УДК 004.652.6

МОДЕЛЬ ВЫБОРКИ, ИНДЕКСИРОВАНИЯ И ОЧЕРЕДИ ОБЪЕКТОВ В СУБД FOUNDATIONDB

Селиванов П.А., Гришунов С.С., Белов Ю.С.

Московский государственный технический университет имени Н.Э. Баумана, Калужский филиал, Калуга, e-mail: maslow.tema@yandex.ru

По мере того как программные системы продолжают распространяться в постоянно растущих масштабах, выходя за пределы географических, организационных и традиционных коммерческих границ, требования, предъявляемые к их коммуникационным инфраструктурам, будут возрастать в геометрической прогрессии. Современные системы работают в сложных средах с несколькими языками программирования, аппаратными платформами, операционными системами и требованиями к динамическим гибким развертываниям с надежностью 24/7, высокой пропускной способностью и безопасностью при сохранении высокого качества обслуживания (QoS). Требуется какое-то надежное решение для обеспечения всех этих параметров, для построения масштабируемой и отказоустойчивой системы. Цель данной статьи – дать более глубокое понимание работы индексов, выборок и работы с очередью. Исходный драйвер от разработчиков FoundationDB предоставляет минимальный набор функций, основные из которых – сохранить ключ-значение, выбрать значение по ключу, выбрать значения по области ключей, подписаться на область ключей. Для реализации приложения данного функционала недостаточно, поэтому требуется разработать модель для хранения, индексирования, быстрых выборок и работы с очередями, которая будет хорошо работать по времени и оптимально использовать ресурсы диска.

Ключевые слова: FoundationDB, FlatBuffer, индексирование данных, выборка объектов, очередь объектов, масштабирование, репликация, отказоустойчивость, серверные курсоры

MODEL FOR SELECTION, INDEXING, AND QUEUING OBJECTS IN DBMS FOUNDATIONDB

Selivanov P.A., Grishunov S.S., Belov Yu.S.

Bauman Moscow State Technical University, Kaluga branch, Kaluga, e-mail: maslow.tema@yandex.ru

As software systems continue to expand at an ever-increasing scale, transcending geographical, organizational, and traditional commercial boundaries, the demands placed on their communication infrastructures will increase exponentially. Modern systems operate in complex environments with multiple programming languages, hardware platforms, operating systems, and requirements for dynamic, flexible deployments with 24/7 reliability, high throughput, and security while maintaining high quality of service (QoS). Some reliable solution is required to provide all these parameters, to build a scalable and fault-tolerant system. The purpose of this article is to provide a deeper understanding of how indexes, samples, and queuing work. The original driver from the FoundationDB developers provides a minimal set of functions, the main ones are save key-value, select value by key, select values by key area, subscribe to key area. This functionality is not enough to implement the application, so you need to develop a model for storing, indexing, quick samples, and working with queues that will work well on time and optimally use disk resources.

Keywords: FoundationDB, FlatBuffer, data indexing, object selection, object queue, scaling, replication, fault tolerance, server cursors

FoundationDB – это распределенная база данных, предназначенная для обработки больших объемов структурированных данных в кластерах серверов. Она организует данные в качестве упорядоченных пар ключ-значение и использует транзакции ACID для всех операций. Эта база данных особенно хорошо подходит для нагрузок чтения/записи, но также имеет отличную производительность при интенсивных операциях записи [1].

Исходный драйвер от разработчиков FoundationDB предоставляет минимальный набор функций, основные из которых – сохранить ключ-значение, выбрать значение по ключу, выбрать значения по области ключей. Для реализации приложения данного функционала недостаточно, поэтому требуется разработать модель для хранения, индексирования, быстрых выборок и работы с очередями, которая будет хорошо работать по времени и оптимально использовать ресурсы диска.

Серверные курсоры. Чтобы преодолеть ограничение по величине и времени транзакции, выборка должна идти в индивидуальных микротранзакциях, сохраняя состояние в специальном объекте БД – курсоре. Перед началом выборки создается новый объект курсора со случайным идентификатором. Все курсоры должны считаться моделями и храниться в БД как отдельный тип данных [2].

Поля модели:

– идентификатор UUID;

– ID типа данных (2 байта);

– момент создания (для сборки мусора неуспешно закончившихся транзакций);

– срок жизни (в секундах);

– последний обработанный ключ (или значение 0x00, если выборка еще не началась).

