Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «**Информатика и системы управления**»

КАФЕДРА «**Программное обеспечение ЭВМ и информационные технологии**»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

Метод блочного хранения данных с возможностью доказательства неправомерного доступа на основе хеш-сумм.

Студент группы **ИУ7-83Б** _____ **П. Г. Пересторонин**
(Подпись, дата) (И.О. Фамилия)

Руководитель ВКР _____ **А. С. Григорьев**
(Подпись, дата) (И.О. Фамилия)

Нормоконтролер _____ _____
(Подпись, дата) (И.О. Фамилия)

***2022** г.*

РЕФЕРАТ

Расчетно-пояснительная записка 73 с., 17 рис., 1 табл., 41 ист., 1 прил.

В работе представлена разработка метода блочного хранения данных с возможностью доказательства неправомерного доступа.

Проведена классификация видов защиты информации. Рассмотрены существующие системы блочного хранения данных с защитой от неправомерного доступа, их ограничения и области применимости, отмечены связи между свойствами методов и видами защиты информации. Рассмотрен метод блочного хранения в СУБД ClickHouse в движке MergeTree, проанализированы потенциальные угрозы. На основе имеющегося метода разработан метод, обеспечивающий возможность доказательства неправомерного доступа. Был проанализирован и протестирован разработанный метод с шифрованием данных и без него.

КЛЮЧЕВЫЕ СЛОВА

защита информации, Blockchain, шифрование, блочное хранение данных, ClickHouse, MergeTree.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Хеш-функция — функция, осуществляющая преобразование массива входных данных произвольной длины в выходную битовую строку установленной длины, выполняемое определенным алгоритмом [1].

Хеш-сумма (хеш-код) — строка бит, являющаяся выходным результатом хеш-функции [2].

Криптографическая стойкость — способность криптографического алгоритма противостоять криптоанализу; стойким считается алгоритм, успешная атака на который требует от атакующего обладания недостижимым на практике объемом вычислительных ресурсов либо настолько значительных затрат времени на раскрытие, что к его моменту защищенная информация утратит свою актуальность [5].

СУБД — система управления базами данных [6].

Движок (подсистема хранения) — компонент СУБД, управляющий механизмами хранения баз данных, или библиотека, подключаемая к программам и дающая им функции СУБД [3][4].

ВВЕДЕНИЕ

Обеспечение защиты от неправомерного доступа — мера по защите информации. При работе с информацией лицо, имеющее доступ к этой информации, ограничено правами, которые определяют способы взаимодействия с информацией. В качестве примера таких прав можно привести права на чтение, изменение и удаление информации.

Цель работы — разработать метод хранения данных с возможностью доказательства неправомерного доступа на основе хеш-сумм.

Для достижения поставленной цели требуется решить следующие задачи:

— описать виды защиты информации, классифицировать их;
— рассмотреть базовые элементы и понятия, используемые при проектировании методов хранения информации с возможностью защиты от неправомерного доступа;
— провести анализ существующих методов хранения информации с защитой от неправомерного доступа;
— провести анализ блочного хранения данных в системе, в которой планируется использование метода, на предмет защиты информации от неправомерного доступа;
— спроектировать и реализовать метод блочного хранения данных с возмозможностью доказательства неправомерного доступа;
— исследовать метод на предмет невозможности реализации угроз при различных конфигурациях системы.

1 Аналитическая часть

В данном разделе анализируется предметная область, классифицируются виды защиты от неправомерного доступа, рассматриваются базовые понятия, используемые при реализации таких методов, анализируются существующие решения, рассматривается метод хранения данных в движке MergeTree СУБД ClickHouse и модификация этого метода, предоставляющая возможность доказательства неправомерного доступа.

1.1 Анализ предметной области

1.1.1 Виды защиты информации при блочном хранении данных

Задача по обеспечению защиты от неправомерного доступа может решаться на нескольких уровнях:

1) Отсутствие возможности неправомерного доступа. Самый желаемый уровень, при котором лицо не имеет возможности превышения прав при работе с информацией. Для чтения, например, может достигаться за счет шифрования данных на диске, где ключ шифрования имеется только у лиц, обладающими правами на чтение.
2) Наличие возможности устранения последствий неправомерного доступа. Данный уровень может быть использован совместно с исключением неправомерного доступа, обеспечивая дополнительную защиту. Примерами методов защиты, обеспечивающими описанную возможность, могут послужить резервное копирование и репликация данных.
3) Наличие возможности доказательства неправомерного доступа. Данный уровень служит для ответа на вопрос, являются ли данные консистентными. Для обеспечения такой возможности может использоваться, например, расчет контрольных сумм.

При этом в общем случае работа с блочным хранилищем данных подразумевает 3 действия:

— чтение;

— изменение;
— удаление блока.

При обеспечении защиты на уровне возможности устранения последствий неправомерного доступа, восстановление данных при изменении и удалении возможно при наличии в системе избыточной информации об удаленных или измененных фрагментах. Данное условие накладывает некоторые ограничения на операции изменения и удаления. Таким образом для рассматриваемого уровня защиты можно ввести 2 дополнительные неправомерные операции:

— частичное удаление блоков;
— частичное изменение (или изменение части блоков).

Стоит также учесть, что меры защиты применимы не ко всем операциям. Например, невозможно устранить последствия неправомерного чтения. С учетом этого и всего вышесказанного можно выделить итоговый список видов защиты информации, которые могут обеспечиваться блочным хранилищем данных:

— исключение неправомерного чтения;
— исключение неправомерного изменения;
— исключение неправомерного удаления блока;
— возможность устраненения последствий частичного неправомерного удаления блока;
— возможность устраненения последствий частичного неправомерного изменения;
— доказательство неправомерного удаления блока;
— доказательство неправомерного изменения.

1.2 Базовые понятия

1.2.1 Хеш-функция

Хеш-функции применяются:

— при решении задачи дедубликации;
— при построении идентификаторов;

— при вычислении контрольных сумм;

— при хранении паролей.

При рассмотрении хеш-функций под алгоритмом подразумевается процесс вычисления значения хеш-функции, а под атакой на алгоритм — решение обратной задачи: нахождение для заданного значения хеш-функции H_1 такого массива входных данных A_1, что $f(A_1) = H_1$. Хеш-функцию, которая является стойкой по определению криптографической стойкости по отношению к такой задаче, называют криптостойкой.

Криптостойкие функции также обладают следующим свойством: при наличии массива входных данных A_1 и значения хеш-функции для него $f(A_1)$, настолько же сложной задачей, что и нахождения обратного значения, является задача нахождения такого отличного от A_1 значения массива входных данных A_2, для которого верно $f(A_1) = f(A_2)$. Такие значения A_1 и A_2, для которых верно равенство $f(A_1) = f(A_2)$, называются коллизиями.

1.2.2 Блокчейн

Блокчейн — выстроенная по определенным правилам непрерывная последовательная цепь блоков — элементов, содержащих информацию. [7] В общем случае такая цепь поддерживает 2 операции:

1) построение цепи (в частном случае элемент добавляется в конец и для цепи вычисляется только новое одно значение);
2) проверка целостности всей цепи.

При добавлении блока вычисляется значения хеша от его содержимого и хеша предыдущего блока. Вычисленное значение считается хешом добавляемого блока.

Пример блокчейна изображен на рисунке 1.

При проверке целостности цепи выполняются следующие действия:

1) Высчитывается хеш от содержимого первого блока и сравнивается со значением хеша, записанным при добавлении данного блока в цепь. Если значения не совпадают, то констатируется факт нарушения целостности.

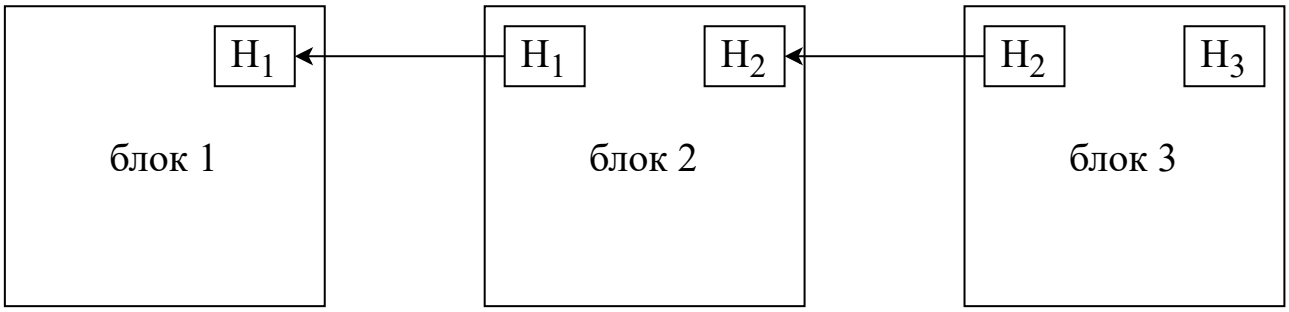

Рисунок 1 – Пример блокчейна

2) Для очередного блока вычисляется значение хеша от его содержимого и хеша предыдущего блока. Вычисленное значение сравнивается со значением хеша, записанным при добавлении блока. При несовпадении констатируется факт нарушения целостности.

При изменении последнего блока достаточно пересчитать и перезаписать значения хеша только для него. Однако при изменении хеша блока, не являющегося последним, потребуется пересчитать значение хеша всех последующих блоков, что в некоторых системах может потребовать больших вычислительных затрат.

1.2.3 Дерево и ориентированный ациклический граф Меркла

Дерево Меркла [8] — двоичное дерево, в листовые вершины которого помещены хеши блоков данных, а внутренние вершины содержат хеши суммы значений в дочерних вершинах. В общем случае операция сложения — произвольная функция от 2 аргументов *f(x, y)*, которая может быть несимметричной (например, конкатенация строк). Корневой узел дерева содержит хеш от всего набора данных, то есть такое дерево является хеш-функцией.

Пример дерева Меркла можно увидеть на рисунке 2.

Применения дерева Меркла:

— Хранение транзакций в блокчейне криптовалют (например, в Bitcoin; позволяет получить значение хеша всех транзакций в блоке, а также эффективно верифицировать транзакции).

— Для множества значений в среде с ограничением по памяти. В среде хра-

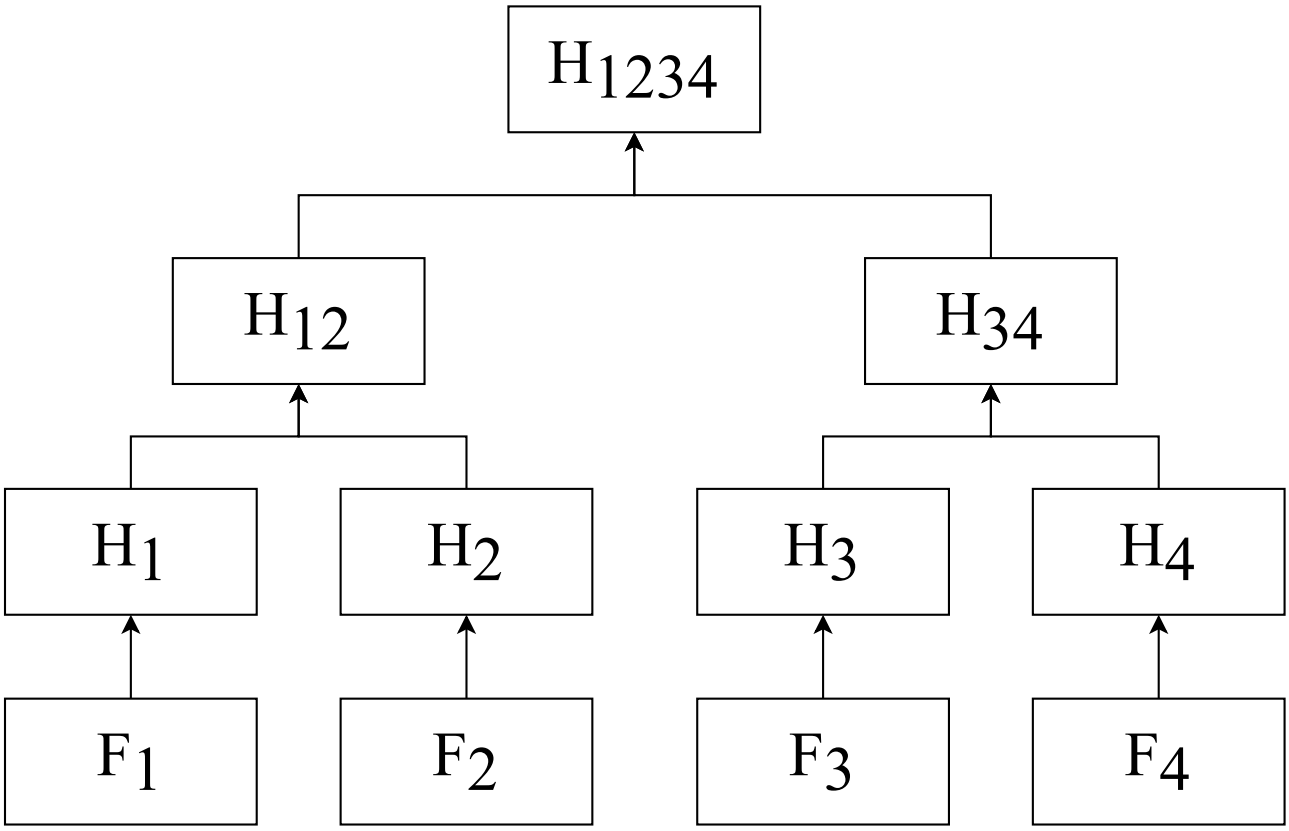

Рисунок 2 – Пример дерева Меркла

нится только корень дерева. Для проверки принадлежности элемента множеству вместе с элементом требуется передать доказательство Меркла — значения всех хешей, с которыми суммируется хеш проверяемого элемента на пути из листа в корень.

Ориентированный ациклический граф Меркла [9] — структура данных, представляющая собой граф, строящийся по следующим правилам:

1) Все вершины графа, степень полуисхода [10] которых равна 0, представляют хеши данных.
2) Вычисление значений остальных вершин графа делается в порядке, обратном топологической сортировке [11]. Хешем очередной вершины будет значение хеша от суммы вершин, в которые есть дуга из рассматриваемой вершины.

Данная структура данных используется в системе версионного контроля Git [12].

1.3 Существующие решения

1.3.1 PASIS

PASIS [13] — распределенное хранилище данных, реализующее метод хранения, защищающий от ошибок, связанных с подменой данных определенными узлами системы, называемыми византийскими. Узлы хранилища хранят данные фрагментами и версионируют информацию. Клиенты читают и пишут фрагменты данных.

Данные клиента распределяются по фрамгентам на N узлов хранилища. При чтении данных клиент обращается к m узлам, которые являются подмножеством узлов, на которые происходила запись, и проверяет валидность полученных фрагментов. Процесс чтения может быть выполнен в несколько итераций, если в результате валидации будет выявлена ошибка. Процесс чтения считается завершенным, если получено m верных фрагментов. Каждый отдельный фрагмент не несет в себе информации, исходные данные можно восстановить только по m и большему числу фрагментам.

