Parallel and Distributed Computing
CS3006 (BDS-6A)
Lecture 21

Instructor: Dr. Syed Mohammad Irteza

Assistant Professor, Department of Computer Science, FAST

18 April, 2023

Previous Lecture

- MPI
    - Odd-Even Sort (BubbleSort variant)
- Fault Handling
    - Fault Tolerance
    - Redundancy
        - HW
            - Passive (TMR)
            - Active (hot/cold sparing), eviction types
            - Hybrid
        - Information
        - Time

File Systems

- File systems *were originally developed for centralized computer systems* and desktop computers.

- The file system was an operating system facility providing *a convenient programming interface to disk storage*.

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.

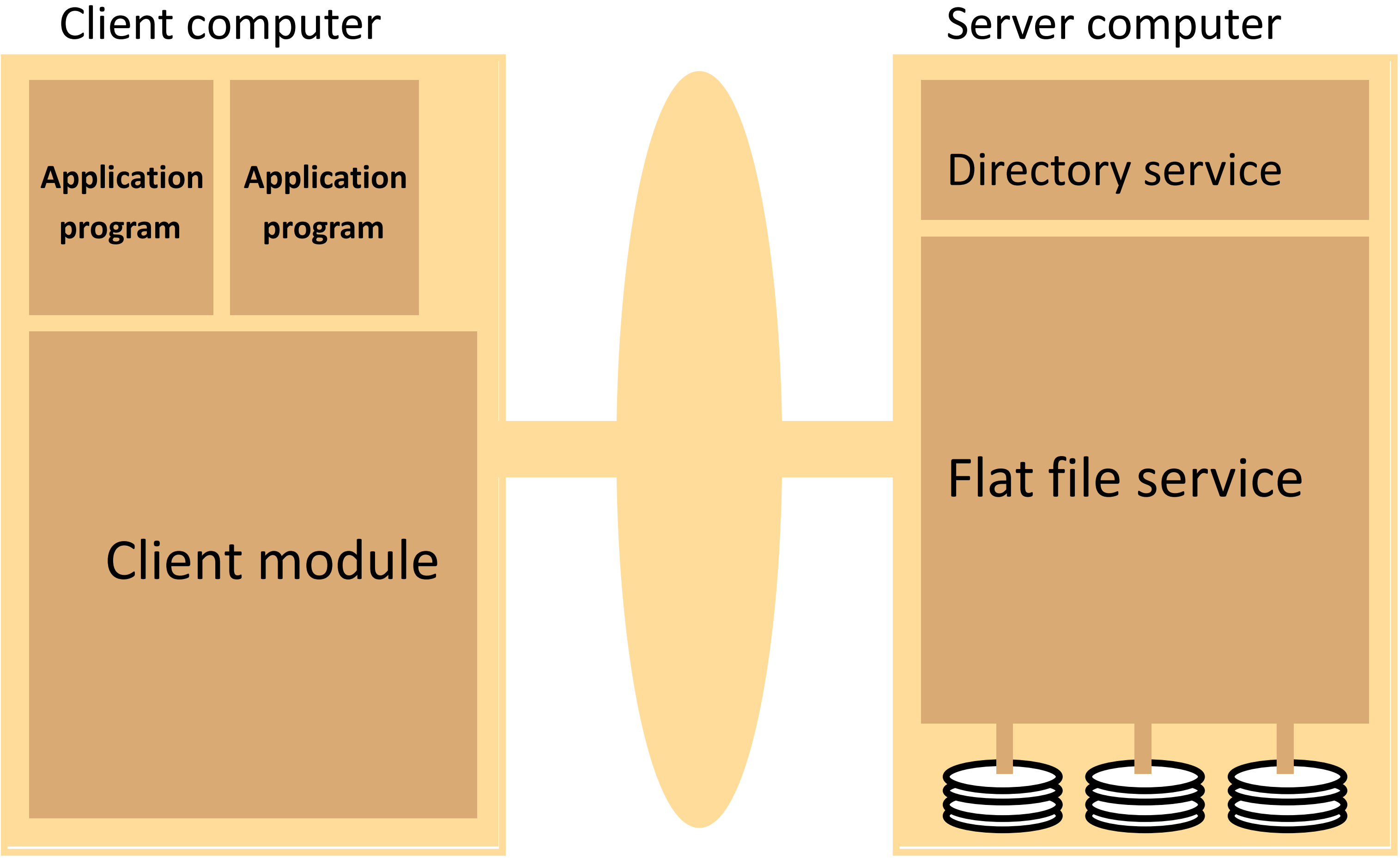
- Files contain both *data* and *attributes*.

Distributed File Systems

- Distributed file systems *support the sharing of information* in the form of files and hardware resources.
- A DFS enables programs to *store and access remote files/storage* exactly as local ones.
- The *performance and reliability* of such access should be comparable to that for files stored locally.
- Recent advances in higher bandwidth connectivity of switched local networks and disk organization have led to higher performance and *highly scalable file systems*.
- *Functional requirements:* open, close, read, write, access control, directory organization, etc.
- *Non-functional requirements:* scalable, fault-tolerant, secure

General Distributed File Service Architecture

- An architecture that offers a *clear separation* of the main concerns in providing access to files is obtained by structuring the file service as three components:
  - A *flat file service*
  - A *directory service*
  - A *client module*



Distributed File Service Architecture

- The *client module* implements exported interfaces of *flat file* and *directory services* available on the server side.

