Аннотация

В современном мире много информации. И при разработке любого приложения рано или поздно возникает вопрос, как и где хранить информацию пользователей и приложения. Здесь на помощь приходит огромное количество уже существующих баз данных. И зачастую данных бывает настолько много, что неоправданно хранить их на одном компьютере. Тогда используется распределенная база данных. К современным распределенным базам данных предъявляется ряд требований. Одно из таких требований — согласованность данных, а также изолированность транзакций.

Поэтому тестирование согласованности и изолированности — это актуальный вопрос для разработчиков распределенных систем. Один из инструментов для проверки гарантий согласованности, который практически не имеет аналогов, это инструмент хаос-тестирования Jepsen.

В этой работе изучен инструмент хаос-тестирования Jepsen. Также рассмотрены различные феномены, нарушающие гарантии согласованности и изолированности, и которые вспомогательный инструмент Jepsen Elle может находить.

Кроме того, проанализирована база данных Azure Cosmos DB с помощью Jepsen. В 15% историй наблюдается G2-item аномалия которая, хоть и не противоречит заявленной в документации изоляции моментальных снимков, также нарушает заявленное в документации «full ACID» свойство транзакций.

Оглавление

Введение

Существует спрос на инструменты проверки согласованности и изоляции транзакций, так как базы данных не обеспечивают тот уровень безопасности и надежности, на который претендуют. Они могут быть полезны по ряду причин:

- Пользователю(разработчику приложения, хранящего свои данные в базе данных) хочется научиться понимать про ту или иную базу данных, насколько она соответствует документации (это необходимо для того, чтобы выбрать базу данных, которая больше всего подходить для потребностей разработчика);
- Удобный инструмент для тестирования согласованности может существенно помочь на этапе разработки распределенных систем, возможно, стать одним из этапов CI/CD процесса;
- Jepsen проводит свои исследования независимо и в соответствии с их этической политикой. Это вносит большой вклад в сообщество, помогает в развитии и совершенствовании распределенных систем.

Большинство распределенных систем стремятся к достижению баланса между временем выполнения операций и согласованностью. Один из инструментов для проверки гарантии согласованности — это инструмент хаос-тестирования Jepsen. Хаос-тестирование — это тестирование путем внесения в систему незапланированных сбоев [5]. Наблюдая за поведением системы, можно понять, как сделать распределенную систему более надежной. Хаос тестирование это важная часть тестирования, потому что помогает выявить состояния гонки (race condition), которые сложно иначе обнаружить в процессе разработки.

2.1 Цели работы

- Получить опыт работы с выбранным инструментом проверки свойств транзакций (Jepsen);

- Изучить выполненные данным инструментом исследования различных баз данных;
- Проанализировать с помощью выбранного инструмента реальную базу данных (Azure Cosmos DB), которая еще не была проанализирована;
- Сравнить уровень согласованности, заявленный в документации, и уровень, установленный с помощью тестов.

2.2 Основные понятия

Параллельная система — это система, состоящая из независимых компонент, которые могут выполнять некоторые операции одновременно.

Распределенная система — это тип параллельных систем, который представляет собой систему с несколькими независимыми компонентами, расположенными на разных узлах в компьютерной сети. Эти узлы способны обмениваться данными, а также они координируют свои действия так, чтобы для конечного пользователя распределенная система работала как единая согласованная система. Система имеет логическое состояние, которое меняется с течением времени.

Хаос-тестирование [5] — это тестирование путем внесения в распределенную систему незапланированных сбоев.

Процесс — это логически однопоточная программа, которая способна выполнять некоторые операции.

Операция — переход из одного состояние в другое.

Атомарная операция — операция в общей области памяти, которая завершается за один шаг относительно других потоков, имеющих доступ к этой области памяти. Во время выполнения такой операции над переменной ни один поток не может наблюдать изменение наполовину завершенным. Неатомарные операции не дают такой гарантии. [6]

Параллелизм — это свойство системы, которое означает, что несколько процессов могут выполняться в одно и то же время.

Сбой — это состояние процесса, в котором тот не может вызывать никаких операций. Если операция по какой-то причине не была завершена и не имеет времени завершения, то ее следует рассматривать одновременно с каждой операцией, которая будет вызвана после. Это нарушает требования на однопоточность процесса, поэтому в этой работе будет говориться о состоянии сбоя процесса.

История — совокупность операций и их параллельной структуры. В этой работе будет рассматриваться истории с точки зрения Jepsen. То есть истории будут представлены в виде упорядоченного списка операций вызова и завершения.

Модель согласованности — набор гарантий, используемый в той или иной распределенной системе, для обеспечения согласованности данных.

Транзакция — некоторый конечный набор операций, переводящий данные из одного согласованного состояния в другое. Либо будет выполнена каждая операция из набора, либо ни одной.

ACID — основные свойства транзакций: атомарность, согласованность, изолированность и прочность.

Согласованность — свойство транзакций, которое гарантирует, что каждая успешно завершенная транзакция фиксирует результат, являющийся допустимым с точки зрения внутренних правил базы данных. Когда же какая-то транзакция пытается записать несогласованные данные, вся транзакция откатывается, транзакция завершается с ошибкой.

Изоляция — свойство транзакций, которое гарантирует, что параллельно исполняющиеся транзакции не влияют на результаты друг друга. Полную изоляцию тяжело поддерживать, поэтому в реальных данных поддерживаются различные более слабые уровни изоляции.

Happens before — A happens-before B означает, что все изменения, выполненные потоком X до момента операции A и изменения, которые повлекла эта операция, видны потоку Y в момент выполнения операции B и после выполнения этой операции. Отметим, что операцию A исполняет поток X и операцию B исполняет поток Y. [7]

Основные теоретические сведения

В этой главе будут рассмотрены основные теоретические сведения, которые будут полезны в практической части.

3.1 Базы данных

База данных — это некоторое хранилище данных. Чаще всего в данной работе будут рассматриваться распределенных базах данных, то есть таких базы данных, которые хранят некоторые части своих данных в различных физических локациях.

Основное требование к базам данных — это поддержание какого-то внутреннего консистентного состояния данных. Иными словами, согласованность и целостность, непротиворечивость данных в каждый момент времени.

Различные базы данных добиваются этого разными способами. Реляционные базы данных предоставляют механизм транзакций, который гарантирует согласованность данных. Чтобы ускорить работу с данными, базы данных используют некоторый механизм блокировок, который теоретически гарантирует консистентное состояние. Однако, это не всегда правда.