Данные восстанавливаются из фрагментов с помощью стирающих кодов [14] и проверяются с помощью объединенных контрольных сумм [13]. Объединенная контрольная сумма — конкатенация хешей всех N записываемых фрагментов. После восстановления данных в наличии у читателя имеются все N фрагментов, для которых можно рассчитать значение хеша и проверить подлинность данных.

Таким образом использование данной системы позволяет защититься от неправомерного изменения или удаления данных на части узлов хранилища с возможностью восстановления данных, а также доказать неправомерное изменение данных.

1.3.2 Криптографические файловые системы

TCFS[15], NCryptFS[16] — примеры криптографических файловых систем. В отличие от обычных файловых систем, данные системы имеют допол-

нительный слой между виртуальной файловой системой и драйвером файловой системы, который шифрует данные. При такой реализации работа слоя шифрования данных будет прозрачна для пользователя, потому что пользователь работает с виртуальной файловой системой.

TCFS использует стандартные для файловых систем способы подключения в ОС `Linux`: системные вызовы `mount`, `ioctl` [17]. При работе в пределах смонтированной файловой системы все операции записи и чтения проходят через слой шифрования, поэтому данные записываются на устройство только в зашифрованном виде.

NCryptFS обладает похожим принципом работы, однако подключается с помощью собственной команды `attach`, не используя стандартные методы, и имеет возможность создания пространства имен для управления правами групп пользователей на доступ к определенным директориям, а также содержит средства авторизации.

Описанные системы решают задачу неправомерного чтения. При случайной записи данных на устройство будет нарушена их целостность, проверка целостности возлагается на приложения, использующие данные файловые системы, но так как данные хранятся в зашифрованном виде, целостность исходных данных может быть нарушена более значительно. Возможности восстановления данных нет.

1.3.3 OceanStore

OceanStore [18] — безопасная распределенная read-only файловая система.

Данное хранилище хранит данные в одноранговой сети, где узлы соединены между собой по принципу точка-точка. Преимуществом данного хранилища является хранение данных по фрагментам с использованием стирающих кодов. Данный подход, как уже было показано в 1.3.1, позволяет получать исходные данные, записанные как N фрагментов, из любых m фрагментов, где $m < N$.

В данной системе используется понятие самопроверяющихся данных [19]:

данные адресуются по идентификатору GUID, который является хешем от содержимого. Данный хеш считается из дерева Меркла, изображенного на рисунке 3. Каждый фрагмент, помимо самих данных, содержит в себе также доказательство Меркла. При восстановлении данных проверяется каждый фрагмент по отдельности (с помощью доказательства Меркла), а также все данные целиком. Порядок действий можно описать следующим образом:

1) Получить *m* фрагментов, проверив у каждого доказательство Меркла.
2) Восстановить *N* фрагментов с использованием стирающих кодов.
3) Восстановить данные из *N* фрагментов.
4) Построить дерево Меркла и убедиться в равенстве корня дерева и GUID.

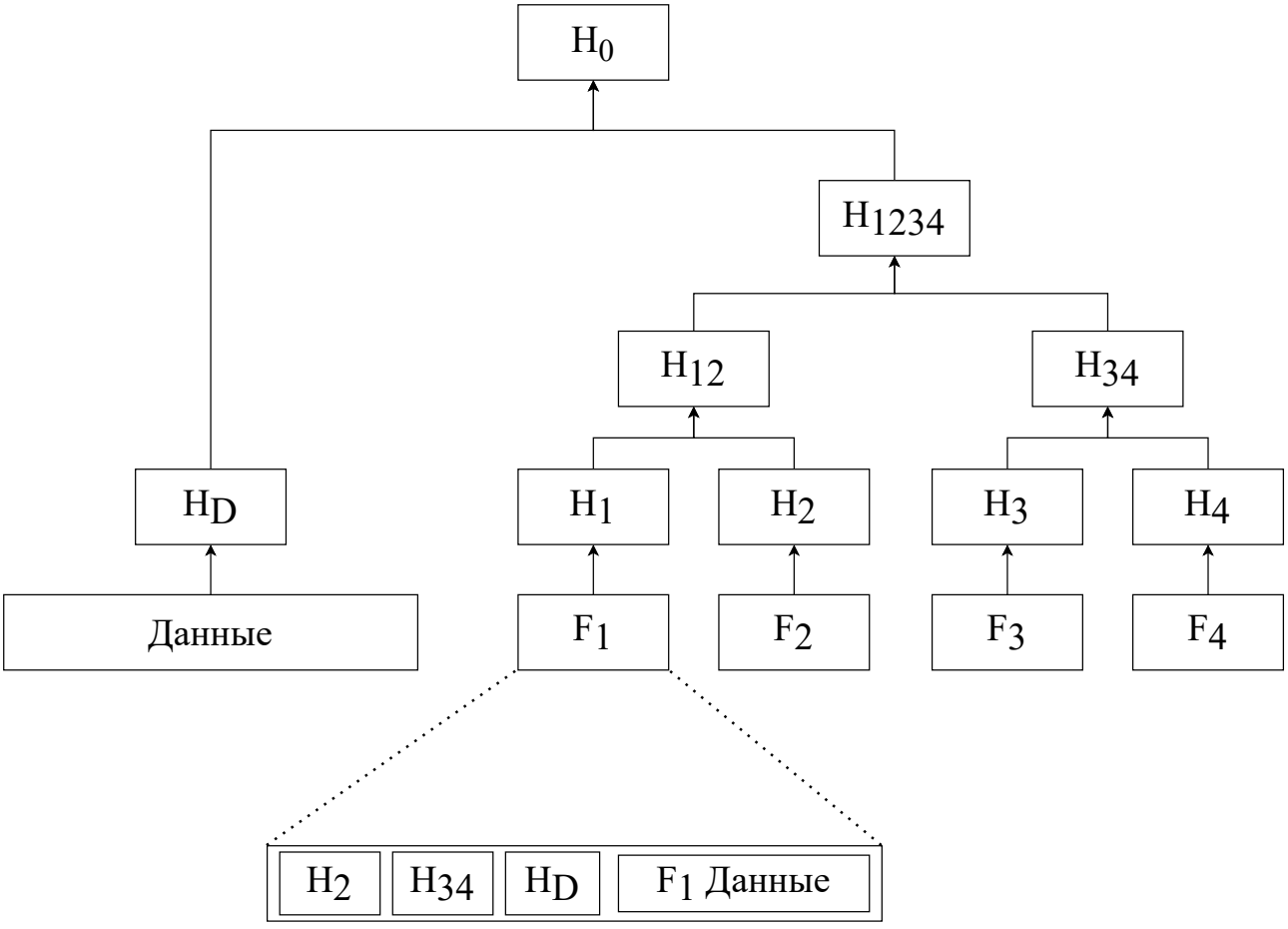

Рисунок 3 – Пример дерева Меркла и фрагментов для OceanStore.

Данная система считается read-only, потому что изменение данных без изменения идентификатора невозможно, так как идентификатор создается на базе содержимого. Таким образом, например, при обновления версии каких-

то данных, старая версия остается неизменной, но появляется новая версия с новым содержимым и GUID.

Данное хранилище, несмотря на внутренее устройство, поддерживает возможность иерархической структуры данных. Для этого требуется в некоторых объектах, которые не являются листьями дерева иерархии данных, хранить хеши дочерних вершин. GUID корня иерархической структуры известен.

На примере иерархической структуры видна сложность изменения данных: изменение элемента дерева приводит к необходимости изменения (замены на новый элемент) всех вершин-предков.

Как и хранилище из раздела 1.3.1, данная система защищена от частичного удаления или изменения данных, обладает возможностью восстановления, а также обладает возможностью доказательства изменения данных.

1.3.4 Git

Git [12] — система контроля версий. Данная система не решает задачу защиты от неправомерного доступа непосредественно, однако делает это косвенно: при нарушении целостности данных система перестанет работать, что будет являться доказательством неправомерного доступа. Внутреннее представление Git — хранилище, адресующее по содержанию. Это означает, что файлы, которыми оперирует Git в рамках системной файловой системы, хранятся в виде ассоциативного массива, доступ в котором осуществляется по ключу. Данная система называется Object Store. В качестве ключа файла выступает хеш от его содержимого, подобно GUID в 1.3.3.

Иерархическую структуру файлов Git поддерживает в виде ориентированного ациклического графа Меркла за счет наличия 3 основных типов объектов:

1) blob — содержит информацию о конкретной версии файла. В ациклическом графе Меркла файлы этого типа всегда обладают нулевой степенью полуисхода, то есть их хеш составлен на основе содержимого файла. Этим же хешем данные файлы адресуются в упомянутом выше ассоциа-

тивном массиве.

2) tree — содержит информацию о конкретной версии директории. Данный объект содержит в себе другие объекты (больше 0), которые могут быть типа blob или tree. Он обладает отличной от обычной функцией суммирования: вместо простой конкатенации хешей вершин, в которые ведут дуги, в функции суммирования также фигурируют имена файлов дочерних вершин (реальные имена, не хеши), а также режимы доступа к данным файлам.
3) commit — содержит информацию о коммите — фиксированном состоянии системы. commit обладает единичной степенью полуисхода и единственная его дуга ведет в объект типа tree, соотвествующей корневой директории репозитория. Объект типа commit хранит информацию об авторе изменений, связанных с коммитом, об имени, хеше и режиме доступа корневой директории и информацию о хеше родительского коммита — то есть предыдущей зафиксированной версии. Адресуется коммит значением хеш функции от его содержимого. Схема, по которой описанный коммит связан с предыдущим, в точности соответствует схеме работы блокчейна.

На рисунке 4 схематично представлен пример графа для 3 основных типов объектов.

Изменение внутренней структуры в процессе изменения данных происходит в 3 этапа:

1) Изначально Git не будет учитывать изменения в рабочей директории, то есть при изменении файла у системы будет иметься копия файла из рабочей директории, которая не будет изменена.
2) Чтобы система учла изменения, требуется добавить измененные файлы в индекс. Данное действие приведет к изменению файла с точки зрения файловой системы, с точки зрения внутреннего состояния системы Git это приведет к созданию нового объекта типа blob в Object Store, а также вир-

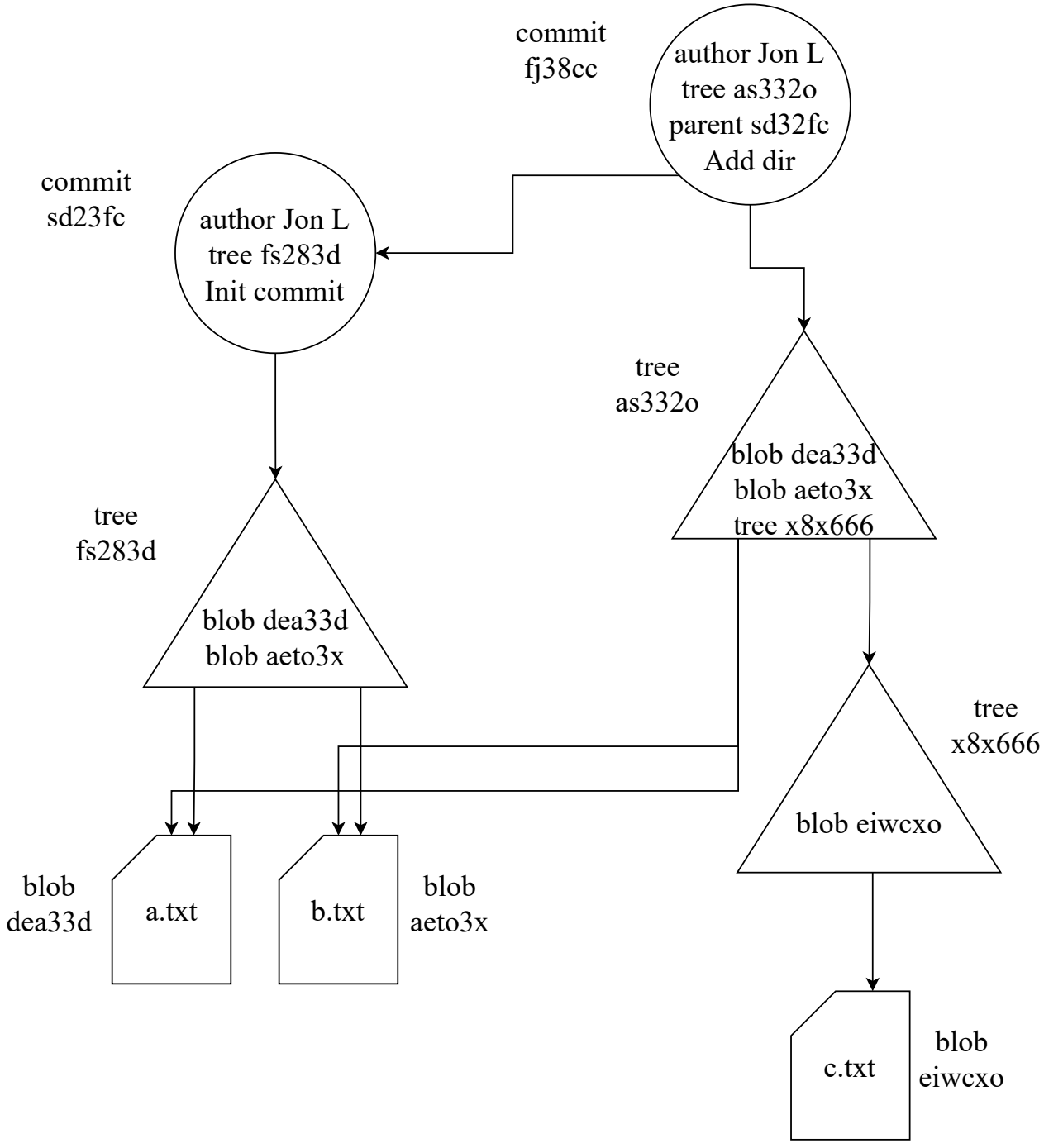

Рисунок 4 – Пример графа Меркла для 3 основных типов объектов

туального объекта типа tree (данный объект не будет виден в Object Store на этом этапе). Возможное состояние системы на этом этапе изображено на рисунке 5.

3) При фиксации изменений виртуальный объект типа tree, соответствующий корню рабочей директории для новых версий файлов, добавленных в индекс, становится реальным, то есть создается объект в Object Store, а также создается объект типа commit, который ссылается (содержит хеш в качестве значения) на вновь созданный объект типа tree. Возможное состояние после всех проделанных действий изображено на рисунке 6.

Неправомерный доступ подразумевает возможность доступа на запись к внутренней структуре, изменения в которой происходят исключительно за счет добавления новых объектов.

Возможны 3 варианта:

1) Изменение содержимого объекта. Имя файла в системе Git — его хеш и любое изменение содержимого приведет к несовпадению рассчитываемого хеша и имени файла, что приведет к ошибке и обнаружению нарушения в данных.
2) Удаление объекта. Такая атака будет обнаружена в момент, когда произойдет попытка обращения к удаленному файлу. Он не будет найден в Object Store и будет ошибка.
3) Добавление объекта. Добавление объекта не будет обнаружено, однако и данные, добавленные с новым объектом, не будут использованы ввиду того, что никакой другой объект не будет ссылаться на вновь созданный. Если добавить коммит в конце цепочки, то появляется возможность посчитать хеш вместе с хешем предыдущего commit объекта, однако подобный формат работы похож на стандартный и может быть осуществлен извне.