- The responsibilities of the various *modules* can be defined as follows:

- *Flat file service:*
  - Concerned with the implementation of *operations on the contents of file*. Unique File Identifiers (***UFID***s) are used to refer to files in all requests for flat file service operations. ***UFID***s are *long sequences of bits* chosen so that each file has a unique ID among all of the files in a distributed system.

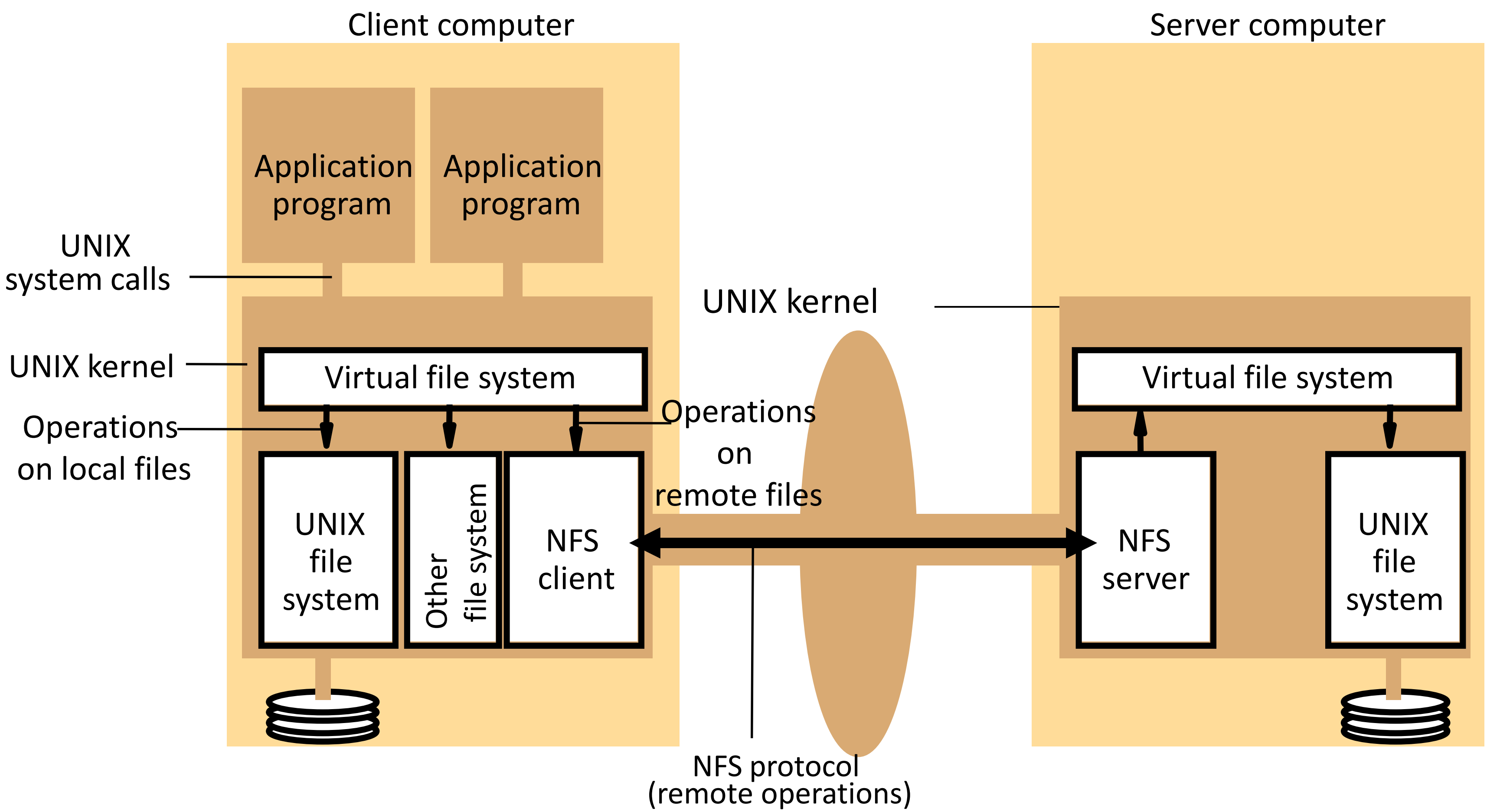
Distributed File Service Architecture

- *Directory service*
  - Provides mapping between *text names for the files* and their ***UFID***s. Clients may obtain the ***UFID*** of a file by quoting its text name to directory service. Directory service *supports functions needed to generate directories and to add new files to directories*.
- *Client module*
  - It runs on each computer and provides an integrated service (flat file and directory) as a *single API to application programs*. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
  - It holds information about the *network locations of flat-file and directory server processes*; and achieves better performance through the implementation of a *cache* of *recently used file blocks at the client*.

