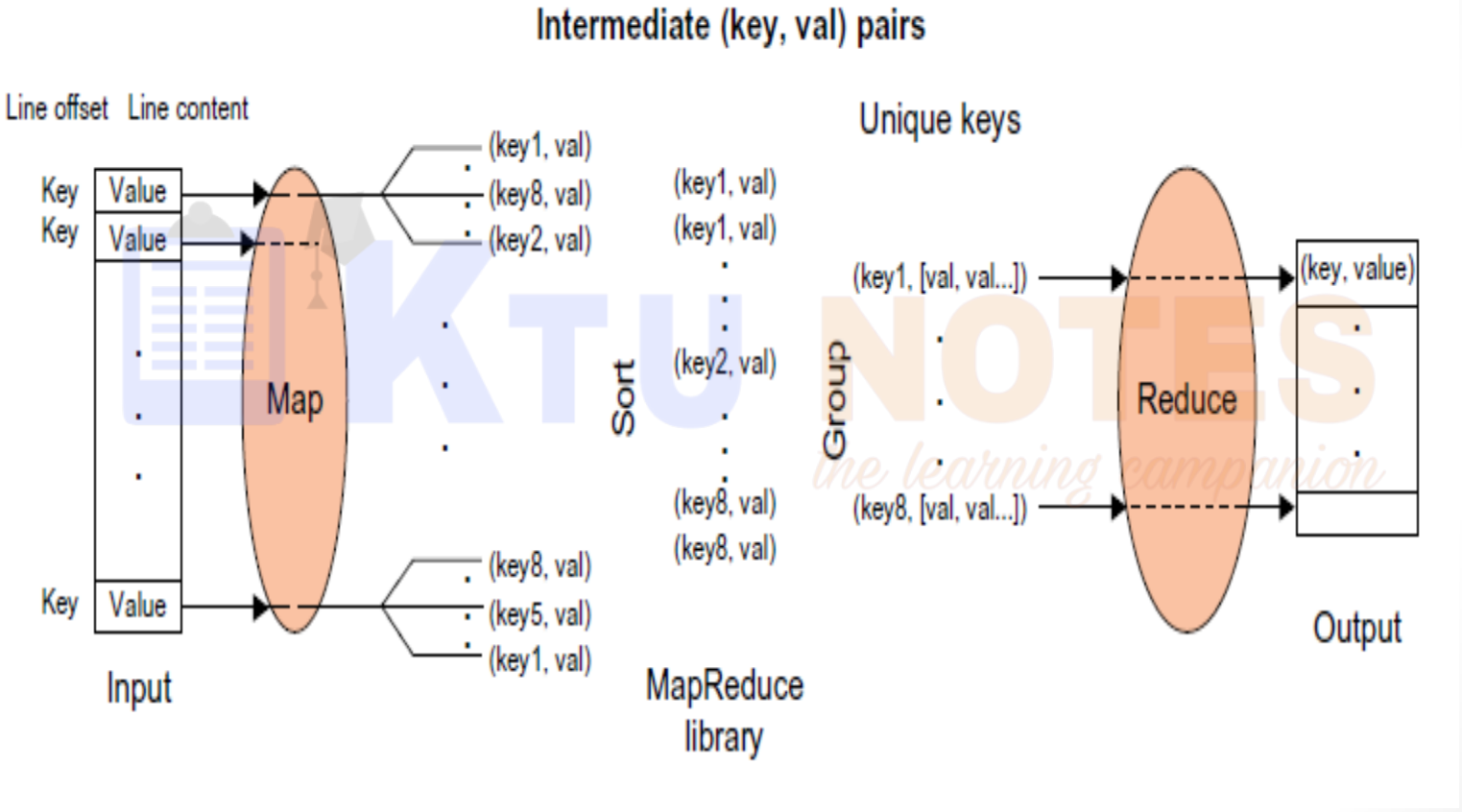
Initialize **Spec** object

... ...

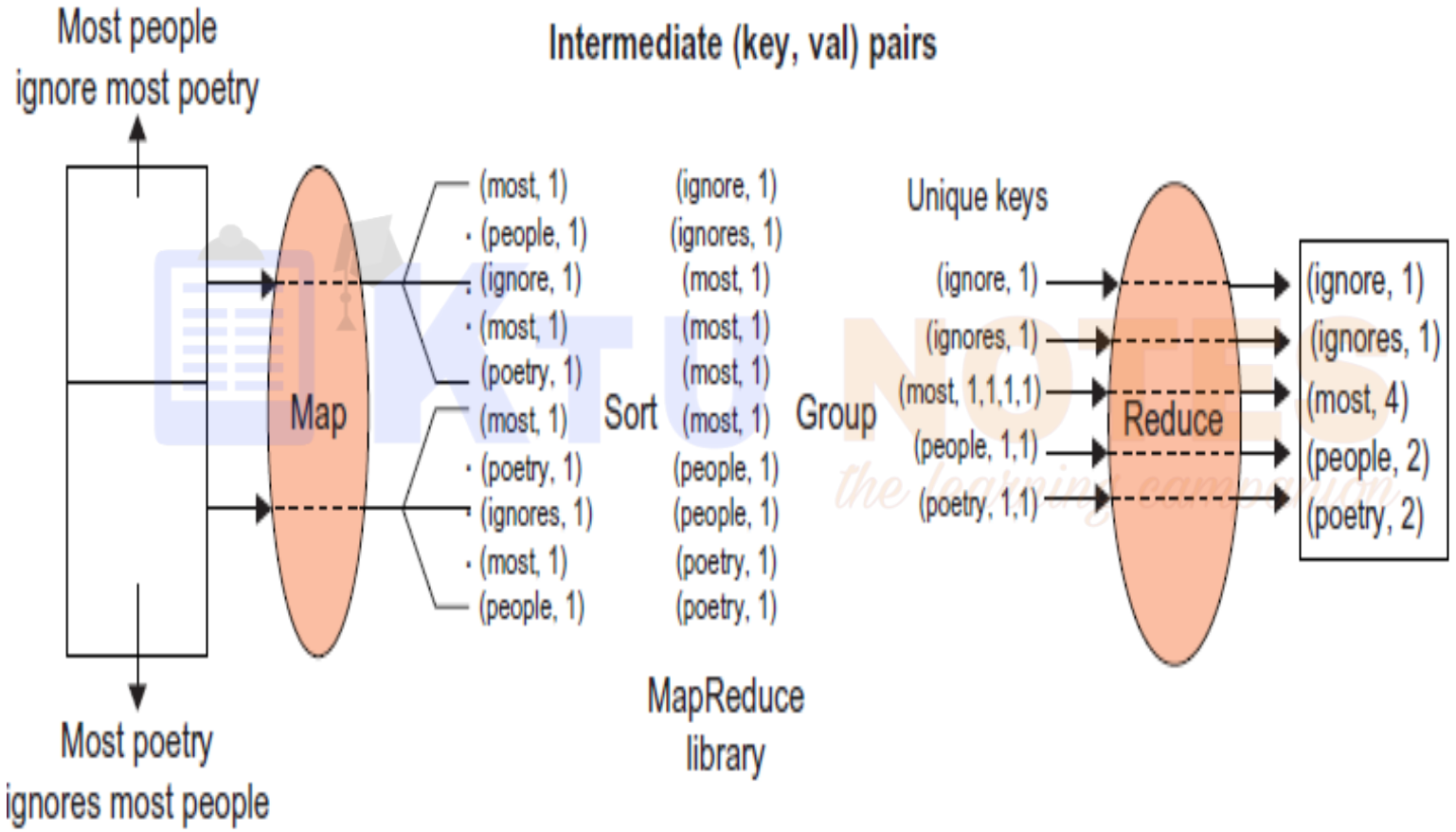
MapReduce (Spec, & Results)