Таким образом данная система обладает возможностью доказательства неправомерного изменения, а также удаления. Однако в общем случае возмож-

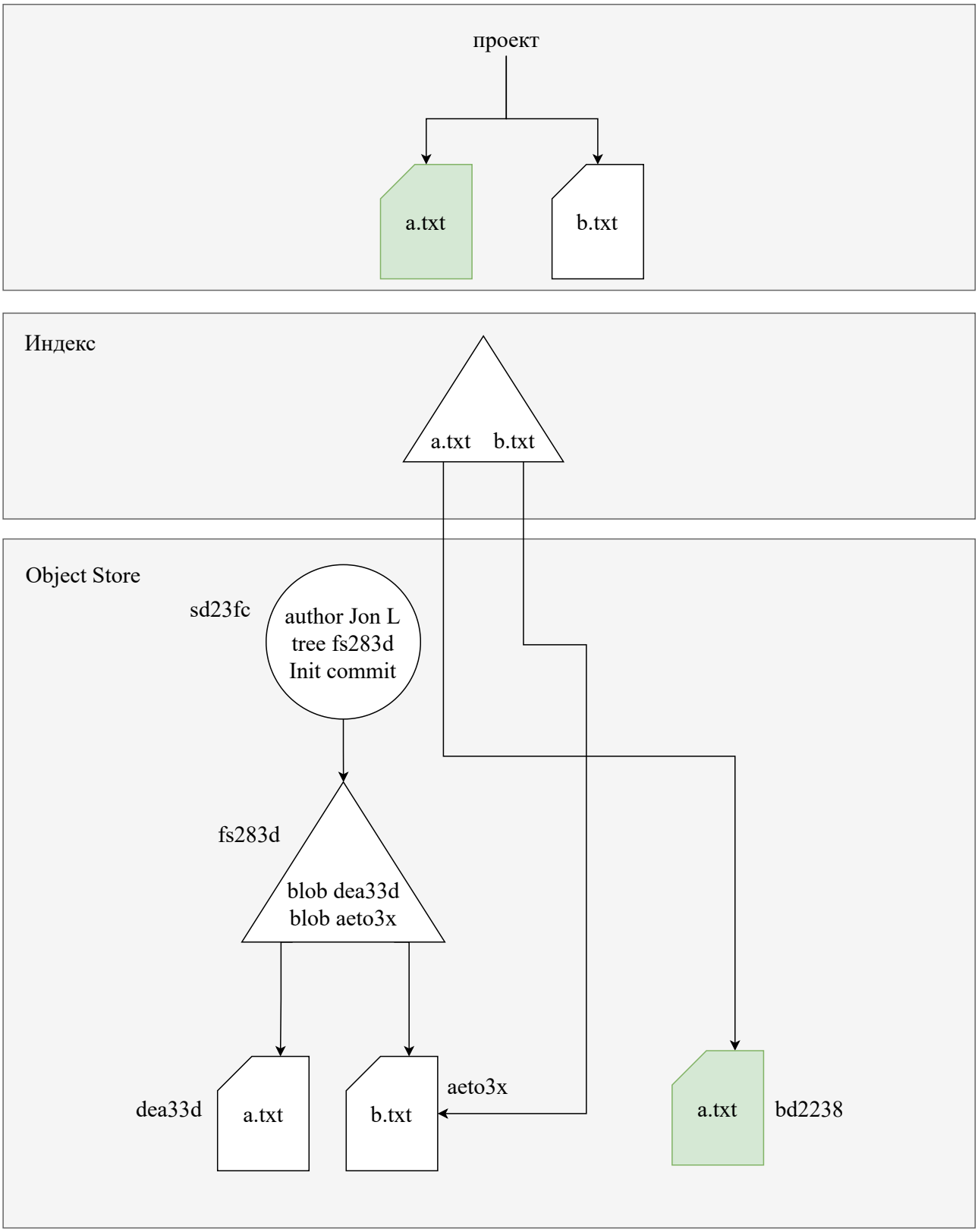

Рисунок 5 – Возможное состояние системы контроля версий Git при добавлении нового файла в индекс.

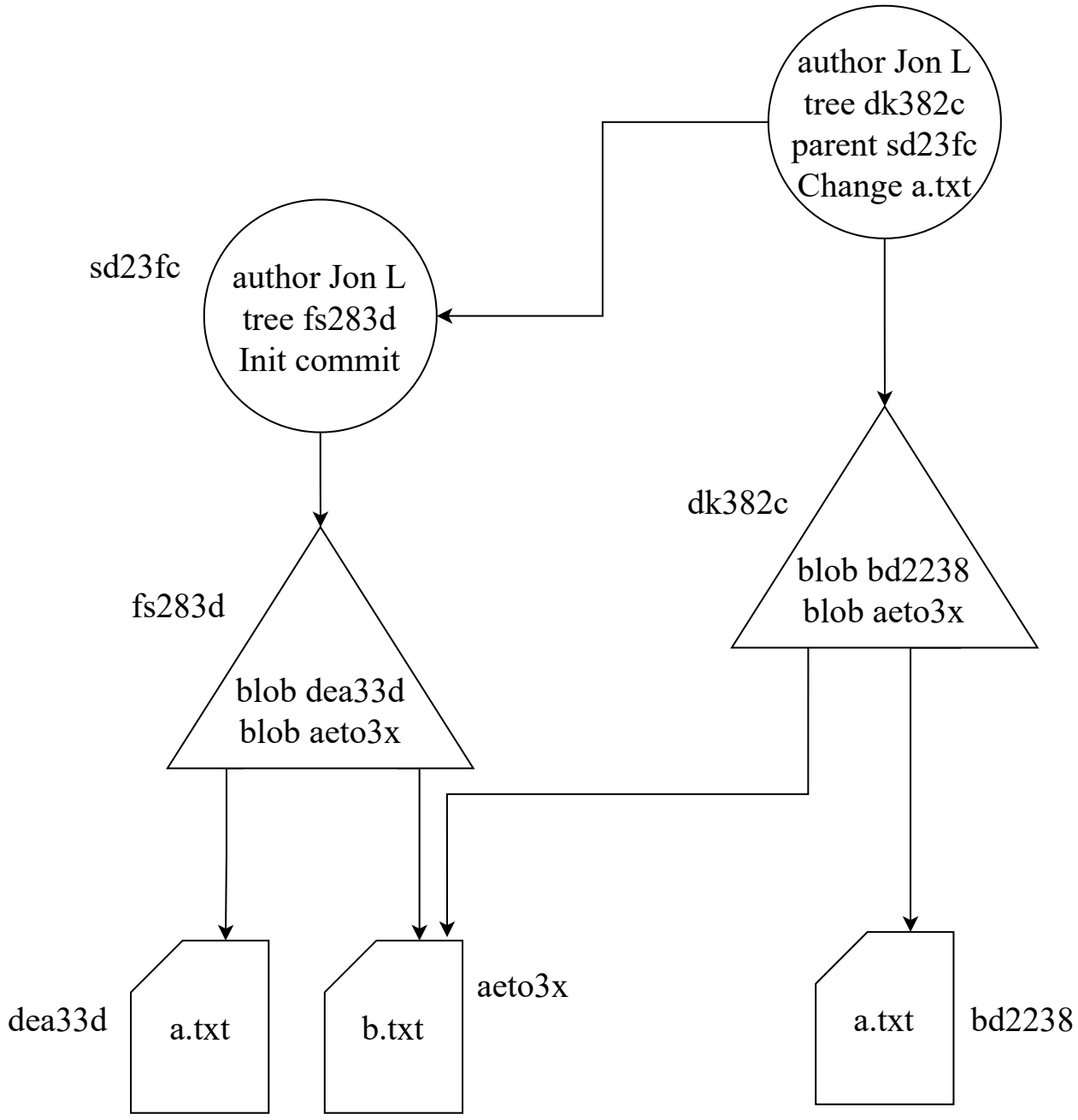

Рисунок 6 – Возможное состояние системы контроля версий Git после фиксирования изменений.

но пересчитать значения хешей, на которые влияет изменение или удаление, что приведет к потере возможности доказательства.

1.3.5 Bitcoin

Bitcoin [7] — одноранговая децентрализованная электронная денежная система.

Для данной системы недопустимы сценарии, при которых данные теряются и приводится только доказательство неправомерного доступа к данным. Ввиду того, что данная система является распределенной, у нее нет единого хранилища данных, что усложняет получение неправомерного доступа к данным и их изменение или повреждение без возможности восстановления.

Транзакция — операция передачи биткойнов с одного адреса на другой. Транзакция осуществляется следующим образом:

1) Владелец биткойн-адреса (который связан с публичным ключом) использует любую совершенную транзакцию, в которой адресом назначения является данный адрес.
2) Создается новая транзакция, которая содержит информацию о наборе адресов и соответсвующем количестве биткоинов, которые требуется перевести в рамках транзакции.
3) Данная транзакция подписывается приватным ключом владельца кошелька, с которого происходит передача биткоинов.

Таким образом невозможность неправомерного изменения транзакции обеспечивается за счет PKI [20].

Транзакция является составным элементом блока. Блок — составленный по определенным правилам набор транзакций.

Блок имеет определенную структуру, в которую входят:

— магическое число;
— размер блока;
— заголовок блока;
— счетчик транзакций;

— набор транзакций.

Заголовок блока состоит из следующих полей:

— версия блока;

— хеш заголовка предыдущего блока;

— хеш корня дерева Меркла, составленного из транзакций, входящих в данный блок;

— текущее время;

— целевое значение хеша, ниже которого должен быть хеш от заголовка;

— число *Nonce*, которое подбирается таким образом, чтобы хеш от заголовка был меньше целевого значения.

Блоки образуют блокчейн — в заголовке очередного блока содержится хеш предыдущего блока, с помощью которого блоки собираются в цепь. Из одного блока может исходить несколько других блоков, которые содержат первый, как предшествующий. Действующей цепочкой считается наиболее длинная. В этом смысл концепции, называемой proof-of-work: большинство пользователей берут наиболее длинную цепочку блоков и коллективно пытаются подобрать требуемое значение *Nonce*, которое бы удовлетворяло условию целевого значения, тем самым концентруя вычислительные мощности и не давая другим цепочкам стать длиннее текущей. Наиболее быстро будет рассчитываться та цепочка, для расчета которой используется наибольшее количество мощностей.

Сложность расчета значений *Nonce* контролируется за счет изменения целевого значения хеша. Можно дать вероятностную оценку сложности вычисления нужного значения хеша в количестве попыток, исходя из следующих условий:

— значение хеш функции является случайным значением;

— число ведущих нулей в бинарном представлении целевого значения равняется числу M.

Тогда вероятность появления такого значения хеша, которое будет меньше целевого значения, равна вероятности выпадения M первых нулей при расчете

хеша и рассчитывается по формуле 1.

$$P = \frac{1}{2^M}, \tag{1}$$

С учетом независимости расчетов для различных $Nonce$ и формулы 1, можно рассчитать количество попыток, требуемых совершить в среднем для получения одного удовлетворяющего числа по формуле 2.

$$N = \frac{1}{\frac{1}{2^M}} = 2^M, \tag{2}$$

Наиболее известный способ атаки на системы, работающие по концепции proof-of-work — атака 51%. Суть атаки заключается в том, что атакующий владеет большинством вычислительных мощностей и делает следующие действия:

1) Сделать блок, который будет содержать некоторую транзакцию T_1.
2) Дождаться, пока система, на счет которой мы перевели биткоины, будет считать транзакцию T_1 завершившейся.
3) Взять блок, предшествующий блоку из пункта 1, и начать делать новую цепочку без транзакции T_1 до тех пор, пока она не станет действующей.

Такая атака позволяет неправомерно изменить данные.

Выводы

Сравнение рассмотренных методов хранения данных с возможностью защиты от неправомерного доступа можно увидеть в таблице 1. Обозначения:

— НД — неправомерное действие;
— КФС — криптографические файловые системы;
— У — удаление блока;
— И — изменение;
— ЧУ — частичное удаление блока;
— ЧИ — частичное изменение.

Полученные результаты можно свести к следующему:

— Защита от чтения возможна в системе, в которой определенная группа людей обладает некоторым уникальным знанием (ключ в КФС).
— Восстановление данных после частичного изменения или удаления блока возможно в распределенных системах, использующих стирающий код, в связи с тем, что фрагменты хранятся в разных местах и допускается неправомерный доступ в части мест.
— Исключает возможность полного неправомерного изменения только Bitcoin, потому что во всех рассмотренных системах количество узлов, в которых хранятся данные, конечно и задано наперед (хоть и в теории может быть очень велико), в то время как в Bitcoin все узлы системы содержат одно состояние.
— Возможностью доказательства неправомерного изменения обладают все системы кроме КФС, используя для этого контрольные суммы и хеши. Однако также как в случае удаления, в локальной системе без шифрования нарушитель может восстановить целостность цепочки блоков при изменении внутреннего состояния Git.
— Возможностью доказательства неправомерного удаления блока обладают 2 системы, в основе которых лежит блокчейн. Однако если данные

не шифруются никаким образом, то в случае с Git человек, нарушающий права, способен повторить действия самой системы, и произвести изменения, поддержав целостность цепочки блоков.

Таблица 1 – Методы хранения и обеспечиваемая защита.

НД	**PASIS**	**КФС**	**OceanStore**	**Git**	**Bitcoin**
чтение (исключение)	-	+	-	-	-
ЧУ или ЧИ (восстановление)	+	-	+	-	-
изменение (исключение)	-	-	-	-	+
изменение (доказательство)	+	-	+	+/-	+
удаление (доказательство)	-	-	-	+/-	+

1.4 Метод хранения данных в СУБД ClickHouse

ClickHouse — колоночная СУБД [21]. Блоки данных, которые хранит СУБД, физически поделены на столбцы, а не на строки. Формат хранения данных не единый и зависит от используемого движка, движок задается для каждой таблицы. Разные движки имеют один и тот же интерфейс, оперируют общими структурами, однако внутреннее строение отличается. В рамках данной работы рассматривается движок MergeTree.

1.4.1 Движок MergeTree

Кусок — блок, в виде которого движок хранит данные [23]. Физически определяется как директория на диске, хранящая в себе файлы, с информацией о данных, индексах, хеш-суммах и другой метаинформацией.

Партиция — логическая группа кусков, определяемая по заранее известному признаку [23]. Таким признаком, например, может быть неделя, к которой принадлежат данные. Физически определяется как составная часть имени куска, каждый кусок принадлежит только 1 партиции. Например, если недели идентифицируются датой понедельника, то название куска хранит в себе дату понедельника недели, в которую он (и все данные, входящие в него) входит. Партиции образуют непересекающиеся группы данных, что позволяет эффективнее выполнять операцию поиска.

Операция слияния кусков — объединение подмножества кусков в 1 кусок, содержащий все данные объединяемого подмножества [23]. Данный подход используется для исключения большого количества перезаписи данных для сохранения требуемого порядка элементов. Из того, что кусок принадлежит только 1 партиции, следует, что объединение блоков, принадлежащих разным партициям не происходит.

Идея движка MergeTree заключается в том, что приходящие блоки с данными разбиваются на куски по признаку партиции, полученные куски отсортированы по первичному ключу, записываются в БД, а затем над некоторыми

подмножествами кусков в асинхронном режиме выполняется операция слияния для более эффективного хранения и поиска.

