Разработка библиотеки для хранения временных рядов с помощью LSM дерева на языке программирования Go

Никита Томилов[0000−0001−9325−0356]

Университет ИТМО, Санкт-Петербург, Россия
mr@itmo.ru
http://micsecs.org/

Аннотация Благодаря недавнему росту популярности Интернета Вещей, количество данных, которые собираются, хранятся и передаются, также существенно возросло. Концепция "граничных устройств" (edge devices) позволяет буферизовать измерения, полученные с датчиков и представимые в виде временных рядов, чтобы избежать проблем в случае нестабильного соединения с Интернетом. Одним из путей такой буферизации данных на граничном устройстве в рамках запущенного на нем приложения является использование встраиваемой базы данных. Однако, такие базы данных могут иметь слабую производительность, особенно на встраиваемых компьютерах. Это приводит к большим задержкам доступа к данным, что вредно для критически важных систем. Поэтому в этой статье предлагается альтернативное решение по буферизации временных рядов, использующее структуру данных LSM-дерево. Данная статья описывает концепцию хранилища на основе LSM-дерева для буферизации временных рядов внутри приложения, написанного на языке программирования Go. Чтобы продемонстрировать эту концепцию, была разработана библиотека GoLSM. Также был произведен сравнительный анализ разработанного решения с традиционной встраиваемой базой данных SQLite. Исследование показало, что разработанная библиотека предоставляет более быстрый доступ на запись и чтение данных, чем SQLite, при условии, что отметки времени во временных рядов упорядочены по возрастанию, что типично для систем логирования данных.

Keywords: Временные ряды · LSM дерево · SST · Golang.
Перевод с английского языка.

1 Теория

1.1 Временные ряды

Обычно временной ряд - повторяющееся во времени измерение какого-либо параметра с указанием времени, когда измерение было произведено [1]. Временные ряды часто состоят из измерений, производимых через определенный постоянный интервал, но постоянность этого интервала не является

обязательной. Например, измерения температуры за последнюю неделю с указанием времени каждого измерения является временным рядом показаний температуры. Временные ряды широко используются в аналитических целях, включая машинное обучение и предиктивную аналитику. Конкретное измерение временного ряда может быть как непосредственно показанием датчика, так и каким-то вычисленным значением, и пока у этого значения есть отметка времени в каком-либо формате, массив таких данных может считаться временным рядом.

1.2 Данные IoT и граничные вычисления

В наши дни термин "временные ряды"широко известен благодаря недавнему росту индустрии Интернета вещей, IoT, и распространению IoT-устройств и систем, поскольку IoT-устройство часто используется именно для снятия каких-либо измерений в формате временного ряда (например, датчик, логирующий температуру). Часто такие данные передаются на сервер для целей аналитики и сбора статистики. Однако, в больших и сложных измерительных системах, применяемых в "производственном"интернете вещей (industrial IoT), количество генерируемых данных приводит к различным проблемам передачи и хранения этих данных. Популярным методом улучшения такой системы является внедрение так называемых граничных устройств (edge devices) [2]. Традиционный подход с использованием этих устройств приведен на рисунке 1.

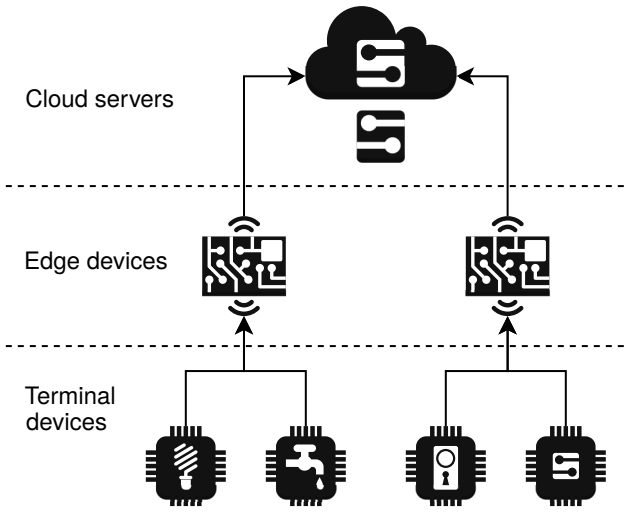


Рис. 1: Архитектура сложной системы.