Distributed File System Examples

- Network File System (NFS) ~ *Sun Microsystems*
- Andrew File System (AFS) ~ *Carnegie Mellon University*
- Sprite File System ~ Berkeley
- Google File System (GFS)
- Hadoop Distributed File System (HDFS)

NFS Architecture



NFS Architecture

- The NFS client and server modules communicate using *Remote Procedure Calls (RPCs)*.

- Sun's RPC system
  - *Sun RPC* was developed for use in *NFS*. It can be configured to use either *UDP* or *TCP*, and the *NFS protocol* is compatible with both

Virtual file system (VFS)

- **NFS provides access transparency through VFS**
  - user programs *can issue file operations for local or remote files without distinction*
  - *Other distributed file systems* may be present that support UNIX system calls, and if so, they *could be integrated via VFS*
  - Distinguishes between local and remote file identifiers
  - *Keeps track of the available file systems* – both local and remote
- The virtual file system layer has *one VFS structure for each mounted file system and one v-node per open file*
  - *A VFS structure relates a remote file system to the local* directory on which it is mounted
  - The *v-node contains an indicator to show whether a file is local or remote*. If the file is local, the v-node contains a reference to the index of the local file (an *i-node* in a UNIX implementation). If the *file is remote*, it contains the file handle of the remote file.

Hierarchic file system

- A hierarchic file system *consists of a number of directories arranged in a tree structure*.
- Any file or directory can be *referenced using a pathname* – a multi-part name
- A UNIX-like file-naming system *can be implemented by the client module using the flat file and directory services* that we have defined.
- A tree-structured network of directories is constructed with *files at the leaves and directories at the other nodes* of the tree. The root of the tree is a directory with a ‘well-known’ ***UFID***.
- A function can be provided in the client module that gets the ***UFID*** of a file given its pathname. The *function interprets the pathname*.
    - Pathname starting from the root, using Lookup to obtain the UFID of each directory in the path.

Mount service

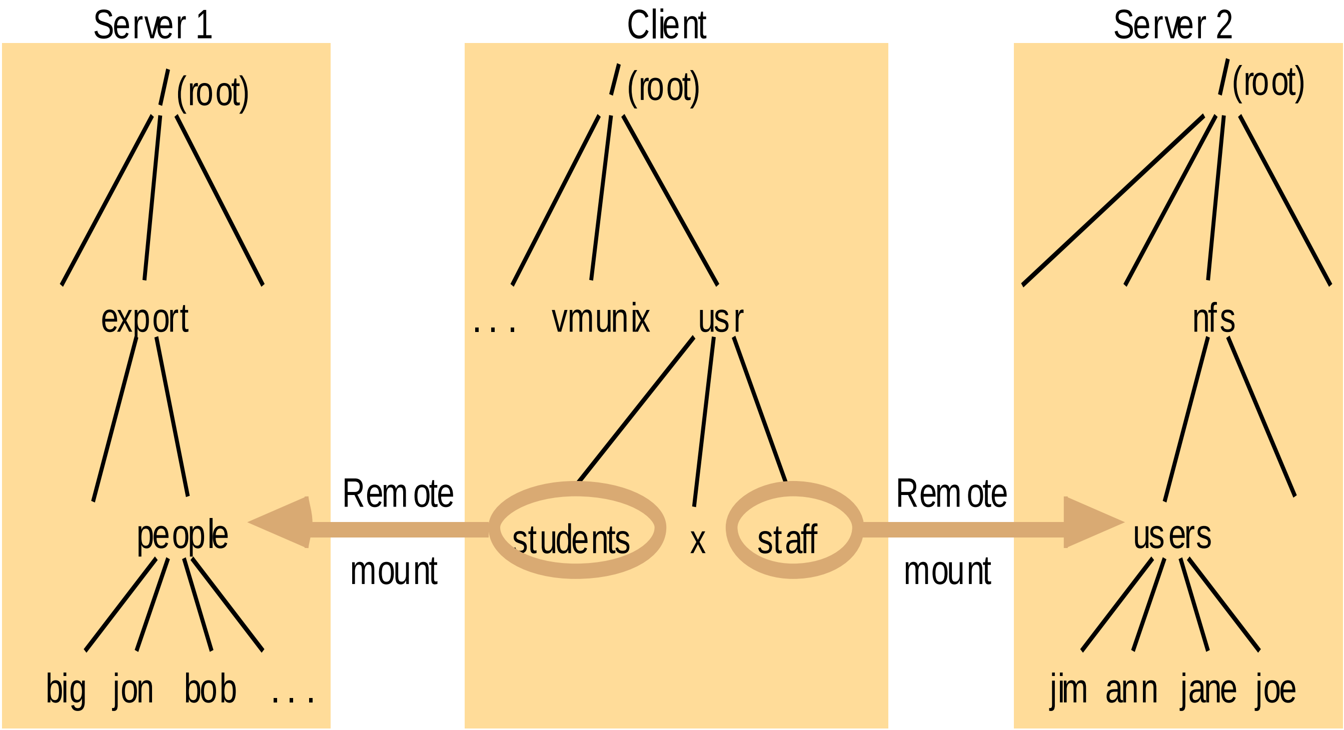
- The *mounting of subtrees* of *remote file systems* by clients is supported by a separate *mount service* process that runs at user level on each NFS server computer.
- On each server, there is a file with a well-known name (/etc/exports) containing the names of *local file systems* that are available for remote mounting.
- An *access list* is associated with each file system name indicating which hosts are permitted to mount the file system
- Mount operation:

```
mount(remotehost, remotedirectory, localdirectory)
```

