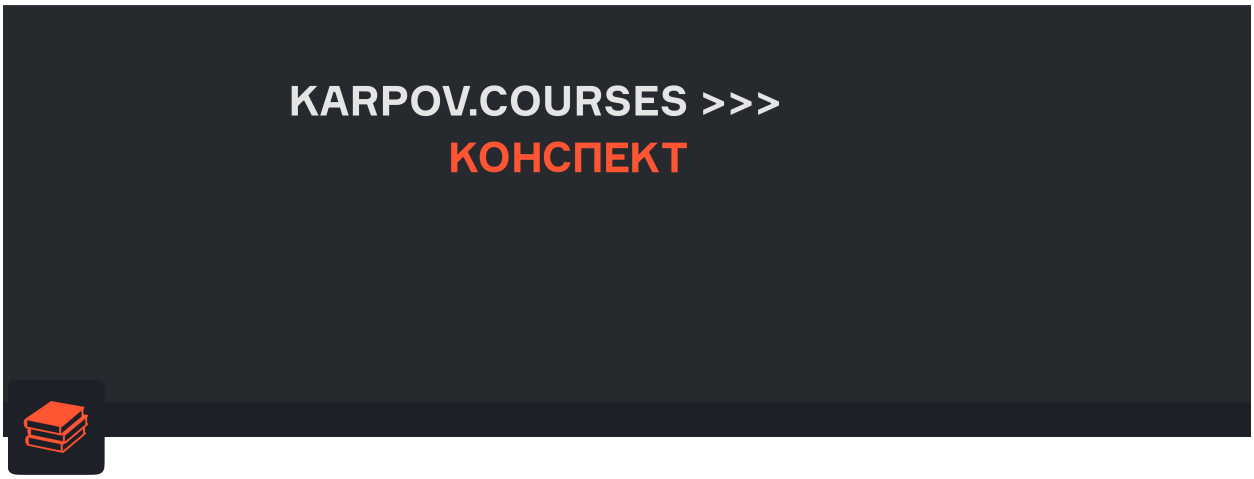


> Конспект > 1 урок > Основы HADOOP.HDFS

> История Hadoop

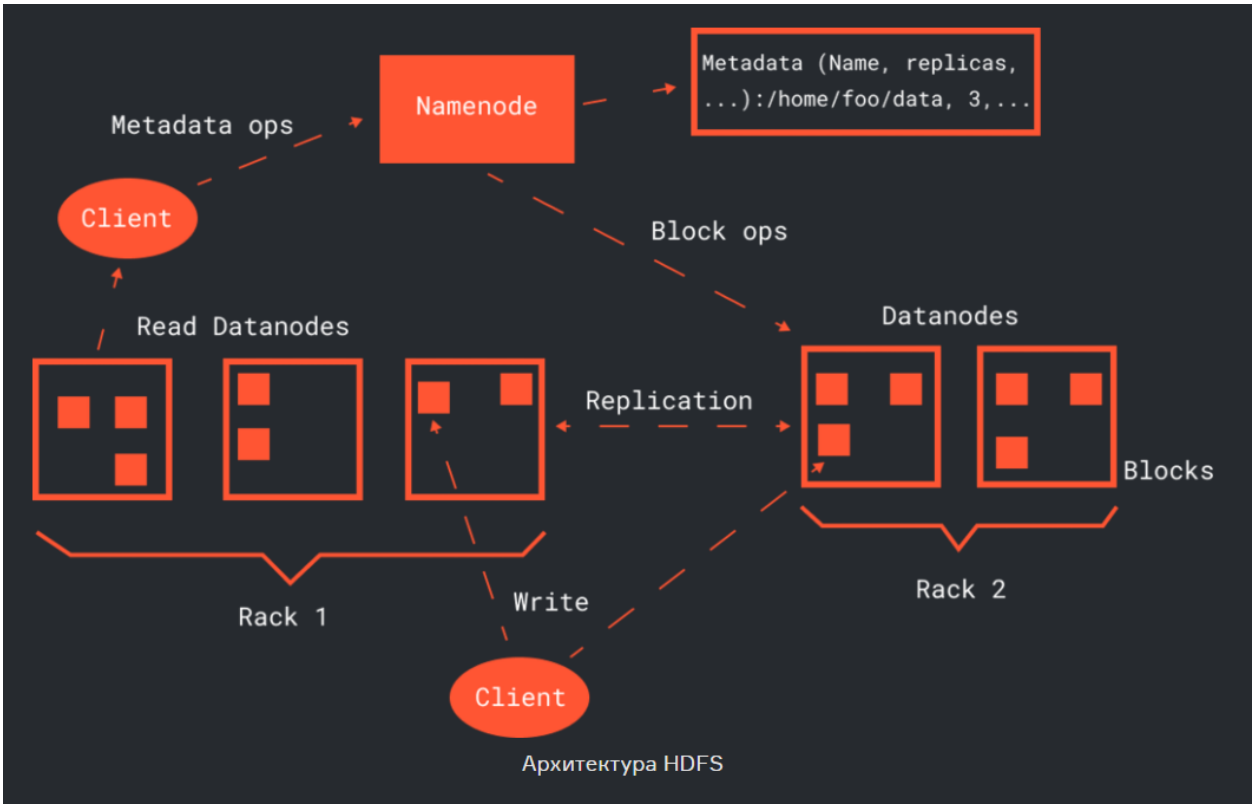
- 2002: запуск проекта Nutch (Yahoo)
- 2003: публикация с описанием GFC
- 2004: создание NDFS (Nutch Distributed File System)
- 2004: публикация Google и MapReduce
- 2005: реализация MR в Nutch
- 2006: выделение подпроекта Hadoop

- 2008: выход Hadoop в лидеры ASF (Apach Software Foundation)

> Архитектура HDFS

> HDFS (Hadoop Distributed File System)

Это распределенная файловая система для хранения файлов больших размеров с возможностью потокового доступа к информации, поблочно распределённой по узлам вычислительного кластера, который может состоять из произвольного аппаратного обеспечения. Как и любая файловая система, HDFS это иерархия каталогов с вложенными в них подкаталогами и файлами