Такая архитектура дает гибкость сбора данных. В случае проблем с соединением между конечным устройством и облаком, данные могут быть бу-

феризованы на граничном устройстве и переотправлены позднее, когда подключение восстановится. Следовательно, такое граничное устройство должно обладать возможностью сбора и хранения данных от конечных устройств на случай таких проблем. Однако, такие граничные устройства не обладают достаточной аппаратной мощностью для использования полноценной СУБД, поэтому возможность буферизации данных должна быть встроена в одно из тех приложений, которые уже запущены на этом граничном устройстве. Следовательно, появляется необходимость во встраивании системы хранения временных рядов в подобные приложения. Здесь следует сделать предположение о том, что подобное приложение написано на языке программирования Go, потому что этот язык является сбалансированным с точки зрения сложности разработки, будучи более простым, чем C/C++, и с точки зрения производительности, будучи менее требовательным к ресурсам, чем Python. Также этот язык программирования был выбран потому, что он является стандартом в той предметной области, в которой работает автор этой статьи.

1.3 Встраиваемые системы хранения

Традиционно, для хранения данных в приложении используется либо какая-нибудь структура данных, располагающаяся целиком в оперативной памяти, либо встраиваемая СУБД. Однако в случае, когда важно не терять данные в случае отключения питания, структуры данных, хранящиеся в оперативной памяти, не подходят, а встраиваемые СУБД гораздо медленнее с точки зрения доступа к данным. В статье описывается подход хранения данных, использующий структуру под названием LSM-дерево. Такой подход похволяет добиться хранения данных на диске, не сильно теряя в производительности по сравнению с in-memory структурами данных.

2 Реализация

2.1 LSM-дерево

LSM-дерево (log-structured merge-tree) - это структура данных «ключ-значение» с хорошими характеристиками производительности. Это хороший выбор для обеспечения индексированного доступа к файлам данных, таким как данные журнала транзакций или данные временных рядов. Дерево LSM хранит данные в двух или более структурах. Каждая из них оптимизирована для конкретного типа памяти, на котором она хранится, а синхронизация между слоями выполняется поблочно [3].

Простое дерево LSM состоит из двух уровней с именами C_0 и C_1. Основное различие между этими уровнями состоит в том, что обычно C_0 представляет собой структуру данных в оперативной памяти, а C_1 хранится на жестком диске. Следовательно, C_1 обычно больше, чем C_0, и когда объем данных в C_0 превышает некоторый предел, данные перемещаются в C_1. Чтобы поддерживать подходящую производительность, обычно рекомендуется

оптимизировать как C_0, так и C_1 для конкретных условий применения. При этом данные рекомендуется переносить между слоями наиболее эффективно, возможно, с использованием алгоритмов, похожих на сортировку слиянием.

Для сохранения эффективности сохранения данных в уровне C_1, было решено использовать SSTable в качестве уровня C_1 и B-дерево для C_0. SSTable, или sorted string table - это файл, содержащий пары ключ-значение, отсортированные по ключу [4]. Использование SSTable для хранения данных временных рядов обычно является хорошим решением, если данные передаются из системы мониторинга или даталоггинга. Из-за этого они сортируются по отметке времени, которая является хорошим кандидатом на ключ, значением же в SSTable может быть само измерение. SSTable всегда является неизменяемой структурой данных, что означает, что данные нельзя напрямую удалить из файла; измерения необходимо помечать «удаленными», а затем пропускать эти данные в процессе уплотнения SST-файла (compaction). Процесс уплотнения также используется для удаления устаревших данных, если для них предполагается определенный срок хранения (expiration).

