Аннотация

Одной из составных частей компьютерных программ являются базы данных. Для обеспечения масштабируемости и надежности базы данных делают распределенными. При использовании распределенных баз данных возникает вопрос, удовлетворяют ли они заявленным характеристикам и свойствам. Это связано с проблемой обеспечения изолированности транзакций — важнейшего инструмента взаимодействия с базой данных.

Тестирование распределенных систем — нетривиальная задача, потому что ошибки сложно обнаружить, так как они зачастую являются результатом сочетания маловероятных событий. Поэтому тестирование изолированности — это актуальный вопрос для разработчиков распределенных систем. Один из инструментов для проверки изоляции транзакций, который практически не имеет аналогов — это инструмент хаос-тестирования Jepsen.

В этой работе изучен инструмент хаос-тестирования Jepsen. Рассмотрены различные феномены, указывающие на нарушения гарантии изолированности, и которые вспомогательный инструмент Jepsen Elle может находить. Кроме того, проведено исследование база данных Azure Cosmos DB с помощью Jepsen. В работе проанализированы найденные аномалии на соответствие заявленным моделям согласованности.

Оглавление

Введение

Изолированность транзакций в базах данных — это важное свойство. Нарушение этого свойства может привести к ошибкам системы, работающей с базой данных. Большинство распределенных систем стремятся к достижению баланса между временем выполнения операций и гарантиями изолированности операций.

Тестирование может помочь в нахождении ошибок. Однако это сложный и кропотливый процесс, потому что зачастую ошибки вызываются маловероятным сочетанием событий, приводящим к нарушениям в заявленной модели согласованности. Один из инструментов для проверки соблюдения изолированности — 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. Удобный инструмент для тестирования соответствия заявленной модели согласованности может существенно помочь на этапе разработки распределенной системы, возможно, стать одним из этапов CI/CD процесса.

2.1 Цели работы

- Научиться применять инструмент проверки свойств транзакций Jepsen;
- Проанализировать с помощью выбранного инструмента реальную базу данных Azure Cosmos DB, которая еще не была исследована;
- Сравнить модель согласованности, заявленную в документации, и модель согласованности, установленную 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 — [6] операция в параллельном программировании, выполняющаяся в общей области памяти потоков и завершающаяся за один шаг относительно других потоков, которые имеют доступ к этой же области памяти. Пока атомарная операция выполняется, ни один поток не может наблюдать частичные, незафиксированные изменения. Неатомарные операции не дают такой гарантии.

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.

История — совокупность операций и их параллельной структуры. В этой работе будут 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. Другими словами, *модель согласованности* — это набор историй, которые считаются корректными с точки зрения данной распределенной системы. Если сказано, что история нарушает какую-то модель согласованности, это означает, что история не входит в соответствующий набор историй. [1]

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.

Отношение «произошло до»(англ. happens before) — [7] это отношение между результатами двух операций. Пусть операцию A исполняет поток X, и операцию B исполняет поток Y. Если операция A «произошла до»(*happens-before*) операции B, то все изменения, совершенные X до операции A, и изменения, выполненные этой операцией, видимы для потока Y в момент выполнения операции B и после.

Основные теоретические сведения

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

База данных — организованная в соответствии с определёнными правилами и поддерживаемая в памяти компьютера(или нескольких компьютеров) совокупность данных, характеризующая актуальное состояние некоторой предметной области и используемая для удовлетворения информационных потребностей пользователей [8]. В данной работе будут рассматриваться распределенные базы данных, то есть таких базы данных, которые хранят некоторые части своих данных в различных физических локациях.

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.

3.1 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)[9]. Изоляция моментальных снимков не накладывается никаких ограничений в реальном времени и не требует упорядочивания процессов между транзакциями.

