Introduction to HDFS

Bachar Wehbi me@bachwehbi.net

Big Data: Hadoop

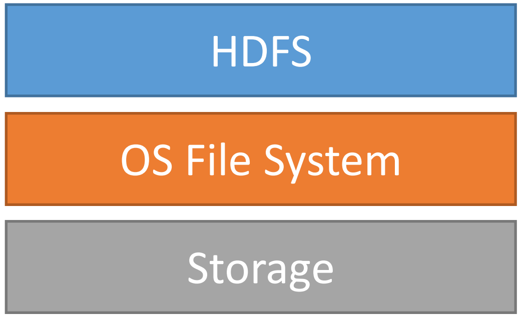
- Hadoop is a framework for distributed data storage and processing at scale
- The core components of Hadoop include HDFS for data storage and YARN for resource management
- Hadoop is also a rich ecosystem with many complementary components
  - Ingestion (Sqoop, Flume, Kafka)
  - Storage (HDFS, Hbase, Kudu)
  - Processing (Spark, MapReduce)
  - Querying (Presto, Drill, Impala, Hive)
  - Exploring and Search (Hue, Solr)

Hadoop Cluster

Cluster of nodes running Hadoop at Yahoo! (Source Yahoo!)

HDFS

- HDFS is a distributed massively scalable file system
- Based on Google File System (GFS)
- Very large files, 128 MB chunks
- Runs on top of native file systems
  - Runs in User Space
  - Heterogeneous Hardware and Software Platforms
- Major improvements and new features in Hadoop V3.x



HDFS

- 21 Petabytes in HDFS as of 2010 at Facebook
- 100+ Petabytes in HDFS as of 2012 at Facebook
- Yahoo! more than 100,000 CPU in over 40,000 servers running Hadoop (multiple clusters, biggest 4500 nodes)
- 455 Petabytes in HDFS at Yahoo!
- Today, most of the Big Names have Hadoop and store their data in HDFS
- HDFS today is the de-facto storage for Data lakes

HDFS Assumptions and Goals

- Designed for large data sets
  - Cluster thousands of nodes, millions of large files, tens of PB
- Hardware failure is the rule not the exception
  - Nodes may fail at any time: uses replication to cope with node failure
    - Mean time between failures for 1 node = 3 years
    - If you have a 1000 nodes cluster ➔ 1 failure per day
  - Fault tolerance: detect failures and recover from them

HDFS Assumptions and Goals

- Data Locality: Moving Computing is Cheaper than Moving Data
  - Network bandwidth is neither unlimited nor free
  - Execute jobs where data is stored instead of moving it to computing nodes
- Optimized for Batch processing
  - Provides very high throughput access: high aggregate read data rate instead of low latency
- Designed for write once read many datasets
  - Not designed for operational data
  - Data can be appended but never updated

HDFS: Storing Data

- Files are divided into blocks of 128 MB (default, can be configured)
- Data Blocks are replicated to 3 DataNodes (default, can be configured)
- Name Node selects Data Nodes to host a block on load time
- Client sends the block to the first Data Node in the list
- The replication process is pipelined from one data node to another
  - A Data Node can be receiving data from a client or another data node and forwarding it to another data node at the same time

HDFS Architecture

Name Node maintains metadata

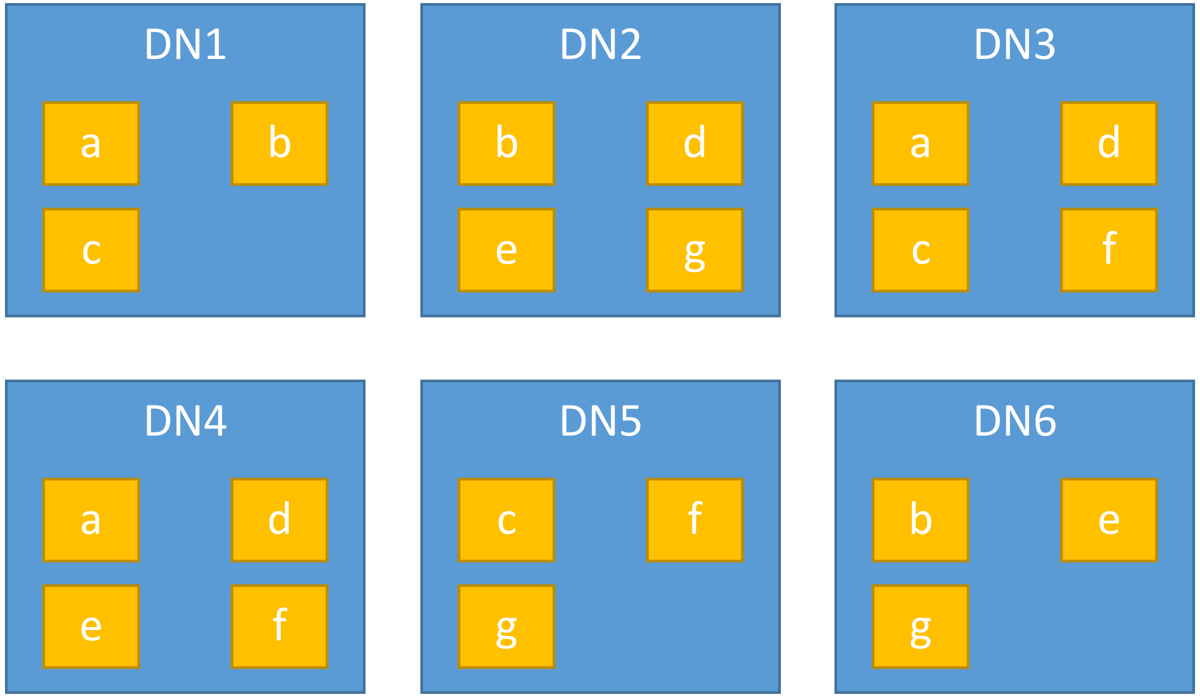
Name Node

Metadata

DN1: a, b, c
DN2: b, d, e, g
DN3: a, c, d, f
DN4: a, d, e, f
DN5: c, f, g
DN6: b, e, g

/log/log.data: e, f, g (3)
/data/f1.csv: a, b, c, d (3)

Data Nodes hold replicated blocks of data



HDFS: Name Node Metadata

- Name Node keeps metadata in memory
  - Very efficient to reply to client requests
- Metadata includes:
  - List of files
  - For every file, list of blocks and block attributes (ex: location)
- Transaction log
  - Reporting: file creation, file deletion, etc.
- HDFS Federation

HDFS: Data Node

- Stores blocks of data on local file system
- Stores metadata of blocks like block ID, CRC32 checksum, block length
- Serves block data and metadata to clients
- Validates periodically block checksum to detect data corruption
- Participates in data pipelining
  - Forwards data to other specified Data Nodes
- Sends period block report to Name Node
  - Includes list of all existing blocks
- Sends periodic heartbeat message to Name Node
  - To indicate it is alive

HDFS: Data Node Heartbeats

- Signal sent by the Data Node to the Name Node at regular interval (by default 3sec)
- Indicates the presence of the Data Node and it is alive
- If after a certain period (default 10 min) of last heartbeat the Name Node do not receive any message from the Data Node, it considers it is dead.
    - It also considers all data blocks hosted by that Data Node to be unavailable
    - Name Node schedules the creation of new replicas of those blocks on other Data Nodes.

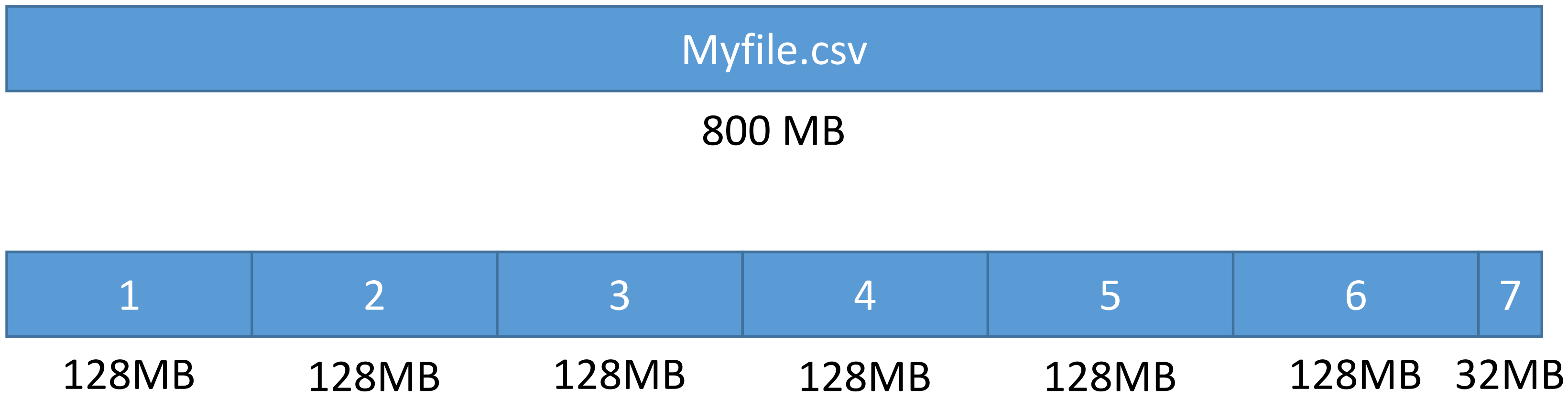
HDFS: Data Node Heartbeats

- Heartbeats includes information about:
  - total storage capacity, fraction of storage in use, and the number of data transfers currently in progress, cache capacity and in use.
  - This is used by the Name Node to select the best Data Node when replying to client requests.
- Based on Heartbeat messages, Name Node can issue following commands to Data Nodes:
  - Block recovery command: to recover specific blocks (when writing to HDFS)
  - Block command: To transfer block to another node or to invalidate certain blocks
  - Cache & Uncach command: to cache or uncache certain blocks