Каждый кусок данных логически делится на гранулы. Гранула — минимальный набор данных (множество элементов), который считывается движком при выборке данных [23]. Гранула также является единицей индексации данных: индексируются только первые элементы каждой гранулы. Для каждого куска данных движок создает файл с засечками (индексный файл). Для каждого столбца, независимо от того, входит он в первичный ключ или нет, сохраняются эти же засечки. Засечки используются для поиска данных напрямую в файлах столбцов. Размер гранул оганичен настройками движка по количеству и байтам. Количество строк в грануле лежит в диапазоне не больше максимально заданного значения, в зависимости от размера строк. Размер гранулы может превышать максимально заданный размер в байтах в том случае, когда размер единственной строки в грануле превышает значение настройки. В этом случае, размер гранулы равен размеру строки.

Описанные выше концепции показаны на рисунке 7.

1.4.2 Хранение данных в файловой системе

Существует 2 вида физического хранения столбцов куска в MergeTree:

— компактный [24];
— широкий [25].

Отличие заключается в том, что компактный вид хранит все столбцы в 1 файле, а широкий — в разных. Примеры компактного и широкого хранения столбцов куска показаны на рисунках 8 и 9. Для всех файлов в движке считается хеш-сумма, для файлов с данными считаются 2 — для данных в сжатом виде и в расжатом. Поэтому при компактном хранении для данных будет 1 хеш-сумма, а при широком будет соответствовать количеству столбцов.

Для записи данных используется класс `WriteBuffer`. При обоих видах хранения данные пишутся через 4 буфера (в скобках указаны соответствующие имена переменных):

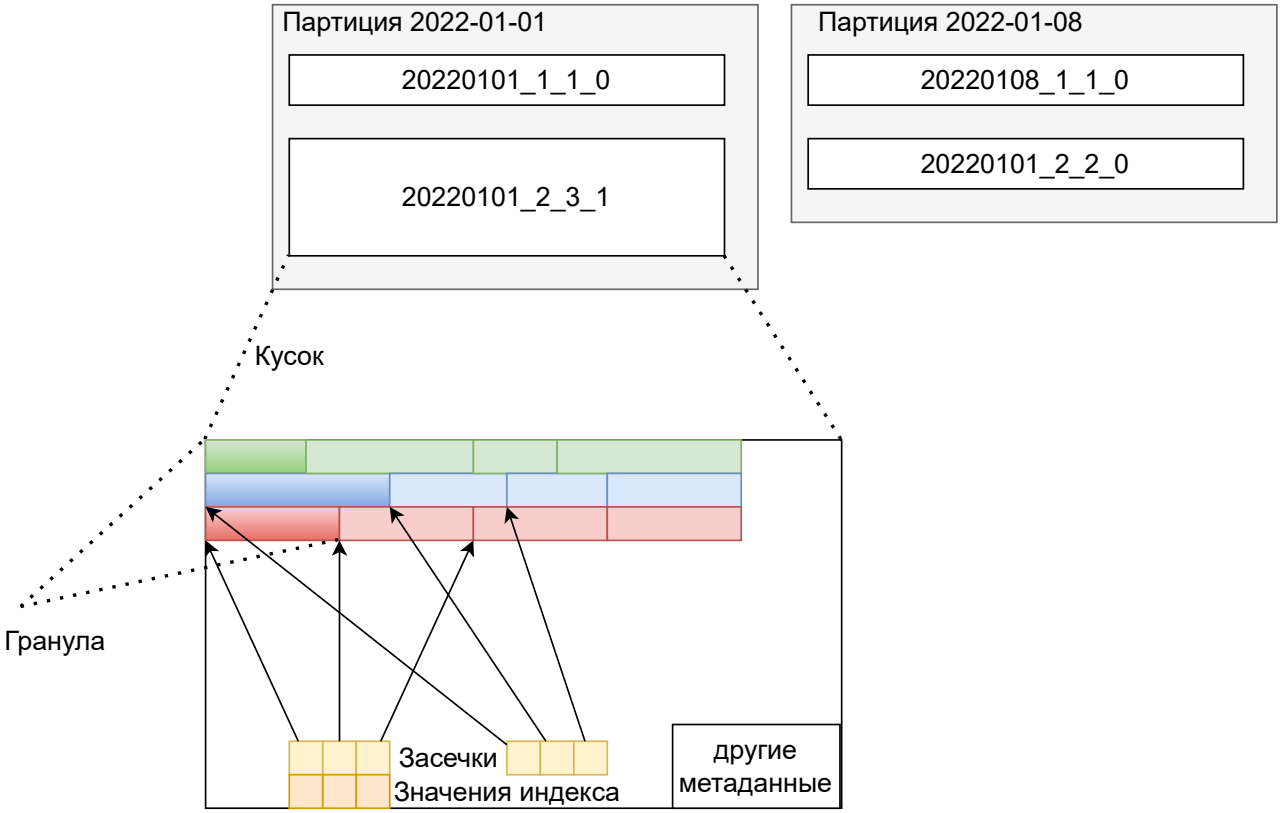

Рисунок 7 – Структура хранения данных в СУБД ClickHouse.

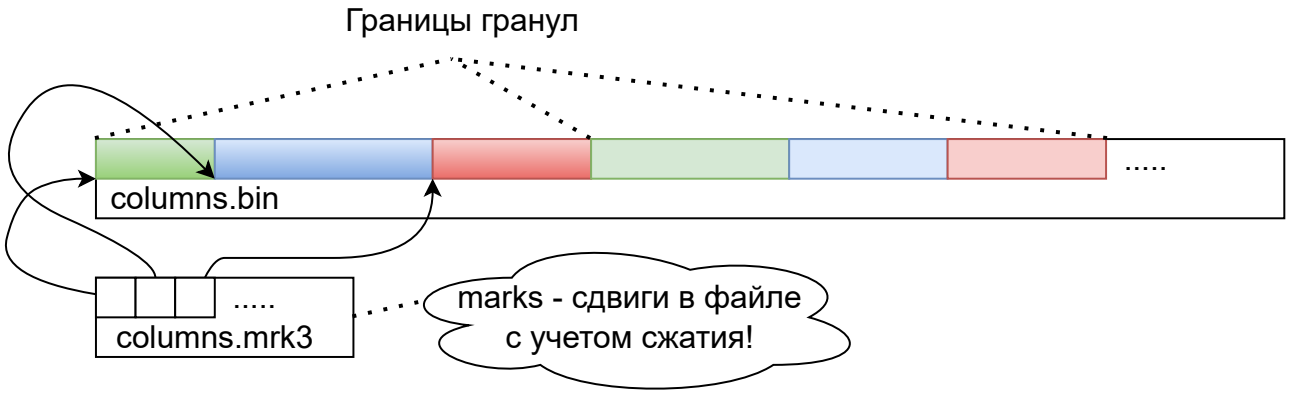

Рисунок 8 – Компактное хранение столбцов.

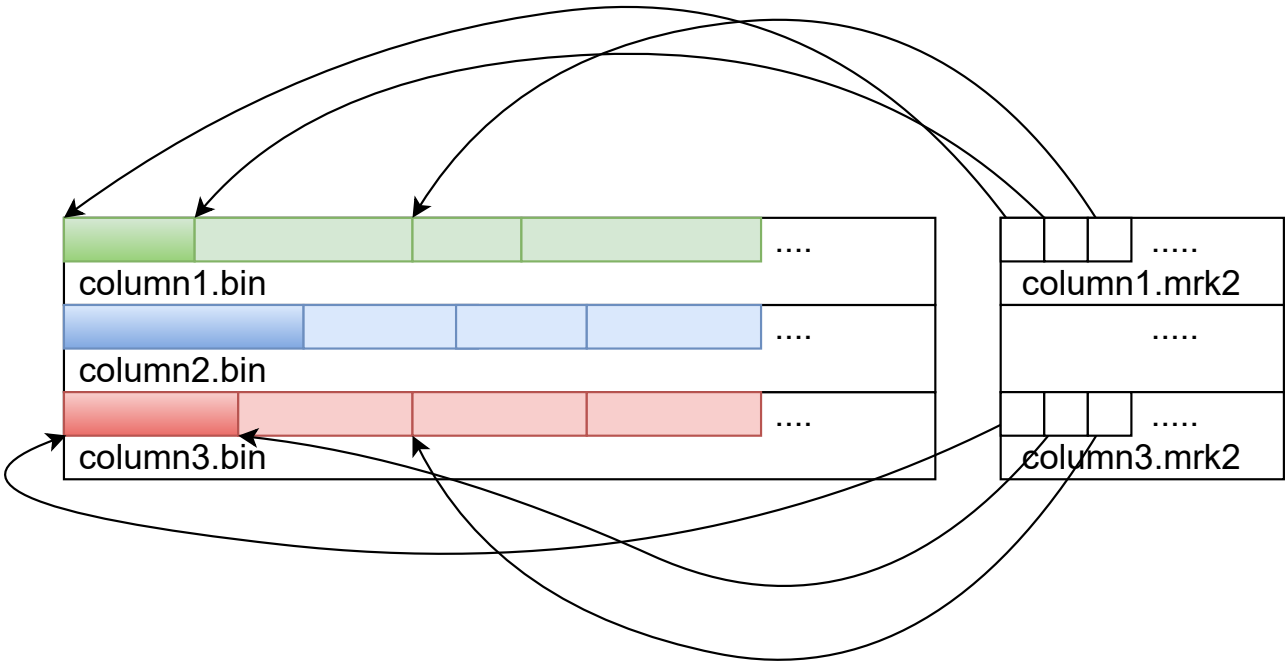

Рисунок 9 – Широкое хранение столбцов.

— хеш-буфер (`hashing_buf`) [26] — считает хеш-сумму несжатых данных;
— сжимающий буфер (`compressed_buf`) [27] — сжимает данные;
— хеш-буфер (`plain_hashing`) [26] — считает хеш-сумму сжатых данных;
— файловый буфер (`plain_file`) [28] — буферизует данные для более эффективной записи в файл.

В действительности данные буферизуются только на 2 уровнях: на уровне сжимающего и файлового буферов. Хеш-буферы при поступлении очередных данных используют поток байт для расчета хеш-суммы и сразу же перенаправляют в нижележащие буферы. Сжимающий буфер, помимо сжатия, считает хеш-сумму для каждого сжатого блока данных. Схема показана на рисунке 10. Размер буферов по умолчанию — 1 Мб. Буфер сбрасывается при достижении 1 из 2 условий:

— буфер переполнился;
— записался целиком столбец гранулы (даже при компактном хранении, для более оптимального чтения).

1.4.3 Вставка данных

Вставка куска данных происходит в 2 шага:

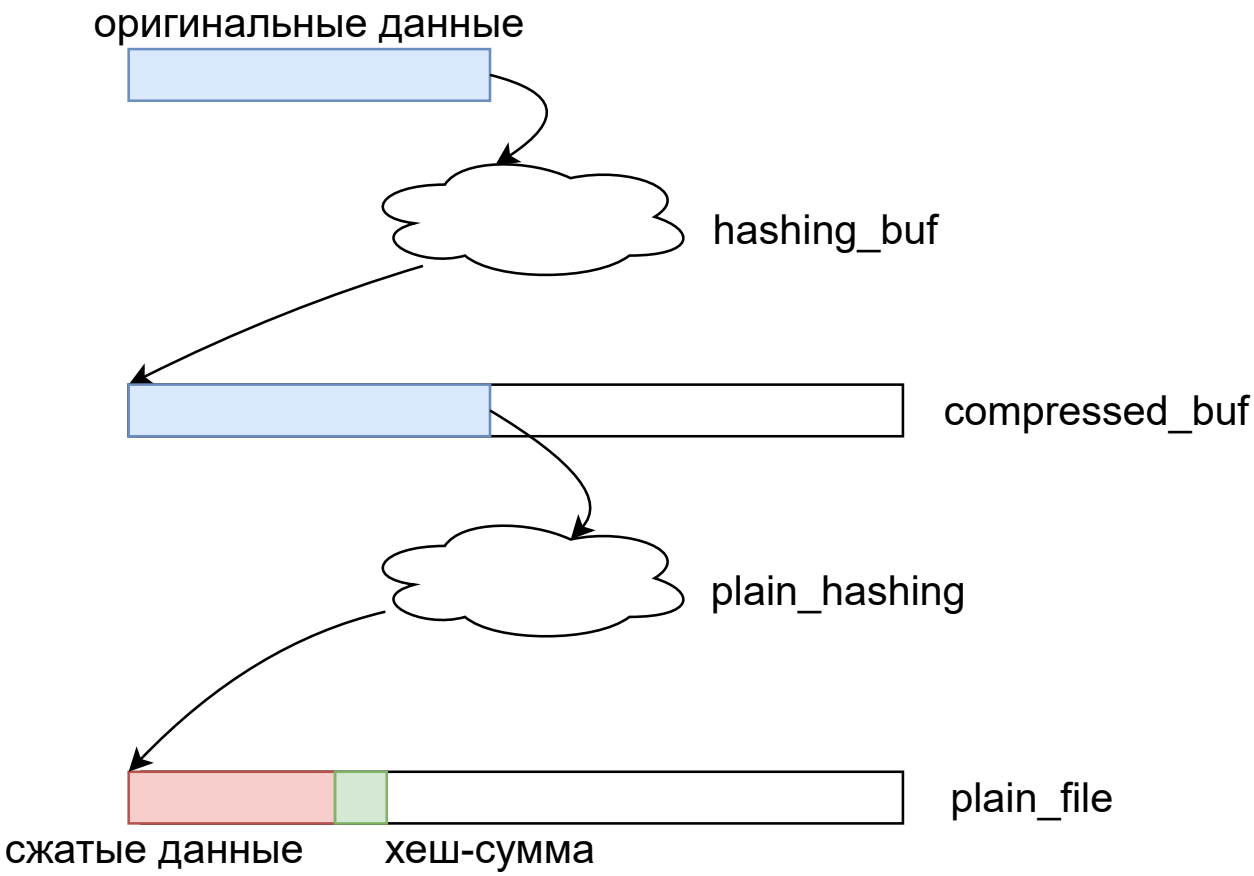

Рисунок 10 – Схема записи с 4 буферами.

— создается временный кусок, данные пишутся в него [29];
— если данные успешно записываются, то временный кусок переименовывается в обычный [30].

Такая схема решает проблему, связанную с битыми кусками: если вставка не проходит, то кусок остается временным и удаляется.

1.4.4 Слияния и мутации кусков

Слияния кусков, как и мутации, происходят в фоновом режиме и состоят из следующих этапов [31]:

— запуск планировщика фоновых задач;
— выбор кусков, которые можно слить или мутировать;
— постановка задачи обработки выбранных кусков в очередь на выполнение фоновому пулу потоков.

Сами задачи выполняются пошагово. Каждая задача имеет метод `bool execute()`, который возвращает `false`, когда задача выполнена, и `true` в обратном случае. Класс, выполняющий задачу, вызывает данный метод до тех пор, пока задача не будет выполнена. Такое дробление задачи на части позволяет откладывать фоновые задачи при повышенной нагрузке системы, лучше планировать, снижая приоритет долгим задачам.