2.2 Журнал фиксации

Любое приложение, которое используется для работы с критически важными данными, должно иметь возможность постоянного сохранения данных в случае отключения электроэнергии или любого другого неожиданного завершения работы приложения. Чтобы поддерживать эту возможность, используется механизм журнала фиксации (commitlog). Это доступный только для записи журнал любых вставок в базу данных. Он записывается перед добавлением данных в файлы данных базы данных. Этот механизм обычно используется в любых реляционных или не связанных СУБД.

Поскольку встраиваемая система хранения данных также должна поддерживать эту возможность, было необходимо реализовать журнал фиксации вместе с деревом LSM. Чтобы гарантировать, что записи в этом журнале фиксации сохраняются в дисковом хранилище, был использован системный вызов fsync, который отрицательно повлиял на производительность полученной системы хранения.

2.3 Разработанная библиотека

Для реализации функции хранения данных временных рядов в приложении, написанном на Go, была разработана библиотека GoLSM. Она предоставляет механизмы для сохранения и извлечения данных временных рядов и использует двухуровневое LSM-дерево, а также механизм журнала фиксации для хранения данных на диске. Архитектура этой библиотеки представлена на рисунке 2.

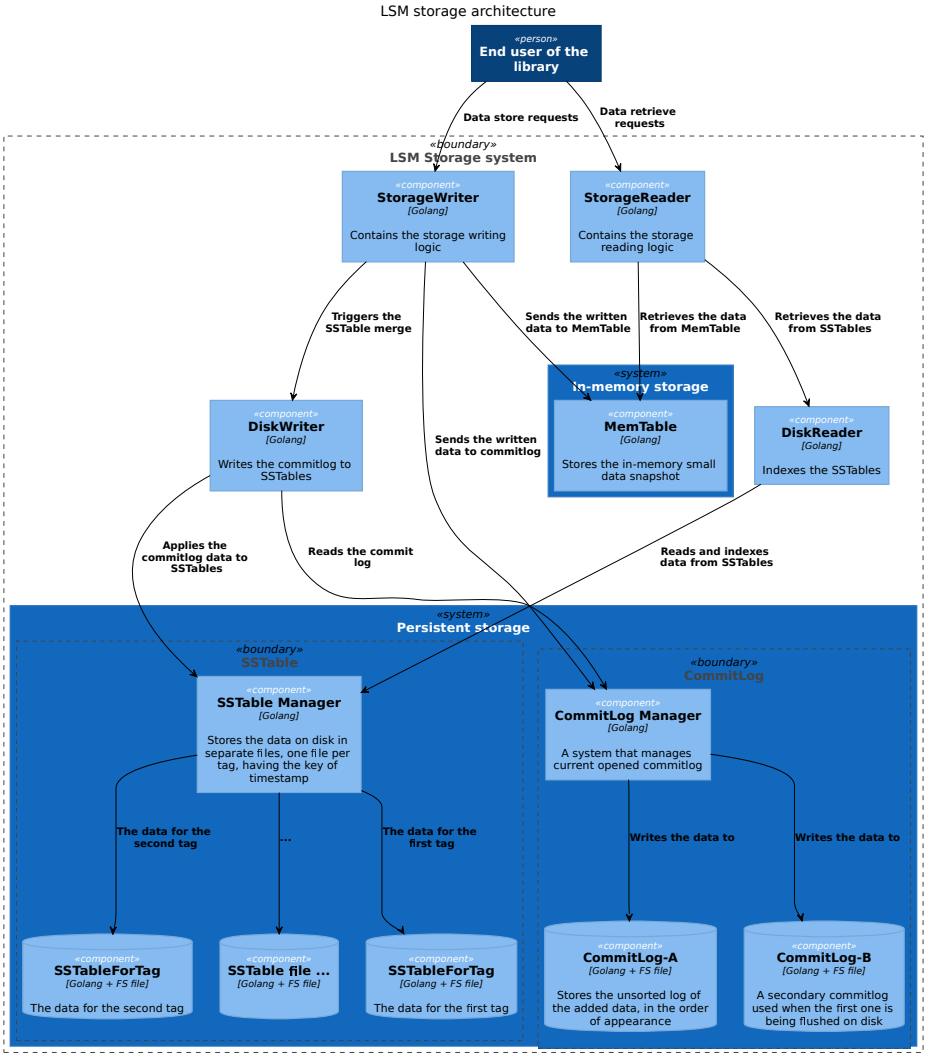


Рис. 2: Архитектура библиотеки GoLSM.