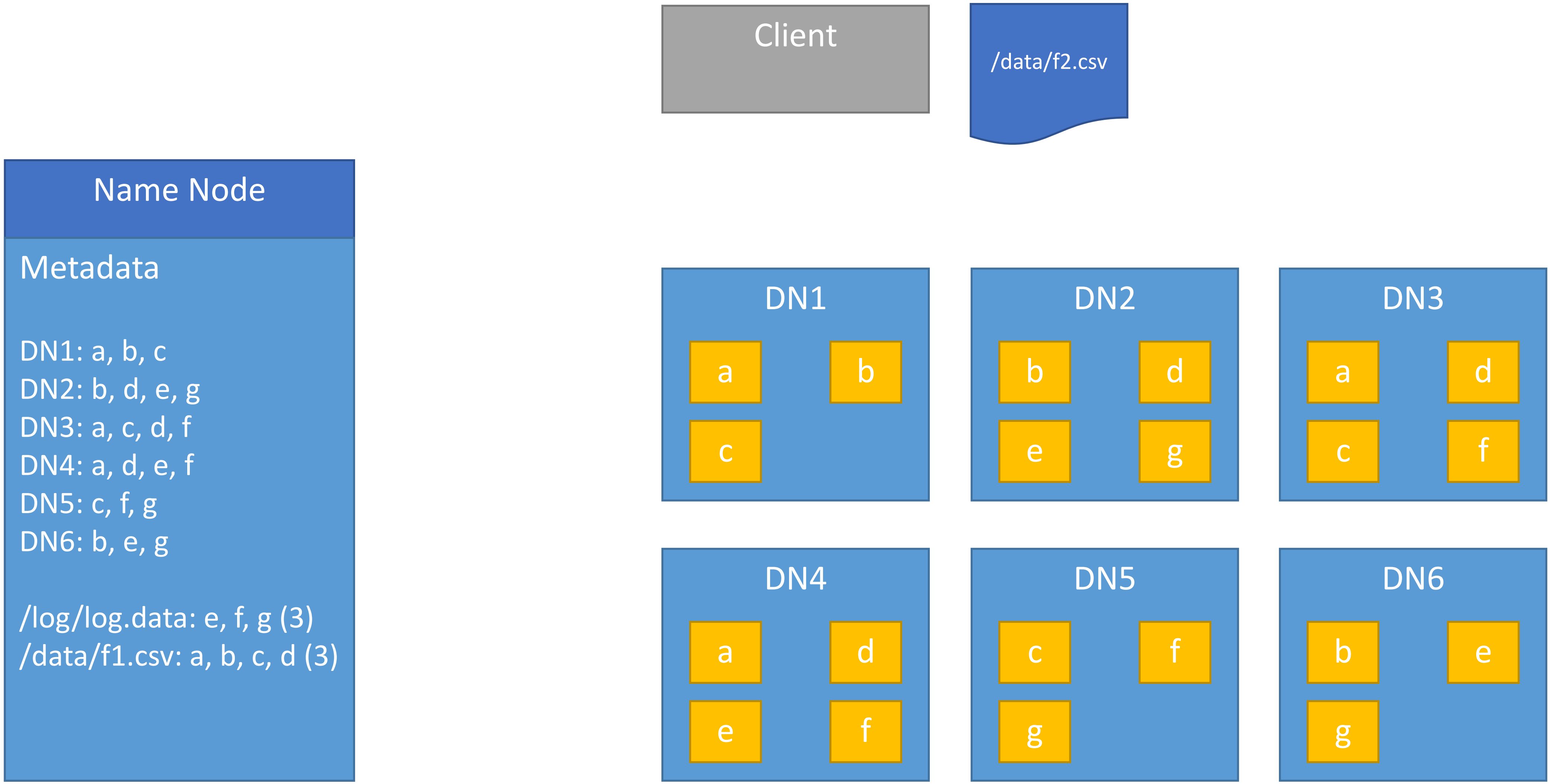
HDFS: Data Blocks

- Data Blocks are continuous location on drive where data is stored (similar to any file system).
  - Blocks in Hadoop v.2+ are by default 128MB large
  - Blocks in HDFS can however be of any size up to the configured maximum
- Why HDFS has a large block size when Linux has 4KB blocks?
  - We are talking about huge datasets (in Terabytes or Petabytes).
  - Having smaller Blocks implies too much metadata (to map files to blocks and blocks to their location)
  - The overhead will be huge!