- Each client maintains *a table of mounted file systems* holding:

```
< IP address, port number, file handle>
```

Mount Service



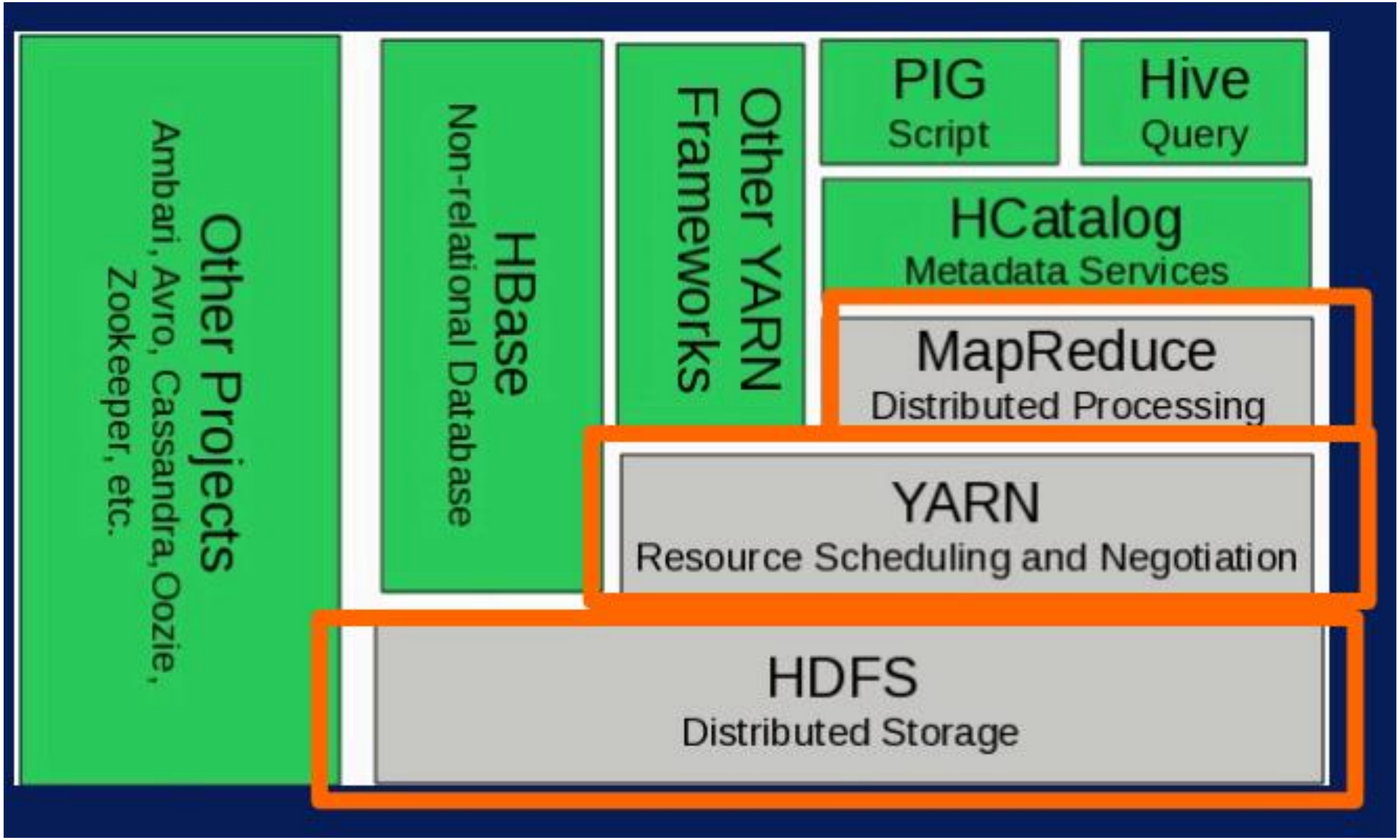
- The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1;
- the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Hadoop

- ***Apache Hadoop*** is an open source software framework for storage and large scale processing of data-sets on *clusters of commodity hardware*.

- It consists of the following basic modules:
  - *Hadoop Distributed File System (HDFS)*
  - *Hadoop YARN*
  - *Hadoop MapReduce*

Hadoop Module



MapReduce

How can we parallelize data processing across many machines?

MapReduce - Parallelizing Programs

- ***Task***: we want to count the frequency of words in a document
- ***Possible Approach***: program that reads document and builds a
  - word → frequency map

How can we parallelize this?