Поскольку эта библиотека изначально разрабатывалась для определенной предметной области и конкретного использования, у нее есть ряд ограничений. Например, у неё нет функций для удаления данных; вместо этого предполагается сохранить измерение с определенной датой истечения срока действия, после чего данные будут автоматически удалены в процессе уплотнения. Данные, которые хранятся с помощью GoLSM, должны состоять из одного или нескольких измерений; каждое измерение представлено именем тега, которое может быть идентификатором датчика или измерительного устройства, отметкой времени (origin), которая является отметкой, когда

измерение было записано, и непосредственно значением измерения, которое сохраняется в виде массива байтов. Этот массив байтов может различаться по размеру, что усложняет процедуру сохранения каждого измерения.

Как видно из рисунка, система хранения состоит из двух уровней: уровня в памяти и уровня постоянного хранения. Уровень в памяти основан на реализации B-дерева от Google [5]. Он хранит небольшую часть данных настраиваемого размера. Уровень хранения состоит из диспетчера журналов фиксации и диспетчера SSTable. Диспетчер журнала фиксации поддерживает два файла журнала фиксации; в то время как один используется для записи текущих данных, другой используется для добавления ранее записанных данных в файлы SSTable, которыми управляет SSTable Manager. Каждый файл SSTable содержит один тэг, а также специальный индекс в памяти, который также основан на B-дереве. Этот индекс используется для ускорения извлечения данных из SSTable, когда запрошенный временной диапазон больше, чем то, что хранится на уровне в памяти.

3 Сравнение с SQLite

3.1 Методология сравнения

Для сравнения LSM-решения с SQLite была разработана простая система хранения на основе SQLite. Она использует базу данных, состоящую из двух таблиц. Первая таблица называется Measurement и используется для хранения измерений. Каждое измерение представлено своим ключом, временной меткой и значением, а ключ - это первичный идентификатор объекта MeasurementMeta, который хранится во второй таблице. Этот объект хранит имя тега измерения; поэтому можно выполнять поисковые операции с фильтрацией по числовому столбцу вместо текстового столбца. В таблице измерений есть индексы как для ключевого столбца, так и для столбца временной метки.

Для следующих тестов использовалась придуманная синтетическая система. Эта система имеет 10 датчиков, которые производят данные с частотой дискретизации 1 Гц. Для тестов чтения данные для этих 10 тегов были сгенерированы для трех часов измерений. Следовательно, в базе данных SQLite содержатся 108000 записей или точек, или 10800 точек на каждый тег. Библиотека LSM также содержит 108000 записей, разделенных на 10 файлов для каждого тега, по 10800 точек в каждом. Для тестов записи данные для этих 10 тегов генерируются для различных временных диапазонов и затем сохраняются в обоих механизмах хранения.

Для тестирования обоих механизмов хранения использовался стандартный механизм тестирования Go. Он запускает целевой код несколько раз, пока тестовая функция не проработает достаточно долго, чтобы время ее выполнения можно было надежно измерить [6]. Этот механизм дает результат, который измеряется в ns/op, наносекундах на итерацию; для данных тестов итерация - это один вызов чтения или записи в хранилище.

Все тесты выполнялись на ПК с Intel Core i5-8400 под управлением Ubuntu 18.04, файлы данных для обоих хранилищ хранились на твердотельном накопителе NVMe.

3.2 Чтение всего диапазона данных

Этот тест измеряет время считывания для различных временных диапазонов, в то время как операция выполняется для полных трех часов измерений. Результаты этого теста представлены на рисунке 3. Как видно, GoLSM примерно в два раза быстрее SQLite по запросам на чтение. Стоит отметить, что уровень C_0 LSM-дерева не играет существенной роли в ускорении извлечения данных, поскольку временной диапазон выбирается случайным образом из всех трех часов измерений, в то время как уровень C_0 содержит только последние две минуты измерений.