}

- **MapReduce Logical Data Flow**



- **The data flow of a word-count problem using the MapReduce functions (Map, Sort, Group and Reduce) in a cascade operations.**



- **Formal Notation of MapReduce Data Flow**

- The Map function is applied in parallel to every input (key, value) pair, and produces new set of intermediate (key, value) pairs as follows:

$$(key_1, val_1) \xrightarrow{Map\ Function} \text{List}\,(key_2, val_2)$$

- Then the MapReduce library collects all the produced intermediate (key, value) pairs from all input (key, value) pairs, and sorts them based on the "key" part.

- It then groups the values of all occurrences of the same key.

- Finally, the Reduce function is applied in parallel to each group producing the collection of values as output as illustrated here:

$$(key_2, \text{List}\,(val_2)) \xrightarrow{Reduce\ Function} \text{List}\,(val_2)$$

- **Strategy to Solve MapReduce Problems**

- Finding unique keys is the starting point to solving a typical MapReduce problem.

- Then the intermediate (key, value) pairs as the output of the Map function will be automatically found.

- The following three examples explain how to define keys and values in such problems:

  - Problem 1: Counting the number of occurrences of each word in a collection of documents
  - Solution: unique “key”: each word, intermediate “value”: number of occurrences

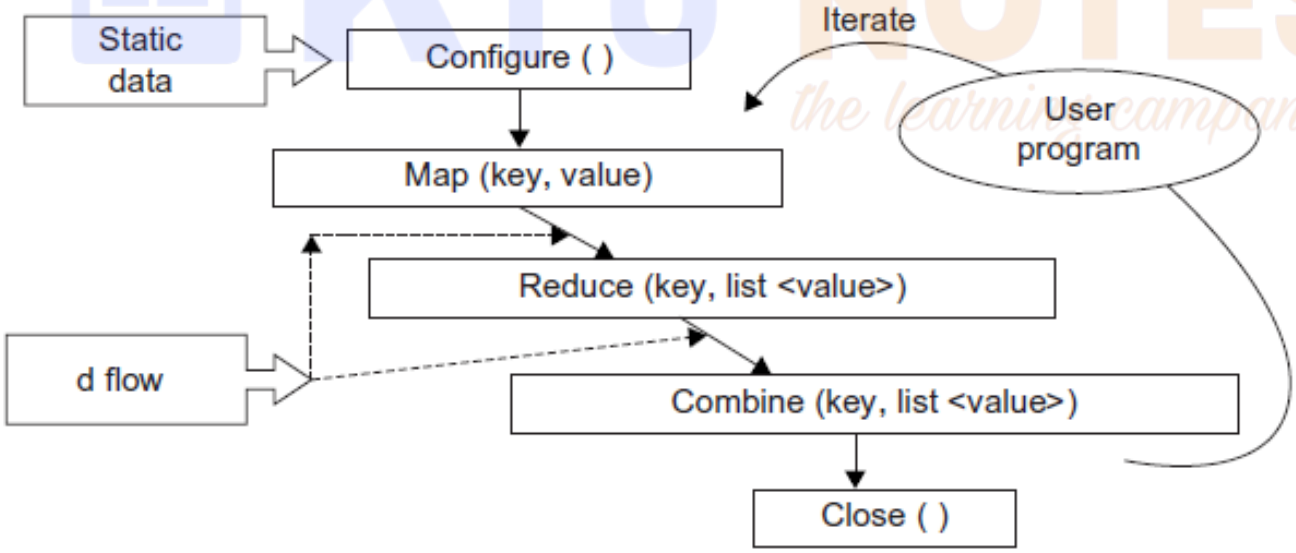
  - Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents
  - Solution: unique “key”: each word, intermediate “value”: size of the word

  - Problem 3: Counting the number of occurrences of anagrams in a collection of documents.
  - Solution: unique “key”: alphabetically sorted sequence of letters for each word (e.g., “eilnst”),intermediate “value”: number of occurrences

Twister and Iterative MapReduce

- A set of extensions to the MapReduce programming model and improvements to its architecture that will expand the applicability of MapReduce to more classes of applications.

- Twister is much faster than traditional MapReduce.



- **Static vs. Dynamic Data**

- Many iterative applications analyzed show a common characteristic of operating on two types of data products.

- Static data is used in each iteration and remain fixed throughout the computation whereas the variable data is the computed results in each iteration and typically consumed in the next iteration.

- **Cacheable Mappers/Reducers**

- Most of the iterative applications operate on moderately sized data sets which can fit into the distributed memory of the computation clusters.