Беренсон, Бернштейн и другие [10] впервые определели *изоляцию моментального снимка* в терминах абстрактного алгоритма:

> *В момент начала транзакции она считывает данные из моментального снимка данных, которые были зафиксированы. Этот момент называется 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 T1 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 T2 с меткой времени фиксации в интервале выполнения T1 [время начала отсчета, время фиксации] не записала данные, которые также записала T1. В противном случае T1 прервется. Это предотвращает потерю обновлений (англ. lost update, P4). Когда T1 будет зафиксирована, эти изменения становятся видимыми для всех транзакций, метки начала отсчета которых больше, чем метка времени фиксации T1.*

В другой формулировке изоляции моментальных снимков определяется как комбинация четырех свойств [11]:

- 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.

3.2 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.

3.2.1 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*.

3.2.2 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.

3.2.3 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).

3.2.4 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.

3.3 Феномены

3.3.1 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 стоит избегать, заключается в том, что они влекут за собой нарушение согласованности распределенной системы. Если на *x* и *y* наложено ограничение (например, $x = y$), то транзакции T_1 и T_2 соблюдают согласованность ограничения, если исполняются отдельно. Но когда эти две транзакции исполняются вместе и изменяют *x* и *y* в разных порядках, то ограничение нарушается. Такое поведение возможно только при наличии грязных записей в системе.

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.

3.3.2 Грязное чтение (англ. *dirty read*, P_1)

Грязное чтение — явление, когда одна транзакция считывает изменения, внесенные другими незафиксированными (или даже прерванными) транзакциями. [12]

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]

3.3.3 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]

3.3.4 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. [12]

3.3.5 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. [12]

3.3.6 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.

3.3.7 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 изоляции транзакций в распределенных системах

В предыдущей главе были рассмотрены некоторые феномены, которые нарушают изоляцию транзакций в 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 изолированности.

4.1 Jepsen

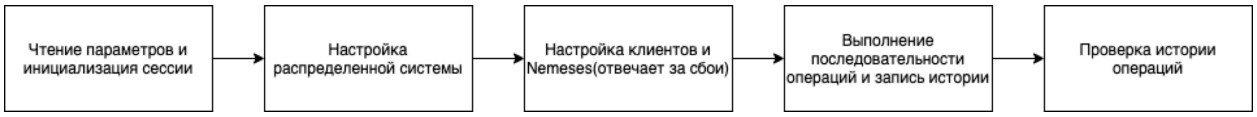

Рис. 4.1: Схема работы Jepsen-тестирования

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 Этапы работы 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

Итак, после генерации операций их выполняют клиенты. В результаты работы клиентов записываются 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 возвращает в исходное рабочее состояние 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.

4.1.2 Формат результатов тестов Jepsen

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.

4.2 Elle

Elle — это инструмент для анализа результатов тестирования. 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 [13]; 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 5 API: Table, SQL, MongoDB, Gremlin, Cassandra. Тут SQL — это документно ориентированный API, называющийся раньше *DocumentDB*, и имеет ряд существенных отличий по сравнению с традиционными реляционными базами 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*). Настройки репликации и пропускной способности указываются для контейнера. Можно определить базу данных Azure Cosmos DB как именованное объединение контейнеров.

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**.

5.2 Дизайн теста

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

В базе данных Cosmos DB хранятся пары ключ-значение. Ключ является уникальным целым числом *id*. Значение — список уникальных целых чисел. Изначально список значений пустой. Также при тестировании на размер каждого списка значений накладывается ограничение сверху (256 или 128 чисел).

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 Описание кластера для тестирования

Для тестирования Azure Cosmos DB использовался кластер Jepsen, который состоит из одного *jepsen-control* узла и пяти *jepsen-nX* узлов. *jepsen-nX* нагружали Cosmos DB, которая находится в облаке.