И в случае с мутациями, и в случае со слияниями результатом задачи всегда является один кусок. Задача сохранения целостности решается похожим на вставку образом: все операции происходят со временным куском, а последним шагом является переименование в обычный. Однако в отличие от вставок, данные операции содержат куски, которые становятся неактуальными и которые нужно убрать. Данная задача в слияниях и мутациях решается по-разному.

Имена кусков создаются по шаблону и состоят из следующих частей [32]:

— имя партиции, к которой принадлежит кусок;
— минимальный номер вставленного куска, который был слит в текущий;
— максимальный номер вставленного куска, который был слит в текущий;
— версия куска с точки зрения слияний (инкрементируется каждый раз, ко-

гда происходит слияние);

— версия куска с точки зрения мутаций (разделяет номер со вставляемыми кусками для возможности определения необходимости применения мутации к отдельным кускам).

В слияниях неактуальные куски узнаются из версии. Пусть имеется 3 куска: `all_1_1_0`, `all_2_2_0`, `all_3_3_0`. При слиянии новый кусок получит имя: `all_1_3_1`. Старые 3 куска будут покрыты [33] новым и будут удалены при завершении работы сервера или при перезапуске.

В мутациях неактуальные куски также узнаются из версии, но из другой. Более того, мутации происходят поочередно над каждым куском (несколько кусков не могут участвовать в мутации), при этом к одному куску может применяться несколько мутаций одновременно.

Покрывающие куски в операциях слияния и мутаций показаны на рисунке 11 зеленым цветом, покрываемые — красным.

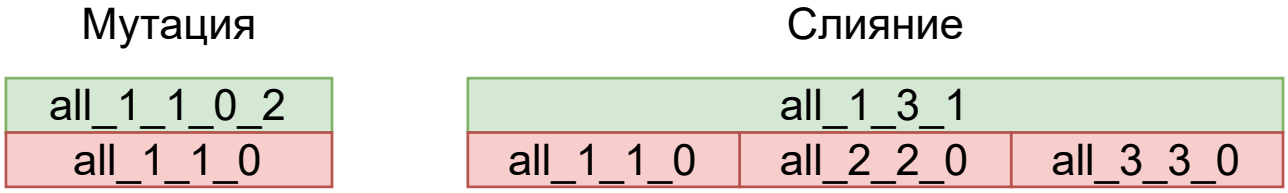

Рисунок 11 – Покрывание старых кусков новыми при мутациях и слияниях.

Такой алгоритм действий позволяет избежать неконсистентного состояния: атомарно выполняется 1 операция (переименования куска), остальные операции выполняются как следствие покрытия.

1.4.5 Шифрование данных

В ClickHouse доступно шифрование на уровне диска, указанного в конфигурационном файле. Реализовано оно схожим со сжатием образом: поверх буфера записи в файл находится буфер шифрования. Алгоритм шифрования — AES с ключами размеров 128, 192, 256 бит. В отчете NIST [34] за 2019 год указано, что AES допустимо использовать как алгоритм блочного шифрования до

2030 года с любой из перечисленных длиной ключа.

Также доступно шифрование столбцов тем же алгоритмом, которое реализовано в виде кодека. Допустимо использования нескольких кодеков одновременно, что позволяет сжимать и шифровать данные одновременно.

1.4.6 Проверка целостности данных

Проверка целостности для каждого куска состоит из следующих шагов:

— проверить вычисленные значения хеш-сумм всех сжатых блоков с записанными значениями;

— проверить вычисленное значение хеш-суммы для сжатых данных из файла с данными (или файлов при широком хранении) с записанным значением;

— проверить вычисленное значение хеш-суммы для расжатых данных файла с данными (или файлов при широком хранении) с записанным значением;

— проверить вычисленные значения хеш-сумм для файлов с метаданными с записанными значениями.

Все метаданные, помимо файловой системы, кэшируются в оперативной памяти, в том числе информация об активных кусках и хеш-суммах. При изменении данных поменять закэшированное в оперативной памяти значение хеш-суммы не получится, однако при перезагрузке СУБД значения можно подменить так, что бы было невозможно доказать изменение или удаление данных.

В случае изменения данных придется пересчитать 3 хеш-суммы (у сжатого блока, у сжатого файла и у расжатых данных), а в случае удаления куска состояние будет оставаться валидным без дополнительных действий.

1.4.7 Анализ защиты от неправомерного доступа в движке MergeTree

С учетом выводов 1.3 и того, что движок MergeTree не является распределенным, можно сделать следующие выводы:

1) При отсутствии шифрования, злоумышленник обладает возможностью делать действия, аналогичные действиям лица, обладающего нужным до-

ступом к информации, в связи с чем система не имеет никакой защиты.

2) При наличии шифрования:
   — исключается возможность неправомерного чтения;
   — есть возможность доказательства неправомерного изменения.

Возможность доказательства неправомерного удаления целого куска отсутствует.

1.5 Добавление возможности доказательства неправомерного удаления в MergeTree

Для обеспечения возможности доказательства неправомерного удаления можно сделать:

1) расчет хеш-суммы на основе информации обо всех кусках;
2) расчет хеш-сумм кусков зависимым от метаданных (хеш-сумм) других кусков.

Данные подходы решают задачу по-разному: в первом случае мы будем иметь одно значение, во втором — количество значений будет совпадать с количеством кусков. Первый подход меньше взаимодействует с файловой системой, в следствие чего быстрее работает, второй подход дает возможность определить, на каком элементе нарушается консистентность (при условии, что данные не будут меняться после нарушения консистентности).

Пусть y_i — искомые хеш-суммы. Пусть для определенности новые хеш-суммы (относящиеся не к конкретному куску, а поддерживающие некоторые связи) будут называться *цепью хеш-сумм* в случае со множеством хеш-сумм, и *результатом цепи хеш-сумм* в случае хранения 1 значения.

Тогда единственная хеш-сумма на основе информации обо всех кусках может рассчитываться по формуле 3, подобно тому, как это делается в 1.3.1, либо по формуле 4 подобно тому, как это делается в 1.3.5.

$$y_0 = hash(x_1|x_2|...|x_n), \tag{3}$$

$$y_0 = hash(x_n|hash(x_{n-1}|...)), \tag{4}$$

где $hash$ — хеш-функция, n — количество кусков данных, | — операция конкатенации строк, x_i — метаданные i-ого куска.

Хеш-суммы кусков, зависимых от метаданных других кусков, могут вычисляться по рекуррентной формуле 5, похожей на 4, но с запоминанием промежуточных состояний.

$$y_i = hash(x_i|y_{i-1}), y_1 = hash(x_1). \tag{5}$$

В рамках данной работы будет использоваться подход с хранением 1 значения для цепи хеш-сумм ввиду того, что при первом нарушении консистентности и происхождении новых событиях информация о месте нарушения связи может стать невалидной и давать неверную информацию.

1.5.1 Атомарные операции для избежания неконсистентного состояния

Операции изменения состава активных партов и пересчета хеш-сумм — сильно связанные операции. Если выполнится одна операция, но не выполнится другая, то состояние будет несогласованным. В связи с чем данные операции должны выполняться атомарно: либо обе выполниться, либо обе не выполниться. В СУБД атомарность [36] — одно из требований, предъявляемых к транзакциям. Одно из решений, позволяющих удовлетворить требованию атомарности — журналирование [35]. Перед началом выполнения операций, данные операции записываются в журнал, и только после этого начинают выполняться. В случае сбоев, состояние БД сверяется с операциями, записанными в журнале, после чего операции либо доделываются до конца транзакции, либо откатываются.

Таким образом атомарность операции будет обеспечиваться за счет следующей последовательности шагов:

1) запись новых значений хеш-сумм в журнал;
2) изменение состава активных кусков с помощью переименовывания;
3) запись новых значений хеш-сумм из журнала в файл.

Данная схема допускает сбой на любом шаге:

1) При сбое на 1 шаге, обе операции будут считаться невыполненными.
2) При сбое на 2 шаге, обе операции все так же будут считаться невыполненными, значение из журнала не используется, потому что значение из файла будет соответствовать составу активных кусков.
3) При сбое на 3 шаге обе операции будут считаться выполненными: произойдет несовпадение хеш-суммы из файла и хеш-суммы активных кусков, однако нужное значение хеш-суммы для файла будет получено из журнала и записано в файл.

1.6 Выводы

В данном разделе были классифицированы неправомерные действия, методы защиты от них, были рассмотрены существующие решения в области хранения данных с защитой от неправомерного доступа, выделены основные элементы, позволяющие обеспечить определенные методы защиты. Также была рассмотрена СУБД ClickHouse, в частности движок MergeTree: была проанализирована структура хранения данных, порядок записи, слияния и изменения порций данных, называемых кусками. Был проведен анализ защищенности от неправомерного доступа в движке MergeTree и предложено улучшение текущего метода хранения данных для обеспечения возможности доказательства неправомерного удаления данных.

2 Конструкторская часть

В данном разделе будут рассмотрены возможные сценарии неправомерных действий и признаки, по которым данные неправомерные действия могут быть обнаружены, будет рассмотрен алгоритм обновления хеш-сумм для связи всех кусков.

2.1 Сценарии неправомерных действий

На рисунке 12 изображены сценарии возможного неправомерного доступа. Здесь рассматриваются только действия с данными, без метаданных.

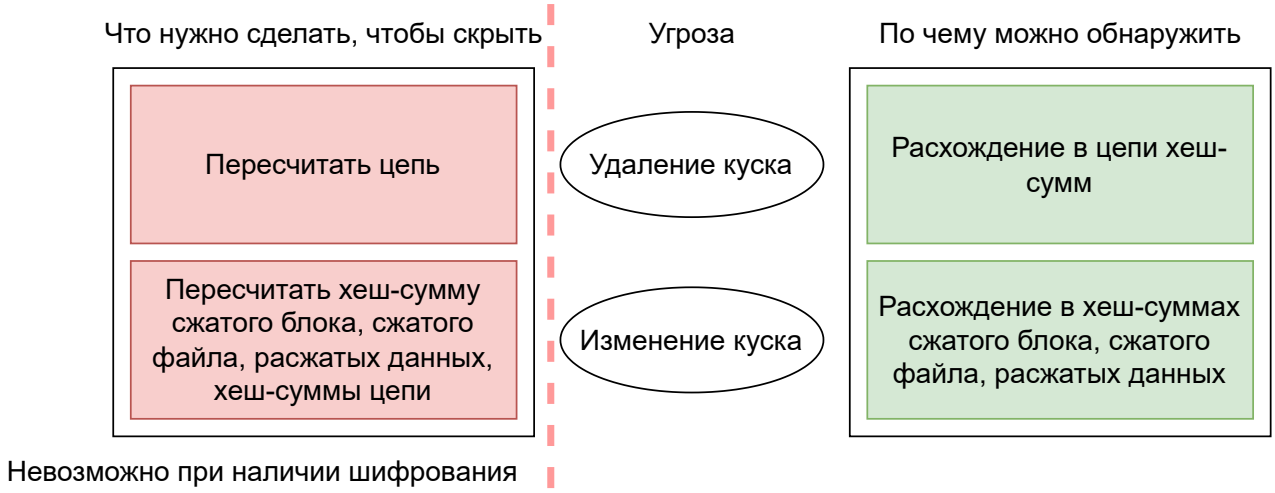

Рисунок 12 – Сценарии неправомерных действий.

2.2 Схемы алгоритмов работы метода

На рисунках 13 и 14 показана схема нового алгоритма изменения состояний кусков. Серым отмечены блоки, которые присутствовали в старом алгоритме. Во внутренней структуре состояния кусков меняются следующим образом:

— кусок, который переименовывается из временного в обычный, переходит из временного состояния в активное;

— куски, которые покрываются переименовываемым куском, переводятся из активного состояния в истекшее и не используются при чтении.

На рисунке 15 показана схема алгоритма расчета цепи хеш-сумм.

На рисунке 16 показана схема алгоритма проверки цепи хеш-сумм.

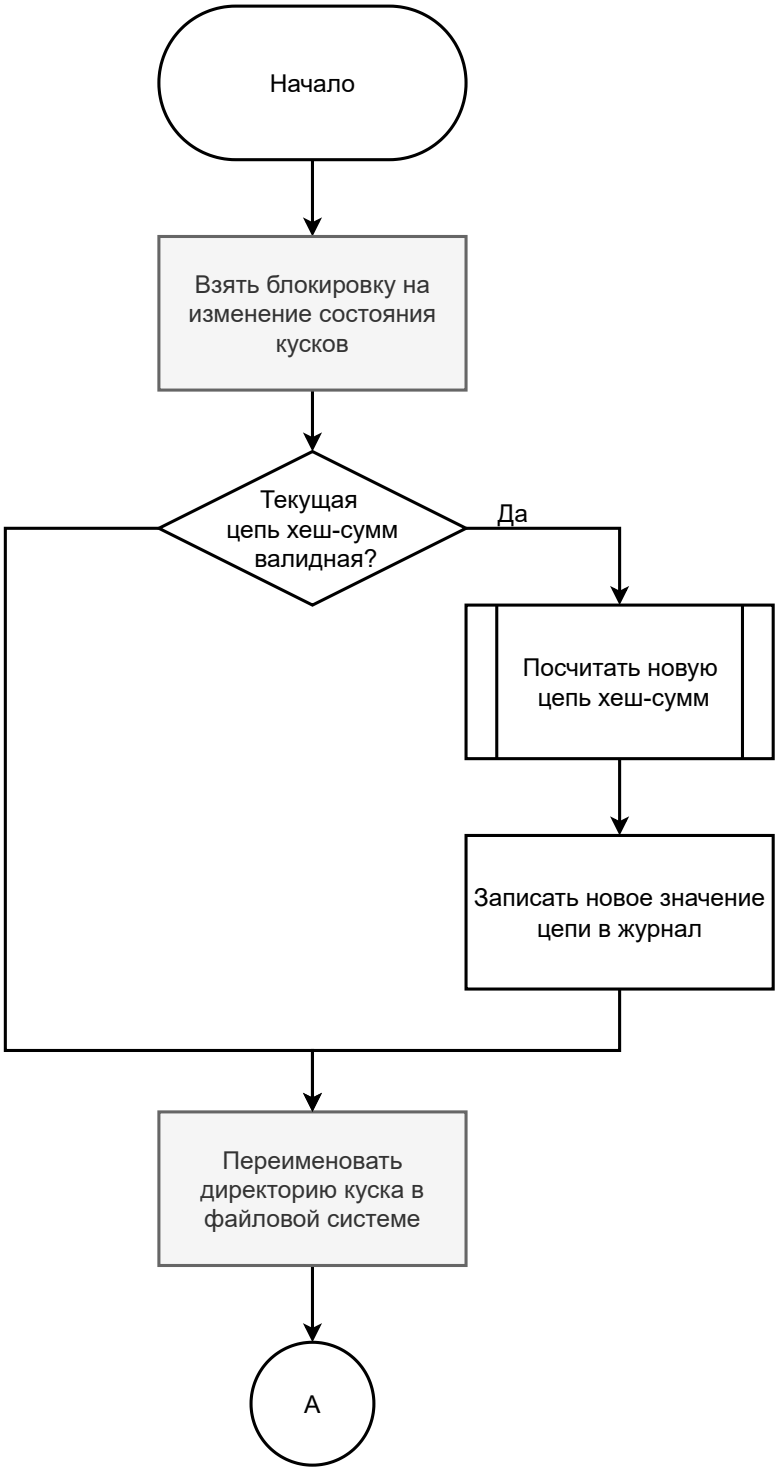

Рисунок 13 – Схема нового алгоритма изменения состояний кусков.