Некоторые базы данных реализуют различные модели согласованности, позволяющие регулировать гарантии, предоставляемые базой данных. Далее будет рассмотрена наиболее часто встречаемая модель согласованности, реализуемая различными базами данных, в том числе Azure Cosmos DB.

3.2 Изоляция моментальных снимков(англ. *Snapshot Isolation*)[1]

Изоляция моментальных снимков — это транзакционная модель. Нет гарантии доступности, то есть распределенная система может быть недоступна во время некоторых типов

сетевых сбоев. Некоторые или все узлы должны приостановить работу, чтобы обеспечить безопасность.

Изменения транзакции видны только этой транзакции до момента фиксации, когда все изменения становятся видимыми атомарно. Если транзакция T_1 изменила объект *x*, а другая транзакция T_2 совершила запись в *x* после начала моментального снимка T_1 и до фиксации T_1, то T_1 должна прерваться.

В отличие от сериализуемости(англ. *Serializability*), которая обеспечивает полный порядок транзакций, изоляция моментальных снимков гарантирует только частичный порядок: подоперации в одной транзакции могут чередоваться с подоперациями из других транзакций. Наиболее заметными явлениями, допускаемыми изоляцией моментальных снимков, являются перекосы записи (англ. *write skews*), которые позволяют транзакциям считывать перекрывающееся состояние, изменять непересекающиеся наборы объектов, а затем фиксировать; и аномалия транзакций только для чтения(англ. *read-only transaction anomaly*), включающая частично непересекающиеся наборы записи.

Данная модель согласованности запрещает грязную запись (англ. *Dirty Write*, P_0) и грязное чтение (англ. *Dirty Read*, P_1), но допустимы неповторяющееся чтение (англ. *Fuzzy Read*, P_2), фантомное чтение (англ. *Phantom*, P_3)[8]. Изоляция моментальных снимков не накладывается никаких ограничений в реальном времени и не требует упорядочивания процессов между транзакциями.

[9] В терминах абстрактного алгоритма можно говорить о данной модели согласованности так: каждая транзакция считывает данные из моментального снимка зафиксированных данных на момент начала транзакции, называемого ее меткой начала отсчета. Это время может быть любым до первого чтения транзакции. Транзакция никогда не блокируется при попытке чтения до тех пор, пока данные моментального снимка из его метки начала отсчета могут быть сохранены. Записи транзакции (обновления, вставки и удаления) также будут отражены в этом моментальном снимке, чтобы их можно было считать снова, если транзакция обращается к данным во второй раз. Обновления другими транзакциями, начатыми после метки начала отсчета транзакции, невидимы для транзакции.

Когда транзакция T_1 готова к фиксации, она получает метку времени фиксации, которая больше любой существующей метки начала отсчета или другой метки времени фиксации. Транзакция будет зафиксирована только в том случае, если ни одна другая транзакция T_2 с меткой времени фиксации в интервале выполнения T_1 [время начала отсчета, время фиксации] не записала данные, которые также записала T_1. В противном случае T_1 прервется. Это предотвращает потерю обновлений. Когда T_1 будет зафиксирована, эти изменения ста-

новятся видимыми для всех транзакций, метки начала отсчета которых больше, чем метка времени фиксации T_1.

В другой формулировке изоляции моментальных снимков определяется как комбинация четырех свойств [10]:

- внутренняя согласованность (англ. *internal consistency*);
- внешняя согласованность (англ. *external consistency*);
- префикс (англ. *prefix*) — здесь и далее это означает, что транзакции становятся видимыми для всех узлов в одном и том же порядке;
- отсутствие конфликта (англ. *NoConflict*) — здесь и далее это означает, что если две транзакции изменяют один и тот же объект, одна должна быть видна другой.

3.3 Граф сериализации [2] [3]

Сначала будут определены различные типы зависимостей, которые возникают между транзакциями, а затем через них определен *граф сериализации*(***DSG***). Здесь и далее T_i/T_j - транзакции.

3.3.1 Зависимость записи (англ. *Directly Write-Depends*)

В дальнейшем будет использоваться обозначение для данного типа зависимости: *ww*. Обозначение в *DSG*: $T_i \xrightarrow{ww} T_j$

Описание: T_j зависит от T_i, когда T_i устанавливает x_i, а T_j устанавливает следующую версию *x*.

3.3.2 Зависимость чтения (англ. *Directly Read-Depends*)

В дальнейшем будет использоваться обозначение для данного типа зависимости: *wr*. Обозначение в *DSG*: $T_i \xrightarrow{wr} T_j$

Описание: T_j зависит от T_i, когда выполняется одно из двух условий:

- T_i устанавливает x_i, T_j читает x_i;
- T_i фиксирует изменение, а затем T_j выполняет чтение на основе предикатов таким образом, что набор объектов, соответствующих предикату, изменяется фиксацией T_i.

Кроме того, T_i — это самая последняя транзакция, в которой было зафиксировано изменение, влияющее на соответствие T_i.

3.3.3 Анти зависимость (англ. *Directly Anti-Depends*)

В дальнейшем будет использоваться обозначение для данного типа зависимости: *rw*. Обозначение в *DSG*: $T_i \xrightarrow{rw} T_j$

Описание: T_j зависит от T_i, когда выполняется одно из двух условий:

- T_i считывает некоторую версию x_i объекта *x*, а затем T_j фиксирует следующую версию *x* в истории версий;
- T_i выполняет чтение на основе предикатов, а T_j перезаписывает это чтение(то есть фиксирует более позднюю, следующую версию объекта).

3.3.4 Граф сериализации(англ. *Direct Serialization Graph*, *DSH*)

DSG имеет один узел для каждой совершенной транзакции. Направленные ребра между этими узлами представляют зависимости чтения/записи/анти. Транзакция T_2 зависит от T_1, если в графе есть путь от T_1 до T_2.

Построение DSG начинается с добавления узлов для каждой зафиксированной транзакции. Затем добавляется ребро *wr*, *rw* или *ww* зависимости для всех пар транзакций, если выполняются условия зависимости.

3.4 Феномены

3.4.1 Грязная запись (англ. *Dirty Write*, P_0) [4]

Грязная запись происходит, когда одна транзакция перезаписывает значение, которое ранее было записано другой транзакцией, все еще находящейся в процессе исполнения.

Одна из причин, по которой грязные записи плохи, заключается в том, что они могут нарушить согласованность базы данных. Если, например, между *x* и *y* существует ограничение (например, $x = y$), и транзакции T_1 и T_2 поддерживают согласованность ограничения, если эти транзакции исполняются отдельно. Однако ограничение может быть легко нарушено, если две транзакции записывают *x* и *y* в разных порядках, что может произойти только при наличии грязных записей.