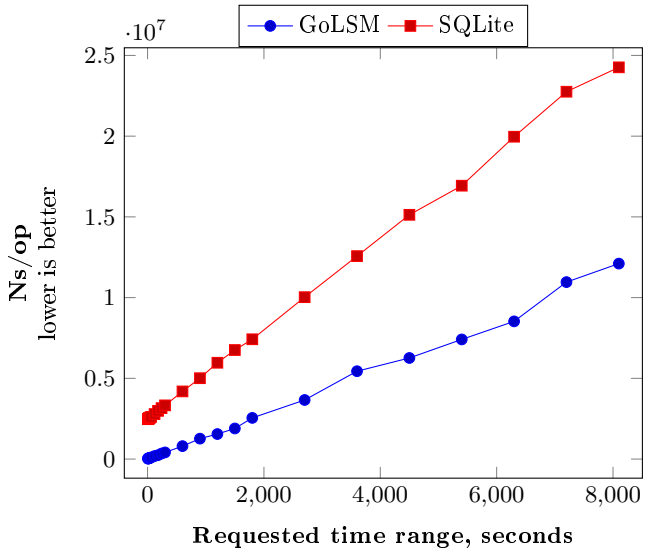


Рис. 3: Чтение всего диапазона данных.

Как видно, хранилище LSM до 50 раз быстрее, чем SQLite в небольшом диапазоне запросов (51251 нс/операцию для GoLSM и 2517353 нс/операцию для SQLite в диапазоне 25 секунд) и до двух раз быстрее в большом диапазоне запросов (12106688 нс/оп и 24262132 нс/оп в диапазоне 2 ч 15 мин). Эта большая разница может быть вызвана использованием индекса в памяти для каждого файла SST, который работает лучше, чем индексация в SQLite. Повышение эффективности для малого диапазона запросов может быть вызвано тем фактом, что малый запрошенный диапазон с большей

вероятностью умещается в пределах уровня C_0 дерева LSM, поэтому медленное извлечение из SST не вызывается.

3.3 Чтение за последние три минуты

Этот тест измеряет время считывания для различных временных диапазонов, в то время как операция выполняется для последних трех минут измерений. Результаты этого теста показаны на рисунке 4. Поскольку уровень C_0 LSM-дерева содержит последние две минуты измерений, более вероятно, что запрос будет соответствовать уровню C_0 без необходимости запрашивать данные с уровня C_1. Таким образом, извлечение данных в два раза быстрее, чем извлечение данных за все три часа измерений, как показано на рисунке 5.