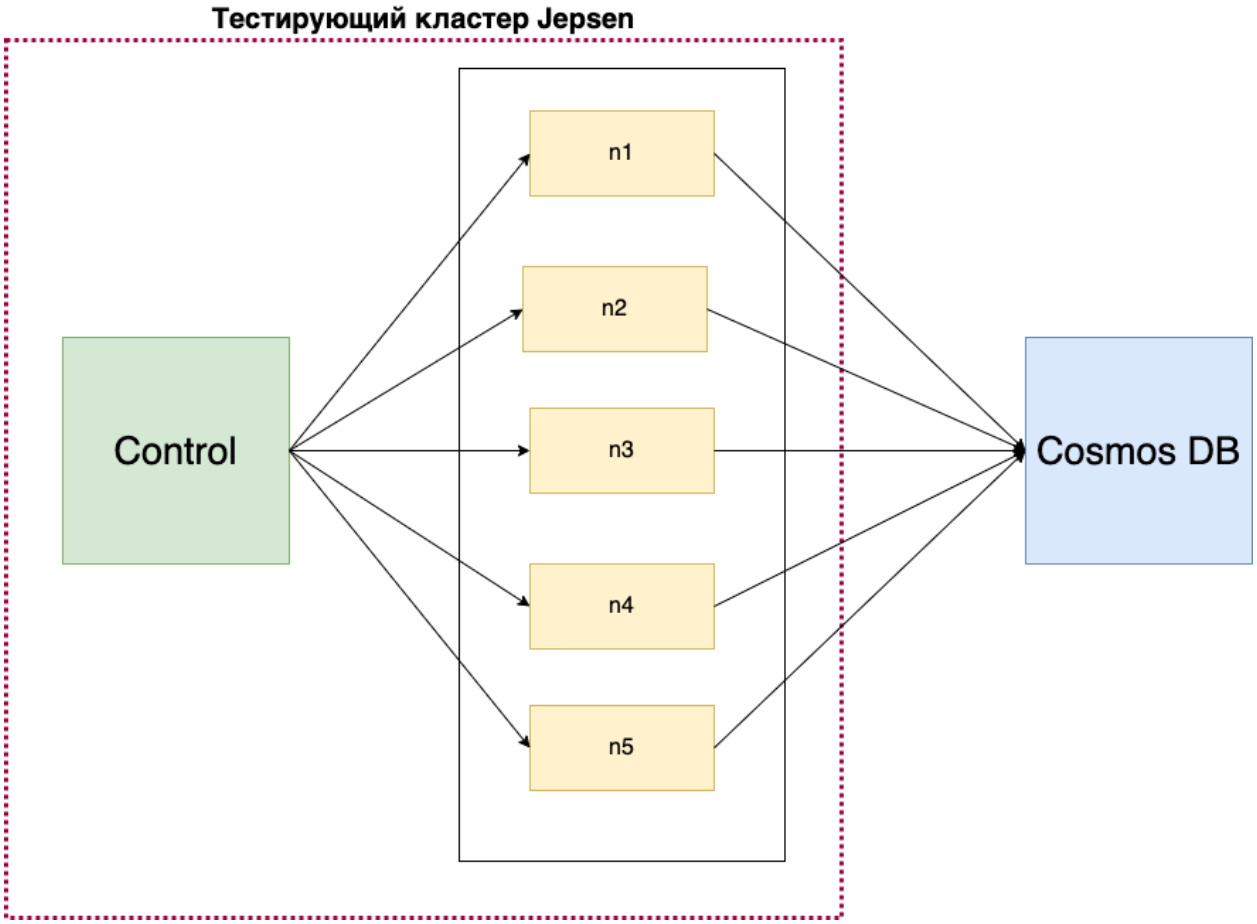

Рис. 5.1: Кластер тестирования Jepsen

В данном исследовании кластер Jepsen будет запущен на одном компьютере, используя docker compose.

Параметры компьютера

- Операционная система — macOS Big Sur версии 11.4
- Центральный процессор — $2,3 GHz$ *Dual-Core Intel Core i5*
- Оперативная память — $8GB$
- Количество ядер — 2

Использование *docker* упрощает и стандартизирует выполнение тестов.

Репозиторий Jepsen предоставляет базовые настройки для запуска тестов в *docker*: репозиторий с docker файлами.