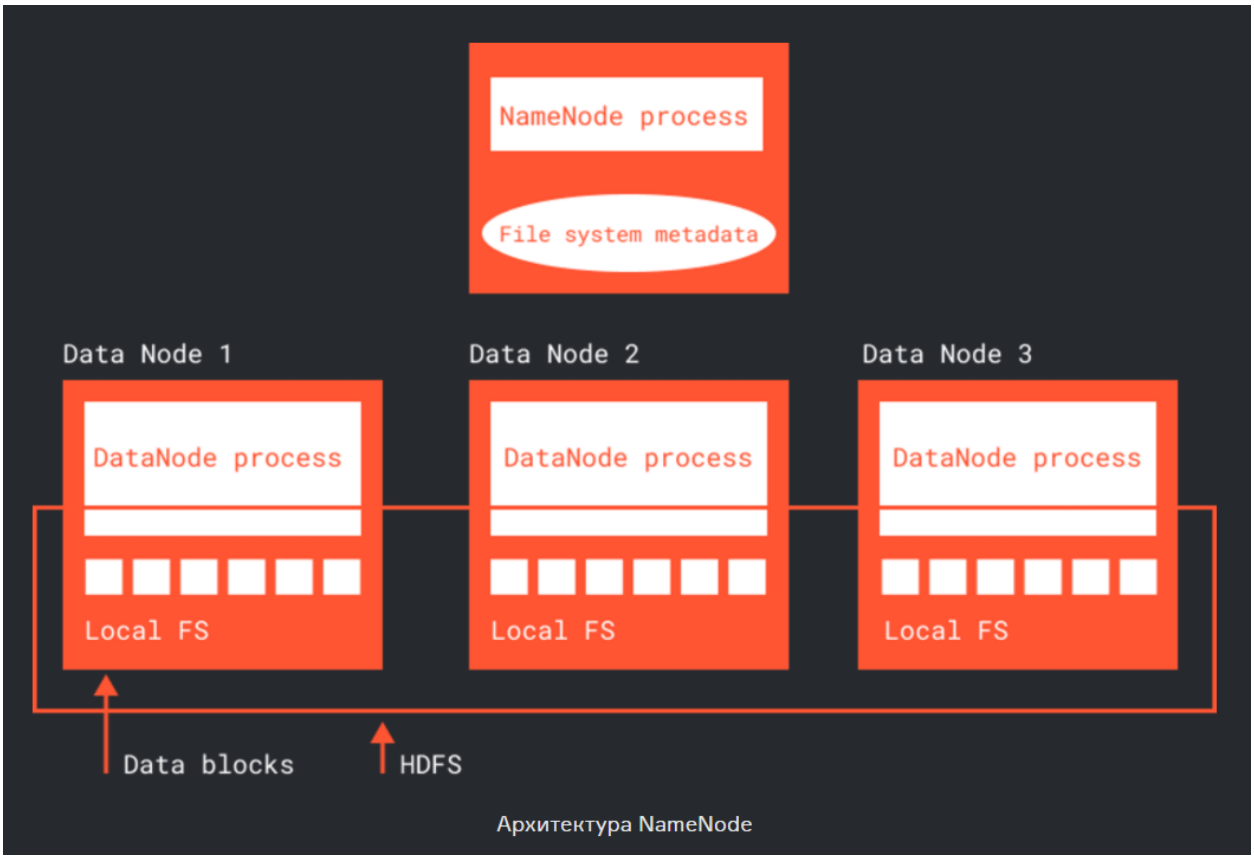


Архитектура HDFS

Кластер HDFS включает следующие компоненты:

Управляющий узел, узел имен или сервер имен (NameNode) – отдельный, единственный в кластере, сервер с программным кодом для управления пространством имен файловой системы, хранящий дерево файлов, а также мета-данные файлов и каталогов.

NameNode – обязательный компонент кластера HDFS, который отвечает за открытие и закрытие файлов, создание и удаление каталогов, управление доступом со стороны внешних клиентов и соответствие между файлами и блоками, дублированными (реплицированными) на узлах данных. Сервер имён раскрывает для всех желающих расположение блоков данных на машинах кластера.



Архитектура NameNode

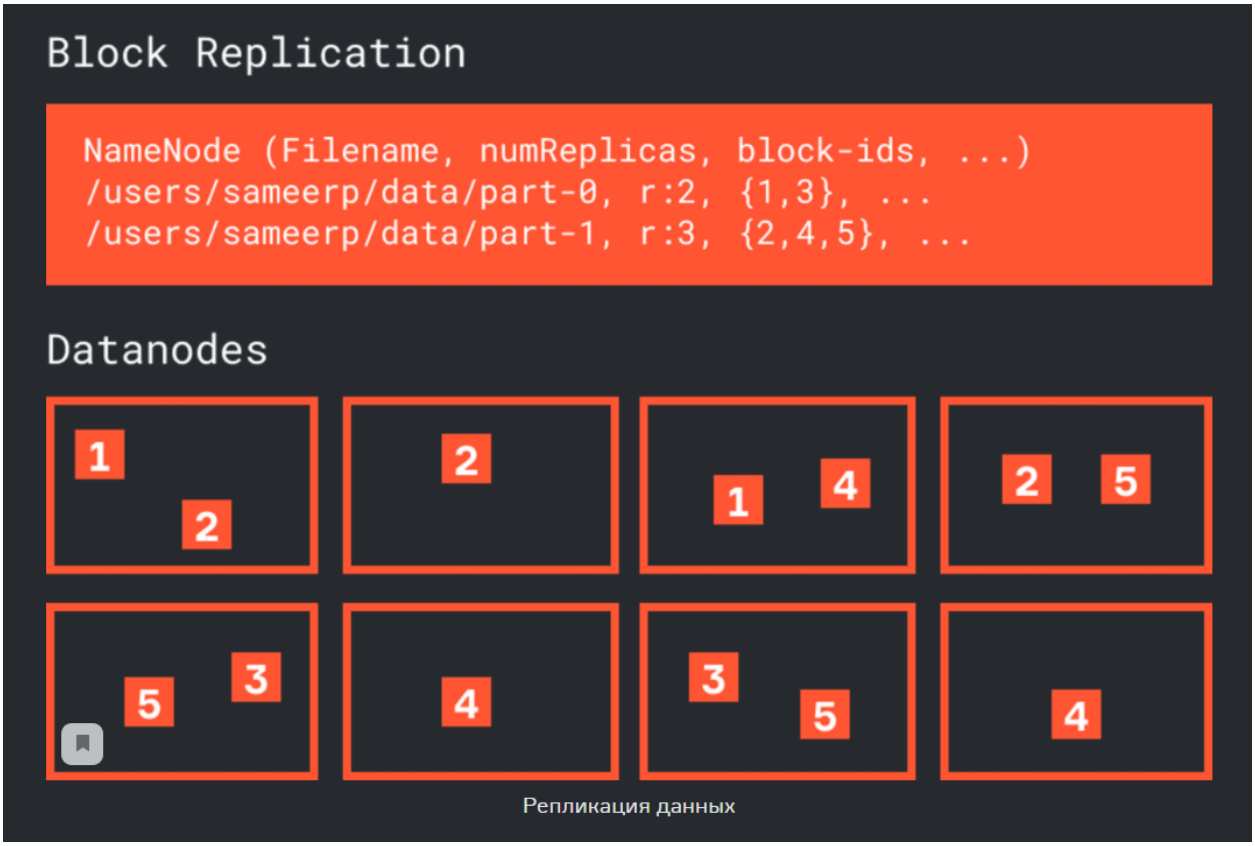
Secondary NameNode — вторичный узел имен, отдельный сервер, единственный в кластере, который копирует образ HDFS и лог транзакций операций с файловыми блоками во временную папку, применяет изменения, накопленные в логе транзакций к образу HDFS, а также записывает его на узел NameNode и очищает лог транзакций.
Secondary NameNode необходим для быстрого ручного восстановления NameNode в случае его выхода из строя.