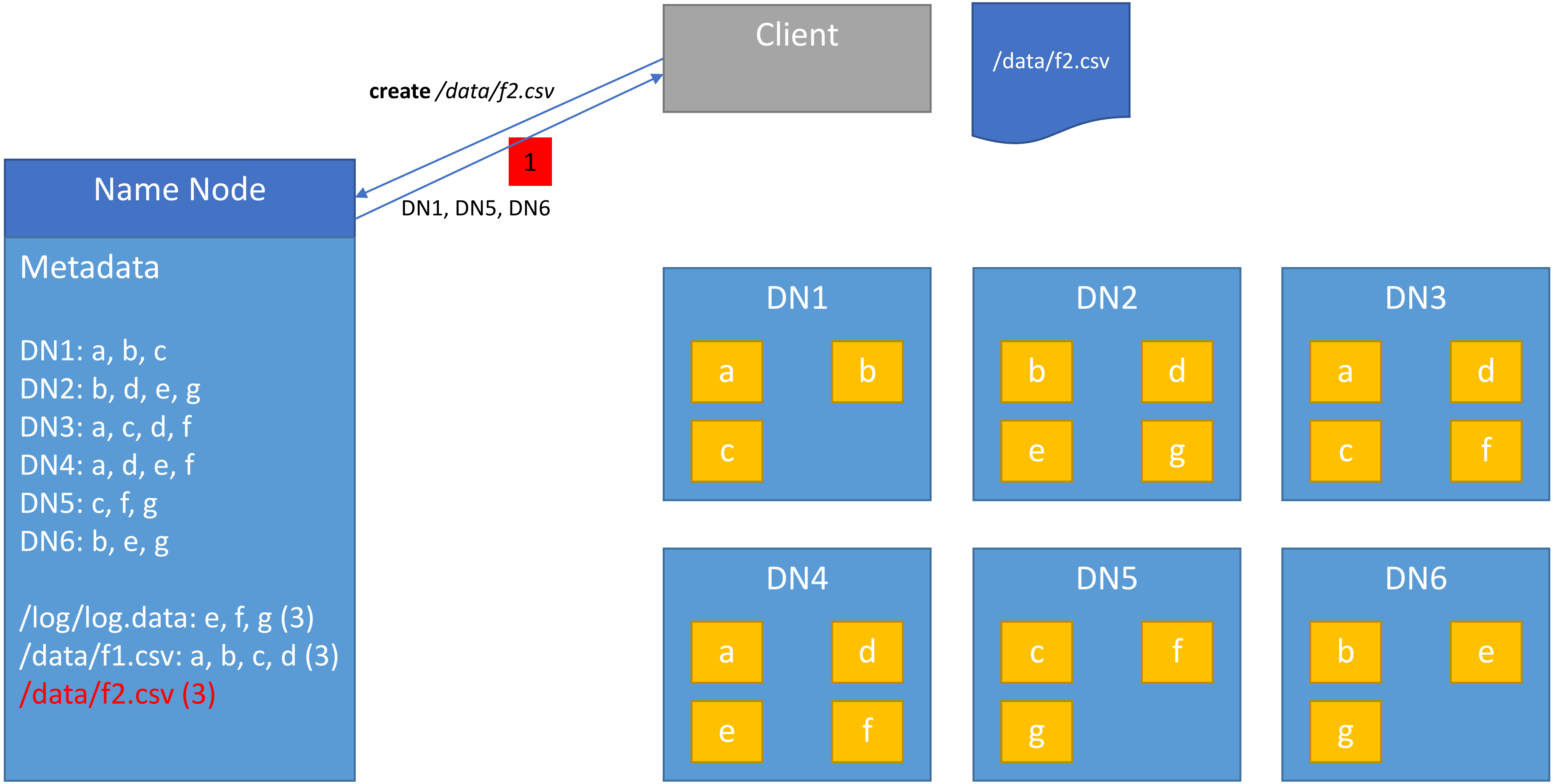
HDFS: Data Blocks



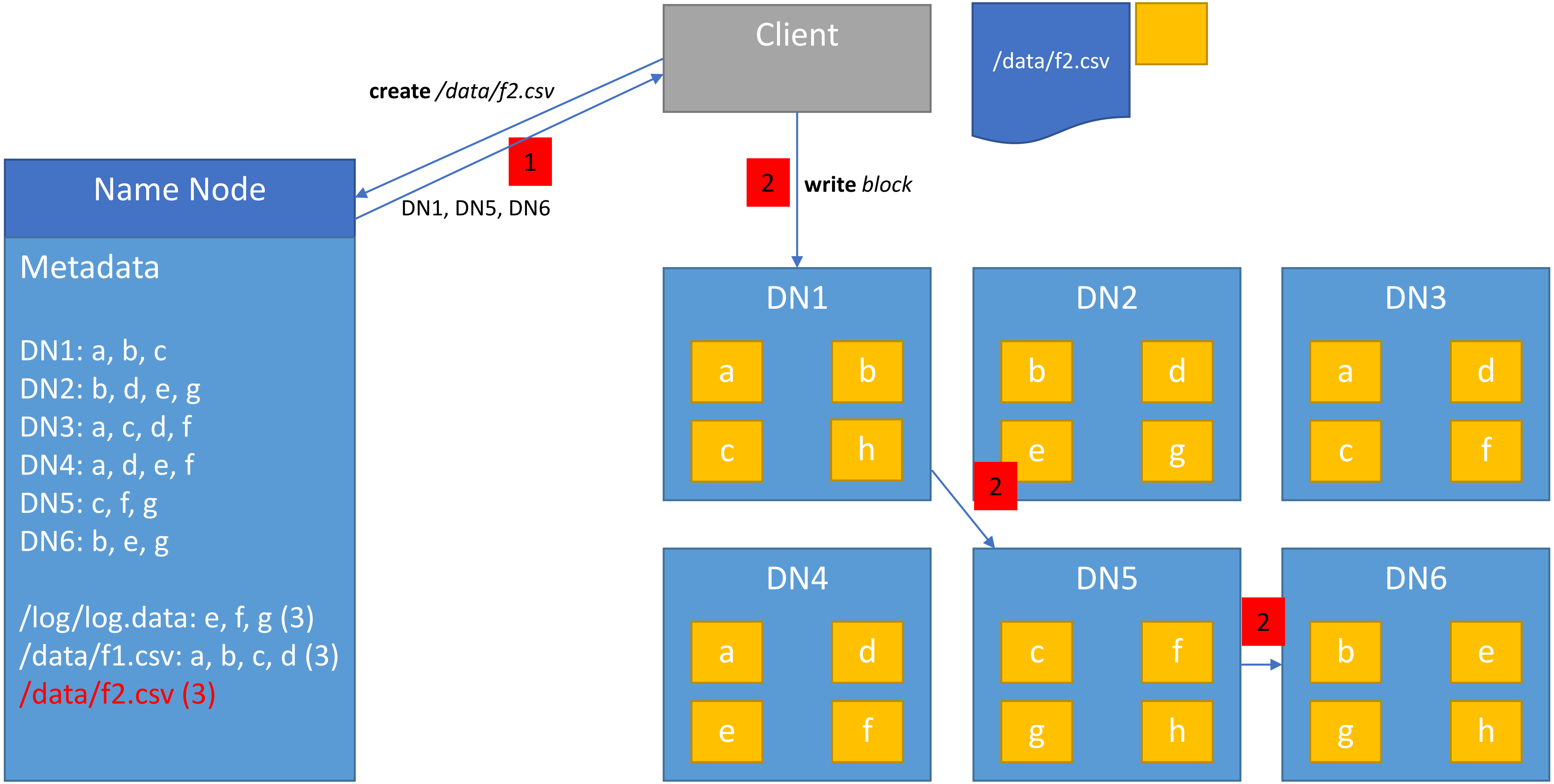
HDFS Writing Data



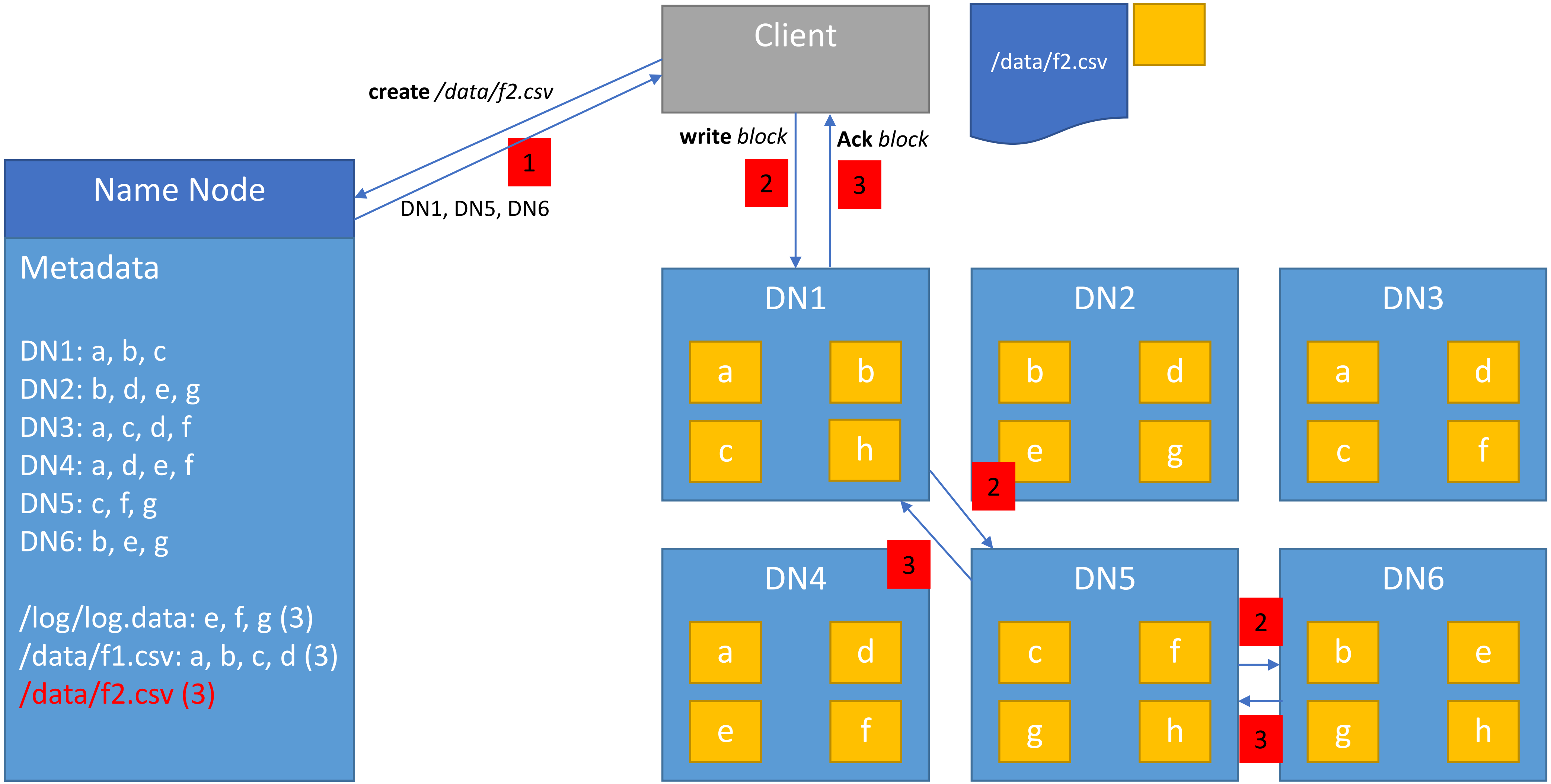
HDFS Writing Data



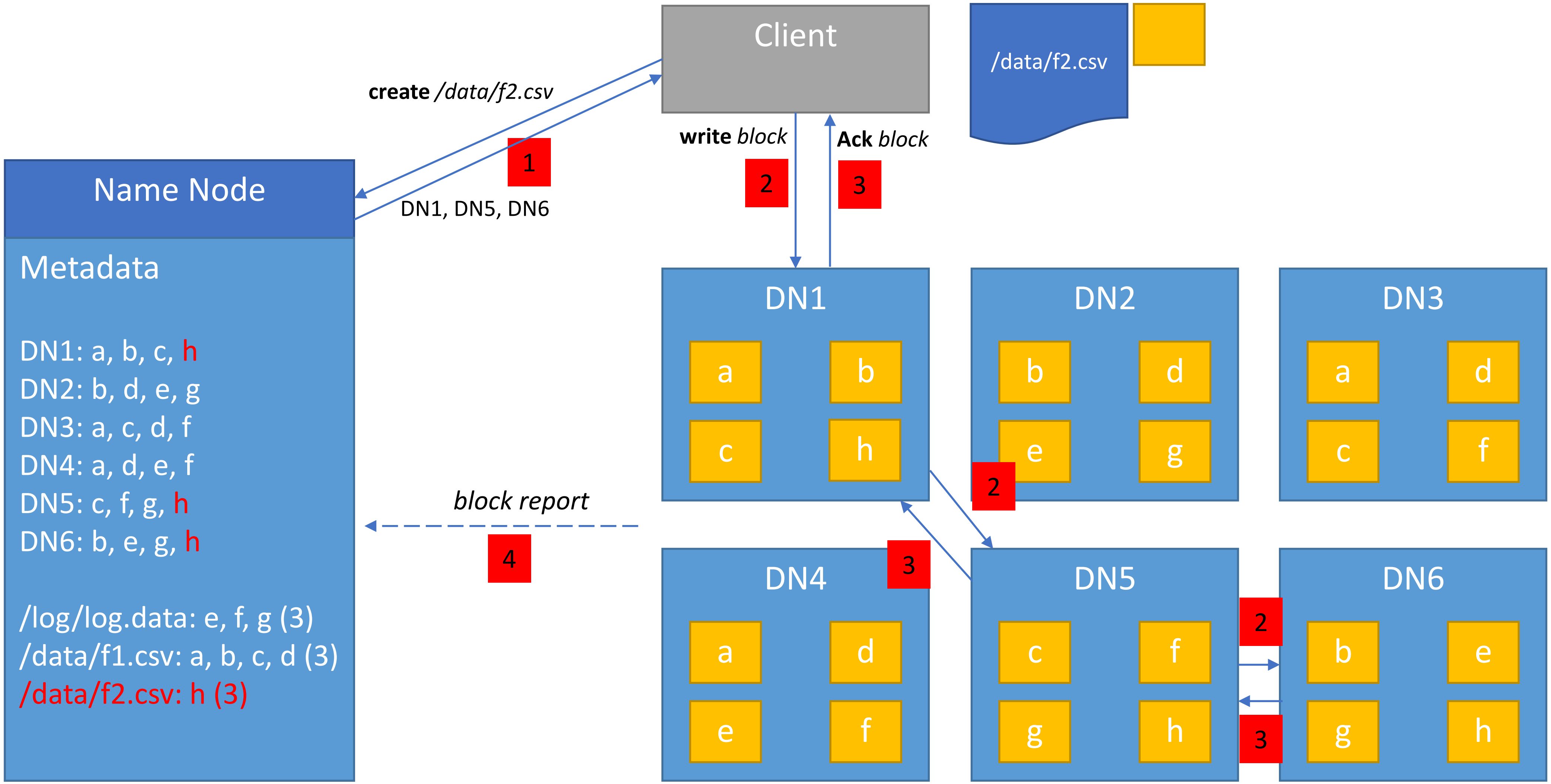
HDFS Writing Data



HDFS Writing Data



HDFS Writing Data

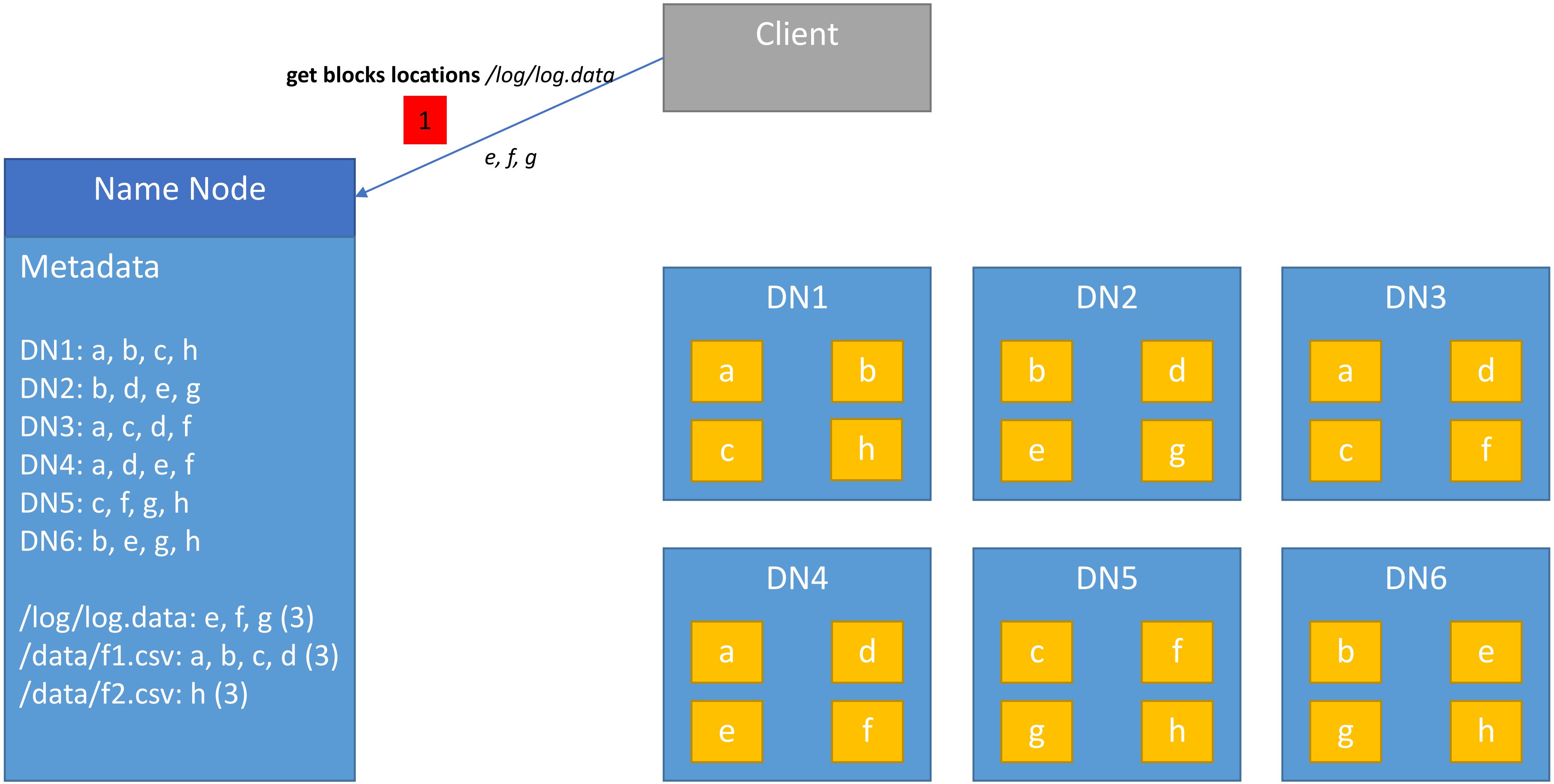


HDFS Writing Data

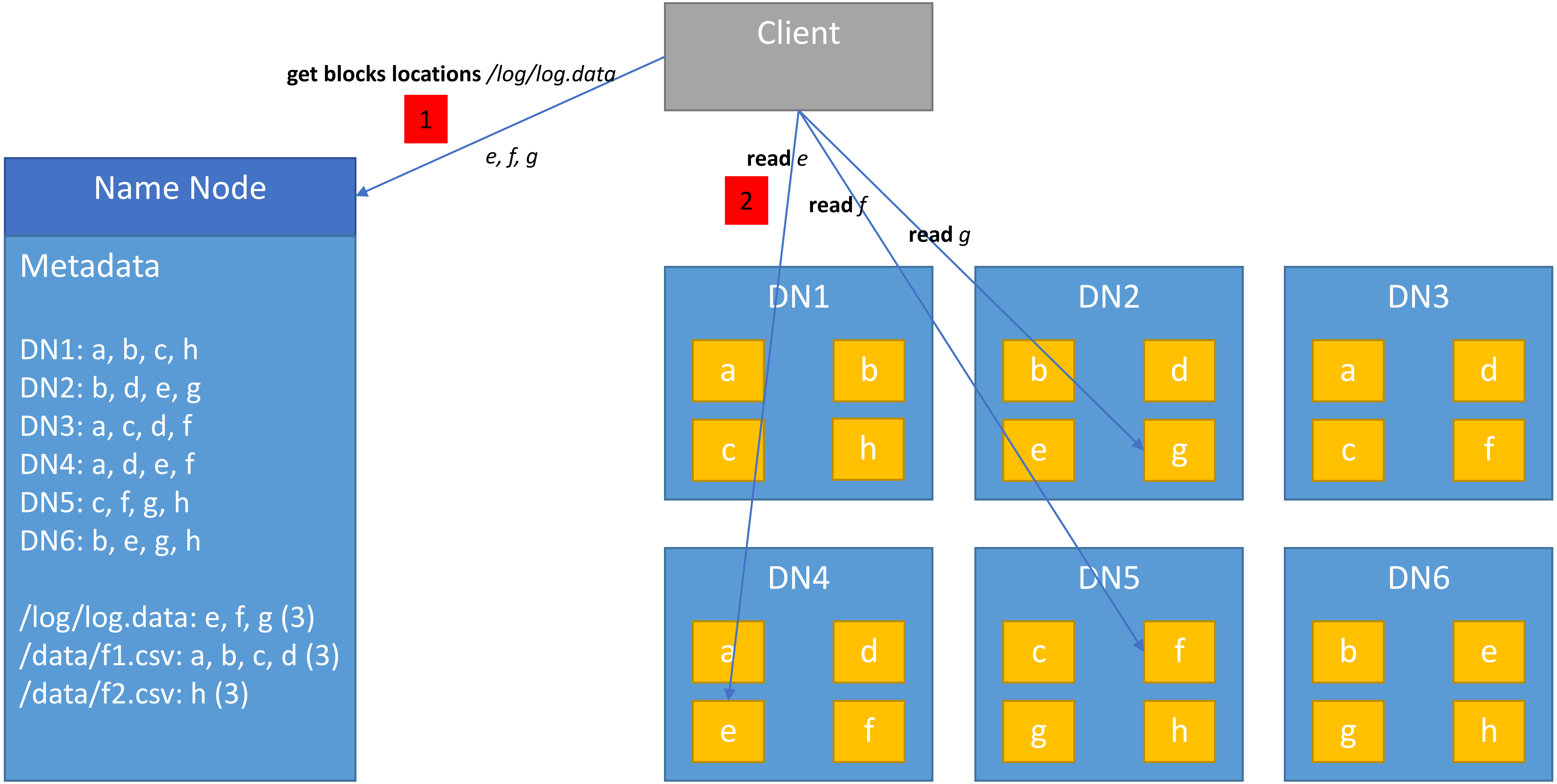
- When an HDFS client wants to write data, it follows the following procedure

1. Client calls the Name Node to create the file
   - Name Node verifies the file does not exist and the client is authorized
   - Adds new file with no blocks to the metadata
   - Name Node provides the Data Nodes where to write the blocks
2. Client writes blocks to Data Nodes
   - Data Nodes implicated in the replication pipeline forward blocks
   - Data Nodes in the pipeline acknowledge the block write
3. Data Nodes report back to the Name Node about new blocks