Еще одна причина необходимости защиты от грязных записей заключается в том, что без защиты от них система не может автоматически откатиться к образу «до» при прерывании транзакции.

3.4.2 Грязное чтение (англ. *Dirty Read*, P_1)

Грязное чтение — явление, когда одна транзакция считывает изменения, внесенные другими незафиксированными (или даже прерванными) транзакциями. [11]

Недостаточно предотвратить только чтение значений, записанных транзакциями, которые в конечном итоге откатываются. Также необходимо предотвратить чтение значений из транзакций, которые в конечном итоге также фиксируются. [4]

3.4.3 Неповторяющееся чтение (англ. *Fuzzy Read*, P_2)

Неповторяющееся чтение — это явление, которое возникает, когда значение считывается дважды во время транзакции, и эти считанные значения отличается между чтениями. Это возможно, когда значение, считанное транзакцией, все еще находящейся в процессе исполнения, перезаписывается другой транзакцией. Даже без повторного считывания значения, которое фактически происходит, это все равно может привести к нарушению инвариантов базы данных. [4]

3.4.4 Фантомное чтение (англ. *Phantom*, P_3)

Фантомное чтение происходит, когда транзакция выполняет два идентичных запроса во время обработки, но возвращаемые результаты этих двух запросов различны. [11]

3.4.5 G0 (цикл записи, англ. *Write Cycle*)

История содержит аномалию *цикл записи*, если ее граф сериализации содержит цикл, полностью состоящий из ребер зависимости записи(*ww*).

Цикл записи происходит, когда две транзакции записывают один и тот же набор данных. Предотвращение циклов записи является минимальным требованием для наличия функциональной базы данных, гарантируя, что записи, выполняемые транзакцией *A*, не перезаписываются транзакцией *B*, пока транзакция *A* все еще выполняется. [11]

3.4.6 G1

G1: включает в себя три феномена:

- G1a (прерванное чтение, англ. *aborted read*) — T_2 считывает некоторый объект (в том числе с помощью чтения предикатов), измененный T_1, и T_1 прерывается. Чтобы предотвратить прерывание чтения, если T_2 читает из T_1 и T_1 прерывается, T_2 также должен прерваться;

- G1b (промежуточное чтение, англ. *intermediate read*) — T_2 считывает версию некоторого объекта (в том числе с помощью чтения предикатов), измененную T_1, и это не было окончательной модификацией этого объекта T_1. Чтобы предотвратить промежуточное чтение, транзакции могут быть разрешены к фиксации только в том случае, если они прочитали окончательные версии объектов из других транзакций;

- G1c (циклический информационный поток, англ. *cyclic information flow*) — граф сериализации содержит направленный цикл, полностью состоящий из ребер зависимостей (чтение и запись). Если на T_1 влияет T_2, то нет никакого пути, по которому T_2 также может влиять на T_1.

3.4.7 G2-item (цикл антизависимости, англ. *anti-dependency cycle*)

G2-item(anti-dependency cycle, цикл антизависимости) возникает, когда граф сериализации *DSG* содержит направленный цикл, имеющий одно или несколько ребер антизависимости(*rw*).

Методология проверки распределенных систем

В предыдущей главе были рассмотрены некоторые феномены, которые нарушают гарантии согласованности распределенной системы. Несмотря на то, что они встречаются довольно часто, их сложно обнаружить. В этой главе будет рассмотрен инструмент, разработанный специально для проверки того, соответствует ли распределенная система и ее транзакции своим гарантиям согласованности и изолированности.

4.1 Jepsen

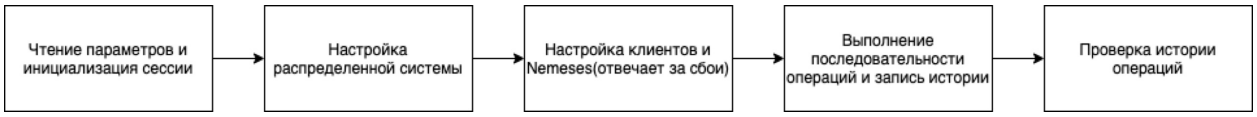

Рис. 4.1: Схема работы Jepsen-тестирования

Jepsen — это библиотека для функционального языка Clojure. Jepsen может доказать только лишь наличие ошибок, а не их отсутствие.

Jepsen проверяет систему, генерируя случайные последовательности операции (например, чтение, запись, cas) в распределенной системе, записывая метку времени и продолжительность каждой операции, а также создавая модель системы в памяти. Также Jepsen может генерировать последовательности транзакций из различных операций. А затем он пытается доказать, имеет ли история событий смысл с учетом заданной модели согласованности.

Jepsen также может генерировать разные сбои в распределенной системе, например, проблемы с сетью, уничтожение компонентов, а также генерацию случайной нагрузки.

Jepsen инкапсулирует код для настройки и демонтажа распределенной системы, которую нужно протестировать. Возможно выполнить настройку и демонтаж вручную, но если

позволить Jepsen справиться с этим, то возможно запускать тесты в системе *CI*, параметризовать конфигурацию базы данных, запускать несколько тестов подряд с чистого листа и так далее.

4.1.1 Как анализировать результаты

При каждом запуске jepsen создает новый каталог в *store/* директории, и можно увидеть последние результаты в папке *store/latest*. Там лежат несколько файлов. Файл *history.txt* содержит операции, которые выполнял тест. Файл *jepsen.log* — копия консоли для этого запуска, *jepsen.log* есть журнал всех операций, выполненных jepsen к тестируемой системе, и, наконец, *test.fressian* — это необработанные данные теста, включающие полную историю операций, *timeline.html* — это html документ, который показывает удобную временную шкалу операций. Эта шкала очень полезный инструмент для понимания порядка операций в тесте и выявления причин несогласованности результатов теста. Синий цвет указывает на то, что операция прошла успешно, красный — на неудачную операцию (состояние системы не изменилось), а оранжевый — на неопределенную операцию.

4.1.2 Подробнее о работе Jepsen

При запуске теста jepsen сначала подключается по ssh к каждому узлу, загрузит, распакует и настроит на них распределенную систему.

После запускаются клиентские процессы и процессы-сбои (англ. *nemesis*). Во время теста в jepsen есть два типа процессов: один — клиент, который будет выполнять различные операции с системой (с интервалом и частотой, заданных в генераторе), а другой — nemesis, вносящий сбои и разрушения и выполнять восстановление системы. После завершения операций jepsen будет использовать checker (после jepsen 2.0 это будет Elle), чтобы проверить правильность истории операций с определенными моделями согласованности.