- So long running map/reduce tasks which last throughout the life of the computation can be used for such iterative applications.

- The long running (cacheable) map/reduce tasks allow map/reduce tasks to be configured with static data and use them without loading again and again in each iteration.

- Twister achieves considerable performance gains by caching the static data across map/reduce tasks.

Hadoop Library from Apache

- Hadoop is an open source implementation of MapReduce coded and released in Java (rather than C) by Apache.
- The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS) as its underlying layer rather than GFS.
- The Hadoop core is divided into two fundamental layers: **the MapReduce engine and HDFS.**
- The MapReduce engine is the computation engine running on top of HDFS as its data storage manager.
- **HDFS:** HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.

- **HDFS Architecture:** HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves).
- To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes).
- The mapping of blocks to DataNodes is determined by the NameNode.
- The NameNode (master) also manages the file system's metadata and namespace.
- In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files.

- **HDFS Features:** Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements, to operate efficiently.

- Because HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system.

- For example, security has never been supported for HDFS systems.

- The following are the two important characteristics of HDFS to distinguish it from other generic distributed file systems.

- **HDFS Fault Tolerance:** One of the main aspects of HDFS is its fault tolerance characteristic.

- Since Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception.

- Hadoop considers the following issues to fulfill reliability requirements of the file system:
  - **Block replication:** To reliably store data in HDFS, file blocks are replicated in this system.
  - HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster.
  - The replication factor is set by the user and is three by default.
  - **Replica placement :** Storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability
  - But the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack.
  - Hence sometimes HDFS compromises its reliability to achieve lower communication costs.

- For example, for the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data

- **Heartbeat and Blockreport messages:** are periodic messages sent to the NameNode by each DataNode in a cluster.

- Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode .

- The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

- **HDFS High-Throughput Access to Large Data Sets (Files):** Because HDFS is primarily designed for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency.

- Because applications that run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64MB) to allow HDFS to decrease the amount of metadata storage required per file.

- This provides two advantages: The list of blocks per file will shrink as the size of individual blocks increases

- By keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

- **HDFS Operation:** The control flow of HDFS operations such as write and read can properly highlight roles of the NameNode and DataNodes in the managing operations.

- The control flow of the main operations of HDFS on files is further described to manifest the interaction between the user, the NameNode, and the DataNodes in such systems.

- **Reading a file :** To read a file in HDFS, a user sends an “open” request to the NameNode to get the location of file blocks.

- For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file.

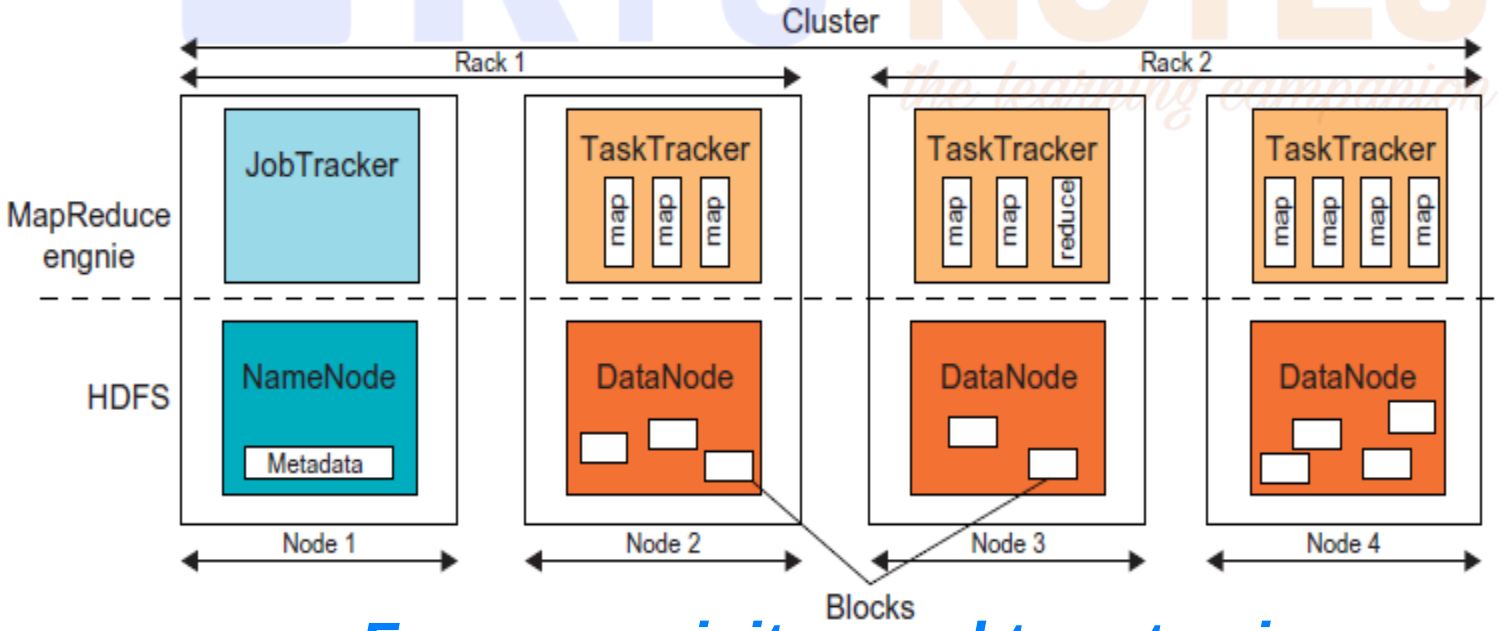
- The number of addresses depends on the number of block replicas.
- Upon receiving such information, the user calls the read function to connect to the closest DataNode containing the first block of the file.
- After the first block is streamed from the respective DataNode to the user, the established connection is terminated and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.
- **Writing to a file**: To write a file in HDFS, a user sends a "create" request to the NameNode to create a new file in the file system namespace.
- If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function.
- The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode.

- Since each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.
- The streamer then stores the block in the first allocated DataNode.
- Afterward, the block is forwarded to the second DataNode by the first DataNode.
- The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.
- Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.

- **Architecture of MapReduce in Hadoop:** The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems.

- The figure shows the MapReduce engine architecture cooperating with HDFS.