4. Name Node updates metadata with file blocks list and their location

HDFS Reading Data



HDFS Reading Data



HDFS Reading Data

- When an HDFS client wants to read data, it follows the following procedure

1. Client calls the Name Node to request blocks of the file to read
    - Name Node verifies the file exist and the client is authorized
    - Name Node replies with list of blocks of requested file
        - For every block, the Name Node indicates locations and best one
2. For every block, client read block from best node

HDFS Reading Data

- What happens when an error is encountered while reading from a Data Node

1. The client will try to fetch data from the next closest Data Node
2. The client will remember about the Data Node in order not to read data from in the future

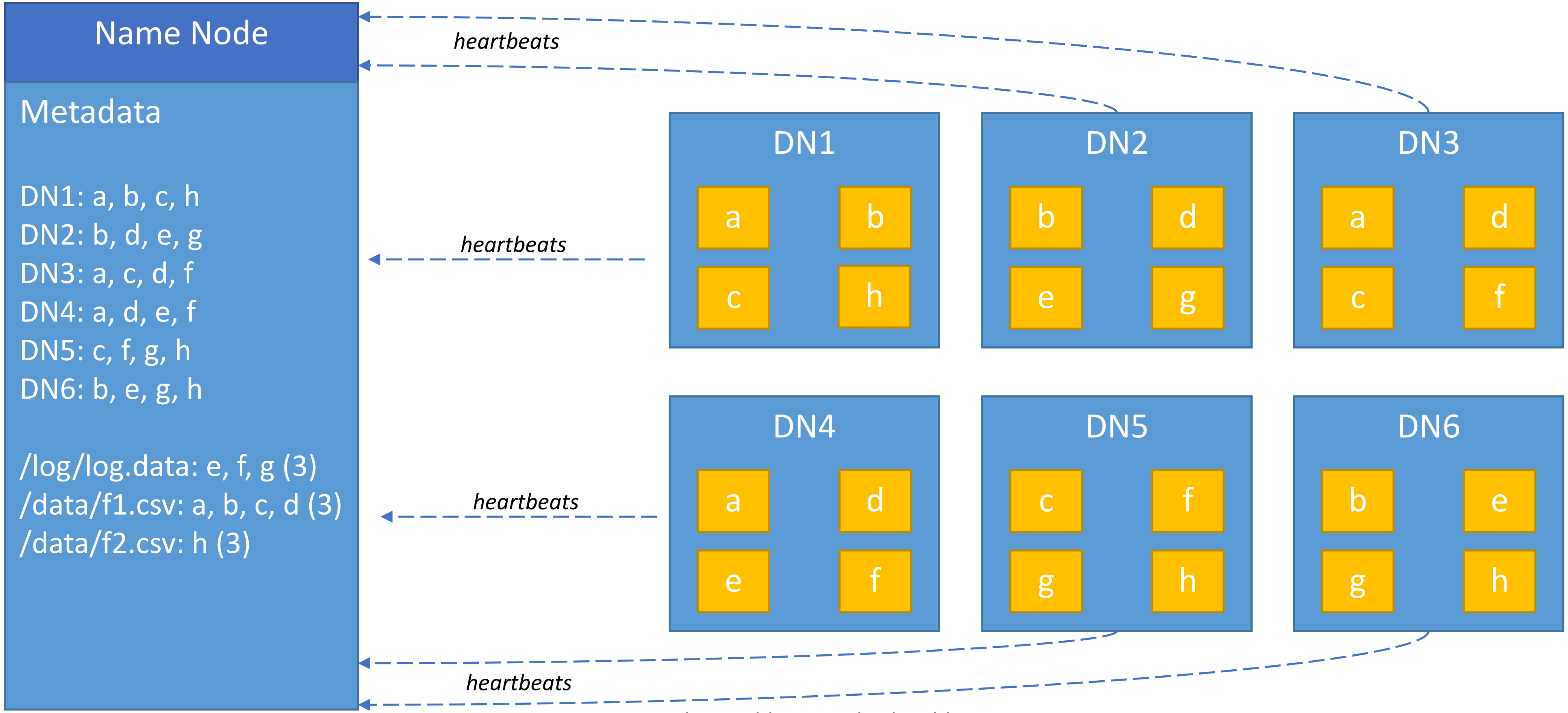
- When reading data, the client checks block checksum