Idea: split document into pieces, count words in each piece concurrently

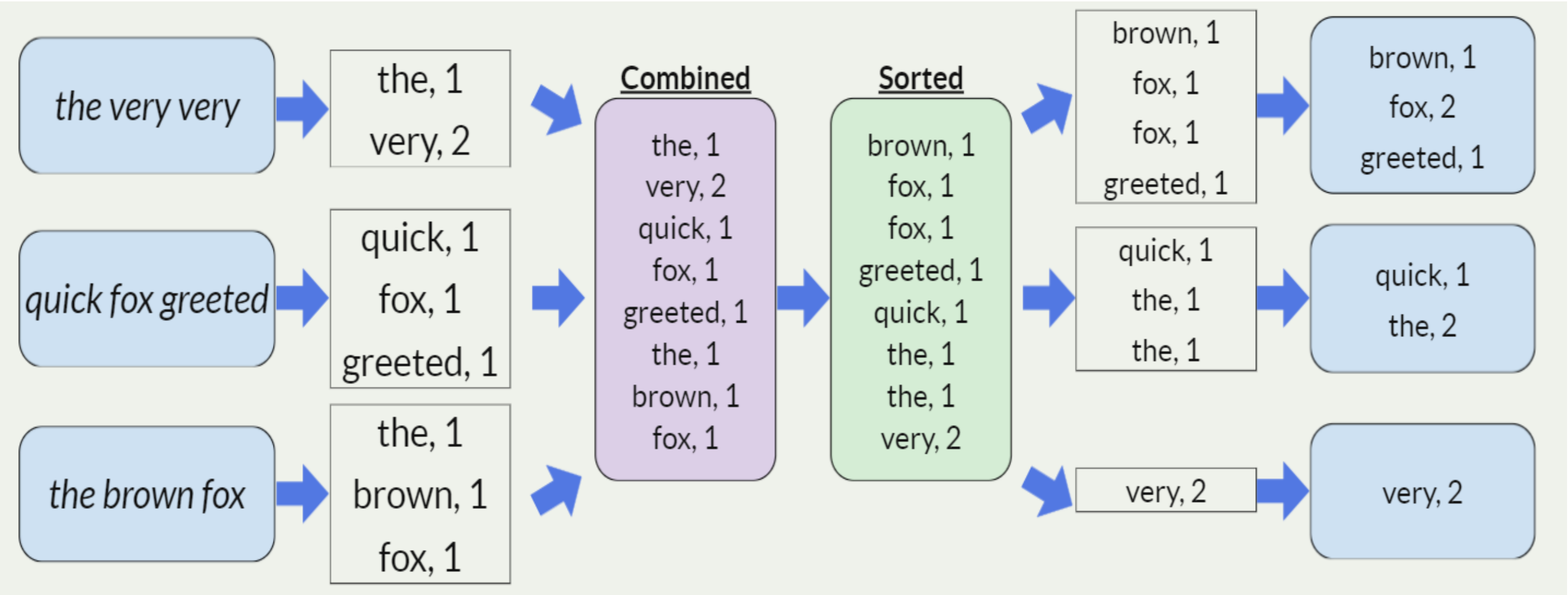
Problem: what if a word appears in multiple pieces? We need to then merge the counts

Idea: combine all the output, sort it, split into pieces, combine in each one concurrently

Example: Counting Word Frequencies

- Idea: *split documents into pieces, count words in each piece concurrently.* Then, *combine* all the text output, *sort* it, *split* into pieces, *sum each one concurrently*.

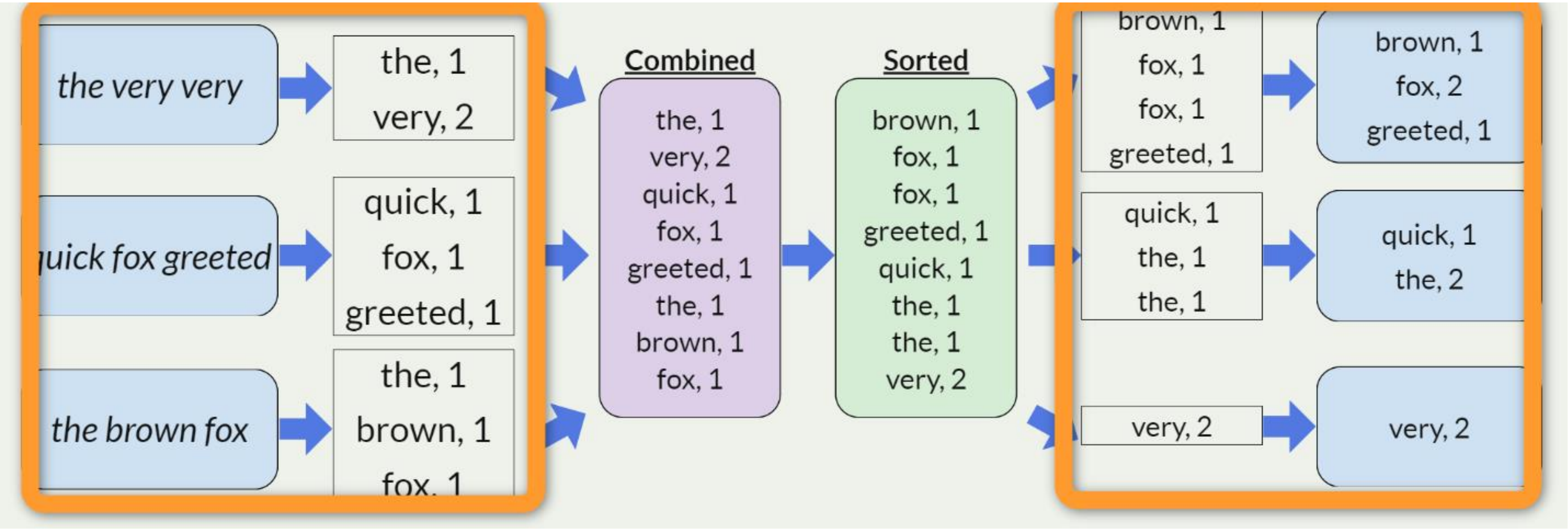
Example: "the very very quick fox greeted the brown fox"