- Similar to HDFS, the MapReduce engine has a master/slave architecture consisting of a single JobTracker as the master and a number of TaskTrackers as the slaves (workers).

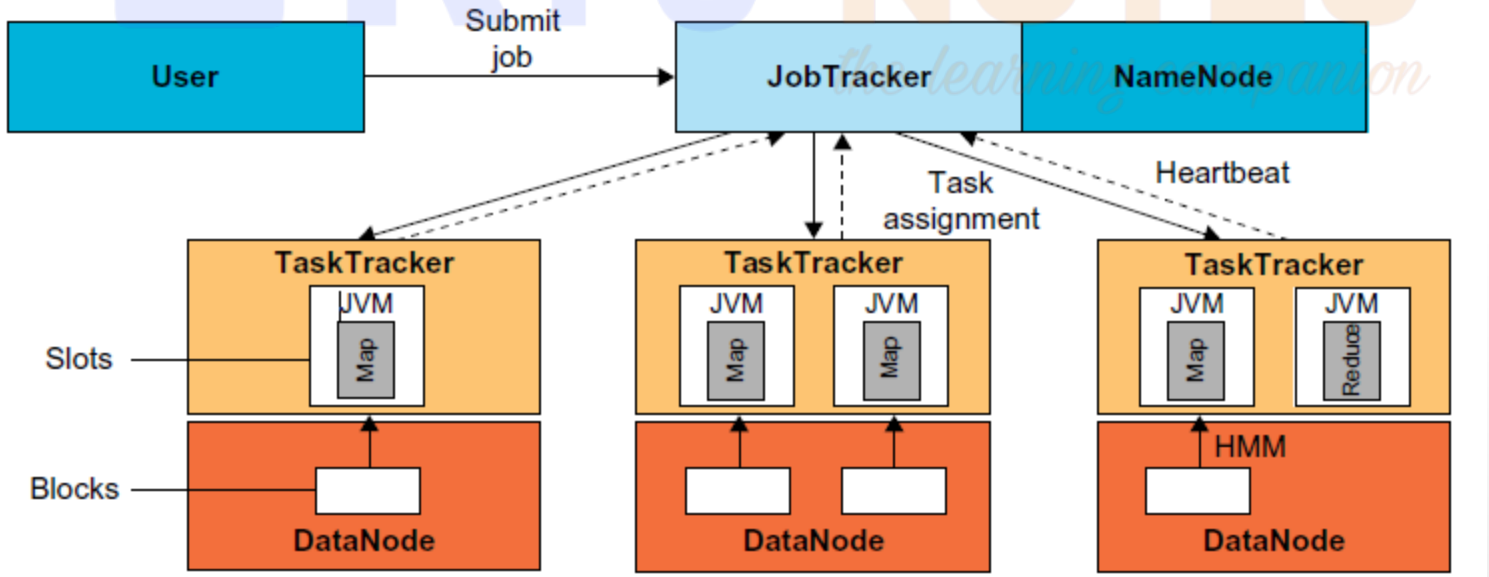


- The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers.
- The TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster.
- Each TaskTracker node has a number of simultaneous execution slots, each executing either a map or a reduce task.
- Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node.
- For example, a TaskTracker node with N CPUs, each supporting M threads, has M * N simultaneous execution slots.
- Each data block is processed by one map task running on a single slot.
- Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

- **Running a Job in Hadoop**: Three components contribute in running a job in this system: a user node, a JobTracker, and several TaskTrackers.

- The data flow starts by calling the *runJob(conf)* function inside a user program running on the user node, in which conf is an object containing some tuning parameters for the MapReduce framework and HDFS.

- The *runJob(conf)* function and *conf* are comparable to the *MapReduce(Spec, &Results)* function and *Spec* in the first implementation of MapReduce by Google



- **Job Submission:** Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:

- A user node asks for a new job ID from the JobTracker and computes input file splits.

- The user node copies some resources, such as the job's JAR file, configuration file, and computed input splits, to the JobTracker's file system.

- The user node submits the job to the JobTracker by calling the submitJob() function.

- **Task assignment**: The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers.

- The JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers.

- The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

- **Task execution:** The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system.

- Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.

- **Task running check**: A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers.

- Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.

Pig Latin

- Pig Latin is a high-level data flow language developed by Yahoo that has been implemented on top of Hadoop in the Apache Pig project.
- Apache Pig is a high-level platform for creating programs that run on Apache Hadoop.
- Pig can execute its Hadoop jobs in MapReduce, Apache Tez, or Apache Spark.
- Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for relational database management systems.
- Pig Latin can be extended using user-defined functions (UDFs) which the user can write in Java, Python, JavaScript, Ruby and then call directly from the language.

- **Pig Latin Data Types**

Data Type	Description	Example
Atom	Simple atomic value	'Clouds'
Tuple	Sequence of fields of any Pig Latin type	('Clouds', 'Grids')
Bag	Collection of tuples with each member of the bag allowed a different schema	{ ('Clouds', 'Grids') ('Clouds', ('IaaS', 'PaaS') }
Map	A collection of data items associated with a set of keys; the keys are a bag of atomic data	['Microsoft' → { ('Windows') ('Azure') } 'Redhat' → 'Linux']

- **Pig Latin Operators**

Command	Description
LOAD	Read data from the file system.
STORE	Write data to the file system.
FOREACH GENERATE	Apply an expression to each record and output one or more records.
FILTER	Apply a predicate and remove records that do not return true.
GROUP/COGROUP	Collect records with the same key from one or more inputs.
JOIN	Join two or more inputs based on a key.
CROSS	Cross product two or more inputs.
UNION	Merge two or more data sets.
SPLIT	Split data into two or more sets, based on filter conditions.
ORDER	Sort records based on a key.
DISTINCT	Remove duplicate tuples.
STREAM	Send all records through a user-provided binary.
DUMP	Write output to stdout.
LIMIT	Limit the number of records.