Узел или сервер данных (DataNode, Node) – один их множества серверов кластера с программным кодом, отвечающим за файловые операции и работу с блоками данных. DataNode – обязательный компонент кластера HDFS, который отвечает за запись и чтение данных, выполнение команд от узла NameNode по созданию, удалению и репликации блоков, а также периодическую отправку сообщения о состоянии (heartbeats) и обработку запросов на чтение и запись, поступающих от клиентов файловой системы HDFS. Стоит отметить, что данные проходят с остальных узлов кластера к клиенту мимо узла NameNode.

Клиент (client) – пользователь или приложение, взаимодействующий через специальный интерфейс (API – Application Programming Interface) с распределенной файловой системой. При наличии достаточных прав, клиенту разрешены следующие операции с файлами и каталогами: создание, удаление, чтение, запись, переименование и перемещение. Создавая файл, клиент может явно указать размер блока файла (по умолчанию 64 Мб) и количество создаваемых реплик (replication factor по умолчанию равен 3-ем).

> Репликация

Создавая файл, клиент может явно указать размер блока файла (по умолчанию 64 Мб) и количество создаваемых реплик (replication factor). Репликация — это избыточность хранения информации, которая позволяет нам потерять одну или несколько о копий. По умолчанию все HDFS-блоки реплицируются 3 раза (replication factor - 3), если клиентом (пользователем или приложением) не задано другое значение коэффициента репликации. С целью повышения надежности для хранения 2-ой и 3-ей реплики выбираются те узлы данных, которые расположены в разных серверных стойках. Последующие реплики могут храниться на любых серверах.



Репликация данных

Репликация данных в HDFS выполняется в следующих случаях:

- создание нового файла (операция записи);
- обнаружение сервером имен отказа одного из узлов данных – если **NameNode** не получает от heartbeat-сообщений, он запускает механизм репликации;

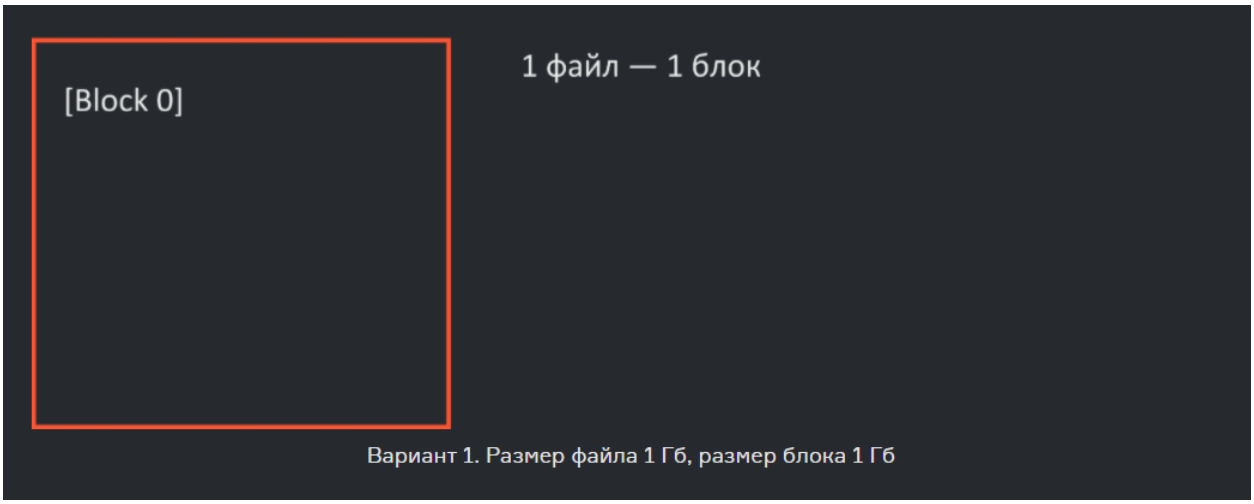
  DataNode
- повреждение существующих реплик;
- увеличение количества реплик, присущих каждому блоку.

Репликация данных выполняется следующим образом:

- NameNode выбирает новые узлы данных для размещения реплик;
- сервер имен выполняет балансировку размещения данных по узлам и составляет список узлов для репликации;
- 1-я реплика размещается на первом узле из списка;
- 2-я реплика копируется на другой узел в этой же серверной стойке;
- 3-я реплика записывается на произвольный узел в другой серверной стойке;
- остальные реплики размещаются произвольным способом.