Example: Counting Word Frequencies

2 "phases" where we parallelize work

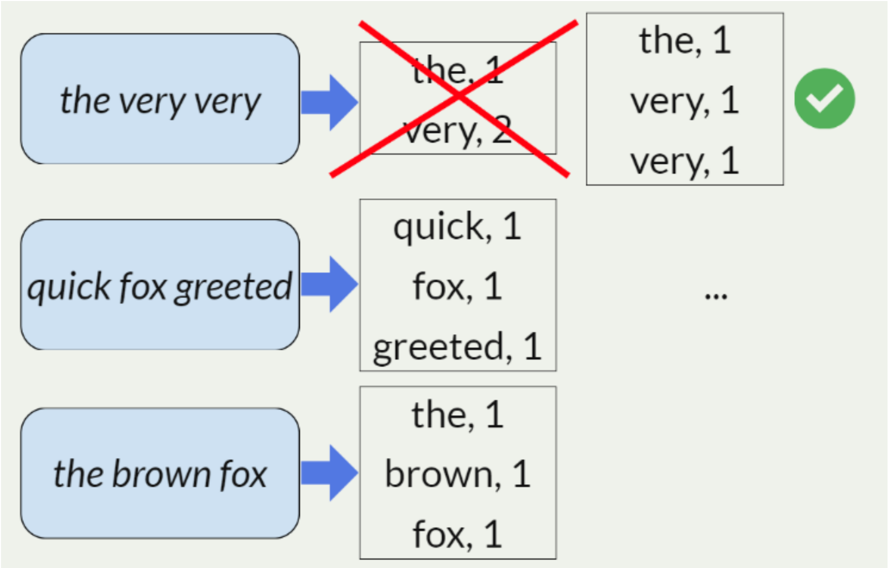
1. ***Map*** the input to some intermediate data representation
2. ***Reduce*** the intermediate data representation into final results



Example: Counting Word Frequencies

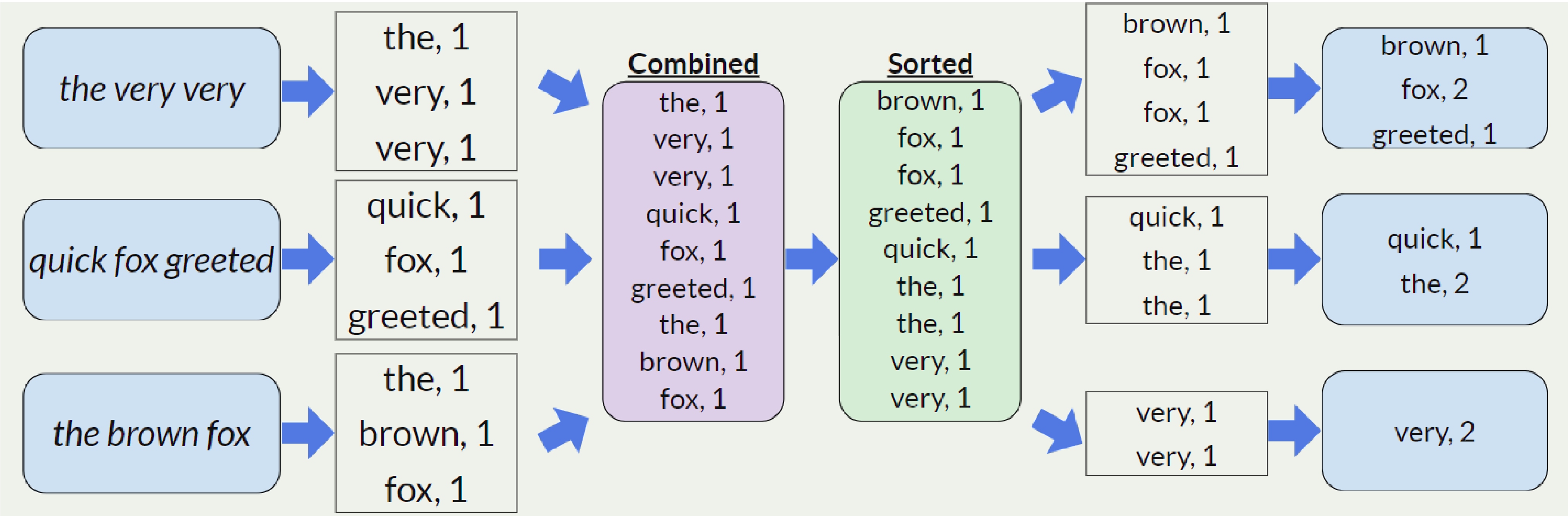
- The first phase focuses on *finding*, and the second phase focuses on *summing*. So the first phase should only *output 1s*, and leave the *summing for later*.