Он поддерживает 2 вида контейнеров:

- *jepsen-control* — управляет другими узлами, настройкой и удалением, генерирует данные и сбои;

- *jepsen-nX* — один из узлов в кластере(будет использовано 5 таких узлов);

5.4 Параметры базы данных

Пропускная способность базы данных — 1000 RU/s

RU (или ЕЗ) — англ. *Request Unit*, единица запросов.

ЕЗ — это единица производительности, которая абстрагирует системные ресурсы (например, ЦП, операции ввода-вывода в секунду и память), необходимые для выполнения операций базы данных, поддерживаемых Azure Cosmos DB.

Например, стоимость выполнения считывания(т. е. выборки одного элемента по его идентификатору и значению ключа раздела) для элемента размером 1 КБ составляет 1 единицу запроса (или 1 ЕЗ). Независимо от того, какие API вы используете для взаимодействия с контейнером Azure Cosmos и независимо от типа операции, затраты всегда измеряются в ЕЗ.

Репликация данных — включена, если задан специальный параметр теста *–regions*. Тогда данные реплицируются в 3 региона, два региона на востоке США(*East US*) и один регион в северной Европе(*North Europe*). Иначе хранится единственная реплика.

5.5 О реализации транзакций в Azure Cosmos DB

5.5.1 Транзакционный пакет(англ. *TransactionalBatch*)

Описание

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.

Тестирование

Был запущен *append* тест.

Транзакции, реализованные с помощью *TransactionalBatch*, состояли из различных комбинаций двух операций: прочитать список значений по ключу *id* и добавить новое уникальное целое число в список значений по ключу *id*. Количество операций в одной транзакции было различное. При тестировании использовались следующие параметры тестируемой системы:

уровень согласованности — тестировалось на всех уровнях: строгий (англ. *strong*), ограниченное устаревание(англ. *bounded staleness*), сеанс(англ. *session*), постоянный префикс(англ. *consistent prefix*) и случайный(англ. *eventual*)

количество потоков — 5

лимит времени на транзакцию — 60 секунд

ограничение на количество элементов по одному ключу — 256

максимальное количество операций в транзакции — 4

минимальное количество операций в транзакции — 1

репликация — отключена

сбои — без сбоев

Результаты

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[14] можно заключить, что из обнаружения в истории *inconsistent version orders* аномалии следует, что *G1a* аномалия там также присутствует.

Кроме того, на уровнях **ограниченное устаревания** и **случайная** была обнаружена *G2-item* аномалия. Обнаружение этой аномалии означает, что граф сериализации(*DSG*) содержит направленный цикл с одним или несколькими ребрами анти зависимости [2].

Транзакции, реализованные через *TransactionalBatch*, 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 для **append** теста использовать нельзя. TransactionalBatch не удовлетворяет свойству внутренней консистентности транзакций.

Рассмотрим другой способ реализации транзакций в Cosmos DB.

5.5.2 Хранимые процедуры(англ. *Stored procedures*)

Azure Cosmos DB предоставляет возможность транзакционного выполнение *JavaScript* кода. При использовании API SQL в Cosmos DB можно реализовать триггеры, определяемые пользователем функции(*UDF*) и хранимые процедуры. Они пишутся на языке *JavaScript*.

Только операции над объектами из одного и того же раздела могут быть включены в одну транзакцию. А значит, операции записи в разные контейнеры не могут быть выполнены транзакционно.

Время выполнения одной хранимой процедуры ограничено пятью секундами. И если транзакция длится дольше этого времени, то она будет отменена. Существует несколько способов для 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 операций* — группировка операций в пакеты и отправка этих пакетов. Это может сократить издержки на пересылку по сети и расходы на создание транзакций для отдельных операций;

- *предварительная компиляция* — определяемые пользователем функции, триггеры и хранимые процедуры заранее компилируются в байт-код. Это позволяет хранимым процедурам работать быстро и занимать мало места.

Хранимые процедуры и триггеры всегда выполняются на первичной реплике контейнера.

5.6 Реализация

Для тестирования Cosmos DB был разработан *append* тест с использованием транзакций, которые реализовывались через хранимые процедуры (англ. *stored procedures*).