1. If a corrupt block is found (reported checksum different than computed one)
2. Client reports error to the Name Node

HDFS - Robustness

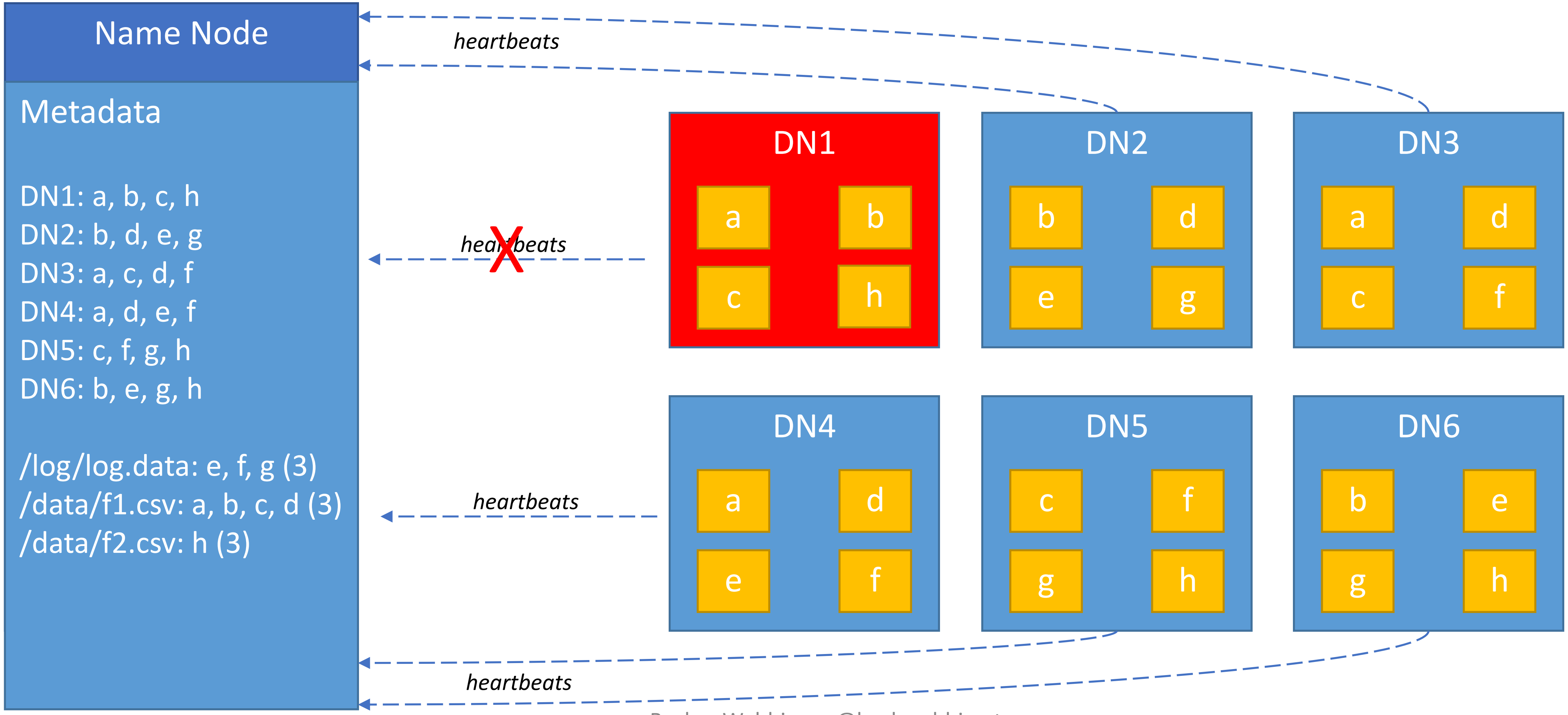
- The main objective of HDFS is to store data reliably even in the presence of failure.
- Three common types of failure are:
  - Name Node failure
  - Data Node failure
  - Network partitioning
    - Causing a subset of Data Nodes to loose connectivity with the Name Node
- Let’s see how HDFS behaves in front of these failures