- **Comparison of High-Level Data Analysis Languages**

	Sawzall	Pig Latin	DryadLINQ
Origin	Google	Yahoo!	Microsoft
Data Model	Google protocol buffer or basic	Atom, Tuple, Bag, Map	Partition file
Typing	Static	Dynamic	Static
Category	Interpreted	Compiled	Compiled
Programming Style	Imperative	Procedural: sequence of declarative steps	Imperative and declarative
Similarity to SQL	Least	Moderate	A lot!
Extensibility (User-Defined Functions)	No	Yes	Yes
Control Structures	Yes	No	Yes
Execution Model	Record operations + fixed aggregations	Sequence of MapReduce operations	DAGs
Target Runtime	Google MapReduce	Hadoop (Pig)	Dryad

- **Word Count Example Using Pig Script**

```
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

Mapping Applications to Parallel and Distributed Systems

- Mapping applications to different hardware and software in terms of 6 application architectures.

- **Category 1** was popular 20 years ago, but is no longer significant. It describes applications that can be parallelized with lock-step operations controlled by hardware. Such a configuration would run on SIMD (single-instruction and multiple-data) machines.

- **Category 2** is now much more important and corresponds to a SPMD (single-program and multiple data) model running on MIMD (multiple instruction multiple data) machines. Each decomposed unit executes the same program, but at any given time, there is no requirement that the same instruction be executed.

- Category 1 corresponds to regular problems, whereas category 2 includes dynamic irregular cases with complex geometries for solving partial differential equations or particle dynamics.

- **Category 3** consists of asynchronously interacting objects and is often considered the people's view of a typical parallel problem. It probably does describe the concurrent threads in a modern operating system, as well as some important applications, such as event-driven simulations and areas such as search in computer games and graph algorithms.

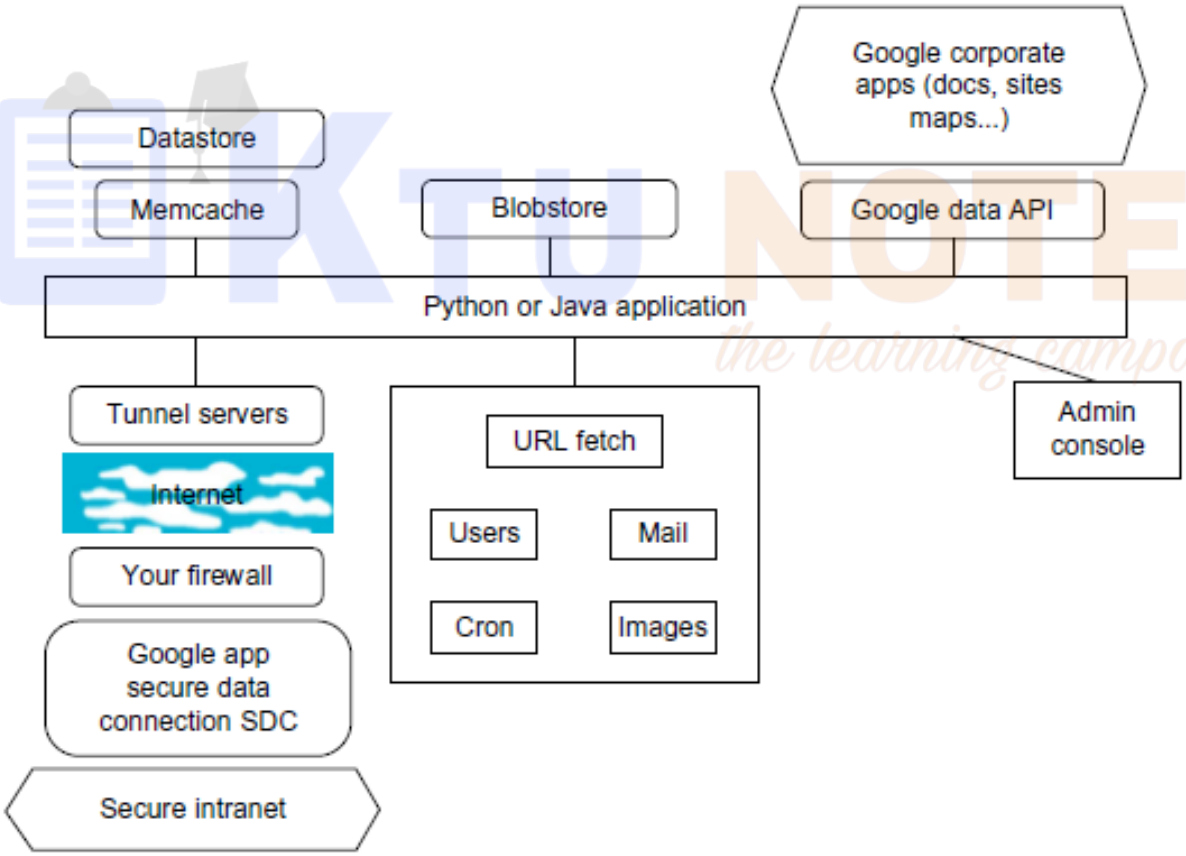
- **Category 4** is the simplest algorithmically, with disconnected parallel components. It is estimated to account for 20 percent of all parallel computing. Both grids and clouds are very natural for this class, which does not need high-performance communication between different nodes.

- **Category 5** refers to the coarse-grained linkage of different "atomic" problems – Category 1-4.

- **Category 6** - MapReduce++, and it has three subcategories: **map only** applications similar to Category 4; the classic MapReduce with file-to-file operations consisting of parallel maps followed by parallel reduce operations; and a subcategory that captures the extended MapReduce version.

Programming the Google App Engine

- The below figure summarizes some key features of GAE programming model for two supported languages: Java and Python.



- A client environment that includes an Eclipse plug-in for Java allows the user to debug GAE on their local machine.
- Developers can use the Google Web Toolkit or any other language using a JVM based interpreter or compiler, such as JavaScript or Ruby.
- Python is often used with frameworks such as Django and CherryPy, but Google also supplies a built in webapp Python environment.
- **Storing and accessing data:** The data store is a NOSQL data management system for entities that can be, at most, 1 MB in size and are labeled by a set of schema-less properties.
- Queries can retrieve entities of a given kind filtered and sorted by the values of the properties.
- Java offers Java Data Object (JDO) and Java Persistence API (JPA) interfaces while Python has a SQL-like query language called GQL.

- The data store is strongly consistent and uses optimistic concurrency control.