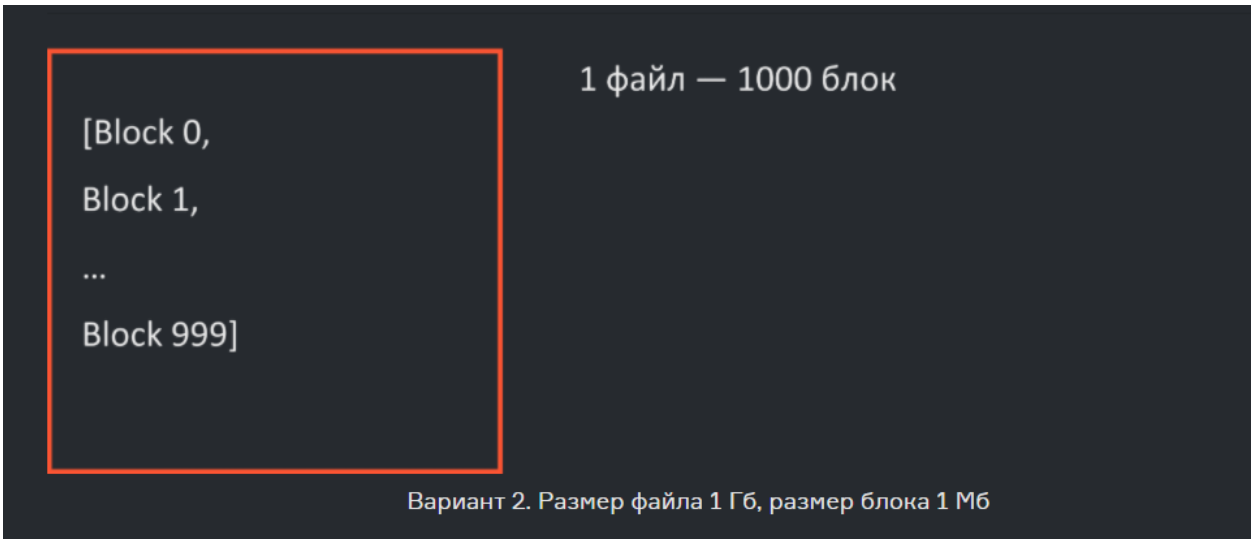
> Проблема мелких файлов

Наличие большого кол-ва файлов/блоков, приводит к тому, что кол-во объектов, которые хранятся в памяти увеличиваются в разы. Рассмотрим несколько вариантов хранения файлов.



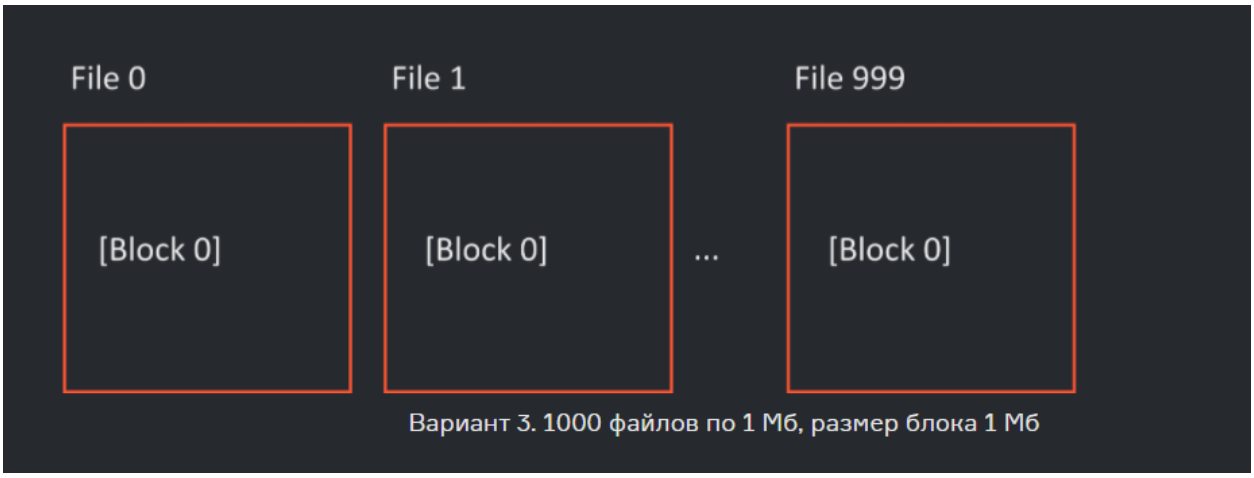
Для описания метаинформации о файле в первом случае нам нужно иметь 3 объекта:

1. Файл - 1 шт.
2. Блок - 1 шт.
3. Массив, который хранит информацию о блоках этого файла - 1 шт.



Если мы уменьшим размер блока, во втором случае нам нужно уже иметь 1002 объекта:

1. Файл - 1 шт.
2. Блок - 1000 шт.
3. Массив, который хранит информацию о блоках этого файла - 1 шт.