HDFS Robustness: Data Node Failure

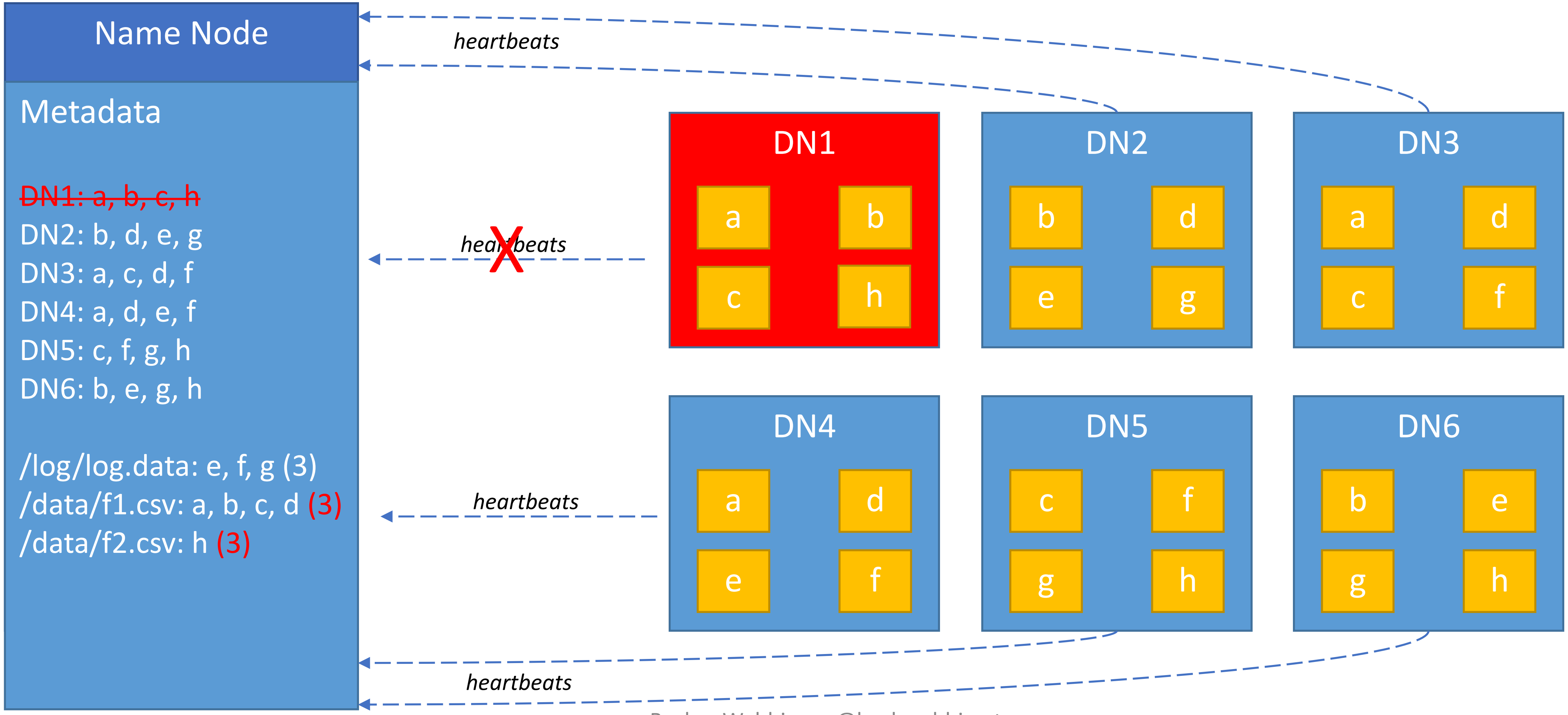


HDFS Robustness: Data Node Failure

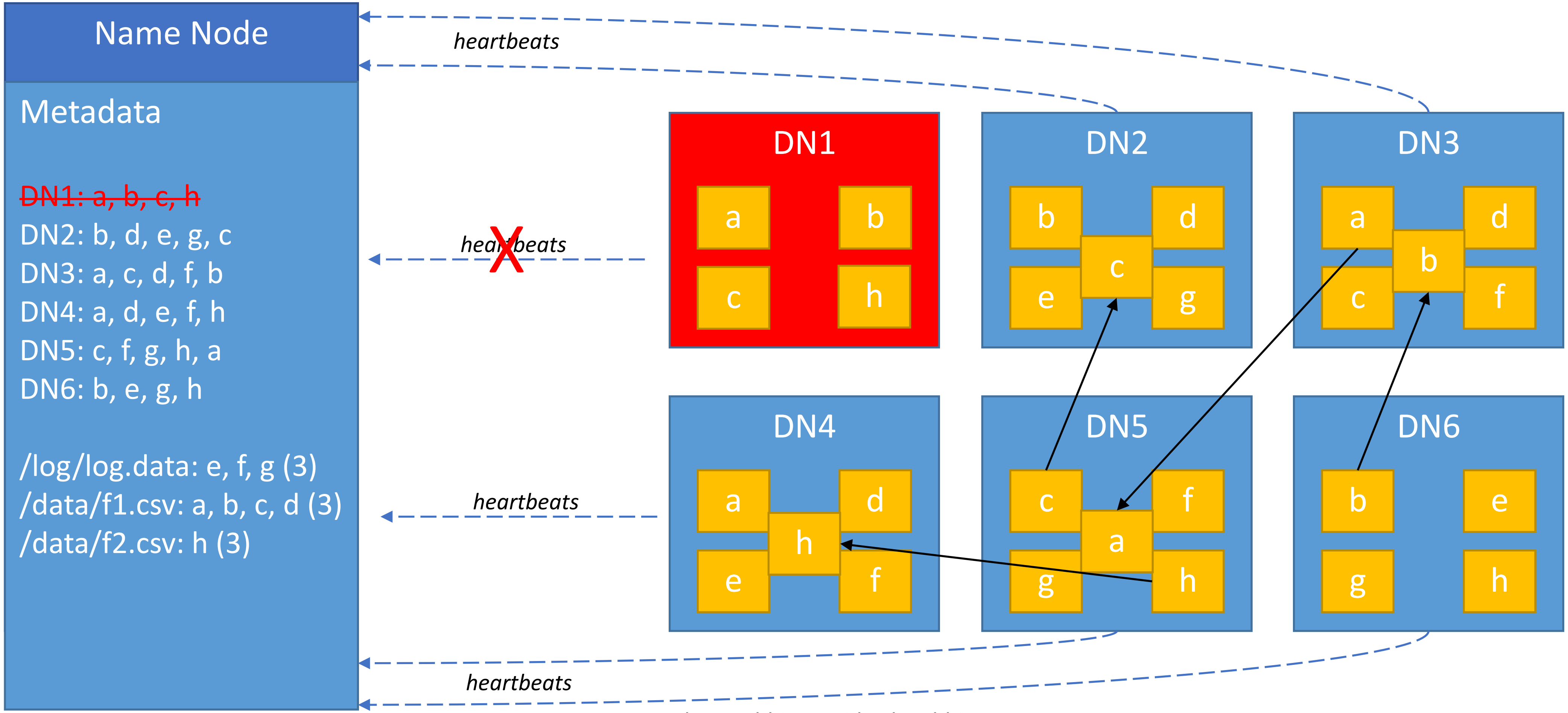
Network Partitions can cause multiple Data Nodes to lose connectivity with the Name Node