Jepsen кластер состоит из *jepsen-control* узла, который управляет другими узлами, настройкой и удалением, генерирует данные и сбои. А также из обычных узлов, далее они будут обозначаться как *jepsen-nX*.

Настройка операционной системы

Первым этапом на узлах *jepsen-nX* вызывается настройка операционной системы, которая была задана в параметрах тестирующей системы. Jepsen поддерживает несколько операционных систем:

1. *jepsen.os.centos*
2. *jepsen.os.debian*
3. *jepsen.os.smartos*
4. *jepsen.os.ubuntu*

Настройка базы данных

Далее необходимо настроить распределенную систему на узлах кластера *jepsen-nX*. Для этого специальная функция *db* использует *reify* для создания нового объекта, удовлетворяющего протоколу баз данных Jepsen (из пространства имен *db*). Этот протокол определяет две функции, которые должны выполнять настройку(*setup!*) и демонтаж(*teardown!*) узла тестирования базы данных.

В функции *setup!* прописывается алгоритм установки базы данных на узел. Как правило, сначала загружается архив в нужными файлами, распаковывается в каталог и запускается нужный двоичный файл.

Для установки пакетов нужно быть в root, поэтому используется *jepsen.control/su*, чтобы получить привилегии root. А для загрузки и установки архива используется *jepsen.control.util/install-archive!*.

Jepsen запускает установку (а после и демонтаж) одновременно на всех узлах. Это может занимать некоторое время, так как каждый узел должен загрузить архив. Но при повторных запусках Jepsen будет использовать кэшированный архив.

Распределенная система должна быть запущена на каждом узле в качестве демона (*daemon*) в фоновом режиме. Рекомендуется использовать функции *jepsen.control.util* для запуска и остановки демонов для запуска распределенной системы.

Демонтаж базы данных

Чтобы убедиться, что действия предыдущего запуска тестовой системы не влияли на текущие результаты, Jepsen выполняет демонтаж базы данных перед настройкой в начале теста. Затем он снова выполняет демонтаж по завершении теста. Для этого используется команда остановки демона, а после удаляются каталоги с файлами. Для этого используется *jepsen.control/exec*. Jepsen автоматически связывает *exec* для работы с нужным узлом, настроенным во время *db/setup!*, но при необходимости можно подключиться к произвольным узлам.

Генератор и вид операций

Jepsen client принимает операцию, применяет ее к тестируемой системе и возвращает соответствующие значения операции завершения. У операций имеются поля *:type*, *:f*, *:value*. $: type =: invoke$ означает, что это операция только собирается быть применена к системе. $: type =: ok$ означает, что операция завершена успешно, а $: type =: fail$ означает, что операция завершилась с ошибкой. *:f* несет в себе информацию о том, какая именно операция должна быть применена к тестируемой системе, например, *read*, *write*, *append*.

Вызовы функций параметризуются их аргументами и возвращаемым значениям. Операции Jepsen параметризуются значениями *:value*, которое может быть любым — Jepsen не проверяет их. Например, $: f =: write, : value = k$ используется, чтобы указать значение, которое записывается. А $: f =: read, : value = k$ используется, чтобы указать значение, которое (в конечном итоге) будет прочитано. Когда операция чтения только вызывается, используется $: value = nil$, так как неизвестно, что будет прочитано.

jepsen.generator генерирует операции описанного вида. И даже последовательности операций, которые должны исполняться как транзакции.

Jepsen Client

После настройки распределенной системы на *jepsen-control* запускается генератор некоторых последовательностей операций. Эти операции передаются на исполняющие узлы *jepsen-nX*, где для их обработки определен *Jepsen client*.

Для этого будет определен новый тип структуры данных **Client**. Клиенты поддерживают клиентский протокол Jepsen, и, как и *reify*, предоставляют реализацию клиентских функций(*open!*, *setup!*, *invoke!*, *teardown!*, *close!*), которые должны быть реализованы.

Жизненный цикл клиента состоит из 5 частей. Сначала *open!* получает копию клиента, привязанную к определенному узлу, и устанавливает соединение с тестируемой системой. Далее *setup!* инициализирует нужные тесту структуры данных, например, создает таблицы. Затем *invoke!* применяет операции, сгенерированные *jepsen.generator*, к тестируемой системе и возвращает соответствующие операции завершения. Потом *teardown!* удаляет и очищает все то, что было создано *setup!*. И затем *close!* закрывает сетевое подключение и завершает жизненный цикл клиента.

Проверка корректности

После генерации операций и выполнения их клиентами у Jepsen есть истории. Они содержат сами операции с их результатом, а также метки времени и продолжительность каждой операции. Дальше эти истории необходимо проанализировать. Jepsen использует модель для представления абстрактного поведения системы и средство для проверки того, соответствует ли история заданной модели. В более старых тестах Jepsen использовался *knossos.model* для проверки корректности. Для Jepsen версии 2.0 и выше используется *Elle*. В этой работе также будет использоваться *Elle*, которая будет подробнее рассмотрена далее.

Также можно использовать *checker/compose* для выполнения анализа линеаризуемости и создания графиков производительности. Кроме того, есть возможность создавать HTML-визуализации истории. Для этого нужно использовать *jepsen.checker.timeline*.

Сбои

Для добавления сбоев в тестируемую систему используется Немезида(англ. *nemesis*). Это специальный клиент, не привязанный к какому-либо конкретному узлу. *jepsen.nemesis* обеспечивает несколько встроенных режимов сбоев. Например, *nemesis/partition-random-halves* разделяет сеть на две половины, выбранные случайным образом, а затем лечит сеть по прошествии какого-то времени. Также можно добавлять в тестируемую систему паузы и сдвиги часов.

Как и для остальных клиентов, операции сбоев генерируются на узле *jepsen-control* с помощью такого же генератора.

4.2 Elle

Elle — это инструмент для анализа транзакций. Он автоматически строит граф сериализации для транзакций и ищет циклы в этом графе для выявления нарушений согласованности. Например, если граф содержит цикл, то невозможно сказать, какая транзакция произошла до и после, а значит, нарушается гарантия линеаризуемости. Дополнительно проверяется наличие прерванных и промежуточных считываний и другие нарушения.

Elle не является полным: он может не идентифицировать аномалии, которые присутствовали в тестируемой системе. Это следствие двух факторов:

- Elle проверяет истории, наблюдаемые в реальных базах данных, где результаты транзакций могут остаться незамеченными, а информация о времени может быть не такой

точной, как хотелось бы;

- проверка сериализуемости является NP-полной задачей; Elle намеренно ограничивает свои выводы теми, которые можно решить за линейное (или *log*-линейное) время.