- An update of an entity occurs in a transaction that is retried a fixed number of times if other processes are trying to update the same entity simultaneously.
- An application can execute multiple data store operations in a single transaction which either all succeed or all fail together.
- The performance of the data store can be enhanced by in-memory caching using the **memcache**, which can also be used independently of the data store.
- Google added the **blobstore** which is suitable for large files as its size limit is 2 GB.
- **For incorporating external resources:** The Google SDC Secure Data Connection can tunnel through the Internet and link the intranet to an external GAE application.

- The **URL Fetch operation** provides the ability for applications to fetch resources and communicate with other hosts over the Internet using HTTP and HTTPS requests.
- There is a specialized mail mechanism to send e-mail from GAE application.
- Applications can access resources on the Internet, such as web services or other data, using GAE's URL fetch service.
- The URL fetch service retrieves web resources using the same high speed Google infrastructure that retrieves web pages for many other Google products.
- There are dozens of Google "corporate" facilities including maps, sites, groups, calendar, docs, and YouTube, among others.
- These support the Google Data API which can be used inside GAE.

- An application can use Google Accounts for user authentication.
- Google Accounts handles user account creation and sign-in, and a user that already has a Google account (such as a Gmail account) can use that account with a GAE app.
- GAE provides the ability to manipulate image data using a dedicated Images service which can resize, rotate, flip, crop, and enhance images.
- An application can perform tasks outside of responding to web requests on a schedule that is configured, such as on a daily or hourly basis using "**cron jobs**," handled by the Cron service.
- A GAE application is configured to consume resources up to certain limits or quotas.
- With quotas, GAE ensures that an application is not exceeding budget, and that other applications running on GAE won't impact the performance of a particular app.
- In particular, GAE use is free up to certain quotas.

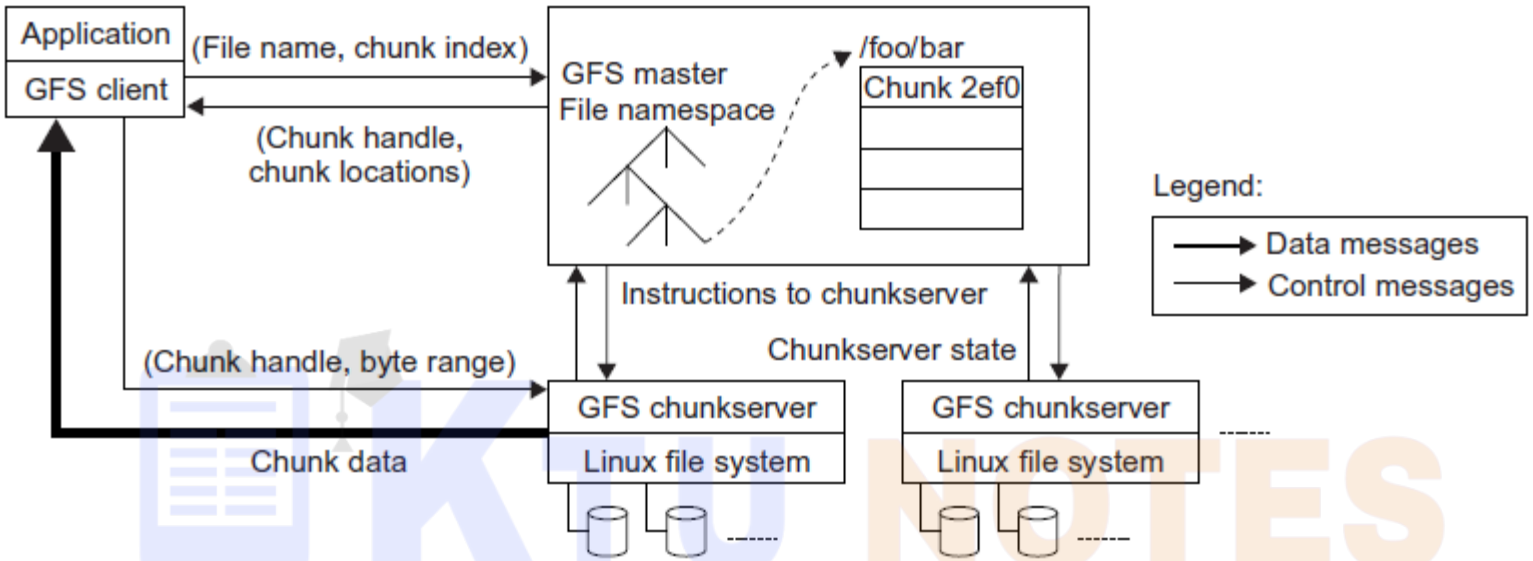
Google File System (GFS)

- GFS was built primarily as the fundamental storage service for Google's search engine.
- As the size of the web data increased Google needed a distributed file system to redundantly store massive amounts of data on cheap and unreliable computers.
- None of the traditional distributed file systems can provide such functions and hold such large amounts of data.
- GFS was designed for Google applications, and Google applications were built for GFS.
- **Characteristic of the cloud computing hardware infrastructure:** As servers are composed of inexpensive commodity components concurrent failures will occur all the time.

- **File size in GFS:** GFS typically will hold a large number of huge files, each 100MB or larger, with files that are multiple GB in size quite common.
- Google has chosen its file data block size to be 64MB instead of the 4 KB in typical traditional file systems.
- **The I/O pattern**: in the Google application is also special.
- Files are typically written once, and the write operations are often appending data blocks to the end of files.
- Multiple appending operations might be concurrent.
- There will be a lot of large streaming reads and only a little random access.
- As for large streaming reads, highly sustained throughput is much more important than low latency.

- **Reliability** is achieved by using replications (i.e., each chunk or data block of a file is replicated across more than three chunk servers).