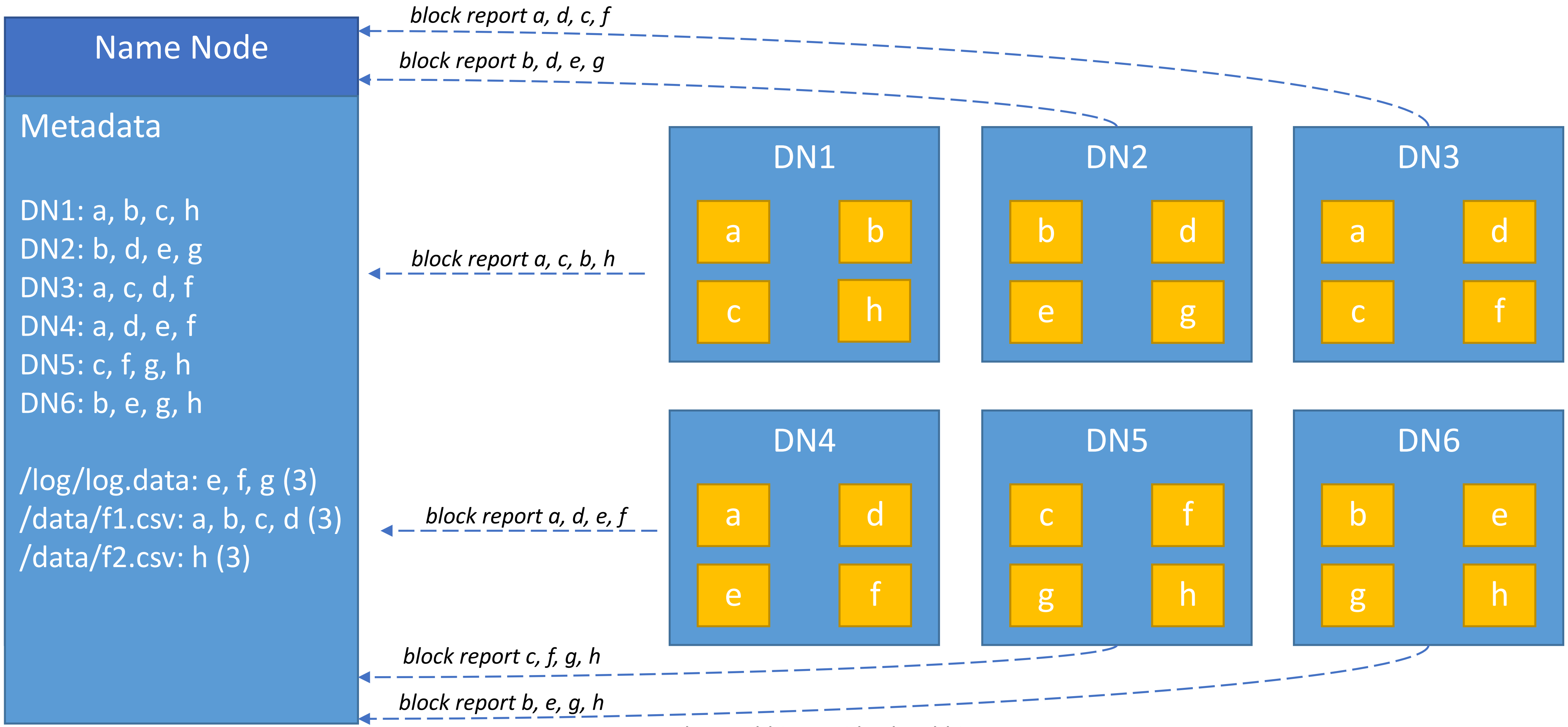
HDFS Robustness: Data Node Failure



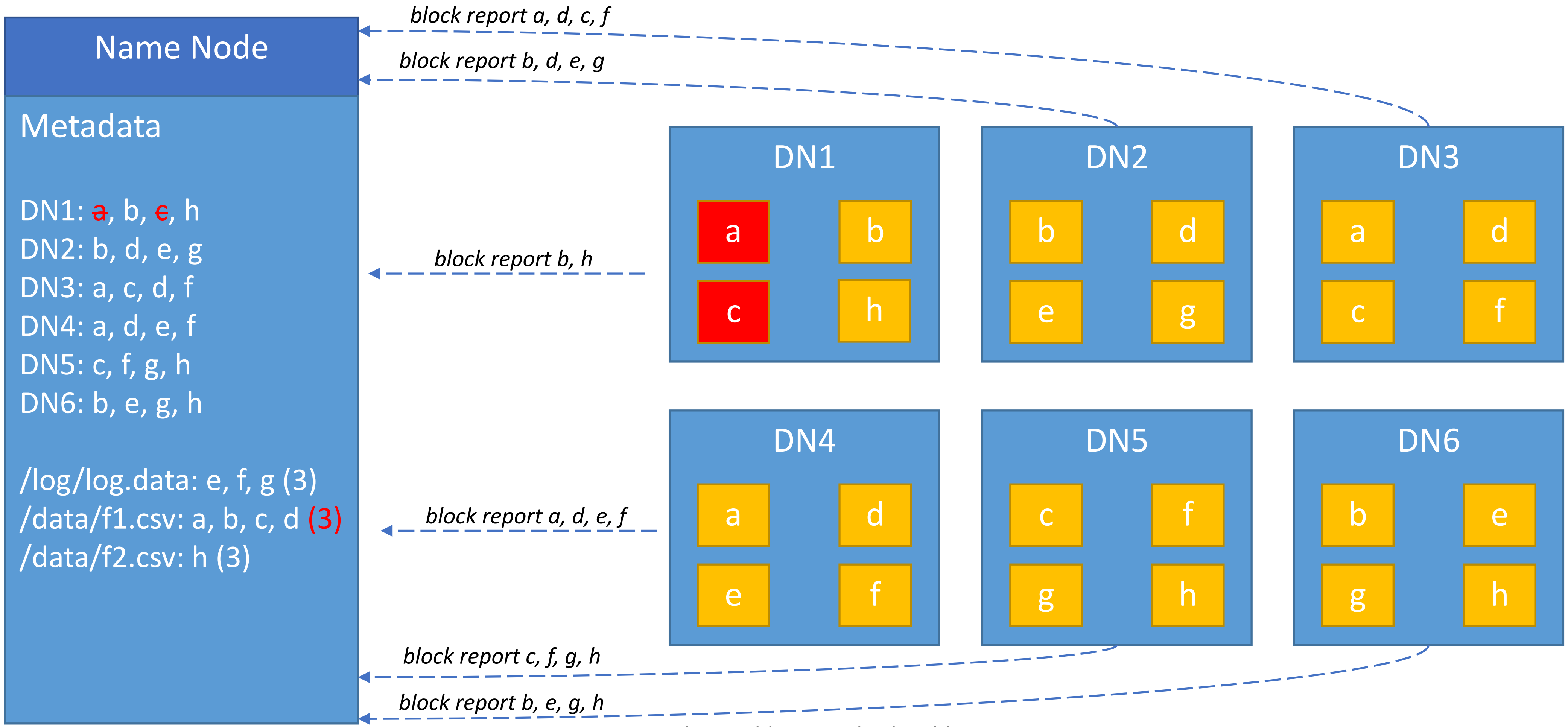
HDFS Robustness: Data Node Failure



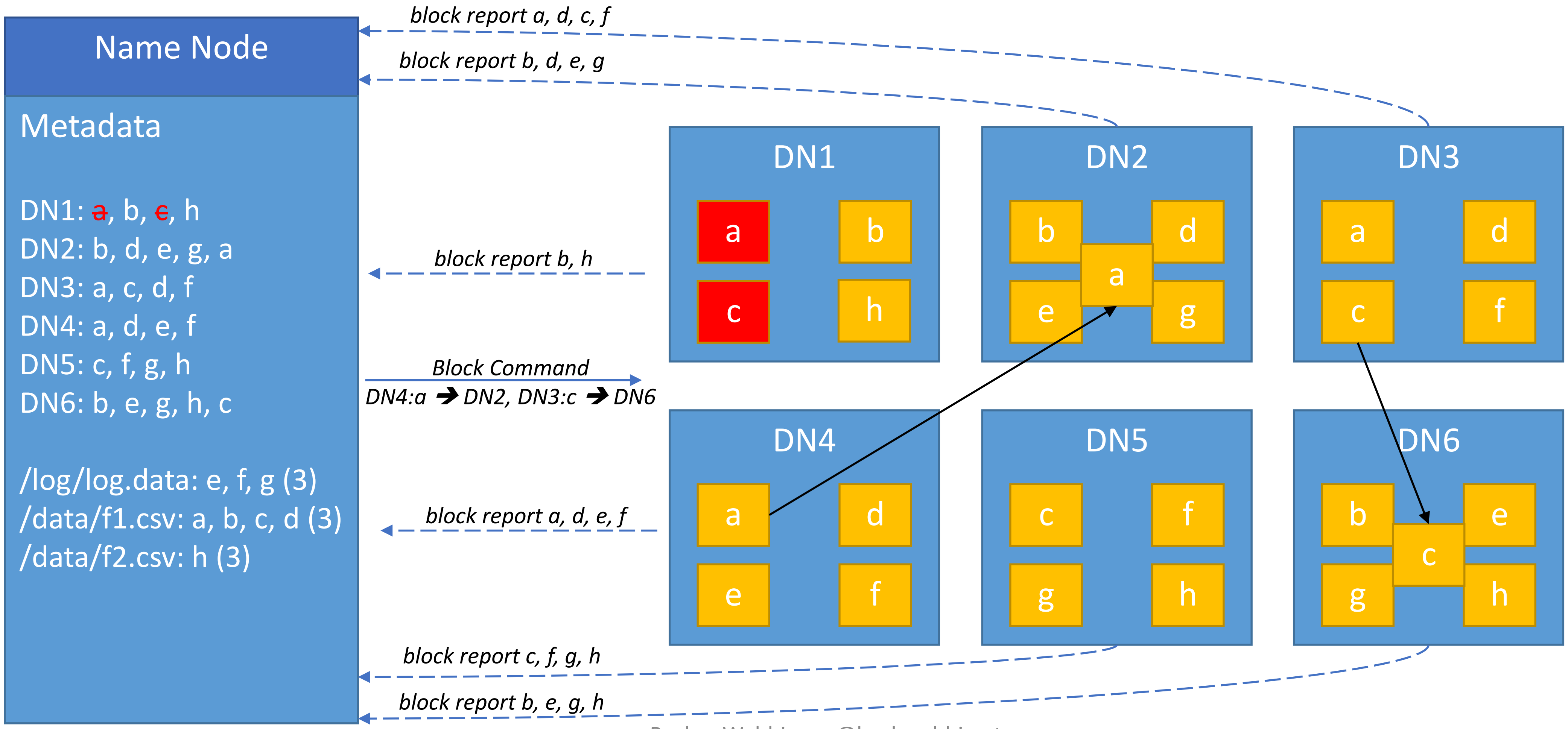
HDFS Robustness: Data Node Disk Failure



HDFS Robustness: Data Node Disk Failure



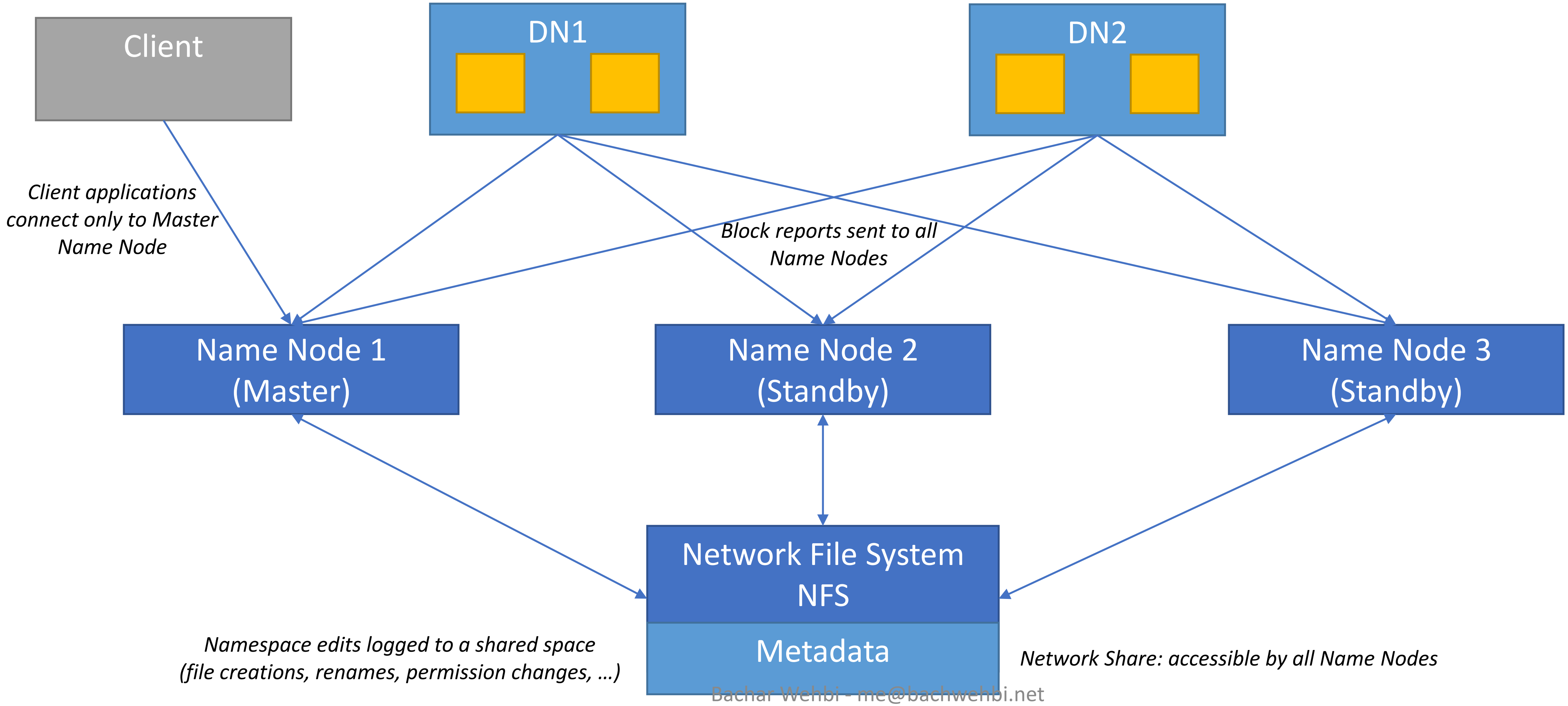
HDFS Robustness: Data Node Disk Failure



HDFS Robustness: High Availability

- Before talking about Name Node Failure, let’s talk about High Availability (HA) in HDFS
- Prior ro Hadoop V2, the Name Node was a Single Point Of Failure (SPOT)
  - If the Name Node was down (failure) or for maintenance, the HDFS cluster would be unavailable (until the Name Node was restarted or recovered)
- Hadoop V2 introduced High Availability to HDFS with the support of 2 redundant Name Nodes in active/passive mode.
- Hadoop V3 allows to have more than 2 redundant Name Nodes for better availability
- But How high availability in HDFS works?

HDFS Robustness: High Availability



HDFS Robustness: Name Node Failover

- Manual process using HDFS's utility tool: *hdfs haadmin*
  - *-transitionToActive <serviceId>*
  - *-transitionToStandby <serviceId>*
  - *-failover <activeId> <standbyId>*
- Automatic Failover: requires ZooKeeper
  - All Name Nodes inform their status to ZooKeeper
  - When Zookeeper detects Master Name Node unavailability
    - Initiates an election process to select the new master Name Node among the standby Name Nodes

Data Storage Formats

How you store data defines how (fast & efficiently) you can read it

Data Storage Formats

- The tools in the Hadoop (Big) Data ecosystem use different formats to store data
  - The selection of the format depends on the use case. There is nothing as the best data format for big data!
- The major formats are
  - Text files including tabular and CSV files
  - JSON files
  - Apache Avro data format
  - Apache ORC data format
  - Apache Parquet data format
- The field is still in active research and development. Apache Arrow is another project to unify the layout of data in Memory. We will not discuss Arrow as it is a project topic
- HDFS does not depend on a particular data format. HDFS considers files simply as a sequence of bytes.

Data Storage Formats: Text Files

- Text files are the most basic and simple format type.
  - It consists of storing everything in its String (text) representation
  - It can therefore be read and written by all programming languages. Just read text from a file
  - CSV and tabular formats are the most commonly used text file formats. They have the advantage of being compatible with most of the tools in the Hadoop ecosystem.
  - Human friendly format as it can be read and dubugged very naturally
- Text files are however very inefficient
  - Inefficient for binary and numeric (non text) types:
    - *1234567890* is a *4 Bytes* integer by consumes *10 Bytes* in string format
    - Inapropritate for binary object data (as images, sound, etc.): convert to Base64
  - Conversion from/to native types is CPU intensive
  - Does not support schema definition as part of the format
    - Off channel schema definition which adds complexity when working with Text files
  - Poor performance in general

Data Storage Formats: JSON Files

- JavaScript Object Notation is a serialization format for the Web
- It is basically a text based format with wide support almost everywhere
- Complete integration between data and Schema
- Row based format
- Dos not include any built in compression
  - Compression should be applied on top
- Human friendly and very easy to debug
- Verbose and take lot of disk space
  - Keys are repeated in every record (every row)
- As with text files, JSON is very inefficient at scale

Data Storage Formats: Apache Avro

- Row based file format
- Cross-language file format for Hadoop
- Binary data storage with optimized encoding
  - Data stored in native types ➔ no conversion required on read or write
- Schema metadata embedded within the file
  - Schema segregated from data (complete seperation)
  - JSON based ➔ language independent
  - Schema evolution as primary goal: good for future changes
- Widely supported in the Hadoop ecosystem and outside
  - Used in streaming systems like Apache Kafka
- Comes with a utility command line tool to work with files
  - Convenient for debugging as the binary format is not human friendly

Data Storage: Columnar vs. Row Formats

ID	Name	City	Country
1	Alice	Paris	France
2	Bob	Lille	France
3	Mike	Berlin	Germany
4	Hiba	Paris	France

- Row Based File Format
- Organizes data into rows
- Traditional way for storing data
- Example: Apache Avro

ID	Name	City	Country
1	Alice	Paris	France
2	Bob	Lille	France
3	Mike	Berlin	Germany
4	Hiba	Paris	France

- Column Based File Format
- Organizes data into columns
- Reads only selected columns – efficient in reading data
  - Example: SELECT ID, Name FROM table
- Example: Apache Parquet, Apache ORC