Чтобы сделать структуру и поведение курсора универсальным между полным перебором и выборкой по индексу, предлагается использовать ID типа данных также и для индексов, нумеруя их как отдельный тип данных.

Предполагается, что создание курсора должно идти в отдельной независимой транзакции, предшествующей выборке, а идентификатор возвращается приложению. Он может использоваться как внутри приложения, так и снаружи. Например, возвращая ID курсора в фронтенд, клиент может использовать любой доступный сервер для продолжения выборки, что повышает отказоустойчивость [3].

Серверные курсоры также могут использоваться для организации постраничных выборок, но эта идея требует подробной проработки.

После использования курсор должен быть закрыт (а запись о нем в БД удалена). В случае ошибок обработки или некорректного использования, записи незакрытых курсоров могут быть проанализированы (для мониторинга системы – метрика корректности работы) и удалены сборщиком мусора.

Полный перебор с предикатом. Главное применение – выборки произвольного объема с любой сложностью условий, в том числе безусловные. Предикат – функция, принимающая на вход буфер объекта и возвращающая bool и error – решение, подходит элемент для выборки или нет. Для выборки только объектов определенного типа потребуется поле ключа «тип данных» (2 байта, описано выше).

type Predicat func(buf []byte) (bool, error)

Функция полного перебора должна принимать на вход контекст (для прерывания выборки), ID базы (2 байта), ID типа данных (2 байта), ID курсора (пустой, если новая выборка), функцию предиката для данного типа. Выходные параметры – канал буферов и ошибка. Канал нужен для того, чтобы а) сгладить разрывы между микротранзакциями запроса и б) корректно выйти в случае прерывания по контексту.

func SeqScan(ctx context.Context, db, objtype, cursor uint16, p Predicat) (<-chan [] byte, error)

Композитные индексы. Главное применение – быстрые выборки по условиям равенства одного или нескольких полей [4]. Эти индексы выбираются в пользу набора простых индексов с реализацией join по следующим причинам:

– индексы по индивидуальным полям менее рационально используют место на диске;

– реализация функции Join для двух индексов будет иметь сложность не менее O(NlogN) в случае использования хеш-функции, а в худшем случае O(N^2). Для трех и более индексов – полиномиально, в то время как сложность выборки по композиту – O(N);

– композитные можно использовать повторно. Например, индекс по полям (A, B, C) можно также использовать для выборок по (A, B) и (A).

Как правило, при проектировании функционала в приложении заранее известна хотя бы часть выборок, которые будут необходимы. Это позволяет некоторые индексы создавать с самого начала. А поддержка драйвером полного перебора с предикатом позволяет создать инструменты миграции и актуализации индексов после запуска приложения в производственную эксплуатацию.

При создании индекса предлагается идентифицировать их записи кодом типа данных, как и структуры, а элементы индекса разделять между собой, чтобы избежать смешения данных. Если не разделять значения, то смешение приведет к ошибке выборки.

Например, пусть есть поля A = "ав" и B = "", а у другого объекта A = "а" и B = "в". Тогда в обоих случаях запись индекса без разделителей будет равна "ав". Для разделения значений предлагается использовать два вида разделителей: 0x00 и 0xFF. Использование 0x00 после пустого значения в сортированной коллекции будет аналогично правилу NULLS FIRST, а 0xFF соответственно NULLS LAST.

Итого, структура записи индекса по полям (A NULLS LAST, B NULLS FIRST, C NULLS LAST) будет выглядеть так:

<ID базы><ID типа индекса><значение A>0xFF<значение B>0x00 <значение C>0xFF<UUID объекта> = ""

Поскольку операции с полями FlatBuffer'a достаточно легковесные, для инвалидации индекса предлагается использовать функцию определения индекса и хранение исходного буфера в памяти [5]. Функция определения – подобна функции предиката, принимает на вход ID базы, буфер объекта, а на выходе дает массив байт ключа индекса (по схеме, как показано выше) и ошибку. Такая функция позволит определять полное значение индекса, в том числе его перенос в другую БД.

Драйвер должен использовать функцию определения индекса дважды – сна-

чала передавая в него оригинальный (исходный) буфер объекта, а затем новый. Это позволит получить старое и новое значения ключа индекса без лишних выборок из БД. Затем драйвер должен удалить старое значение и записать новое. Важное ограничение – это должно быть сделано в той же транзакции, что и сохранение объекта, т.е. быть его неотъемлемой частью.