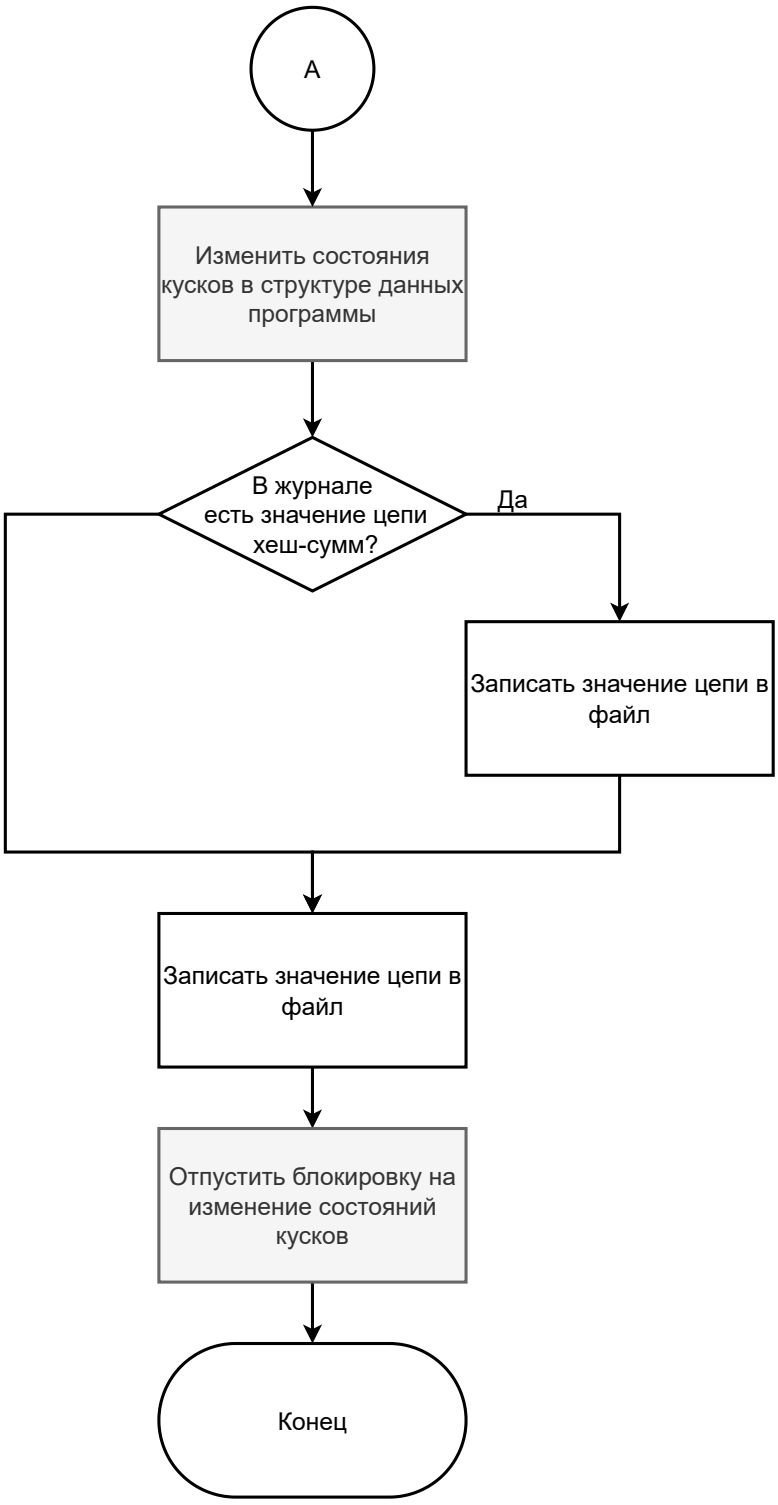

Рисунок 14 – Схема нового алгоритма изменения состояний кусков. Продолжение.

Рисунок 15 – Схема алгоритма расчета цепи хеш-сумм.

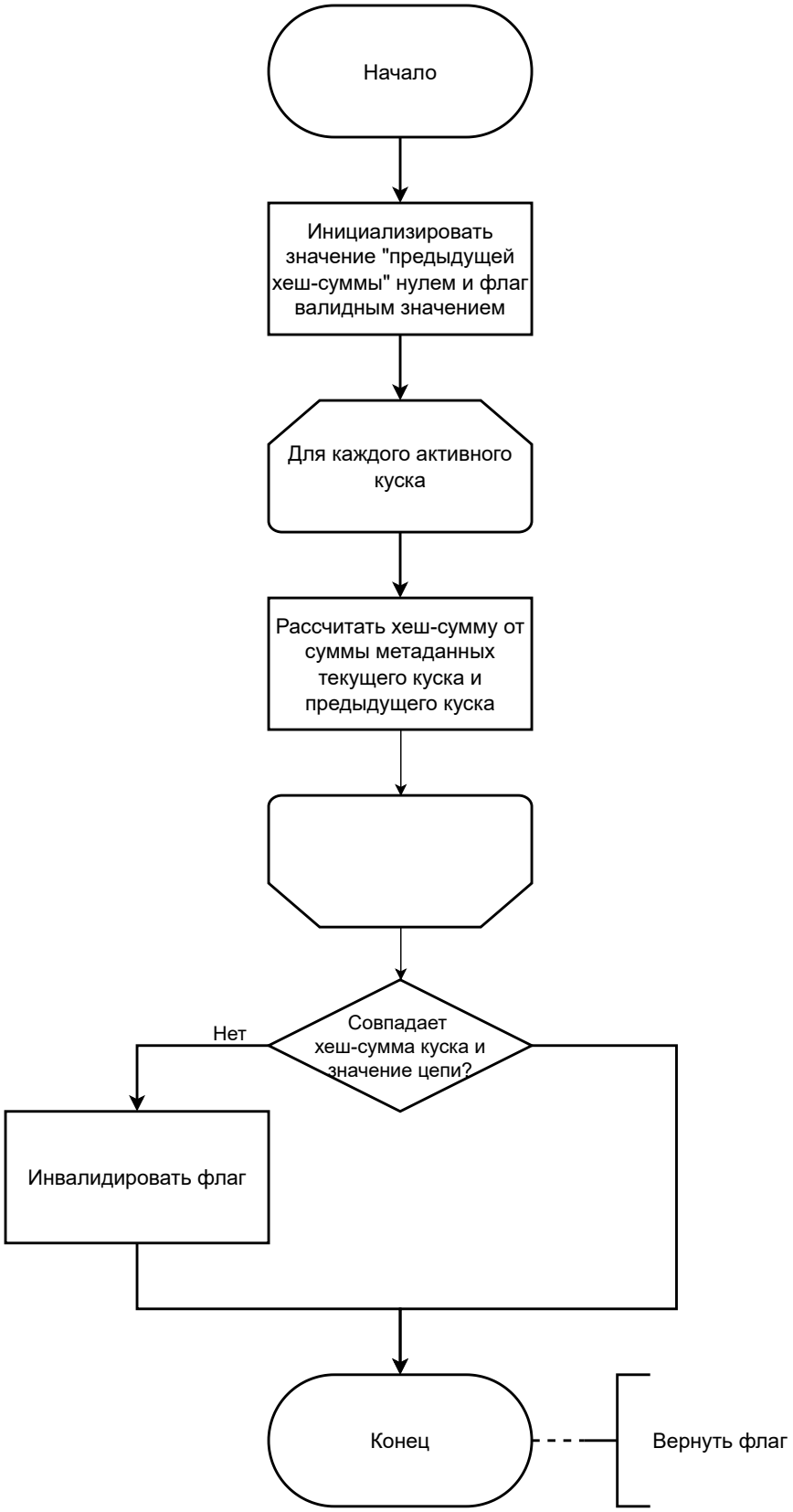

Рисунок 16 – Схема алгоритма проверки валидности цепи хеш-сумм.

2.3 Выводы

В данном разделе были рассмотрены возможные сценарии неправомерного доступа, а также построены схемы алгоритмов, необходимых для реализации разрабатываемого метода.

3 Технологическая часть

В данном разделе рассматриваются используемые инструменты и технологии, диаграмма классов для реализуемого метода, а также сама реализация.

3.1 Выбор инструментов и технологий

В качестве языка программирования был выбран C++ [37] в связи с тем, что на этом языке программирования написана СУБД ClickHouse, для которой данный метод реализуется.

В качестве компилятора был выбран компилятор Clang [39] 12 версии в связи с тем, что на официальном сайте ClickHouse рекомендуется использовать Clang выше 11 версии [38].

Для сборки проекта используются утилиты CMake [40] и Ninja [41], в проекте уже присутствуют требуемые конфигурационные файлы и сборка полностью автоматизирована.

3.2 Диаграмма классов реализуемого метода

На рисунке 17 представлена диаграмма классов реализуемого метода.

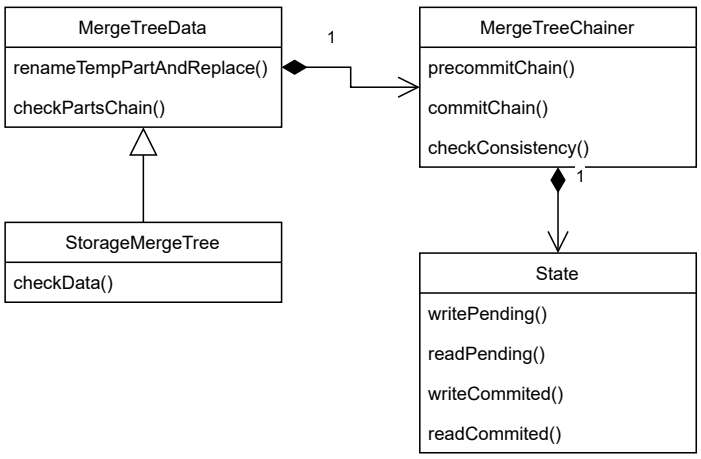

Рисунок 17 – Диаграмма классов реализуемого метода.

3.3 Код объекта, отвечающего за построение цепи хеш-сумм

Для проверки и расчета цепи хеш-сумм используется класс `MergeTreeDataChainer`. Его интерфейс представлен в листинге 1. Публичный интерфейс состоит из следующих методов:

1) `precommitChain` — проверяет, что текущая цепь хеш-сумм валидна и записывает в журнал значение новой. В качестве аргументов получает текущую цепь, элемент, который добавляется в цепь, вектор элементов, которые удаляются из цепи, а также блокировку на использование кусков. Возвращает булево значение, было ли записано значение в журнал (не записывается в случае неверной цепочки на момент вызова метода).
2) `checkConsistency` — проверяет цепь. На вход получает набор кусков, а также блокировку на использование кусков.
3) `commitChain` — фиксирует значение цепи хеш-сумм, перенося его из журнала в файл цепи. На вход получает блокировку на использование кусков.

Листинг 1: Класс MergeTreeDataChainer.

```
1  class MergeTreeDataChainer
2  {
3      using Checksum = UInt64;
4
5  public:
6      MergeTreeDataChainer(DiskPtr disk, String relative_storage_path,
       Poco::Logger * log = nullptr);
7      bool precommitChain(DataParts & data_parts, const DataPartPtr &
       part_to_add,
8          const DataPartsVector & parts_to_remove, const DataPartsLock &
       lock);
9      CheckResult checkConsistency(const DataParts & data_parts, const
       DataPartsLock & /*lock*/);
10     void setIgnoreOnInconsistency(const bool new_value) {
       ignore_on_inconsistency = new_value; }
11     void setThrowOnInconsistency(const bool new_value) {
       throw_on_inconsistency = new_value; }
12     void commitChain(const DataPartsLock & /*lock*/);
13
14 private:
15     Checksum calculateChain(const DataParts & data_parts);
16     static void transformToFutureState(DataParts & data_parts, const
       DataPartPtr & part_to_add,
17         const DataPartsVector & parts_to_remove);
18     void precommitChain(const DataParts & data_parts, const
       DataPartsLock & /*lock*/);
19     void updateFromOnePart(SipHash & hash, const DataPart & data_parts
       );
20
21     State state;
22     Poco::Logger * log;
23     bool throw_on_inconsistency = false;
24     bool ignore_on_inconsistency = false;
25 };
```

Метод, отвечающий за проверки корректности цепи хеш-сумм, представлен в листинге 2. Данный код сначала проверяет соответствие цепочки зафиксированному в файле значению. Если обнаруживается несовпадение, то результат сравнивается с значением из журнала. Если не совпадает со значением из журнала, то возвращается ошибка.

Листинг 2: Валидация цепи хеш-сумм.

```
CheckResult MergeTreeData::MergeTreeDataChainer::checkConsistency(
    const DataParts & data_parts, const DataPartsLock & /*lock*/) {
     if (data_parts.size() == 0) {
         const auto pending = state.readPending(), commited = state.
    readCommited();
         return pending == 0 && commited == 0
             ? CheckResult("parts_chain", true, "")
             : CheckResult("parts_chain", false, "no parts, but not
    empty checksums: " + std::to_string(pending) + " and " + std::
    to_string(commited));
     }
     const auto checksum = calculateChain(data_parts);
     const auto commited = state.readCommited();
     if (commited == checksum)
         return CheckResult("parts chain", true, "");

     const auto pending = state.readPending();
     if (pending == checksum) {
         if (log)
             log->warning("parts chain is verified with pending value,
    rewriting commited");
         state.writeCommited(checksum);
         return CheckResult("parts chain", true, "");
     }
     return CheckResult("parts chain", false, "Parts chain is not valid
    !");
}
```

Сравнение цепей хеш-сумм сводится к сравнению 2 векторов. Код, рассчитывающий значение цепи хеш-сумм, представлен в листинге 3. Цепь представляется вектором хешей, где хеш — число размером 8 байт. Для расчета хешей используется хеш-функция `SipHash`. Для каждого очередного в качестве значения используются хеш-суммы всех файлов, имеющихся в куске, а также хеш-предыдущего куска (для первого куска используется 0).

Листинг 3: Расчет цепи хеш-сумм.

```
1  MergeTreeData::MergeTreeDataChainer::Checksum
2  MergeTreeData::MergeTreeDataChainer::calculateChain(const DataParts &
       data_parts) {
3      if (data_parts.size() == 0)
4          return 0;
5      Checksum prev = 0;
6      for (const auto& part : data_parts) {
7          SipHash hash{0, 0};
8          hash.update(prev);
9          updateFromOnePart(hash, *part);
10         prev = hash.get64();
11     }
12     return prev;
13 }
14 void MergeTreeData::MergeTreeDataChainer::updateFromOnePart(SipHash &
       hash, const DataPart & data_part) {
15     for (const auto& [file, checksum] : data_part.checksums.files) {
16         hash.update(file);
17         hash.update(checksum);
18     }
19 }
```

Код метода `precommitChain` представлен в листинге 4. В данном методе происходят следующие действия:

1) проверятся консистентность цепи для текущего списка активных кусков;
2) если проверка успешна, то к текущему списку добавляется кусок на добавление, и удаляются куски, которые требуется удалить;
3) для нового состояние активных блоков высчитывается цепочка и записывается в журнал.