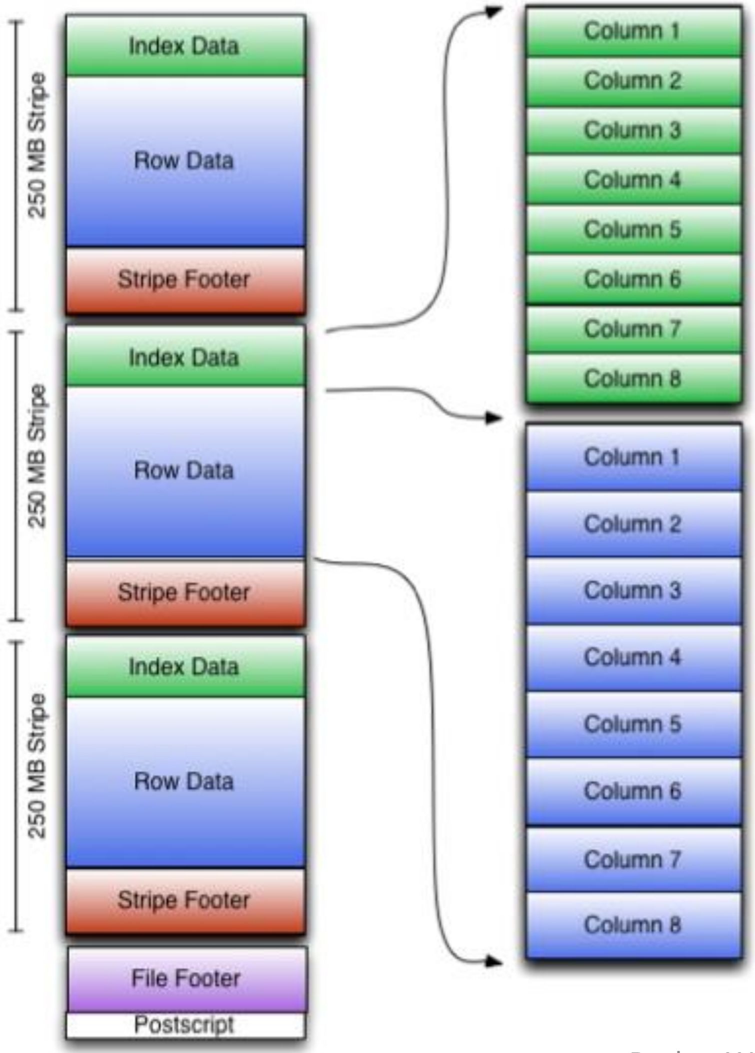
Data Storage Formats: Apache ORC

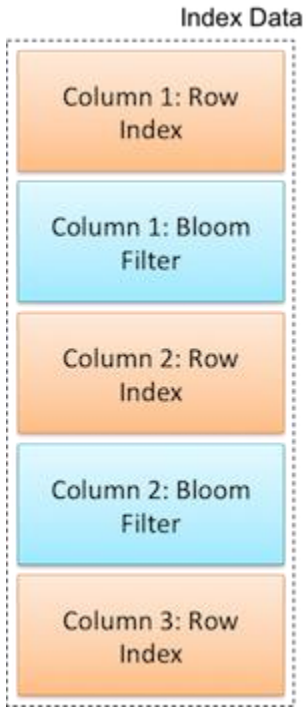
- ORC is a column based file format developpped originally by Hortonworks to optimize storage for Hive data
  - Facebook has 300PB of ORC data (improved compression from 5x to 8x, and saved 1400 servers)
  - The main focus was on enabling high speed processing and reducing file sizes.
  - Lightweight indexes stored in each file
    - min, max, avg for numeric columns, enumerations for string based columns
  - Schema seperated from data and stored into footer
  - Rich type model
  - Integrates compression, indexes and statistics
- Wide support in the Hadoop ecosystem
- Comes with a utility command line tool to work with files
  - Convenient for debugging as the binary format is not human friendly

Data Storage Formats: Apache ORC

- ORC data layout: efficient read and write & reduce storage needs
- Stripes: Divided into row groups
  - The default stripe size is 250 MB. Large stripe sizes enable large, efficient reads from HDFS.
  - Each column is stored in several streams
    - Integer columns have 2 streams: PRESENT (bitmask) to indicate if the value is non null and DATA stream that include the non Null values
    - For binary data ORC uses 3 streams: PRESENT, DATA, and LENGTH
  - Index Data: includes min and max for every column in a row group and an optional bloom index
  - Stripe Footer: includes the encoding (data type) of each column and the location of the streams
- File Footer: it contains
  - List of stripes in the file, the number of rows per stripe, and type information (data Schema).
  - Column-level statistics: cunt, min, max, and sum and has Null.
- Postscript: At the end of the file (after the file footer). It contains compression parameters and the size of the compressed footer.

Data Storage Formats: Apache ORC





Data Storage Formats: Apache Parquet

- Parquet is a column based format designed for efficient storage and retrieval of data
  - Developped initially by Cloudera and Twitter
  - Embeds Schema metadata in the file
  - Inspired by Google Dremel Paper to enable efficient storage of data and improve read efficiency
  - Optimized for Batch write as it identifies repeated patterns to reduce storage space
- Wide support in the Hadoop ecosystem
- Comes with a utility command line tool to work with files
  - Convenient for debugging as the binary format is not human friendly

Data Storage Formats: Apache Parquet

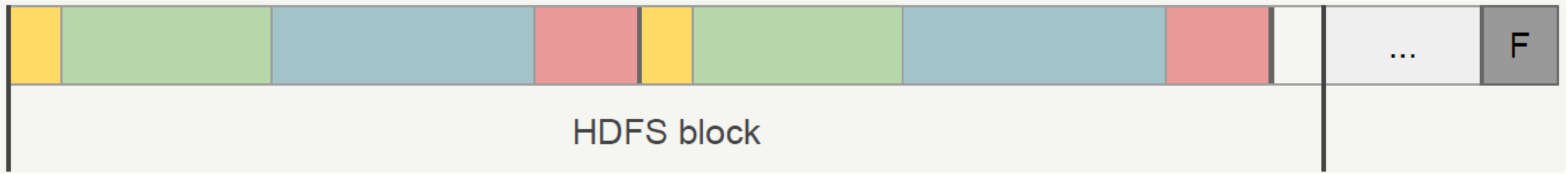
Parquet Data Layout: advanced techniques to read less data

- Row Groups
  - Data needed for a group of rows to be reassembled
  - Smallest task or input split size
  - Made of Column Chunks
- Column Chunks
  - Contiguous data for a single column
  - Made of data pages and an optional dictionary page
- Data Pages
  - Encoded and compressed runs of values
- Dictionary Page
  - Includes the list of unique values in a data page of a column
  - Useful to skip entire row groups when the search criteria (where statement) is missing in the dictionary
    - Example: SELECT * FROM table WHERE city=Paris

Data Storage Formats: Apache Parquet

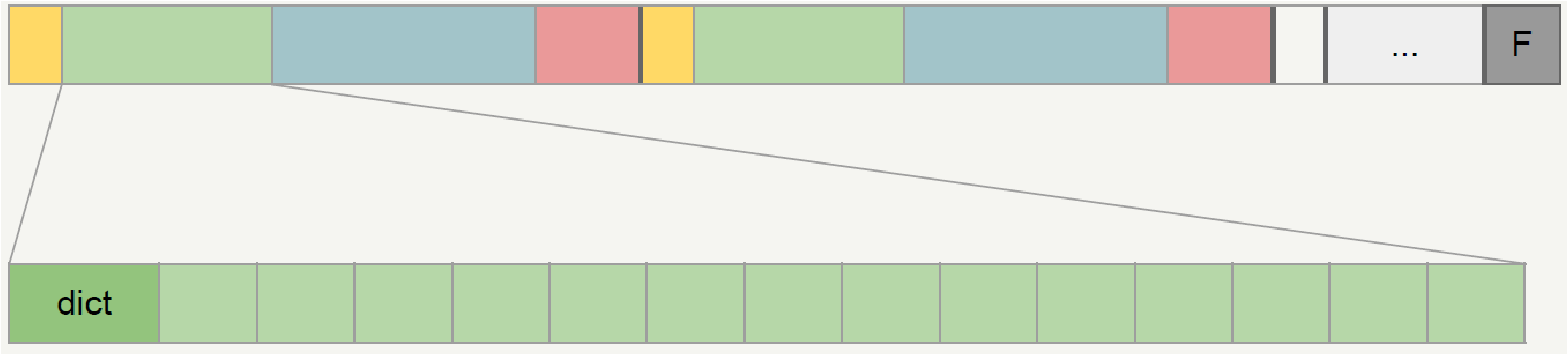
Row Group

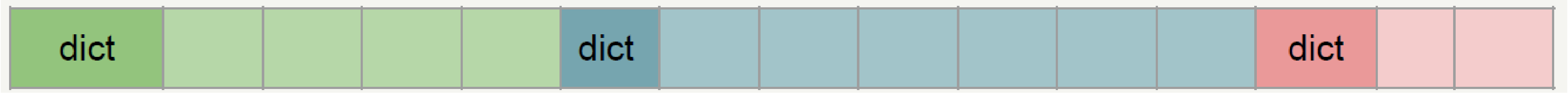
A	B	C	D
a1	b1	c1	d1
...	...	...	...
aN	bN	cN	dN
...	...	...	...



Data Storage Formats: Apache Parquet

Column Chunk & Data Pages





Data Storage Formats: Summary

	Text Files	JSON Files	Apache Avro	Apache ORC	Apache Parquet
Human friendly	X	X			
Tools compatibility	X	X	X	X	X
Performance			X	X	X
Binary format			X	X	X
Embedded Schema		X	X	X	X
Column Based				X	X

Data Compression

- Reduces disk space required for data storage
- Compression is a tradeoff between disk space/bandwidth and CPU
    - High compression rate codecs take more CPU time but save more disk space and require less I/O
    - Lower compression rate codecs are much faster but vase less disk space and require more I/O compared to agressive algorithms
- Compression can significantly improve performance of Big Data jobs:
    - Allows to handle more I/O operations
    - Improves the performance when reading/writing over the network (Cloud storage)
    - Improves the performance when reading/writing to a magnetic Disk
- The selection of the compression codec is use case specific
- Hadoop Natively supports multiple compression Codecs
- Compression Codecs include: bzip, gzip, lzo, lz4, snappy, brotli