Исходный код опубликован на *github*:
https://github.com/Alaska-666/jepsen.cosmos-db/tree/main.

5.7 Тестирование

В этом параграфе будут описаны основные этапы тестирования Cosmos DB с помощью Jepsen.

При запуске тестирующей системы Jepsen выполнит настройку операционной системы на узлах *jepsen-nX*. Будет использована *jepsen.os.debian*.

Затем будет выполнено подключение к базе данных Cosmos DB, которая расположена «в облаке». На этом этапе будет создан контейнер. В контейнере будут храниться пары ключ-значение, где ключ есть уникальное целое число *id*, а значение — список целых чисел.

Дальше на *jepsen-control* узле будут генерироваться транзакции. Транзакции будут состоять из некоторого количества операций, где каждая операция это либо операция чтения списка чисел по ключу, либо добавление нового уникального целого числа в конец списка по ключу.

Сгенерированные транзакции передаются на исполнение пять *jepsen-nX*, где за их исполнение отвечает *Client*. Транзакции применяются к тестируемой системе. Если операция чтения обращается к несуществующему ключу, то будет прочитан пустой список. Если операция добавления обращается к несуществующему ключу, то создается список из одного элемента, который нужно было добавить в конец.

При тестировании использовались транзакции минимум из одной операции и максимум — из 20. Также использовались ограничения на количество чисел, которые могут быть записаны по одному *id*. По умолчанию максимальная длина списка чисел — 256.

После завершения работы *client*-ов у Jepsen имеется список историй, где хранится также временная метка каждой операции и ее длительность. Jepsen проводит проверку корректности историй с помощью Elle. Elle строит граф сериализации для каждой истории, затем ищет циклы для выявления аномалий.

5.7.1 Параметры тестирования

Используемые параметры:

уровень согласованности — тестировалось на всех уровнях

количество потоков — от 5 до 15

лимит времени на транзакцию — 60 секунд или 120 секунд

ограничение на количество элементов по одному ключу — 256 или 128

максимальное количество операций в транзакции — от 4 до 20

минимальное количество операций в транзакции — 1

репликация — отключена

сбои — сдвиг часов

5.8 Результаты

При тестировании Azure Cosmos DB, где транзакции реализованы с помощью хранимых процедур(англ. *Stored procedures*), на всех уровнях согласованности(тесты запускались с разными параметрами, в том числе на разных уровнях согласованности: сильная, ограниченное устаревание, сеанс, префикс и случайная) были замечены G2-item аномалии.

5.8.1 Обозначения для графиков

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

Чтение (англ. *read*)

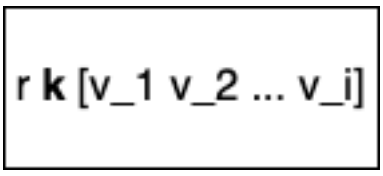

Рис. 5.2: Обозначение для операции чтения

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*)