Листинг 4: Метод `precommitChain`.

```
1  bool MergeTreeData::MergeTreeDataChainer::precommitChain(DataParts &
       data_parts,
2       const DataPartPtr & part_to_add, const DataPartsVector &
       parts_to_remove, const DataPartsLock & lock) {
3       if (!checkConsistency(data_parts, lock).success) {
4           if (log)
5               log->error("Parts chain is inconsistent (ignoring
       inconsistency: " + std::to_string(ignore_on_inconsistency) + ")");
6           if (throw_on_inconsistency)
7               throw Exception(ErrorCodes::LOGICAL_ERROR, "Parts chain is
        inconsistent and throw mode is set.");
8           if (!ignore_on_inconsistency)
9               return false;
10      }
11      transformToFutureState(data_parts, part_to_add, parts_to_remove);
12      precommitChain(data_parts, lock);
13      return true;
14 }
15 void MergeTreeData::MergeTreeDataChainer::precommitChain(const
       DataParts & data_parts, const DataPartsLock & /*lock*/) {
16      state.writePending(calculateChain(data_parts));
17 }
18 void MergeTreeData::MergeTreeDataChainer::transformToFutureState(
       DataParts & data_parts,
19      const DataPartPtr & part_to_add, const DataPartsVector &
       parts_to_remove) {
20      data_parts.insert(part_to_add);
21      for (const auto& part : parts_to_remove)
22          data_parts.erase(part);
23 }
```

3.4 Код изменения состояний активных блоков

Как было описано в аналитической части и в схемах алгоритмов, улучшение метода хранения, которое позволяет доказывать неправомерное удаление, работает в тот момент, когда происходит изменения состояний блоков. Это происходит в функции `renameTempPartAndReplace` класса `MergeTreeData`. Код с использованием цепи хеш-сумм, описан в листингах 23 и 24 в приложении А.

3.5 Выводы

В данном разделе были рассмотрены используемые инструменты и технологии, диаграмма классов для реализуемого метода, а также сама реализация.

4 Исследовательская часть

В данном разделе будет показаны действия, которые требуется сделать, чтобы обойти защиту системы без шифрования данных, а также будет показано, что система обнаруживает неправомерные действия с наличием шифрования.

4.1 Обход механизма защиты при отсутствии шифрования данных

Рассмотрение для 2 форматов хранения:

1) компактное;
2) широкое.

4.1.1 Компактное хранение

Необходимо создать таблицу и поместить в нее данные. Эти действия выполняются запросами, представленными в листинге 5.

Листинг 5: Создание таблицы с компактным хранением.

```
CREATE TABLE cats
(
    `id` UInt64 CODEC(NONE),
    `name` String CODEC(NONE),
    `age` UInt64 CODEC(NONE)
)
ENGINE = MergeTree
ORDER BY id
SETTINGS min_bytes_for_wide_part = 1000000000, min_rows_for_wide_part
   = 1000000000, use_parts_chainer = 1

INSERT INTO cats VALUES (1, 'Murka', 11), (2, 'Elsa', 18), (3, 'Cleo',
    5)
```

Как было показано в конструкторской части, при изменении какого-то блока данных требуется пересчитать 3 хеш-суммы куска и цепь хеш-сумм. Пусть требуется выставить столбцу значение $age = 4$, где $id = 3$. Данные на диске выглядят так, как показано в листинге 6.

Листинг 6: Данные записанного куска в сыром виде.

```
hexdump -C all_1_1_0/data.bin
39 32 97 f2 20 a1 0f 9e  d5 82 4c 8a 1c 64 a7 74  |92.. .....L..d.t|
02 21 00 00 00 18 00 00  00 01 00 00 00 00 00 00  |.!..............|
00 02 00 00 00 00 00 00  00 03 00 00 00 00 00 00  |................|
00 5c 43 37 3f 94 e6 65  eb 4d fb 60 68 3a 35 85  |.\C7?..e.M.`h:5.|
4d 02 19 00 00 00 10 00  00 00 05 4d 75 72 6b 61  |M..........Murka|
04 45 6c 73 61 04 43 6c  65 6f d8 cd aa 90 8a 0a  |.Elsa.Cleo......|
4a 0a 5f c3 97 cd bd e9  58 3d 02 21 00 00 00 18  |J._.....X=.!....|
00 00 00 0b 00 00 00 00  00 00 00 12 00 00 00 00  |................|
00 00 00 05 00 00 00 00  00 00 00                 |...........|
```

Если разобрать по байтам, получится:

1) 16 байт — хеш-сумма для сжатого блока (1 столбца);
2) 9 байт — заголовок кодека, состоящий из:
   — 1 байт — метод;
   — 4 байта — размер сжатых данных (с заголовком);
   — 4 байта — размер исходных данных.
3) 24 байта — 3 числа по 8 байт — столбец `id`;
4) 16 и 9 байт аналогично на хеш-сумму и заголовок кодека для 2 столбца;
5) 16 байт — строки «Murka», «Elsa» и «Cleo» и перед ними числа размером 1 байт;
6) 16 и 9 байт аналогично на хеш-сумму и заголовок кодека для 3 столбца;
7) 24 байта — 3 числа по 8 байт — столбец `age`.

Таким образом требуется поменять:

1) 1 байт — значение числа, которое хотим поменять;
2) 16 байт — хеш-сумма столбца (блока сжатых данных).

После замены файл `data.bin` будет выглядеть как в листинге 7.

Листинг 7: Данные записанного куска в сыром виде после исправления.

```
39 32 97 f2 20 a1 0f 9e  d5 82 4c 8a 1c 64 a7 74  |92.. .....L..d.t|
02 21 00 00 00 18 00 00  00 01 00 00 00 00 00 00  |.!..............|
00 02 00 00 00 00 00 00  00 03 00 00 00 00 00 00  |................|
00 5c 43 37 3f 94 e6 65  eb 4d fb 60 68 3a 35 85  |.\C7?..e.M.`h:5.|
4d 02 19 00 00 00 10 00  00 00 05 4d 75 72 6b 61  |M..........Murka|
04 45 6c 73 61 04 43 6c  65 6f 75 37 d2 0b 34 77  |.Elsa.Cleou7..4w|
72 89 c2 7a f3 1c 62 ef  ee 97 02 21 00 00 00 18  |r..z..b....!....|
00 00 00 0b 00 00 00 00  00 00 00 12 00 00 00 00  |................|
00 00 00 04 00 00 00 00  00 00 00                 |...........|
```

При проверке состояния таблицы получается результат, описанный в листинге 8.

Листинг 8: Результат проверки.

```
CHECK TABLE cats
format Vertical

Query id: dbcbc244-10f0-491d-85e7-849377ac73e0

Row 1:

part_path: all_1_1_0
is_passed: 0
message: Checksum mismatch for file data.bin in data part
```

Была заменена хеш-сумма сжатого блока, однако не были заменены хеш-суммы сжатых и расжатых данных, описанные в `checksums.txt`, который описан в листинге 9.

Листинг 9: Хеш-суммы, описанные в файле `checksums.txt`.

```
63 68 65 63 6b 73 75 6d  73 20 66 6f 72 6d 61 74  |checksums format|
20 76 65 72 73 69 6f 6e  3a 20 34 0a 67 cf 30 a7  | version: 4.g.0.|
25 ce b0 4e 0e 75 28 87  2a 91 be 42 82 8e 00 00  |%..N.u(.*..B....|
00 84 00 00 00 f1 3c 04  09 63 6f 75 6e 74 2e 74  |......<..count.t|
78 74 01 96 fc d9 4b 53  ab 22 6f 33 9a be c4 ce  |xt....KS."o3....|
28 da 5d 00 08 64 61 74  61 2e 62 69 6e 8b 01 5c  |(.]..data.bin..\|
70 1c a6 71 6a e6 e7 f3  d7 70 27 28 d3 99 12 01  |p..qj....p'(....|
40 24 e8 86 e7 dd 22 a9  de f5 3b 07 c6 0f 91 6d  |@$....".....;....m|
7e 09 2d 00 f0 25 6d 72  6b 33 70 eb db 30 3d 91  |~.-..%mrk3p..0=.|
74 86 b7 33 12 22 5a 25  89 00 cb 00 0b 70 72 69  |t..3."Z%.....pri|
6d 61 72 79 2e 69 64 78  10 f6 15 da b1 a5 50 1e  |mary.idx......P.|
18 54 ae 68 d7 22 30 aa  ee 00                    |.T.h."0...|
```

Хеш-суммы подсчитываются для файла с данными, файла с индексами, файла с количеством элементов в куске и файла с засечками.

Данные хранятся в сжатом виде, жмутся кодеком по умолчанию (LZ4). Порядок записи следующий:

— строка «checksums format version: 4»;
— хеш-сумма сжимающего буфера (16 байт);
— заголовок кодека (9 байт);
— количество файлов (как число переменной длины, в данном случае 1);
— для каждого файла:
    — имя файла;
    — размер файла;
    — хеш-сумма файла;
    — булевый признак сжатия;
    — размер расжатых данных (если булевый признак сжатия — истина);
    — хеш-сумма расжатых данных (если булевый признак сжатия — истина).

С учетом того, что менялся только `data.bin`, требуется поменять хеш-сумму только для него. Содержимое файла после изменений представлено в листинге 10. Хеш-сумма не была сжата, поэтому размер файла не изменился.

Листинг 10: Хеш-суммы, описанные в файле `checksums.txt` после внесения изменений.

```
 1 63 68 65 63 6b 73 75 6d  73 20 66 6f 72 6d 61 74  |checksums format|
 2 20 76 65 72 73 69 6f 6e  3a 20 34 0a 67 cf 30 a7  | version: 4.g.0.|
 3 25 ce b0 4e 0e 75 28 87  2a 91 be 42 82 8e 00 00  |%..N.u(.*..B....|
 4 00 84 00 00 00 f1 3c 04  09 63 6f 75 6e 74 2e 74  |......<..count.t|
 5 78 74 01 96 fc d9 4b 53  ab 22 6f 33 9a be c4 ce  |xt....KS."o3....|
 6 28 da 5d 00 08 64 61 74  61 2e 62 69 6e 8b 01 5c  |(.]..data.bin..\|
 7 70 1c a6 71 6a e6 e7 f3  d7 70 27 28 d3 99 12 01  |p..qj....p'(....|
 8 40 24 e8 86 e7 dd 22 a9  de f5 3b 07 c6 0f 91 6d  |@$....".../....m|
 9 7e 09 2d 00 f0 25 6d 72  6b 33 70 eb db 30 3d 91  |~.-..%mrk3p..0=.|
10 74 86 b7 33 12 22 5a 25  89 00 cb 00 0b 70 72 69  |t..3."Z%.....pri|
11 6d 61 72 79 2e 69 64 78  10 f6 15 da b1 a5 50 1e  |mary.idx......P.|
12 18 54 ae 68 d7 22 30 aa  ee 00                    |.T.h."0...|
```

Вывод проверки после проведенных изменений показан в листинге 11.

Листинг 11: Результат проверки после исправления `checksums.txt`

```
CHECK TABLE cats
FORMAT Vertical

Query id: dbcbc244-10f0-491d-85e7-849377ac73e0

Row 1:

part_path: all_1_1_0
is_passed: 1
message:

Row 2:

part_path: parts chain
is_passed: 0
message:   Parts chain is not valid!
```

Проверка показывает, что теперь нарушена цепь хеш-сумм блоков. Полный алгоритм и его реализация построения цепи хеш-сумм и код алгоритма описаны в конструкторском и технологическом разделах. Файл со значением цепи хеш-сумм содержит в себе число размером 8 байт. После пересчета хеш-суммы и замены значения в файле `commited_parts_chain.bin` вывод не поменяется, так как данные о значениях цепи хеш-сумм и отдельных хеш-сумм кэшируются в оперативной памяти. Данный факт вынуждает злоумышленника сделать так, чтобы СУБД перезапустилась. При перезапуске сервера значения цепи хеш-сумм поменяются на нужные и получится вывод, показанный в листинге 12.

Листинг 12: Результат проверки после исправления `commited_parts_chain.bin`.

```
CHECK TABLE cats
FORMAT Vertical

Query id: 95e1ae1c-203e-488b-a0a2-f3885ab4aff1

Row 1:

part_path: all_1_1_0
is_passed: 1
message:

Row 2:

part_path: parts chain
is_passed: 1
message:
```

Как итог, чтобы проверка куска проходила успешно, требуется:

1) пересчитать значение хеш-суммы сжатого блока, в который было внесено изменение;
2) пересчитать значение хеш-суммы файла `data.bin` в файле `checksums.txt`;
3) пересчитать значение цепи хеш-сумм в файле `commited_parts_chain.bin`.

При удалении потребуется сделать только последний шаг. Аналогично потребуется поступить при замене куска (то есть при удалении и вставке куска с аналогичной структурой). Подобные сценарии неправомерного доступа проходили бы при старом методе хранения данных.

4.1.2 Широкое хранение

Для широкого хранения следует рассмотреть отличия от приведенного выше алгоритма действий:

1) изменение сжатого блока и его хеш-суммы будет происходит в соответсвующем нужному столбцу файле;
2) изменение в файле `checksums.txt` будет происходить для файла, соответствующего меняемому столбцу.

4.2 Работа системы с шифрованием данных

Шифрование можно настроить на нескольких уровнях:

— на уровне отдельных столбцов;

— на уровне виртуального диска (пути в файловой системе, который будет передан ClickHouse через конфигурационный файл).

Шифрование на уровне столбцов позволяет злоумышленнику менять файлы с индексами, а также файлы с засечками, что в совокупности позволяет сделать так, чтобы при выборке данных с условием на первичный ключ данные какого-либо куска не читались вовсе, что аналогично их удалению. Однако на запросы, которые не имеют ограничений на первичный ключ, данный подход не подействует.

Пусть имеется шифрование на уровне виртуального диска. Пример конфигурации для такого шифрования приведен в листинге 13.

Листинг 13: Параметры конфигурации для шифрования на уровне диска.

```
<storage_configuration>
    <disks>
        <disk1>
            <type>local</type>
            <path>/ClickHouse/build/programs/disk/</path>
        </disk1>
        <default>
            <type>encrypted</type>
            <disk>disk1</disk>
            <path>enc/</path>
            <algorithm>AES_256_CTR</algorithm>
            <key>1234000000000001234000000000000</key>
        </default>
    </disks>
</storage_configuration>
```

Тогда в случае повторения запросов из листинга 5, данные, записанные в файл, будут выглядеть так, как показано в листинге 14.