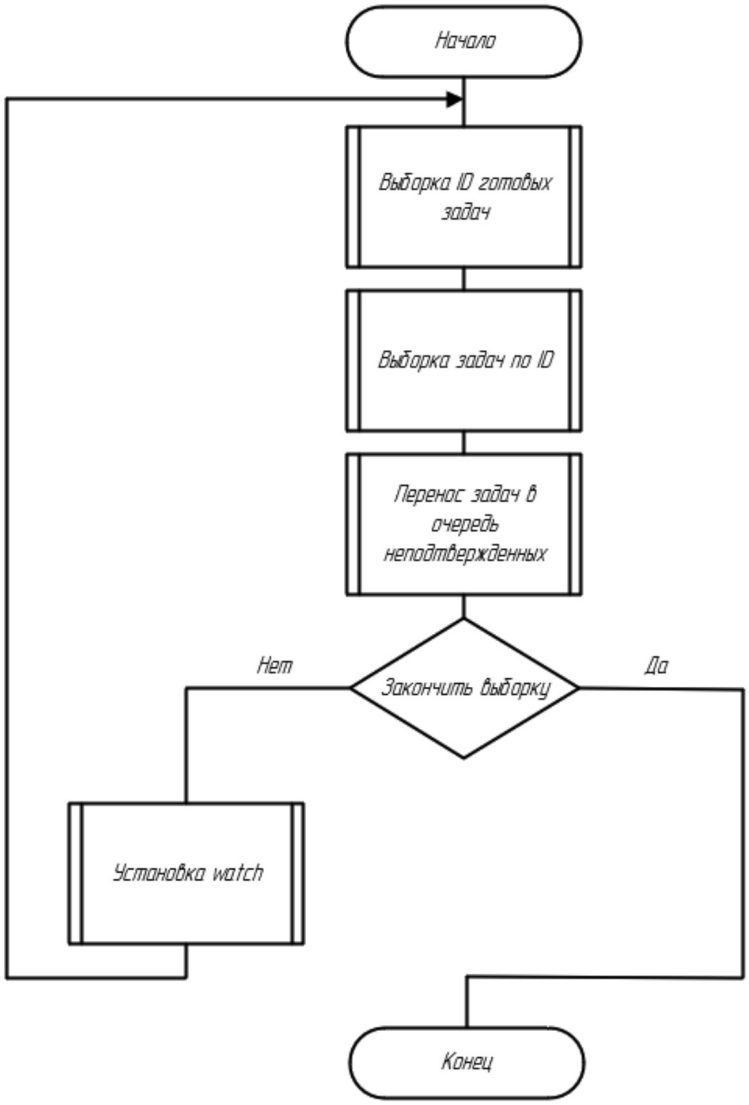
Модель очереди. Основная идея в том, что очередь – это специальный индекс, сортированный по дате, с отслеживанием изменения. Отслеживать легко с помощью инструмента watch клиента fdb.

Для удобства применения элемент очереди создается с моментом времени его предполагаемого выполнения. При этом механизм подписки должен возвращать только элементы с временем выполнения меньше текущего. Это позволит реализовать отложенные операции без лишних действий [6].

Для обеспечения гарантии обработки, при получении элемент очереди должен быть перемещен в особую коллекцию записей, которые еще не были подтверждены. Записи в этой коллекции не должны зависеть от времени, чтобы их можно было легко удалять без дополнительных данных.

Как и в случае типов данных и индексов, для идентификации очереди нужно использовать ID базы и ID типа данных. Для списка неподтвержденных элементов можно использовать тот же ID типа, но оградить его от основной очереди разделителем 0xFF.

Элемент очереди – это ID записи заранее известного приложению типа данных объектов, которые и являются реальными объектами задач.



Блок-схема алгоритма получения данных из очереди

Структуры элементов очереди:

● Обычный элемент. Выборка задач, готовых к выполнению: время задачи меньше или равно текущему времени

```
<ID базы><ID типа данных><Время выполнения задачи><ID объекта задачи> = ""
```

● Элемент списка неподтвержденных. Выборка неподтвержденных: 0xFF<задачи> меньше или равно 0xFF0xFF

```
<ID базы><ID типа данных>0xFF<ID объекта задачи> = ""
```

● // Элемент отслеживания изменений (для watch)

```
<ID базы><ID типа данных>0xFF0xFF = "<момент последнего обновления>"
```

Алгоритм получения данных из очереди:

1. Выбрать ID готовых задач (задачи меньше или равно текущее время). Либо согласно установленному лимиту, либо все готовые.
2. По списку ID выбрать сами задачи и вернуть их.
3. Переместить записи в список неподтвержденных.
4. Если требуется продолжать выборку, устанавливается watch.
5. Как только watch срабатывает, переход к п. 1.