- A single master coordinates access as well as keeps the metadata.
- This decision simplified the design and management of the whole cluster.
- Developers do not need to consider many difficult issues in distributed systems, such as distributed consensus.
- GFS provides a similar, but not identical, POSIX file system accessing interface.
- The distinct difference is that the application can even see the physical location of file blocks.
- The customized API can simplify the problem and focus on Google applications.

- **Architecture of Google File System (GFS):**



- There is a single master in the whole cluster.
- Other nodes act as the chunk servers for storing data, while the single master stores the metadata and manages the file system namespace and locking facilities
- The master periodically communicates with the chunk servers to collect management information as well as give instructions to the chunk servers to do work such as load balancing or fail recovery.

- The master has enough information to keep the whole cluster in a healthy state.
- With a single master, many complicated distributed algorithms can be avoided and the design of the system can be simplified.
- **Weakness:** The single GFS master could be the performance bottleneck and the single point of failure.
- To mitigate this, Google uses a shadow master to replicate all the data on the master, and the design guarantees that all the data operations are performed directly between the client and the chunk server.
- The control messages are transferred between the master and the clients and they can be cached for future use.
- With the current quality of commodity servers, the single master can handle a cluster of more than 1,000 nodes.

BigTable - Google's NOSQL System

- BigTable was designed to provide a service for storing and retrieving structured and semistructured data.
- BigTable applications include storage of web pages, per-user data, and geographic locations.
- Web pages are used to represent URLs and their associated data, such as contents, crawled metadata, links, anchors, and page rank values.
- Per-user data has information for a specific user and includes such data as user preference settings, recent queries/search results, and the user's e-mails.
- Geographic locations are used in Google's well-known Google Earth software. Geographic locations include physical entities (shops, restaurants, etc.), roads, satellite image data, and user annotations.

- The scale of such data is incredibly large. There will be billions of URLs, and each URL can have many versions, with an average page size of about 20 KB per version.
- The user scale is also huge - hundreds of millions of users and thousands of queries per second.
- Geographic data might consume more than 100 TB of disk space.
- It is not possible to solve such a large scale of structured or semistructured data using a commercial database system.
- This is one reason to rebuild the data management system; the resultant system can be applied across many projects for a low incremental cost.
- The other motivation for rebuilding the data management system is performance.
- Low-level storage optimizations help increase performance significantly, which is much harder to do when running on top of a traditional database layer.

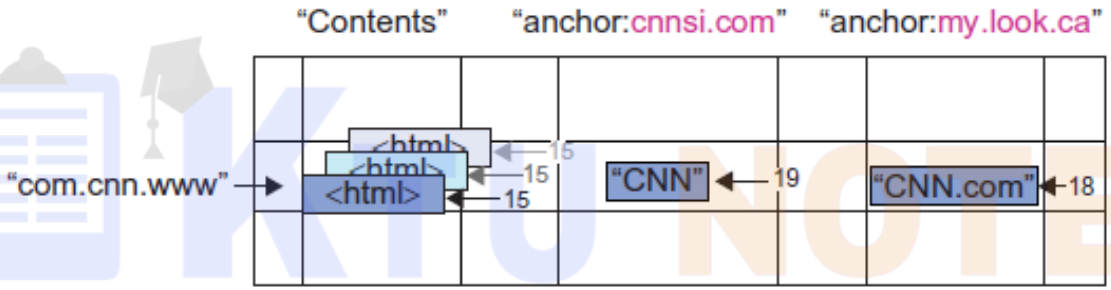
- The design and implementation of the BigTable system has the following goals:
- The applications want asynchronous processes to be continuously updating different pieces of data and want access to the most current data at all times.
- The database needs to support very high read/write rates and the scale might be millions of operations per second.
- Also, the database needs to support efficient scans over all or interesting subsets of data, as well as efficient joins of large one-to-one and one-to-many data sets.