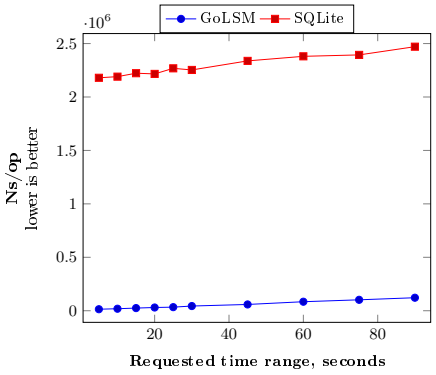


Рис. 4: Чтение всего диапазона данных

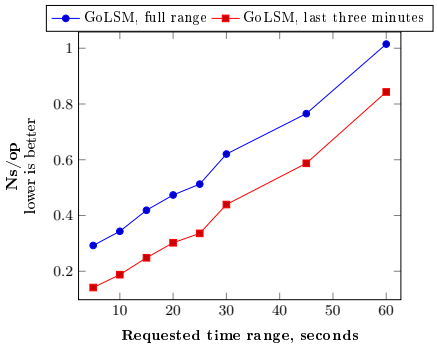
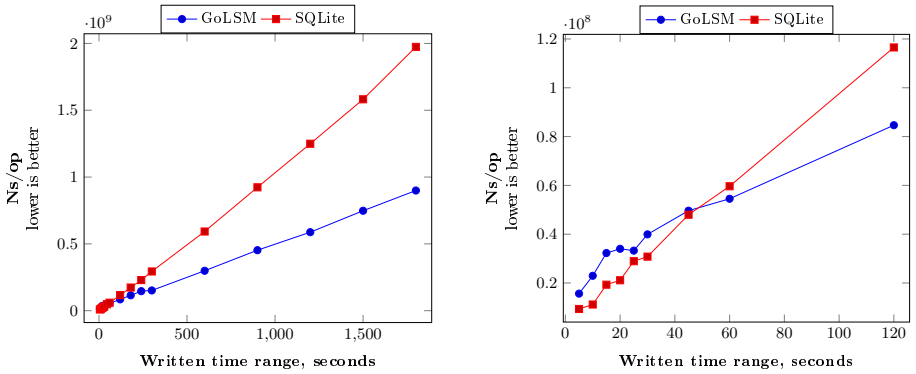


Рис. 5: Разница в скорости для разных интервалов.

Если в системе предполагается чтение только самых последних данных, увеличение емкости C_0 может значительно улучшить производительность чтения и снизить нагрузку чтения на постоянное хранилище диска.

3.4 Запись упорядоченных данных

Этот тест измеряет время записи для различных временных диапазонов. В то время как данные являются упорядоченными, минимальная временная метка следующего записываемого пакета данных всегда больше, чем максимальная временная метка ранее записанного пакета. Это означает, что для каждого SST-файла в GoLSM данные добавляются только в конец SST-файла без необходимости повторять сортировку SST-файла. Результаты этого теста представлены на рисунке 6.



(а) Для большого диапазона (б) Для маленького диапазона

Рис. 6: Запись упорядоченных данных.

На рисунке 6, (а) видно, что GoLSM записывает данные в два раза быстрее, чем SQLite, в течение большого записываемого диапазона времени. Однако на рисунке 6, (б) для небольшого временного диапазона GoLSM на самом деле медленнее. Причина этого в том, что для LSM измерялось время до момента фактической записи данных в журнал фиксации, а не непосредственно в файлы SST; и когда количество записей в журнале фиксации достигает определенного порога, оно должно быть передано в SST, что вызывает скачки времени записи. Однако для записываемых больших временных диапазонов эти накладные расходы не важны по сравнению с обычно медленными пакетными вставками в SQLite.

Стоит отметить, что для вставки в SQLite необходимо составить оператор SQL Insert, который включает преобразование значения типа double в строку. Эта операция выполняется очень медленно для больших пакетов вставок (пакет из 10 тегов за 30 минут составляет 18000 точек), и использование ORM, такого как GORM, не улучшает ситуацию.

3.5 Запись неупорядоченных данных

Этот тест измеряет время записи для различных временных диапазонов, но, поскольку данные являются неупорядоченными, нет никаких гарантий, что все временные метки следующего пакета больше, чем временные метки ранее записанного пакета. Таким образом, пакеты могут быть получены до или после друг друга с точки зрения их временных меток. Это означает, что каждый раз, когда данные передаются из файла журнала фиксации в файлы SST, библиотека должен пересортировать весь файл SST, замедляя процесс записи. Результаты этого теста представлены на рисунке 7. Сравнение записи данных в LSM, когда данные являются упорядоченными или неупорядоченными, доступно на рисунке 8.

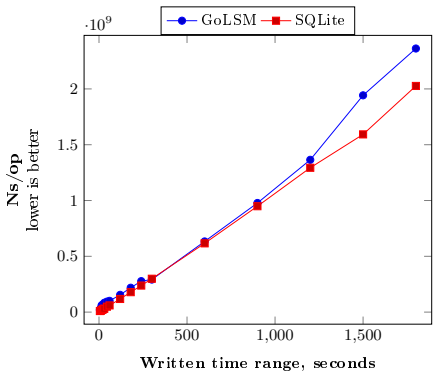


Рис. 7: Запись неупорядоченных данных.