В зависимости от того, какие ребра содержались в найденном цикле в графе сериализации(*ww*, *wr* или *rw*), делается вывод о том, какая найдена аномалия.

4.2.1 Список возможных аномалий

- G0 (цикл записи, англ. *Write Cycle*);
- G1a (прерванное чтение, англ. *aborted read*);
- G1b (промежуточное чтение, англ. *intermediate read*);
- G1c (циклический информационный поток, англ. *cyclic information flow*);
- G-single (перекос чтения, англ. *read skew*);
- G2-item (цикл анти зависимости, англ. *anti-dependency cycle*).

Инструмент также умеет проверять согласованность внутри одной транзакции: то есть можно проверить, что транзакции считывают значения, соответствующие их собственным предыдущим записям, нет дублирующихся элементов и неожиданных элементов (например, элементов, которые никогда не были записаны).

Исследование согласованности Azure Cosmos DB

5.1 Azure Cosmos DB

Azure Cosmos DB - это коммерческий (с закрытым исходным кодом) глобально распределенный многомодельный сервис баз данных Microsoft «для управления данными в планетарном масштабе», запущенный в мае 2017 года. Он не зависит от схемы, горизонтально масштабируем и обычно классифицируется как база данных NoSQL.

Cosmos DB поддерживает несколько API: SQL, Cassandra, MongoDB, Gremlin, Table. Здесь под SQL подразумевается документно ориентированный API, который раньше назывался *DocumentDB*, и он значительно отличается от привычных реляционных баз данных.

Также Cosmos DB поддерживает 5 уровней согласованности: строгий (англ. *strong*), ограниченное устаревание(англ. *bounded staleness*), сеанс(англ. *session*), постоянный префикс(англ. *consistent prefix*) и случайный(англ. *eventual*).

Документная модель Cosmos DB хранит данные в контейнерах (*containers*), состоящих из элементов (*items*). Все настройки масштабирования, пропускной способности, индексирования указываются на уровне контейнера. База данных, по большому счету — именованное объединение контейнеров.

5.2 Дизайн теста

Был разработан тест с использованием библиотеки для тестирования распределенных систем Jepsen. Он будет использован для оценки безопасности транзакций в Azure Cosmos DB.

5.2.1 Append тест

Транзакции параллельно читают и добавляют в списки уникальные целые числа. Каждый список хранится по уникальному *id*. Генератор Jepsen генерирует случайную последовательность транзакций, где каждая транзакция состоит из набора операций произвольной длины. Максимальная и минимальная длина набора операций в транзакции задается в параметрах теста.

Операции в транзакции могут быть двух видов:

1. чтение — считывает список значений по заданному *id*;
2. добавление — добавляет в конец списка чисел по ключу *id* уникальное целое число.

5.3 Этапы тестирования

Далее, используя Elle для анализа транзакций, Jepsen строит граф сериализации для каждой истории и ищет циклы для выявления аномалий.

5.4 Список возможных параметров для теста

5.5 О реализации транзакций в Azure Cosmos DB

5.5.1 TransactionalBatch

TransactionalBatch — это, как утверждает документация, способ задания транзакции из нескольких операций (create, read, update, upsert, delete). Эти операции либо успешно выполнятся все вместе, либо завершатся сбоем. В TransactionalBatch операции выполняются с одним и тем же ключом секции в контейнере. Итак, если все операции выполняются успешно в том порядке, в котором они описаны в транзакционной пакетной операции, транзакция будет зафиксирована. Однако при сбое любой операции выполняется откат всей транзакции.

TransactionalBatch способствует увеличению производительности транзакций, как заявлено в документации.

При тестировании транзакций с использованием TransactionalBatch Elle обнаружила следующие аномалии:

- внутренняя несогласованность (англ. *Internal Inconsistency*) — транзакция не соблюдает свои собственные предыдущие операции чтения и записи.

- несогласованный порядок версий (англ.*Inconsistent Version Orders*) — правила вывода предполагают циклический порядок обновления одного ключа.

Также, из отношений между аномалиями Elle[12] можно заключить, что из обнаружение в истории *Inconsistent Version Orders* аномалии следует, что *G1a* аномалия там также присуствует.

Кроме того, на уровнях **Ограниченное устаревания** и **Случайная** была обнаружена *G2-item* аномалия.

Напомним, что обнаружение этой аномалии означает, что граф сериализации(*DSG*) содержит направленный цикл с одним или несколькими ребрами анти зависимости [2].

Транзакции теряют подтвержденные записи. Кроме того, оказывается, что транзакции не изолированы. То есть,транзакции в такой реализации могли влиять на результаты других транзакций. А значит, TransactionalBatch в нашей задаче использовать нельзя.

5.5.2 Хранимые процедуры(англ. *Stored procedures*)

Azure Cosmos DB обеспечивает транзакционное выполнение JavaScript кода. При использовании API SQL в Cosmos DB можно писать хранимые процедуры, триггеры и определяемые пользователем функции (UDF) на языке JavaScript.

Только документы из одного и того же логического раздела могут быть включены в одну транзакцию. Соответственно, операции записи в разные контейнеры не могут быть выполнены транзакционно. Время выполнения одной хранимой процедуры ограничено (5 секунд), и если длительность транзакции выходит за эти рамки, она будет отменена. Есть способы реализации «долгоживущих» транзакций через несколько обращений к серверу, но они нарушают атомарность.

Помимо того, что написанный JavaScript код будет выполняться атомарно, также данный способ реализации транзакций обещает хорошую производительность. Можно назвать следующие преимущества:

- *пакетная обработка* — это сократит затраты сетефого трафика и накладные расходы на хранение

- *предварительная компиляция* — хранимые процедуры, триггеры и определяемые пользователем функции неявно предварительно скомпилированы в формат байтового кода, чтобы избежать затрат на компиляцию во время каждого вызова скрипта. Благодаря предварительной компиляции скорость хранимой процедуры высокая, а занимаемая память небольшая.

Хранимые процедуры и триггеры всегда выполняются на основной реплике контейнера Azure Cosmos. Эта возможность гарантирует, что операции чтения в хранимых процедурах обеспечивают сильную согласованность.

5.6 Модель согласованности транзакций

Что же утверждает документация Azure Cosmos DB касательно транзакций? Давайте посмотрим. Транзакции базы данных обеспечивают безопасную и предсказуемую модель программирования для обработки одновременных изменений данных. Традиционные реляционные базы данных позволяют писать бизнес-логику с помощью хранимых процедур и/или триггеров, отправлять ее на сервер для выполнения непосредственно в ядре базы данных.