Example: "the very very quick fox greeted the brown fox"



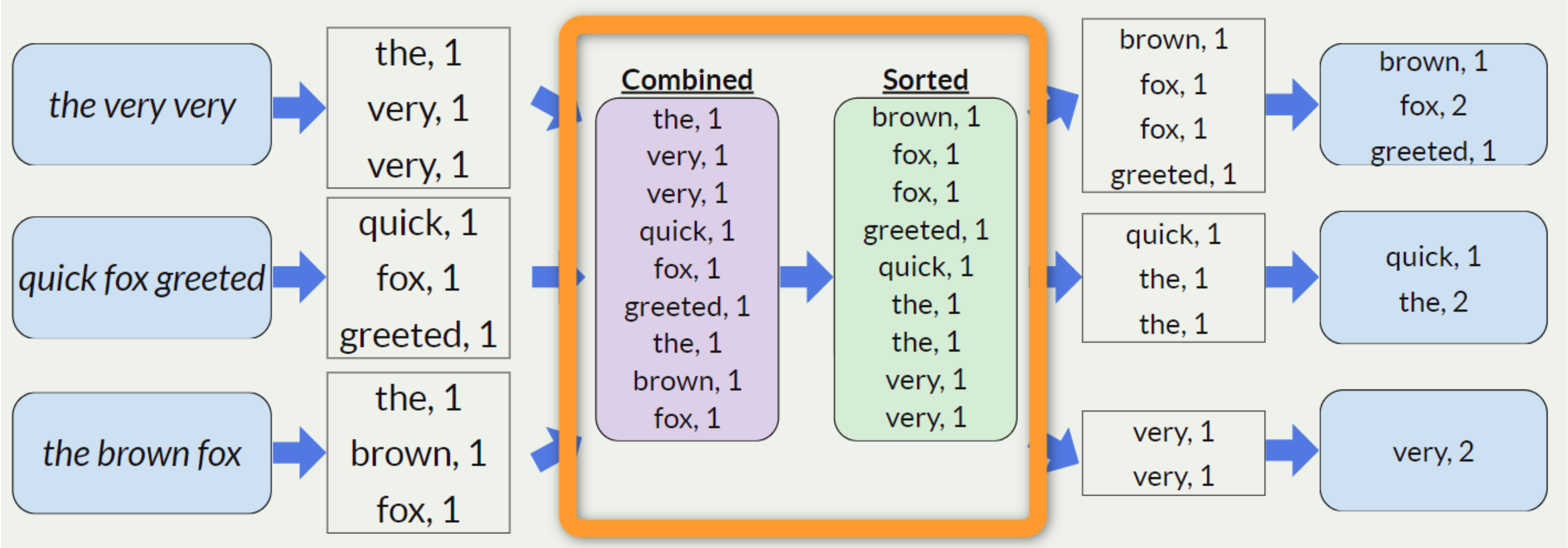
Example: Counting Word Frequencies

- 2 “phases” where we parallelize work
  - ***Map*** the *input* to some *intermediate data representation*
  - ***Reduce*** the *intermediate* data representation into ***final results***



Example: Counting Word Frequencies

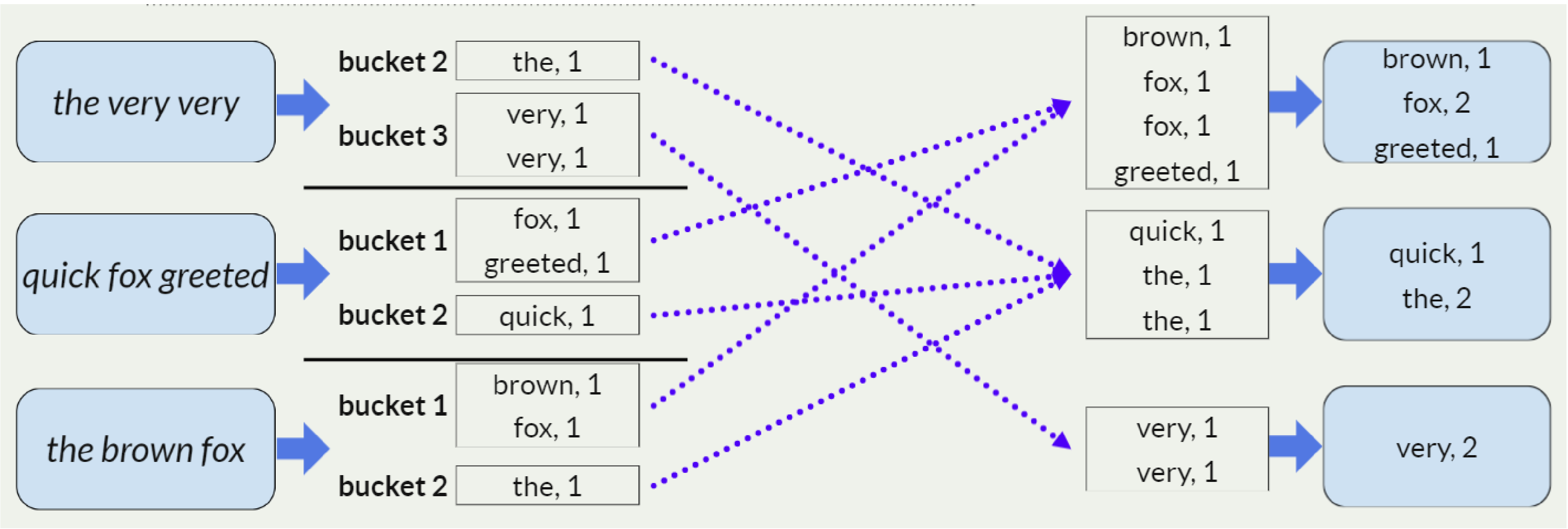
- Question: is there a way to *parallelize* this *operation* as well?
- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.