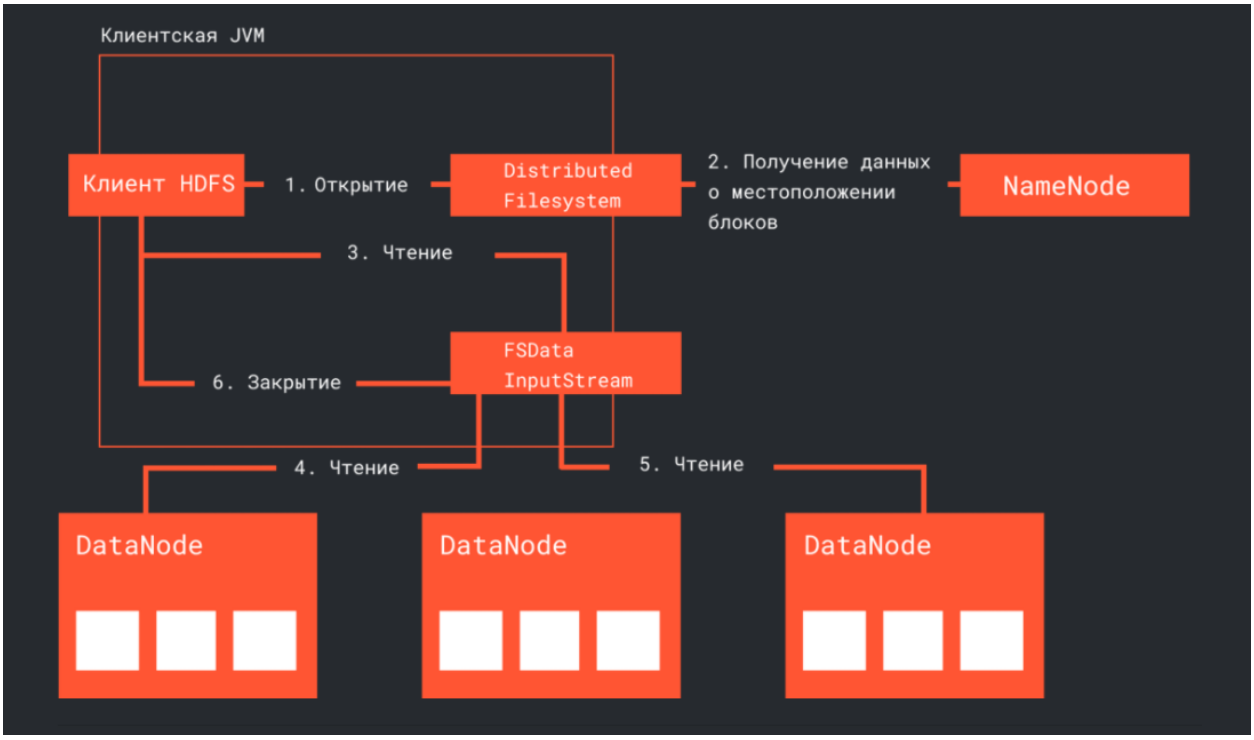


Теперь представим, что размер блока остался с прошлого примера, а файл мы разбили по частям. В таком случае нам нужно 3000 объектов:

1. Файл - 1000 шт.
2. Блок - 1000 шт.
3. Массив, который хранит информацию о блоках этого файла - 1000 шт.

> Процесс чтения и записи

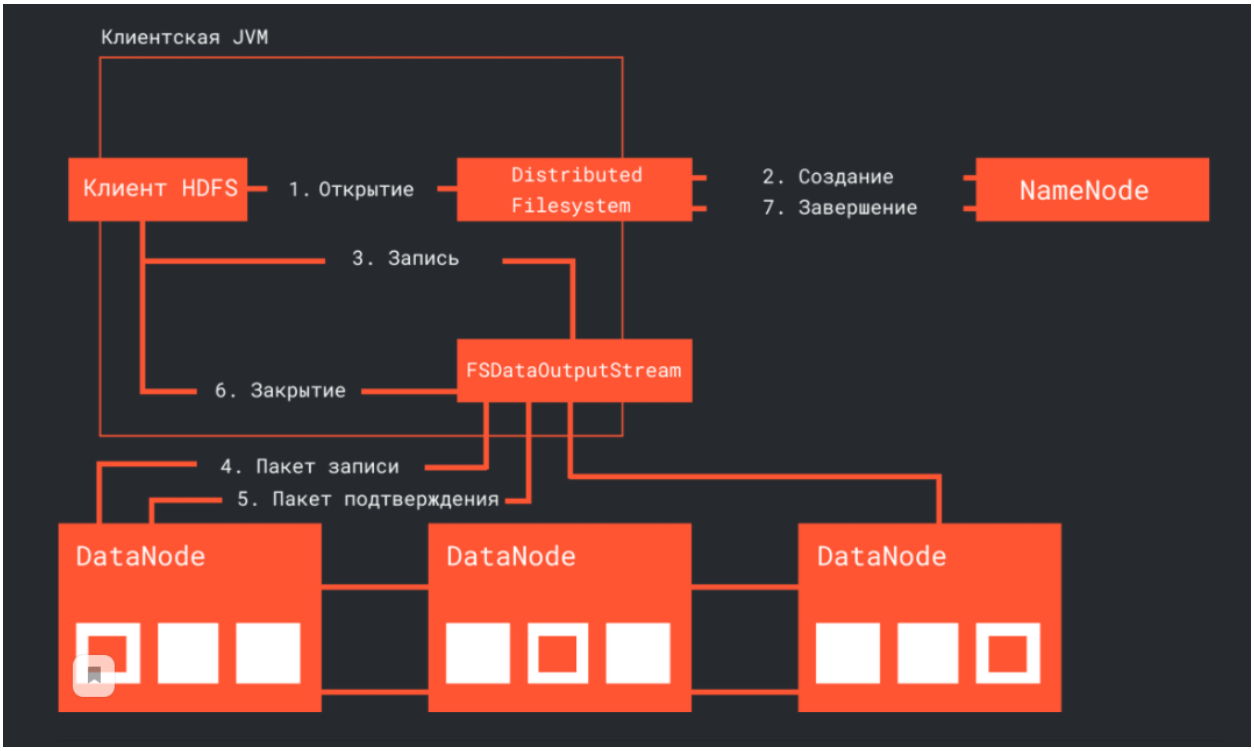
> Процесс чтения



Клиент HDFS через специальный интерфейс Distributed Filesystem запрашивает у NameNode какой-либо файл. NameNode проверяет существует ли этот файл, его права доступа и т.п. Далее если все проверки были успешны - выдает клиенту информацию о файле (где он находится, из каких блоков состоит). Потом клиент самостоятельно используя интерфейс FSData InputStream идет на каждую конкретную DataNode из выданного списка, которого ему вадала NameNode и выкачивает нужную ему информацию.

Важно отметить, что основной трафик идет между клиентом и конкретными DataNode, а не через NameNode.

> Процесс записи



1. Клиент инициирует запрос для записи файла к NameNode.
2. NameNode проверяет, существует ли файл и есть ли у клиента права для записи, и в случае успеха создает запись для файла.
3. Клиент делит файл на несколько пакетов в соответствии с размером блока и управляет ими в форме «очереди данных» и получает количество репликаций блока.
4. Запишите пакеты для всех репликаций в форме конвейера, сначала запишите в первый DataNode. После того, как DataNode сохранит пакеты, передайте их следующему DataNode в конвейере до последнего DataNode (конвейерная форма).
5. После успешного сохранения последнего DataNode он вернет подтверждение и передаст его клиенту в конвейере.
6. Если DataNode выходит из строя во время передачи, текущий конвейер будет остановлен, отказавший DataNode будет удален, а оставшиеся DataNode будут по-прежнему передаваться в форме конвейера.
7. После того, как клиент заканчивает запись данных, он вызывает метод close (), чтобы закрыть поток данных.