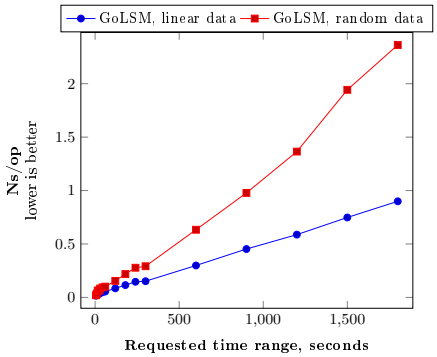


Рис. 8: Разница во времени при записи упорядоченных и неупорядоченных данных.

Как видно на рисунке 7, накладные расходы на пересортирование файлов SST при каждой записи данных настолько велики, что они занимают больше времени, чем медленные пакетные вставки в SQLite. Сравнение записи данных с этими накладными расходами с записью данных без накладных расходов показано на рисунке 8. Когда временные метки данных всегда линейно увеличиваются, это показывает, что вставки почти в три раза медленнее при записи в большом временном диапазоне (747870163 нс/операцию, или 747 мс, для записи 25 минут линейных данных и 1942545058 нс/операцию, или 1942 мс, для записи 25 минут случайных данных).

4 Заключение

Как показано в статье, разработанная библиотека предоставляет более быструю запись и чтение данных временных рядов по сравнению с SQLite. При этом сохраняется возможность не терять данные в случае непредвиденного отключения питания благодаря механизму коммитлогов. В лучшем случае, эта библиотека работает в 50 раз быстрее, чем SQLite, благодаря уровню хранения C_0, который фактически используется как быстрый кэш. Также, эта библиотека работает до двух раз быстрее чем SQLite в операциях записи, благодаря способу организации записи на диск. Однако, это преимущество в скорости записи имеется только в случае, если во временных рядах отметки времени у измерений линейно упорядочены по возрастанию, то есть у следующего пакета данных все отметки времени больше, чем у предыдущего пакета данных. Если это условие не выполняется, запись может быть в несколько раз замедлена, что делает разработанное решение более медленным, чем SQLite. Однако, несмотря на это, преимущество в более быстром чтении из хранилища сохраняется. Управляя размером уровня C_0 можно добиться необходимого баланса скорости чтения и используемого объема оперативной памяти.

В случае использования данного решения в условиях логирования данных, когда приложение должно буферизовать данные, поступающие от датчиков, отметки времени всегда будут возрастать. Это убирает потенциальную проблему записи "случайных"по времени данных и делает разработанное решение хорошей альтернативой SQLite, ускоряющей работу с данными.

Потенциальные улучшения разработанной библиотеки включают в себя механизм разделения одного большого SST файла на несколько маленьких, что позволило бы пересортировывать эти файлы быстрее, и механизм более гибкой настройки уровня C_0, чтобы размер этого уровня был разным для разных тэгов, для случаев, когда одни тэги записываются гораздо чаще или, наоборот, реже, чем другие.

Список литературы

1. Dunning, T., Friedman, E.: Time Series Databases: New Ways to Store and Access Data. O'Reilly Media, CA, USA (2014)
2. B. Chen, J. Wan, A. Celesti, D. Li, H. Abbas and Q. Zhang, "Edge Computing in IoT-Based Manufacturing,"in IEEE Communications Magazine, vol. 56, no. 9, pp. 103–109 (2018). https://doi.org/10.1109/MCOM.2018.1701231
3. O'Neil, P., Cheng, E., Gawlick, D. et al. The log-structured merge-tree (LSM-tree). Acta Informatica 33, 351–385 (1996). https://doi.org/10.1007/s002360050048
4. Adhikari, M., Kar, S. NoSQL databases. In Handbook of Research on Securing Cloud-Based Databases with Biometric Applications (pp. 109-152). IGI Global. (2015)
5. BTree implementation for Go, https://github.com/google/btree. Last accessed 31 Oct 2020
6. Go benchmark documentation, https://golang.org/pkg/testing/. Last accessed 1 Nov 2020