Example: Counting Word Frequencies

- Question: is there a way to *parallelize* this *operation* as well?
- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.

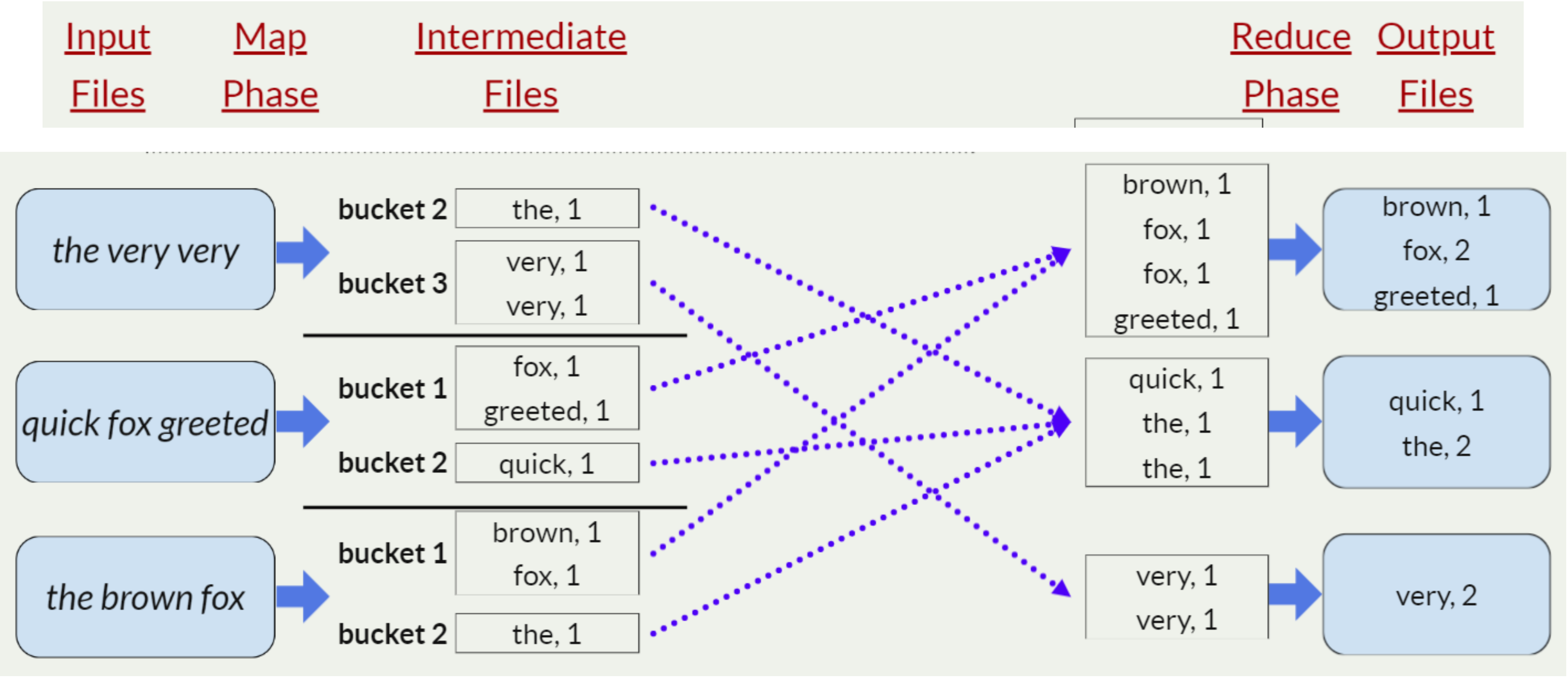
bucket # = *hash(key) % R*   where R = # reduce tasks (3)



Example: Counting Word Frequencies

- Idea: have each *map* task separate its data in advance for each *reduce* task. Then each *reduce* task can *combine* and *sort* its own data.

bucket # = *hash(key) % R* where R = # reduce tasks (3)



References

1. Slides of Dr. Rana Asif Rehman & Dr. Haroon Mahmood

Helpful Links:

1. https://hadoop.apache.org/
2. https://www.cloudera.com/products/open-source/apache-hadoop.html
3. https://www.techtarget.com/searchenterprisedesktop/definition/Network-File-System