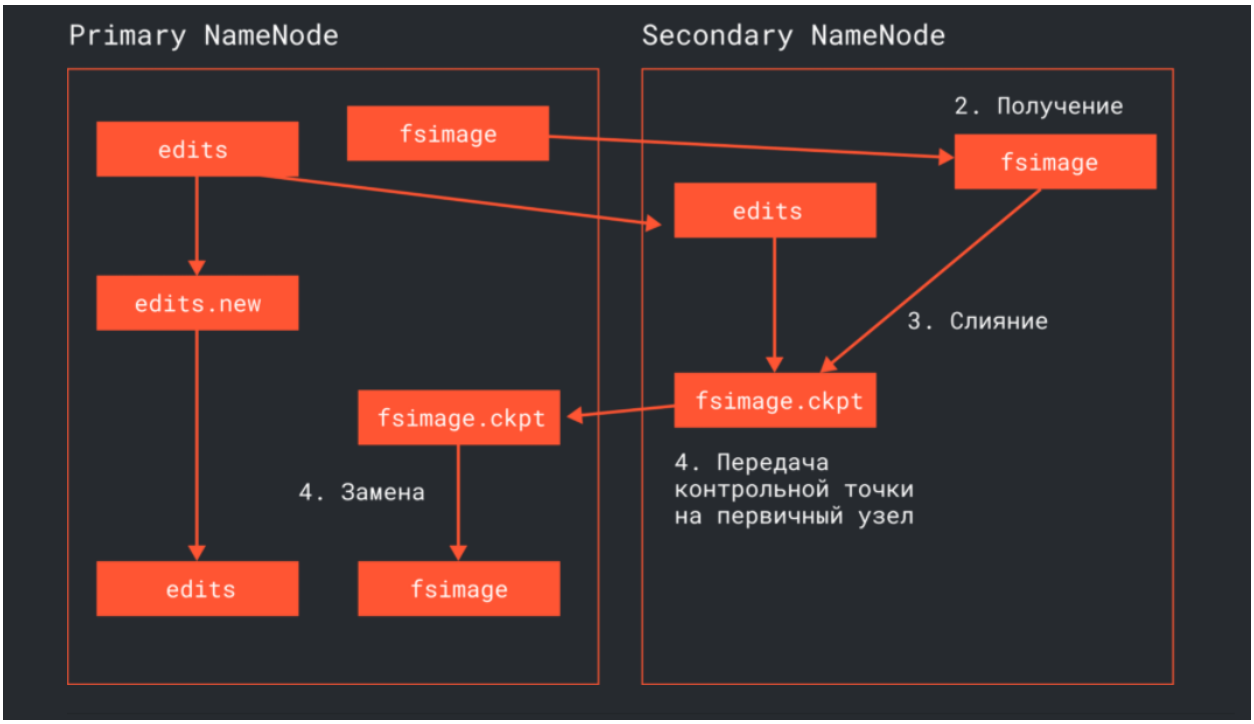
> Хранение данных NameNode

Основные группы файлов, которые хранятся в NameNode (можно найти в ${dfs.name.dir}/):

- VERSION: информация о версии HDFS

- edits: журнал изменений
- fsimage: контрольная точка метаданных
- fstime: время создания контрольной точки

> Secondary NameNode



Secondary NameNode не является репликой NameNode. Состояние файловой системы хранится непосредственно в файле `fsimage` и в лог файле `edits`, содержащим последние изменения файловой системы (похоже на лог транзакций в мире РСУБД). Работа Secondary NameNode заключается в периодическом мерже `fsimage` и `edits` — Secondary NameNode поддерживает размер `edits` в разумных пределах.

Secondary NameNode необходима для быстрого ручного восстановления NameNode в случае выхода NameNode из строя.

Посмотрим как физически выглядят данные в NameNode. Для этого нам нужно через командную строку выполнить следующую команду:

```
hdfs fsck path_to_file -files -blocks -locations
```

В качестве вывода мы получим что-то похожее на:

```
hdfs fsck /user/hive/warehouse/big_cdr_parquet/000000_0 -files -blocks -locations
Connecting to namenode via http://hdp-7:50070
FSCK started by root (auth:SIMPLE) from /192.168.91.141 for path /user/hive/warehouse/big_cdr_parquet/000000_0
at Mon May 18 14:00:22 MSK 2015
/user/hive/warehouse/big_cdr_parquet/000000_0 1133129924 bytes, 9 block(s):  OK
0. BP-1972162810-192.168.91.133-1428693610895:blk_1073747244_6432 len=134217728 repl=3
[192.168.91.141:50010, 192.168.91.139:50010, 192.168.91.133:50010]
1. BP-1972162810-192.168.91.133-1428693610895:blk_1073747245_6433 len=134217728 repl=3
[192.168.91.136:50010, 192.168.91.139:50010, 192.168.91.141:50010]
2. BP-1972162810-192.168.91.133-1428693610895:blk_1073747246_6434 len=134217728 repl=3
[192.168.91.142:50010, 192.168.91.136:50010, 192.168.91.141:50010]
 …

8. BP-1972162810-192.168.91.133-1428693610895:blk_1073747252_6440 len=59388100 repl=3
[192.168.91.141:50010, 192.168.91.137:50010, 192.168.91.135:50010]
```

В нашем примере строка `/user/hive/warehouse/big_cdr_parquet/000000_0 1133129924 bytes, 9 block(s): OK` говорит нам о том, что наш файл имеет размер 1133129924 байт, состоит из 9 блоков и его состояние OK.

Далее идет информация о каждом блоке. Рассотрим на примере 1го блока какую информацию мы о нем получили:

- название блока (blk_1073747244_6432),
- размер (134217728 байт),
- replication factor (3),
- IP-адреса на которых находятся блоки ([192.168.91.141:50010, 192.168.91.139:50010, 192.168.91.133:50010]).

Отдельно стоит обратить внимание на размер последнего блока. Он отличается от всех остальных. Если размер файла меньше размера блока, например файл 10 Мб, а размер блока 128 Мб. В таком случае блок будет размером в 10 Мб, а не 128 Мб. Никакой избыточности с размером блока на HDFS нет.