- The application may need to examine data changes over time.
- BigTable can be viewed as a distributed multilevel map.
- It provides a fault-tolerant and persistent database as in a storage service.

- The BigTable system is scalable, which means the system has thousands of servers, terabytes of in-memory data, petabytes of disk-based data, millions of reads/writes per second, and efficient scans.
- Also, BigTable is a self-managing system (i.e., servers can be added/removed dynamically and it features automatic load balancing).
- Design/initial implementation of BigTable began at the beginning of 2004.
- BigTable is used in many projects, including Google Search, Orkut, and Google Maps/Google Earth, among others.
- One of the largest BigTable cell manages ~200 TB of data spread over several thousand machines.
- The BigTable system is built on top of an existing Google cloud infrastructure. BigTable uses the following building blocks:
  - GFS: stores persistent state
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - MapReduce: often used to read/write BigTable data

- **BigTable Data Model Used in Mass Media:**

- BigTable provides a simplified data model compared to traditional database systems.

- The below figure shows the data model of a sample table, Web Table.

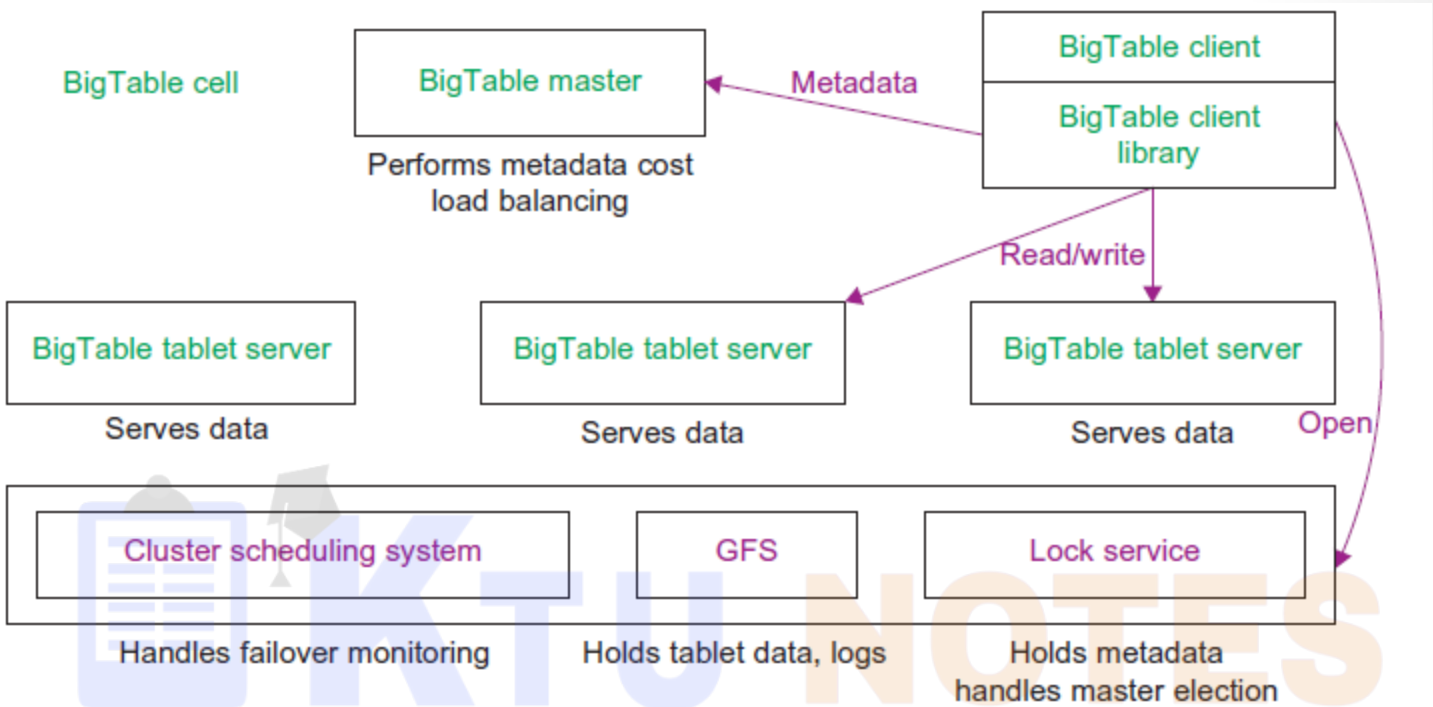


(a) BigTable data model

- Web Table stores the data about a web page. Each web page can be accessed by the URL.

- The URL is considered the row index. The column provides different data related to the corresponding URL—for example, different versions of the contents, and the anchors appearing in the web page.

- The map is indexed by row key, column key, and timestamp—that is, (row: string, column: string, time: int64) maps to string (cell contents).
- Rows are ordered in lexicographic order by row key. The row range for a table is dynamically partitioned and each row range is called “Tablet.”
- Syntax for columns is shown as a (family: qualifier) pair. Cells can store multiple versions of data with timestamps.
- Large tables are broken into tablets at row boundaries. A tablet holds a contiguous range of rows.
- The system aims for about 100MB to 200MB of data per tablet. Each serving machine is responsible for about 100 tablets.
- This can achieve faster recovery times as 100 machines each pick up one tablet from the failed machine.
- Similar to the design in GFS, a master machine in BigTable makes load-balancing decisions.



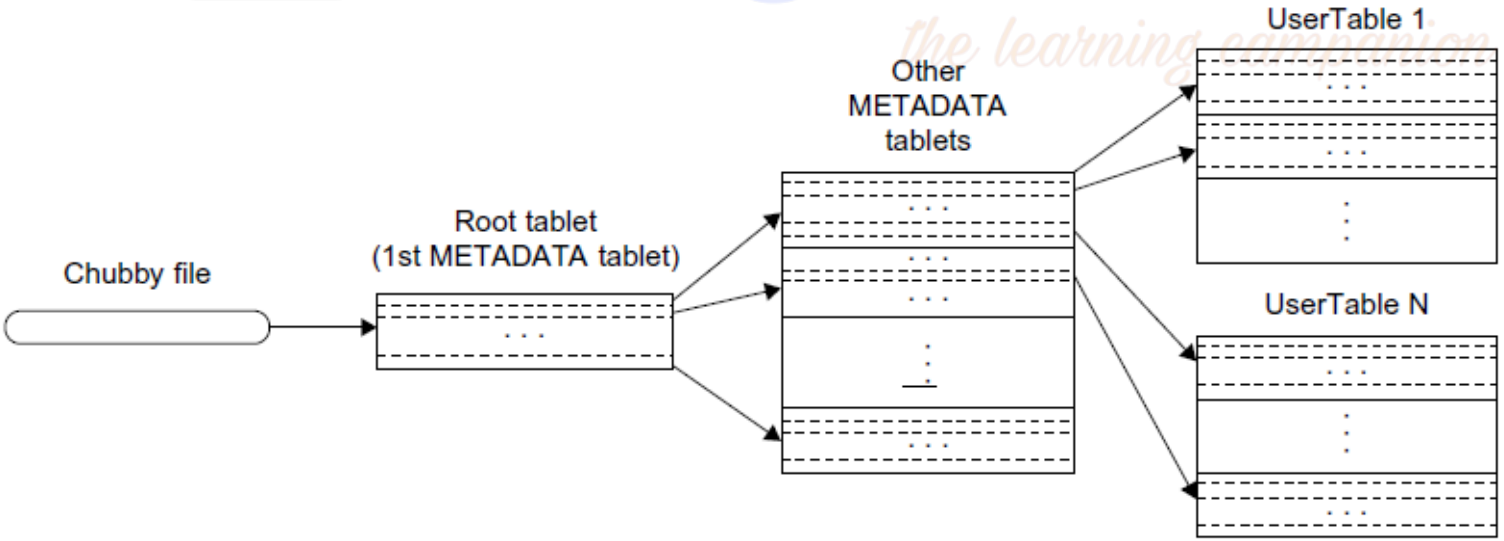
- The above figure shows the BigTable system structure.
- A BigTable master manages and stores the metadata of the BigTable system.
- BigTable clients use the BigTable client programming library to communicate with the BigTable master and tablet servers.

- **Tablet Location Hierarchy:**

- The below figure shows how to locate the BigTable data starting from the file stored in Chubby.

- The first level is a file stored in Chubby that contains the location of the root tablet.

- The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets.



- The root tablet is just the first tablet in the METADATA table, but is treated specially; it is never split to ensure that the tablet location hierarchy has no more than three levels.
- The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier and its end row.
- BigTable includes many optimizations and fault-tolerant features.
- Chubby can guarantee the availability of the file for finding the root tablet.
- The BigTable master can quickly scan the tablet servers to determine the status of all nodes.
- Tablet servers use compaction to store data efficiently.
- Shared logs are used for logging the operations of multiple tablets so as to reduce the log space as well as keep the system consistent.