Блок-схема алгоритма получения данных из очереди представлена на рисунке.

Исходя из алгоритма, функция получения элементов очереди должна принимать на вход ID базы, ID типа данных очереди, ID типа данных задач, а возвращать канал буферов объектов и ошибку. Объект очереди должен иметь возможность выдать все неподтвержденные задачи. Также объект очереди должен иметь функцию подтверждения получения задачи.

Возникает вопрос, абстрагировать ли работу с очередями от коллекций объектов? Если да, то работа везде идет только с обезличенными идентификаторами. Это упрощает код и вид реализации, но забота о загрузке и сохранении объектов ложится на вызывающую сторону. Отрицательный момент: поскольку метод возвращает канал, это будет стимулировать разработчиков организовывать загрузку объектов из БД по одному, что пагубно скажется на производительности [7].

Пример интерфейса:

```
// Конструктор конкретной очереди в конкретной базе
func NewQueue(db, qtype uint16) Queue {}

type Queue interface {
  Ack(uuid.UUID) error
  Pub(time.Time, uuid.UUID) error
  Sub() (<-chan uuid.UUID, error)
  Lost() ([]uuid.UUID, error)
}
```

С другой стороны, на стороне обработчика очереди Sub реально загрузка из БД идет порциями. Оптимистично можно предполагать, что размер порции больше одного объекта, что делает возможной пакетную загрузку объектов из БД на стороне очереди, но усложняет код и интерфейс взаимодействия, а также может оказаться потенциально лишней операцией, к примеру, если нужно получить только изменения в идентификаторах, без подгрузки объекта.

Пример интерфейса:

```
// Конструктор конкретной очереди в конкретной базе
func NewQueue(db, qtype, otype uint16) Queue {}

type Queue interface {
  Ack(uuid.UUID) error
  Pub(time.Time, uuid.UUID, []byte) error
  Sub() (<-chan []byte, error)
  Lost() ([][]byte, error)
}
```

Компромиссный интерфейс, объединяющий оба подхода, но из-за этого менее понятный и избыточный:

```
// Конструктор конкретной очереди в конкретной базе
func NewQueue(db, qtype, otype uint16) Queue {}

type Queue interface {
  Ack(uuid.UUID) error
  Pub(time.Time, uuid.UUID, []byte) error
  PubID(time.Time, uuid.UUID) error
  Sub() (<-chan []byte, error)
  SubIDs() (<-chan uuid.UUID, error)
  Lost() ([][]byte, error)
  LostIDs() ([]uuid.UUID, error)
}
```

Заключение

В результате исследования была реализована модель для выборок, индексирования объектов и работы с очередями, которая максимизирует скорости работы с неопределенно большими объемами объектов, минимизирует занимаемое место в БД, преодолевает лимиты FoundationDB на длительность и размер транзакций, минимизирует выделение памяти на индексы.

Список литературы

1. Селиванов П.А., Белов Ю.С. ОБЗОР FOUNDATIONDB // В сборнике избранных статей по материалам научных конференций ГНИИ «Нацразвитие»: материалы Международных научных конференций. 2020. С. 107–109.

2. Додонов Г.М., Чумак Б.Б. Динамика перехода от реляционной модели на NOSQL решения // Научный альманах 2018. С. 24–29.

3. Парамонов И.Ю., Смагин В.А., Косых Н.Е., Хомоненко А.Д. Методы и модели исследования сложных систем и обработки больших данных: монография / Под ред. В.А. Смагина и А.Д. Хомоненко. СПб.: Лань, 2020. 236 с.

4. Тарасов С.В. СУБД для программиста // Базы данных изнутри. М.: СОЛОН-Пресс, 2015. 320 с.

5. Эрик Р., Джим Р.У. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL / Под ред. Ж. Картер; пер. с англ. А.А. Слинкина. М.: ДМК Пресс, 2013. 384 с.

6. Маркин Е.И., Рябова К.М., Артюшина Е.А. Современные технологии NOSQL для реализации баз данных. 2017. С. 1240–1243.

7. Григорьев Ю.А., Плутенко А.Д., Бурдаков А.В., Цвященко Е.В. Анализ процессов согласования версий записей в базах данных NOSQL // Радиопромышленность. 2017. С. 125–134.