> Основные команды CLI

Aa Команда	Пример
appendToFile	hdfs dfs -appendToFile localfile /user/hadoop/hadoopfile
cat	hdfs dfs -cat hdfs://nn1.example.com/file1
copyFromLocal	hdfs dfs -copyFromLocal localfile /user/hadoop/data/
copyToLocal	hdfs dfs -copyToLocal localfile /tmp/data/ localfile
cp	hdfs dfs -cp [-f] [-p | -p[topax]] URI [URI ...] <dest>

Команда	Пример
du	hdfs dfs -du -s /tmp/test.data
expunge	hdfs dfs -expunge
get	hdfs dfs -get /user/hadoop/file localfile
getmerge	hdfs dfs -getmerge <src> <localdst> [addnl]
ls	hdfs dfs -ls /user/hadoop/file1
mkdir	hdfs dfs -mkdir /user/hadoop/dir1 /user/hadoop/dir2
mv	hdfs dfs -mv /user/hadoop/file1 /user/hadoop/file2
put	hdfs dfs -put localfile /user/hadoop/hadoopfile
rm	hdfs dfs -rm [-f] [-r|-R] [-skipTrash] URI [URI ...]
tail	hdfs dfs -tail pathname
setrep	hdfs dfs -setrep [-R] [-w] <numReplicas> <path>

> HDFS Erasure Coding

> Кодирования со стиранием (Erasure Coding, EC)

Кодирования со стиранием (Erasure Coding, EC) экономит место на жестком диске по сравнению с репликацией. Стандартная схема 3-кратной репликации в HDFS имеет 200% накладных расходов на пространство хранения и пропускную способность сети.

Erasure Coding снижает накладные расходы на хранение данных до 50% независимо от коэффициента репликации за счет чередования, которое разделяет логически последовательные данные файла на более мелкие блоки (бит, байт или блок), сохраняя их на разных дисках массива. Для каждой полосы исходных ячеек данных вычисляется и сохраняется определенное количество ячеек четности. Этот процесс называется кодированием.

Ошибка в любой чередующейся ячейке устраняется через обратную операцию декодирования на основе сохранившихся данных и ячеек четности. Например, файл с 3-кратной репликацией с 6 блоками HDFS без Erasure Coding будет занимать 6*3 = 18 блоков дискового пространства. А с поддержкой EC и 3-мя ячейками четности он занимает в 2 раза меньше дискового пространства – всего 9 блоков

Таким образом, кодирование со стиранием может снизить накладные расходы на хранилище HDFS примерно на 50% по сравнению с репликацией, сохраняя те же гарантии долговечности и позволяя хранить вдвое больше данных на том же объеме хранилища.

Что касается математики, которую реализует технология Erasure Coding, то здесь используется алгоритм Рида-Соломона (Reed-Solomon, RS), который преодолевает ограничение алгоритма XOR.

Операция XOR является ассоциативной и генерирует 1 бит четности из произвольного количества битов данных.

Например, если данные столбца Y потеряны, то используя данные столбца X и ячейку четности, можно декодировать потерянные данные, сгенерировав Y снова. Но, поскольку алгоритм XOR генерирует только 1 бит четности для любого количества входных ячеек данных, он может выдержать только 1 сбой. Этого недостаточно для обеспечения высокой надежности HDFS, когда необходимо обрабатывать несколько сбоев. Отказоустойчивость алгоритма XOR равна единице, а эффективность хранения составляет 2/3, то есть 67%.

Алгоритм Рида-Соломона использует операцию линейной алгебры для создания нескольких ячеек четности, чтобы выдерживать множественные отказы. Здесь выполняется умножение m ячеек данных на матрицу генератора, чтобы получить расширенное кодовое слово с m ячейками данных и n ячейками четности.

Если хранилище выходит из строя, то его можно восстановить, умножая обратную матрицу порождающей матрицы на расширенные кодовые слова до тех пор, пока доступны m из (m + n) ячеек. Отказоустойчивость RS-алгоритма приближается к n, то есть количество ячеек с проверкой четности и эффективность хранения составляет m/(m + n), где m – ячейка данных, а n – ячейка четности

> XOR

XOR (исключающее ИЛИ) – операция, которая принимает значение «истина» только если всего один из аргументов имеет значение «истина».

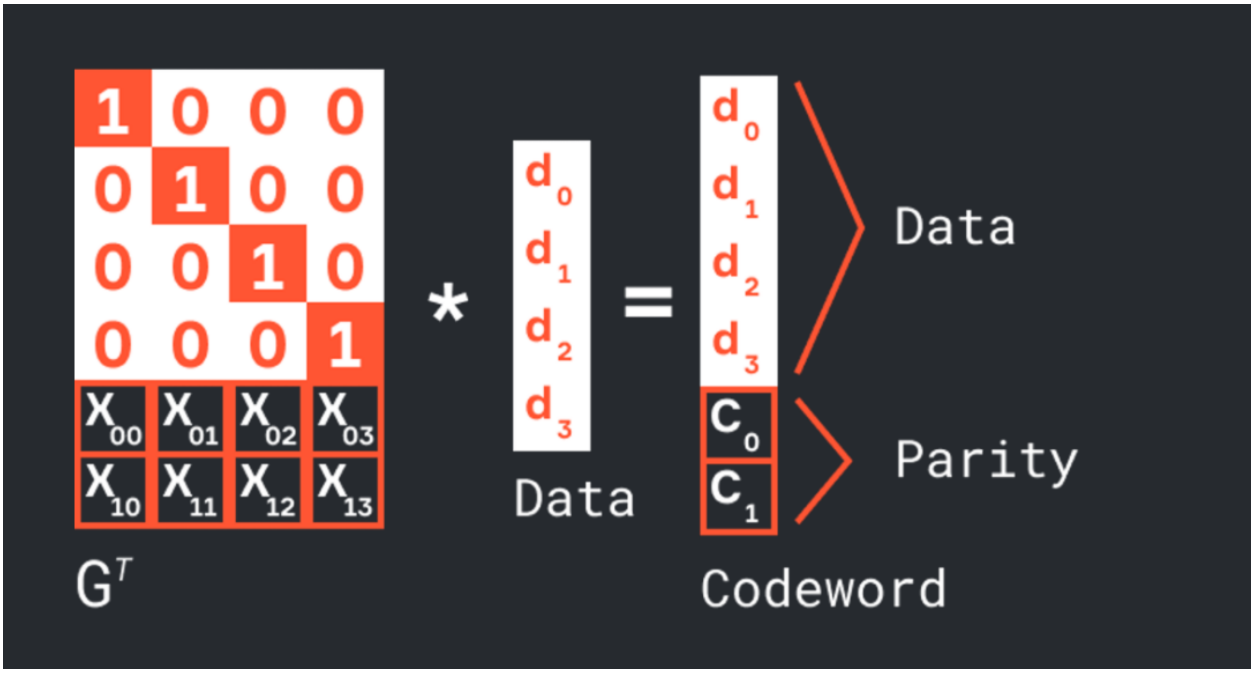
Input		AND	OR	XOR
A	B			
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Матрица истинности