В Azure Cosmos DB поддерживаются транзакции, полностью совместимые с ACID (атомарность, согласованность, изоляция, прочность). В документации утверждается, что поддерживаемый уровень изоляции транзакций - **изоляция моментальных снимков**. Это достаточно сильная модель, которая представляет собой базовый уровень согласованности для таких систем, как PostgreSQL.

5.7 Описание кластера для тестирования

В данном исследовании мы будем запускать кластер Jepsen на одном компьютере, используя docker compose. Это упрощает и стандартизирует выполнение тестов.

Репозиторий Jepsen предоставляет базовые настройки для запуска тестов в докере. Он поддерживает 3 вида контейнеров:

- jepsen-control: управляет другими узлами, настройкой и удалением, генерирует данные и сбои;
- jepsen-nX: один из узлов в кластере(по умолчанию таких узлов 5);
- jepsen-node: используется агентом для создания сбоев (nemesis).

5.8 Результаты

При тестировании Azure Cosmos DB, где мы реализовали транзакции с помощью хранимых процедур(англ. *Stored procedures*), на всех уровнях согласованности(мы запускали тесты с разными параметрами, в том числе на разных уровных согласованности: силь-

ная, ограниченное устаревание,сеанс, префикс и случайная) были замечены G2-item аномалии. Давайте посмотрим на несколько примеров примеры таких аномалий и попытаемся понять,противоречит ли данная аномалия требованиям к Azure Cosmos DB, заявленным в документации.

5.8.1 Тестирование базового уровня

Мы начали с того, что запустили наши тесты в упрощенном варианте, чтобы получить базовый уровень: каждая транзакция Jepsen выполняла только одну операцию чтения или добавления. Полученные истории оказались совместимыми с **изоляцией моментальных снимков**.

5.8.2 Обозначения для графиков

Транзакции, изображенные на графиках аномалий, могут состоять из двух типов операций.

Чтение (англ. *read*)

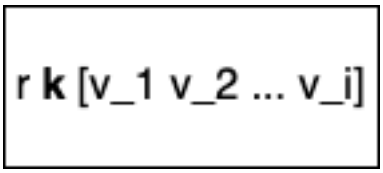

Рис. 5.1: Обозначение для операции чтения

Так будет обозначаться операция чтения в транзакции. *r* — операция чтения, ***k*** — *id* объекта в таблице, который считывается. $[v_1 v_2 ... v_i]$ - сам объект, который был считан, массив целых чисел. Допускается, что может быть считан пустой массив, тогда он обозначается как $[]$.

Добавление (англ. *append*)

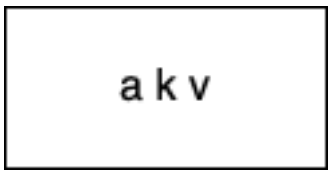

Рис. 5.2: Обозначение для операции добавления

Так будет обозначаться операция добавления нового элемента в транзакции. *a* — операция добавления, ***k*** — *id* объекта в таблице, к которому требуется добавить значение. *v* - значение, которое должно быть добавлено в конец массива целых чисел, который уже хранится.

5.8.3 G2-item (англ. *anti-dependency cycle*, цикл антизависимости)

Пример 1

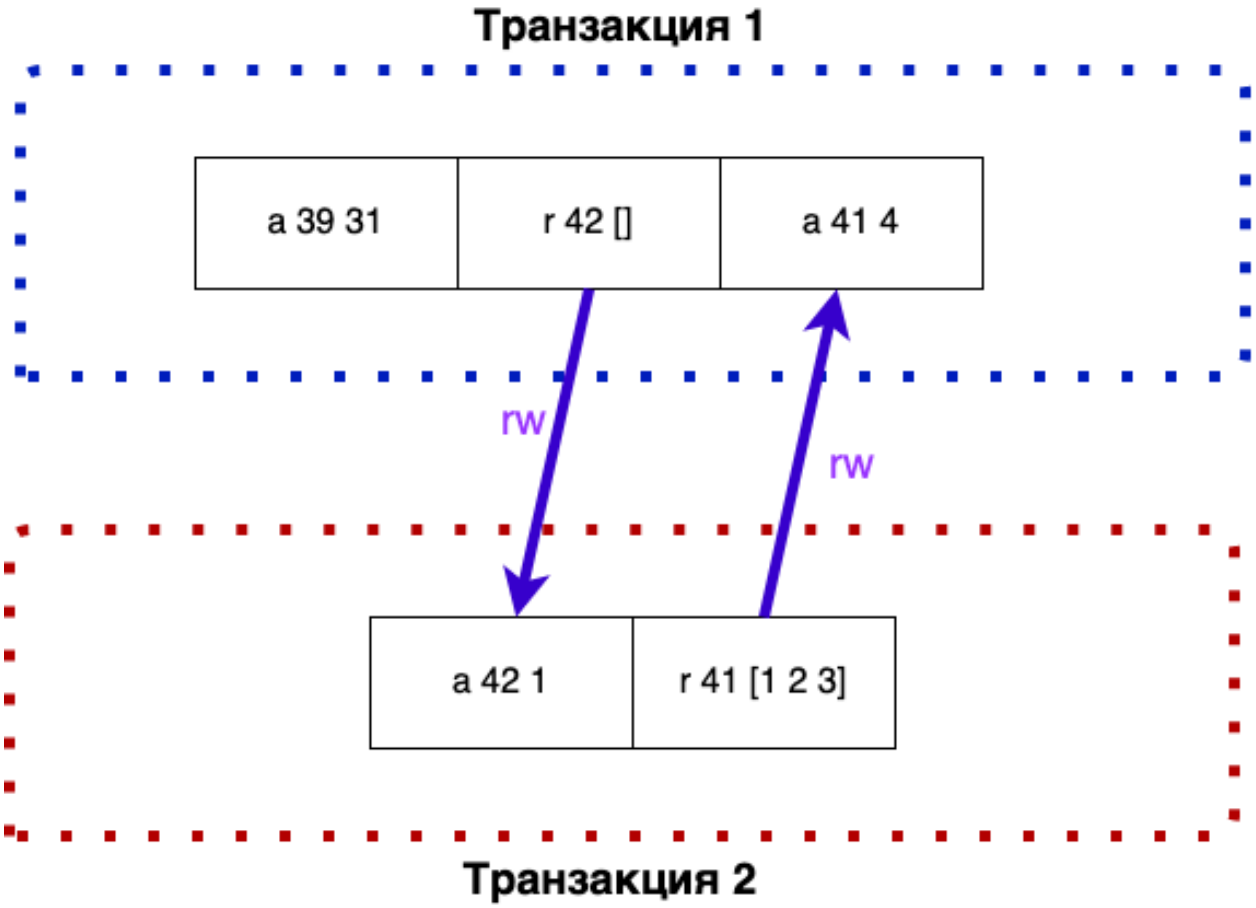

Рис. 5.3: G2-item

Данная аномалия найдена при тестировании с использованием следующих параметров:

уровень согласованности — ограниченное устаревание(англ. *bounded staleness*)

количество потоков — 15

лимит времени на транзакцию — 120 секунд

ограничение на количество элементов по одному ключу — 128

максимальное количество операций в транзакции — 4

Транзакция 1 выполняет операции:

1. добавление числа *31* к массиву под *id* ***39***

2. чтение массива значений по *id* ***42*** $\rightarrow$ получен пустой массив *[ ]*

3. добавление числа *4* к массиву под *id* ***41***

Транзакция 2 выполняет операции:

1. добавление числа *1* к массиву под *id* ***42***

2. чтение массива значений по *id* ***41*** $\rightarrow$ получен массив $[123]$

Ребро анти зависимости *rw* добавляется между операцией 2 транзакции 1 и операций 1 транзакции 2 потому что операция 2 транзакции 1 считала некоторую(*[ ]*) версию объекта с *id* **42**, а операция 1 транзакции 2 изменила этот объект, добавив в массив значений новое число ***1***. Также ребро анти зависимости *rw* добавляется между операцией 2 транзакции 2 и операцией 3 транзакции 1, потому что операция 2 транзакции 2 считала некоторую(*[1 2 3]*) версию объекта с *id* **41**, а операция 3 транзакции 1 изменила этот объект, добавив в массив значений новое число ***4***. В полученном графе сериализации наблюдается направленный цикл, содержащий 2 ребра анти зависимости. Значит, по определению, найдена *G2-item* аномалия.

Эти две транзакции невозможно изолировать: если бы первая транзакция выполнялась первой, изолированно, ее запись с *id* 41 была бы видна второй транзакции — и наоборот. Но поскольку эти транзакции не записывались в один и тот же *id*, им разрешено (при **изоляции моментальных снимков**) выполняться одновременно.

Пример 2

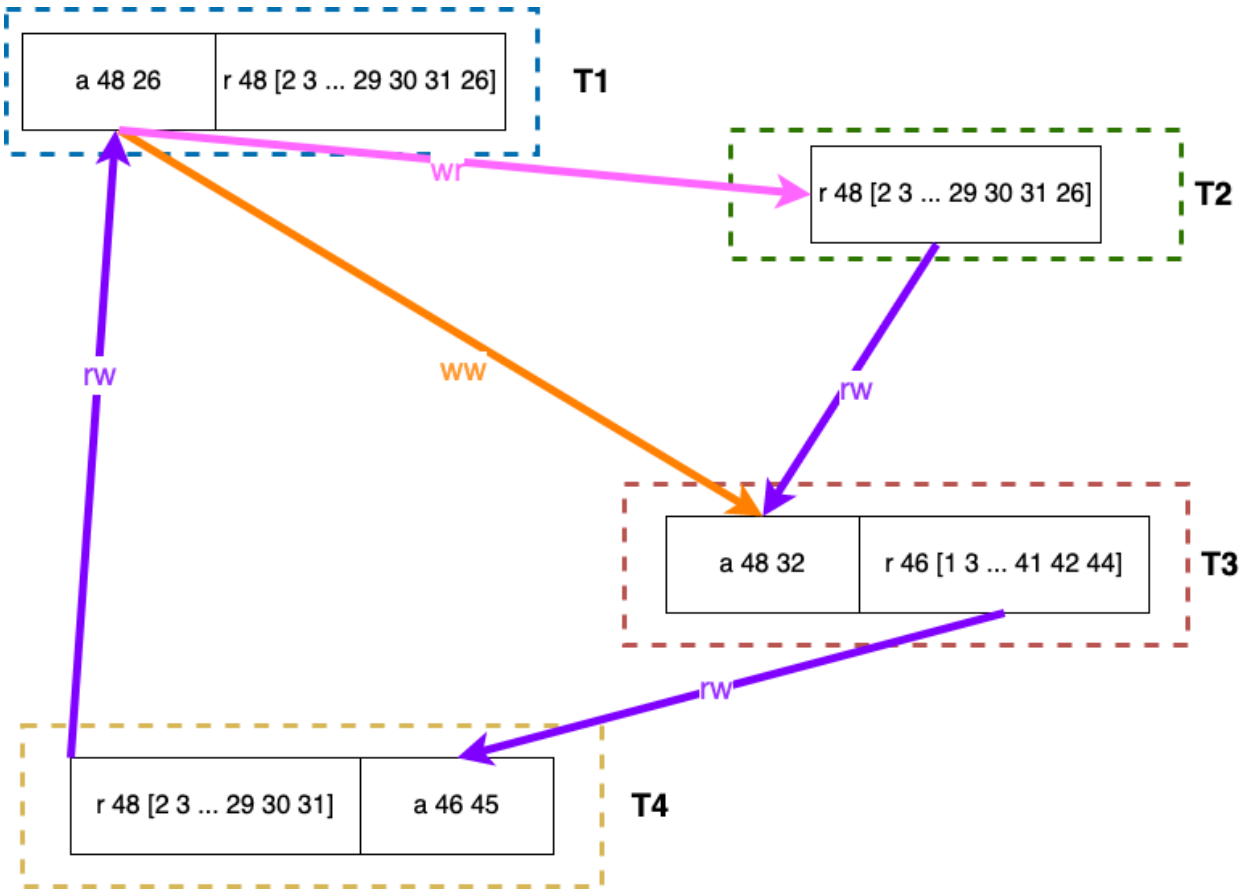

Рис. 5.4: G2-item

Данная аномалия найдена при тестировании с использованием следующих параметров:

уровень согласованности — постоянный префикс(англ. *consistent prefix*)

количество потоков — 15

лимит времени на транзакцию — 120 секунд

ограничение на количество элементов по одному ключу — 128

максимальное количество операций в транзакции — 7

Это более сложный цикл, состоящий из 4 транзакций. Каждая из этих транзакций зависит от другой.