Листинг 14: Данные при шифровании диска.

```
45 4e 43 01 00 02 00 00  00 00 00 00 00 00 00 05  |ENC.............|
31 8d 4f 11 c1 c1 3b 78  ae 20 b6 59 7b ff 67 f4  |1.O...;x. .Y{.g.|
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
f9 82 0a 75 5f 80 01 de  36 e4 22 e9 78 b1 5a 8e  |...u_...6.".x.Z.|
8a 24 e2 2b 2b 57 5b cc  71 18 96 ea af ce 21 e8  |.$.++W[.q.....!.|
e7 20 5c 8b 26 20 98 8d  e9 34 17 cb a9 82 72 43  |. \.& ...4....rC|
ba fc b6 3b f6 f2 78 a2  89 a0 a2 80 b5 4e e5 3b  |...;..x......N.;|
b3 30 f9 e5 6b 20 f5 c0  ef 77 a2 6b 2f ce 4e 0c  |.0..k ...w.k/.N.|
e5 31 af 8c 1c 2f 55 e1  27 9d db 70 24 bc 92 de  |.1.../U.'..p$...|
3e 8c 91 12 83 1c 90 1a  f7 2b a1 1a ce c4 56 c3  |>........+....V.|
a7 e0 62 f8 92 d7 c1 e5  23 eb 64 df 79 5f 15 3d  |..b.....#.d.y_.=|
02 b4 c0 e6 de 12 f0 b2  a4 f9 7e                 |..........~|
```

Шифрование не позволяет исправить хеш-суммы подобно тому, как это выполнялось без него. Требуется проверить:

— что работают все стандартные операции:
  — вставка;
  — слияние;
  — мутация.
— что неправомерные удаление и изменение блока обнаружаются.

4.2.1 Проверка работы стандартных операций

Для проверки корректности работы цепи хеш-сумм со вставкой можно сделать 3 вставки командой из листинга 5. Результат проверки таблицы в листинге 15.

Листинг 15: Результат проверки после вставки 4 блоков.

```
CHECK TABLE cats
FORMAT Vertical

Query id: 70296b3b-a0ec-4938-b731-a47310bc32b8

Row 1:

part_path: all_1_1_0
is_passed: 1
message:

Row 2:

part_path: all_2_2_0
is_passed: 1
message:

Row 3:

part_path: all_3_3_0
is_passed: 1
message:

Row 4:

part_path: all_4_4_0
is_passed: 1
message:

Row 5:

part_path: parts chain
is_passed: 1
message:
```

Для проверки слияния можно сделать еще 3 вставки. Результат проверки в листинге 16. Из проверки видно, что произошло слияние блоков с 1 по 5, цепь осталась валидной.

Листинг 16: Результат проверки после вставки 7 блоков.

```
CHECK TABLE cats
FORMAT Vertical

Query id: c0888295-5a25-4528-8a96-7e8ba7762d00

Row 1:

part_path: all_1_5_1
is_passed: 1
message:

Row 2:

part_path: all_6_6_0
is_passed: 1
message:

Row 3:

part_path: all_7_7_0
is_passed: 1
message:

Row 4:

part_path: parts chain
is_passed: 1
message:
```

Для проверки работы мутаций следует выполнить запрос из листинга 17. Результат проверки таблицы после проведения мутации представлен в листинге 18, цепь хеш-сумм осталась валидной.

Листинг 17: Запрос мутации.

```
ALTER TABLE cats UPDATE age=age + 1 WHERE age < 17;
```

Листинг 18: Проверка таблицы после мутации.

```
CHECK TABLE cats
FORMAT Vertical

Query id: aed93f24-7417-4424-8d16-cac981c1d240

Row 1:

part_path: all_1_5_1_8
is_passed: 1
message:

Row 2:

part_path: all_6_6_0_8
is_passed: 1
message:

Row 3:

part_path: all_7_7_0_8
is_passed: 1
message:

Row 4:

part_path: parts chain
is_passed: 1
message:
```

4.2.2 Проверка обнаружения неправомерных удаления и изменения кусков

Для проверки обнаружения неправомерного удаления можно удалить кусок `all_7_7_0_8`. Вывод проверки представлен в листинге 19, вывод проверки после перезагрузки системы представлен в листинге 20. В обоих случаях показано, что данные были изменены где-то вне системы.

Листинг 19: Вывод системы после неправомерного удаления куска.

```
CHECK TABLE cats
FORMAT Vertical

Query id: 6aa87471-30e7-4c8c-8c79-e3243ee936ae

Row 1:

part_path: all_1_5_1_8
is_passed: 1
message:

Row 2:

part_path: all_6_6_0_8
is_passed: 1
message:

Row 3:

part_path: all_7_7_0_8
is_passed: 0
message:   Check of part finished with error: 'Cannot open file /
   ClickHouse/build/programs/disk/enc/store/cf0/cf01ce53-cba4-4afb-8
   e11-26b487355f84/all_7_7_0_8/columns.txt, errno: 2, strerror: No
   such file or directory'
```

Листинг 20: Вывод системы после неправомерного удаления куска после перезагрузки системы.

```
CHECK TABLE cats
FORMAT Vertical

Query id: 362f64b5-a480-4f2a-b1ad-0c021a110c53

Row 1:

part_path: all_1_5_1_8
is_passed: 1
message:

Row 2:

part_path: all_6_6_0_8
is_passed: 1
message:

Row 3:

part_path: parts chain
is_passed: 0
message:   Parts chain is not valid!
```

Для проверки обнаружения неправомерного изменения можно изменить последовательность байт в зашифрованном файле с данными одного из кусков, например, `all_7_7_0_8`. После изменения данных командой из листинга 21 получается результат проверки из листинга 22. Данное было бы обнаружено и до оптимизации метода, потому что, как было замечено в констуркторской части, при изменении данных изменяются еще и хеш-суммы файлов и сжатых блоков, которые были и до модификации метода.

Листинг 21: Неправомерное изменение данных.

```
printf '\x31' | dd of=cats/all_7_7_0_8/data.bin bs=1 seek=183 count=1
   conv=notrunc
```

Листинг 22: Результат проверки после неправомерного изменения данных.

```
CHECK TABLE cats
FORMAT Vertical

Query id: 3b86fbe1-b13e-4006-8e92-0d460a52784f

Row 1:

part_path: all_1_5_1_8
is_passed: 1
message:

Row 2:

part_path: all_6_6_0_8
is_passed: 1
message:

Row 3:

part_path: all_7_7_0_8
is_passed: 0
message:   Parts chain is not valid!
```

Вывод

В данном разделе были рассмотрены способы обхода защиты системы, работающей без шифрования, было показано правильное функционирование с шифрованием и применением стандартных операций, а также проверена возможность обнаружения неправомерных действий.

ЗАКЛЮЧЕНИЕ

В рамках данной работы:

— были классифицированы виды защиты информации при блочном хранении данных;

— были рассмотрены основные понятия и структуры, используемые при проектировании методов хранения информации с возможностью защиты от неправомерного доступа;

— был проведен анализ существующих методов блочного хранения данных с защитой от неправомерного доступа;

— были выявлены зависимости между свойствами системы и видами защиты информации при блочном хранении данных;

— была рассмотрена архитектура СУБД ClickHouse и структура хранения данных в движке MergeTree;

— был проведен анализ блочного хранения данных в СУБД ClickHouse в движке MergeTree на предмет возможности защиты от неправомерного доступа, в частности — возможности доказательства неправомерного доступа;

— был спроектирован новый метод блочного хранения данных на основе имеющегося, предоставляющий возможность доказательства неправомерного доступа;

— спроектированный метод был реализован, протестирован на предмет обхода защиты при различных конфигурациях системы, проверен на правильность при наличии шифрования в системе.

Таким образом цель работы — разработать метод хранения данных с возможностью доказательства неправомерного доступа на основе хеш-сумм была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Брюс Шнайер. «Прикладная криптография. 2-е издание. Протоколы, алгоритмы и исходные тексты на языке С».

2. ГОСТ. Информационная технология. Криптографическая защита информации. Функция хеширования. [Электронный ресурс]. – Режим доступа: https://protect.gost.ru/v.aspx?control=8id=172313 свободный – (13.10.2021).

3. Движки таблиц СУБД ClickHouse [Электронный ресурс]. – Режим доступа: https://clickhouse.com/docs/ru/engines/table-engines/ свободный – (23.09.2021).

4. Справочное руководство MySQL [Электронный ресурс]. – Режим доступа: https://dev.mysql.com/doc/refman/8.0/en/storage-engines.html свободный – (26.10.2021).

5. Мао В. «Современная криптография: Теория и практика» — М.: Вильямс, 2005. — 768 с.

6. Когаловский М.Р. «Энциклопедия технологий баз данных.» — М.: Финансы и статистика, 2002. — 800 с.

7. Bitcoin: A Peer-to-Peer Electronic Cash System [Электронный ресурс]. – Режим доступа: https://bitcoin.org/bitcoin.pdf свободный – (15.09.2021).

8. R.C. Merkle, ”Protocols for public key cryptosystems,”In Proc. 1980 Symposium on Security and Privacy, IEEE Computer Society, pages 122-133, April 1980.

9. Merkle Directed Acyclic Graphs (DAGs) [Электронный ресурс]. – Режим доступа: https://docs.ipfs.io/concepts/merkle-dag/ свободный – (17.09.2021).

10. Дистель, Рейнхард (2005), ”Graph Theory (3rd ed.)”, Berlin, New York: Springer-Verlag.

11. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Топологическая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 220—224. — 576 с.

12. Git Internals — Git Objects [Электронный ресурс]. – Режим доступа: https://git-scm.com/book/en/v2/Git-Internals-Git-Objects свободный – (28.09.2021).

13. G. Goodson, J. Wylie, G. Ganger, and M. Reiter, ”Efficient Byzantine-tolerant Erasure-coded Storage,” Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004). Florence, Italy, 2004. Supercedes Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-03-104, December 2003

14. Dave K. Kythe, Prem K. Kythe. Algebraic and Stochastic Coding Theory. — 1-е изд. — CRC Press, 2012. — С. 375—395. — 512 с.

15. G.Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano, ”A Transparent Cryptographic File System for Unix,” In Proceedings of the USENIX Annual Technical Conference, Freenix Track. Boston, MA, 2001.

16. C. Wright, M. Martino, and E. Zadok. “NcryptFS: A Secure and Convenient Cryptographic File System,” In Proceedings of the USENIX Conference General Track, San Antonio, TX, June 2003.

17. Уорд Б. Внутреннее устройство Linux. — СПб.: Питер, 2016. — 384 с.

18. J. Kubiatowitz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, ”Oceanstore: An Architecture for Global-Scale Persistent Storage,” In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, Nov 2000.

19. H. Weatherspoon, C. Wells, and J. Kubiatowicz, “Naming and Integrity: Self-Verifying Data in Peer-to-Peer Systems,” In Proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo 2002), June 2002.

20. Полянская О. Ю., Горбатов В. С. Инфраструктуры открытых ключей. Учебное пособие., Москва, 2007.

21. СУБД ClickHouse [Электронный ресурс]. – Режим доступа: https://clickhouse.com/ свободный – (03.09.2021).

22. MergeTree движок СУБД ClickHouse [Электронный ресурс]. – Режим доступа: https://clickhouse.com/docs/ru/engines/table-engines/mergetree-family/mergetree/ свободный – (05.09.2021).

23. Архитектура СУБД ClickHouse. Движок MergeTree. [Электронный ресурс]. – Режим доступа: https://clickhouse.com/docs/ru/engines/table-engines/mergetree-family/mergetree/ свободный – (05.09.2021).

24. Компактный вид хранения данных куска. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83
свободный – (23.02.2022).

25. Широкий вид хранения данных куска. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d1
свободный – (23.02.2022).

26. Хеширующий буфер. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83
свободный – (23.02.2022).

27. Сжимающий буфер. [Электронный ресурс]. – Режим доступа:

https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83
свободный – (23.02.2022).

28. Файловый буфер. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83 свободный – (23.02.2022).

29. Запись временного куска при вставке. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d1 свободный – (23.02.2022).

30. Переименование куска при вставке. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144 свободный – (23.02.2022).

31. Метод планирования фоновых задач. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d1 свободный – (23.02.2022).

32. Структура представления куска. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83 свободный – (23.02.2022).

33. Покрываемые куски. [Электронный ресурс]. – Режим доступа: https://github.com/ClickHouse/ClickHouse/blob/009e71e273238e968be5f5d3d144d83 свободный – (23.02.2022).

34. Transitioning the Use of Cryptographic Algorithms and Key Lengths [Электронный ресурс]. – Режим доступа: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf свободный – (23.02.2022).

35. К. Дж. Дейт «Введение в системы баз данных» — 8-е изд. — М.: «Вильямс», 2006. — С. 1328.

36. Amsterdam, Jonathan. «Atomic File Transactions, Part 1». O'Reilly. 2016.

37. Руководство по языку C++. [Электронный ресурс]. – Режим доступа: https://en.cppreference.com/w/ свободный – (21.03.2022).

38. Выбор компилятора для сборки ClickHouse. [Электронный ресурс]. – Режим доступа: https://clickhouse.com/docs/en/development/developer-instruction/c-compiler свободный – (24.03.2022).

39. Официальный сайт Clang. [Электронный ресурс]. – Режим доступа: https://clang.llvm.org/ свободный – (24.03.2022).

40. Официальный сайт системы сборки CMake. [Электронный ресурс]. – Режим доступа: https://cmake.org/ свободный – (27.03.2022).

41. Официальный сайт системы сборки Ninja. [Электронный ресурс]. – Режим доступа: https://ninja-build.org/ свободный – (27.03.2022).

ПРИЛОЖЕНИЕ А

Листинг 23: Изменение состояний блоков (переименование блока).

```
bool MergeTreeData::renameTempPartAndReplace(
    MutableDataPartPtr &part, MergeTreeTransaction *txn,
    SimpleIncrement *increment, Transaction *out_transaction,
    std::unique_lock<std::mutex> &lock, DataPartsVector *
   out_covered_parts,
    MergeTreeDeduplicationLog *deduplication_log,
    std::string_view deduplication_token)
{
    // ...
    part->name = part_name;
    part->info = part_info;
    part->is_temp = false;
    part->setState(DataPartState::PreActive);

    if (parts_chainer) {
        DataParts active_parts;
        auto range = getDataPartsStateRange(DataPartState::Active);
        active_parts.insert(range.begin(), range.end());
        auto need_commit_chain =
                parts_chainer->precommitChain(active_parts, part,
   covered_parts, lock);

        part->renameTo(part_name, true);

        if (need_commit_chain)
            parts_chainer->commitChain(lock);
    } else {
        part->renameTo(part_name, true);
    }

    auto part_it = data_parts_indexes.insert(part).first;
    MergeTreeTransaction::addNewPartAndRemoveCovered(shared_from_this
   (), part,
        covered_parts, txn);
```

Листинг 24: Изменение состояний блоков (переименование блока). Продолжение.

```
    if (out_transaction) {
        out_transaction->precommitted_parts.insert(part);
    } else {
        size_t reduce_bytes = 0;
        size_t reduce_rows = 0;
        size_t reduce_parts = 0;
        auto current_time = time(nullptr);
        for (const DataPartPtr &covered_part : covered_parts) {
            covered_part->remove_time.store(current_time, std::
    memory_order_relaxed);
            modifyPartState(covered_part, DataPartState::Outdated);
            removePartContributionToColumnAndSecondaryIndexSizes(
    covered_part);
            reduce_bytes += covered_part->getBytesOnDisk();
            reduce_rows += covered_part->rows_count;
            ++reduce_parts;
        }

        modifyPartState(part_it, DataPartState::Active);
        addPartContributionToColumnAndSecondaryIndexSizes(part);
        // ...
    }
}
```