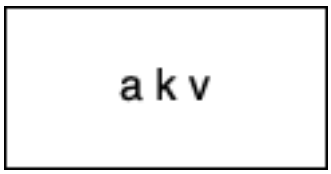

Рис. 5.3: Обозначение для операции добавления

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

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

Пример 1

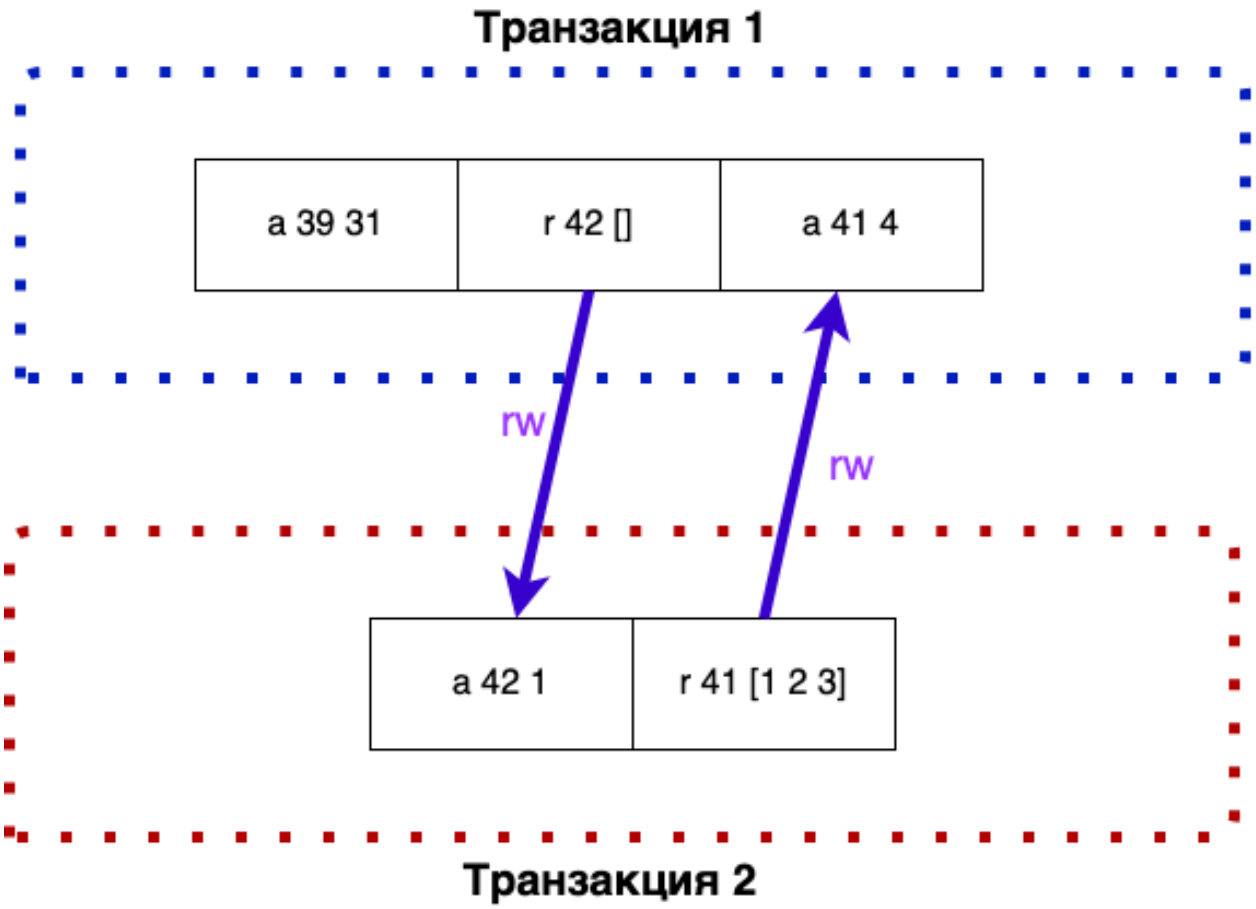

Рис. 5.4: G2-item

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

репликация — отключена

сбои — нет

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

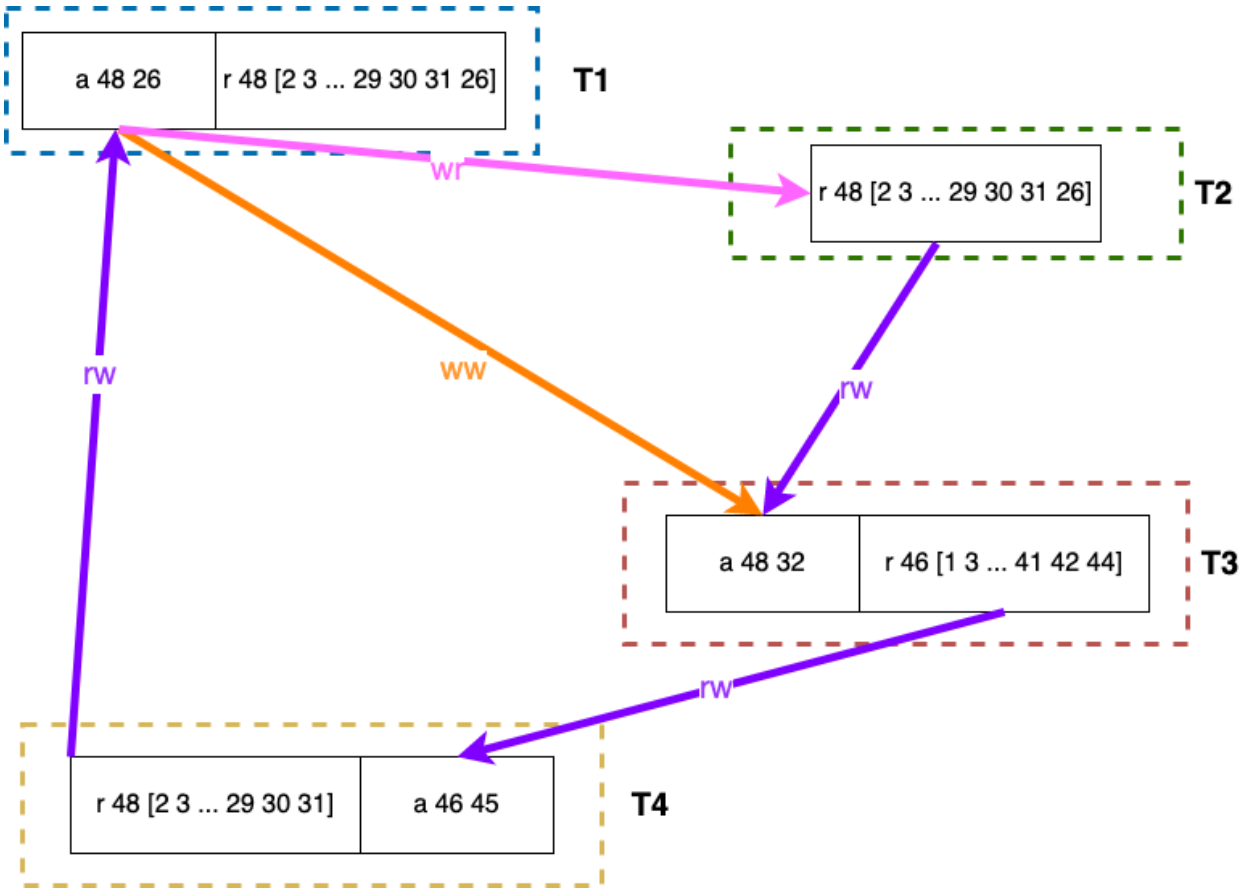

Рис. 5.5: G2-item

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

репликация — отключена

сбои — нет

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*** $\rightarrow$ 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*** $\rightarrow$ 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*** $\rightarrow$ 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*** $\rightarrow$ 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.

5.8.3 Анализ результатов

G2-item аномалия не редкость. Примерно в 15% транзакций наблюдались аномалии во время нормальной работы, без сбоев.

В документации Azure Cosmos DB сказано, что поддерживаемый уровень изоляции транзакций — **изоляция моментальных снимков**.

При *изоляция моментальных снимков* транзакциям, которые не записывают в один и тот же *id*, разрешено выполняться одновременно. То есть, полученные истории, по-видимому, не нарушают *изоляцию моментальных снимков*, но, тем не менее, демонстрируют циклические зависимости транзакций.

База данных Azure Cosmos DB соответствует заявленному уровню изоляции.

Заключение

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

Также сложность обнаружения различных нарушений изоляции 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 изоляцию моментальных снимков как уровень изоляции транзакций. Однако использование этих транзакций осложняется запутанной документацией и API. 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. Такое поведение корректно при изоляции моментальных снимков. А значит, база данных Azure Cosmos DB соответствует заявленному уровню изоляции.

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] База данных. "https://ru.wikipedia.org/wiki/%D0%91%D0%B0%D0%B7%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85".

[9] 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.

[10] 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.

[11] 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.

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

[13] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

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