Транзакция 1(T_1) выполняет операции:

1. добавление числа *26* к массиву под *id* ***48***

2. чтение массива значений по *id* ***48*** → получен массив *[2 3 4 5 6 7 8 9 1 10 11 12 13 14 15 16 17 18 19 20 21 22 23 25 27 28 29 30 31 26]*

Транзакция 2(T_2) выполняет операцию:

1. чтение массива значений по *id* ***48*** → получен массив *[2 3 4 5 6 7 8 9 1 10 11 12 13 14 15 16 17 18 19 20 21 22 23 25 27 28 29 30 31 26]*

Транзакция 3(T_3) выполняет операции:

1. добавление числа *32* к массиву под *id* ***48***
2. чтение массива значений по *id* ***46*** → получен массив *[1 3 4 5 6 2 8 9 10 7 11 13 12 14 15 17 21 22 18 19 16 23 24 25 26 27 29 30 31 20 33 36 35 32 28 34 38 37 39 40 41 42 44]*

Транзакция 4(T_4) выполняет операции:

1. чтение массива значений по *id* ***48*** → получен массив *[2 3 4 5 6 7 8 9 1 10 11 12 13 14 15 16 17 18 19 20 21 22 23 25 27 28 29 30 31]*
2. добавление числа *45* к массиву под *id* ***46***

Далее обозначение $T_1.1$ означает операцию 1 транзакции 1.

Ребра:

- *rw* — $T_4.1 \xrightarrow{rw} T_1.1$, так как $T_4.1$ считывает массив по $id = 48$ и получает массив значений без числа 26, а $T_1.1$ изменяет массив, добавляя в него новое число 26;
- *rw* — $T_2.1 \xrightarrow{rw} T_3.1$, так как $T_2.1$ считывает массив по $id = 48$ и получает массив значений без числа 32, а $T_3.1$ изменяет массив, добавляя в него новое число 32;
- *rw* — $T_3.2 \xrightarrow{rw} T_4.2$, так как $T_3.2$ считывает массив по $id = 46$ и получает массив значений без числа 45, а $T_4.2$ изменяет массив, добавляя в него новое число 45;
- *ww* — $T_1.1 \xrightarrow{ww} T_3.1$, так как транзакции изменяют один и тот же объект $id = 48$, одна транзакция добавляет число 26, а другая 32;
- *wr* — $T_1.1 \xrightarrow{wr} T_2.1$, так как $T_2.1$ считала массив значений объекта $id = 48$ и в массиве содержалось число 26, добавленное транзакцией $T_1.1$.

В полученном графе сериализации наблюдается направленный цикл, содержащий 3 ребра анти зависимости. Значит, по определению, найдена *G2-item* аномалия.

Если пытаться упорядочить данные транзакции, то наблюдается следующее поведение ($T_i < T_j$ — транзакция T_i произошла раньше T_j):

- $T_2 < T_3$, так как T_2 не наблюдает число 32, добавленное в транзакции T_3;
- $T_3 < T_4$, так как T_3 не наблюдает число 45, добавленное в транзакции T_4;
- $T_4 < T_1$, так как T_4 не наблюдает число 26, добавленное в транзакции T_1;
- $T_1 < T_2$, так как T_2 наблюдает число 26, добавленное в транзакции T_1.

Получили противоречие. Эти четыре транзакции невозможно изолировать.

5.8.4 Выводы

В этом параграфе будет рассмотрено, что означает найденная *G2-item* аномалия. В документации Azure Cosmos DB сказано, что транзакции полностью атомарны, изолированы,согласованы и прочны(«*full ACID*»). Однако это не совсем так, потому что транзакции, поддерживающие **изоляцию моментальных снимков** не являются полностью изолированными.

То есть, полученные истории, по-видимому, не нарушают **изоляцию моментальных снимков**, но, тем не менее, демонстрируют циклические зависимости транзакций.

Можно ли говорить, что эти истории удовлетворяют требованиям «*full ACID*»? Возможно, но если это так, тогда из этого следует, что «**I**» в *ACID* означает только частичную изоляцию, или «*full*» означает несколько меньше, чем полная.

Данная аномалия не редкость. Примерно в 15% транзакций наблюдались аномалии во время нормальной работы, без сбоев.

Выводы

В данной работе был обозначен ряд проблем, которые возникают в распределенных системах. Зачастую они носят случайный характер. Также их сложно выявить на этапе разработки распределенной системы. Это обуславливает необходимость введения формальных определений различных моделей согласованности.

Также сложность обнаружения различных нарушений изоляции и согласованности обуславливает появление Jepsen как инструмента для проверки гарантий выполнения важнейших свойств распределенных систем. Этот инструмент был представлен в данной работе.

Кроме того, в этой работе с помощью Jepsen была проанализирована реальная база данных — Cosmos DB. Cosmos DB утверждает, что поддерживает «полностью ACID транзакции» через изоляцию моментальных снимков. Однако использование этих транзакций осложняется запутанной документацией и API. А также изоляция моментальных снимков плохо совместима с маркетинговой фразой «*full ACID*». В процессе тестирования наблюдались истории, которые казались совместимыми с изоляцией моментальных снимков, но также включали аномалии G2-item (циклы анти зависимости), в которых транзакции не наблюдали эффектов друг друга. Такие аномалии были замечены в 15% историй. Это корректно при изоляции моментальных снимков, но спорно, что эти транзакции являются полностью изолированными в смысле *ACID*.

Итак, в этой работе было проведено лишь краткое исследование. В дальнейшем в рамках развития данной работы возможно реализовать другие тесты, например, *register* тест, а также исследовать поведение базы данных при различных сбоях.

Литература

[1] Consistency models. "https://jepsen.io/consistency".

[2] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. pages 67–78, 01 2000.

[3] Adrian Colyer. Generalized isolation level definitions. "https://blog.acolyer.org/2016/02/25/generalized-isolation-level-definitions/".

[4] Adrian Colyer. A critique of ansi sql isolation levels. "https://blog.acolyer.org/2016/02/24/a-critique-of-ansi-sql-isolation-levels/".

[5] Serdar Benderli. Chaos testing a distributed system with jepsen. "https://medium.com/appian-engineering/chaos-testing-a-distributed-system-with-jepsen-2ae4a8bdf4e5".

[6] Атомарные и неатомарные операции. "https://habr.com/ru/post/244881/".

[7] Модель памяти в примерах и не только. "https://habr.com/ru/post/133981/".

[8] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, Cambridge, MA, USA, March 1999. Also as Technical Report MIT/LCS/TR-786.

[9] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[10] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In Luca Aceto and David de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[11] Kevin Sookocheff. Paper review: Generalized isolation level definitions. "https://sookocheff.com/post/databases/generalized-isolation-level-definitions/".

[12] Kyle Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14:268–280, 2020.