C0	C1	C2	OUTPUT
1	0	1	0

Результат операции XOR

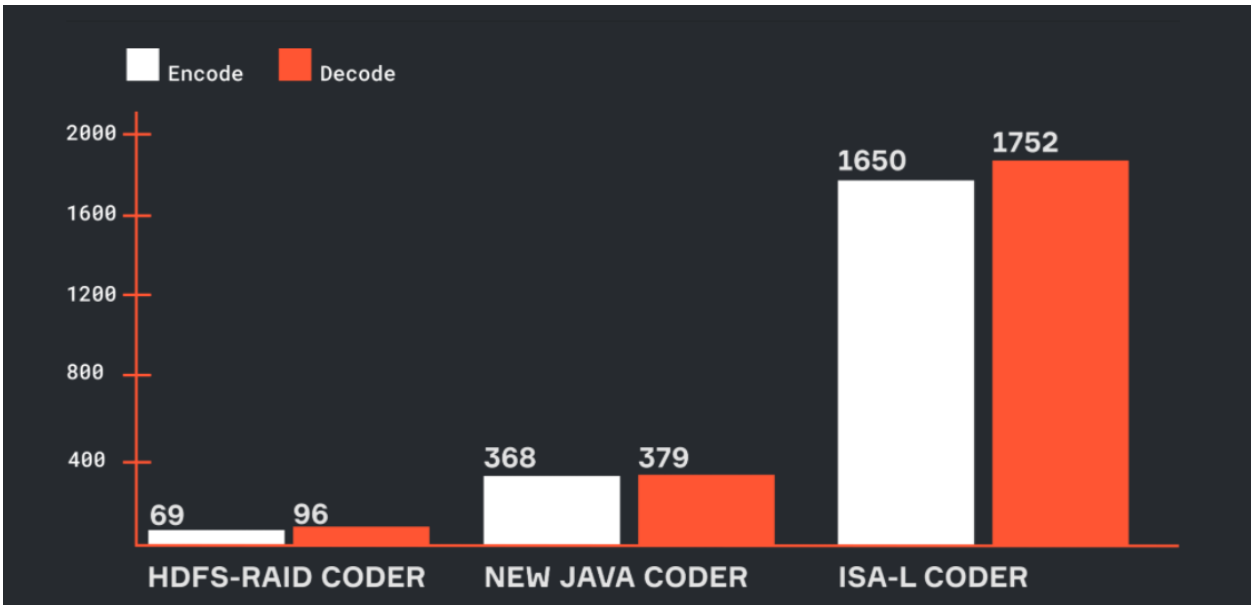
C0 ⊕ C2 ⊕ OUTPUT = 1 ⊕ 1 ⊕ 0 = 0



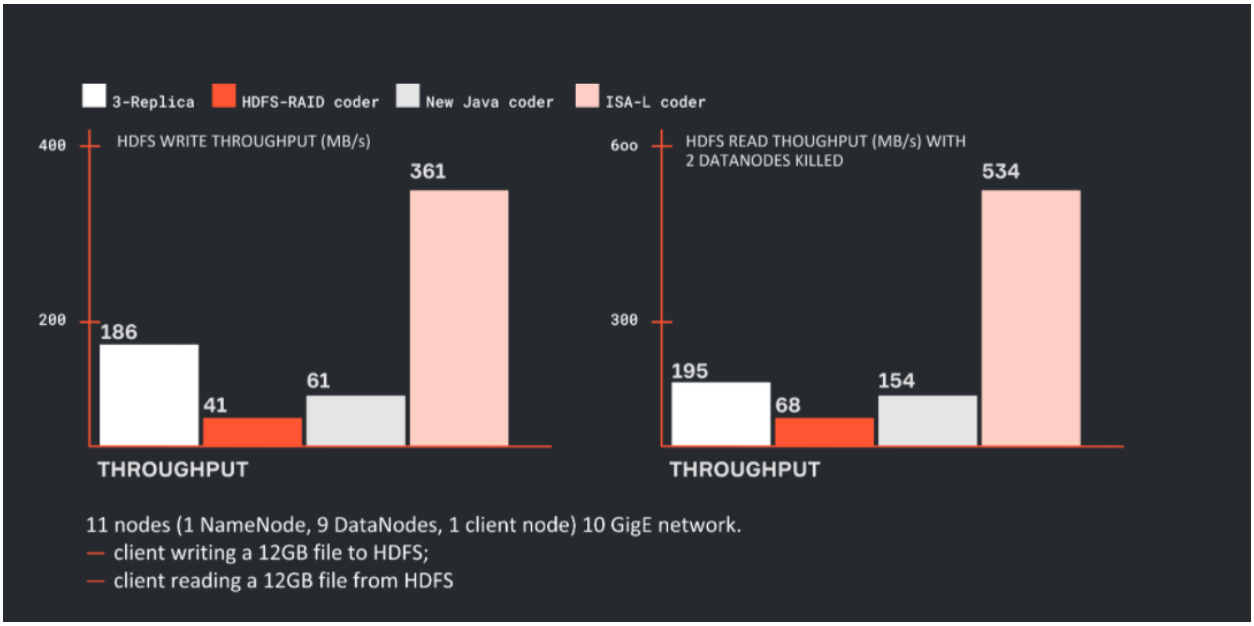
Сравнительная таблица методов хранения данных в HDFS

Mode	Data durability	Storage efficiency
Single replica	0	100%
3-way Replication	2	33%
XOR with 6 data cells	1	86%
RS(6,3)	3	67%
RS(10,4)	4	71%

На практике для хранения холодных данных (которые лежат давно и редко используются) применяется технология хранения через EC. Для горячих данных (которые часто используются) применяется репликация.



Бенчмарки разных механизмов Erasure Coding



Бенчмарки чтения и записи