Реализация random write cache на основе красно-черных деревьев

Смирнов М.А., СПбГУ, sm_m_a@mail.ru

Введение

Рост объёмов данных, возросшие требования к надёжности хранения и быстродействию доступа к данным стали причинами возникновения систем хранения данных (СХД). Правильной организацией работы и структуры этой системы можно улучшить её быстродействие.

Целью данной работы была оптимизация работы кэша СХД, с использованием красно-чёрных деревьев. В ходе работы СХД выделяют 2 типа нагрузки: последовательная, когда блоки памяти следуют непосредственно друг за другом, и случайная, когда невозможно предсказать расположение блоков памяти относительно друг друга. Определением типа нагрузки занимаются уже существующие детекторы. Последовательная запись эффективна, вследствие механической природы жёсткого диска, потому появилась задача о преобразовании случайной записи в последовательную.

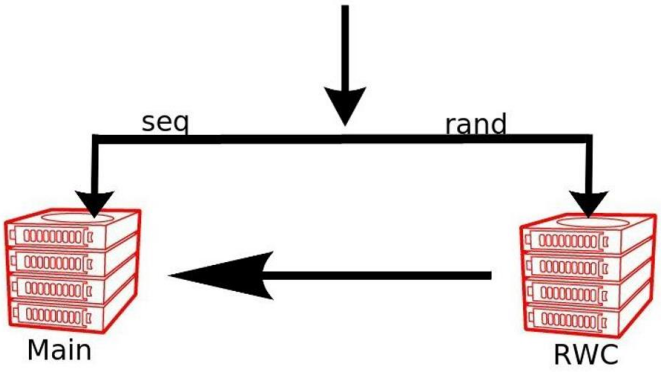

Рисунок 1: Организация работы с последовательными и случайными данными.

На рис 1. видно, что последовательные данные записываются на основной том (Main), а случайные записываем на вспомогательный (Random Write Cache), с сохранением их исходной позиции на основном томе. Запись продолжается в таком ключе до момента заполнения RWC, либо пока идёт сильная нагрузка. Когда нагрузка ослабевает или

вспомогательный том заполнен, данные вытесняются из RWC на Main.

Реализация

На данный момент вспомогательный том реализован по типу хэш-таблицы, поэтому вытеснение на основной том будет происходить не последовательно, а это противоречит основной идеи работы. Вследствие было принято решение о построении RWC на базе красно-чёрного дерева, что даст нам последовательную запись на всех этапах работы и ускорение всей системы в целом. Для полноценной работы красно-чёрного дерева понадобятся следующие методы: добавления, вытеснения и корректировки значений. Корректировка значений используется как при случайной нагрузке, так и при последовательной. Новые данные могут пересекаться с существующими, и, в случае пересечения, необходимо удалить или обновить устаревшую информацию в дереве.

Добавление элемента в RBTree

Ключом при добавлении в дерево будет положение блока памяти на основном томе, в процессе могут возникнуть четыре ситуации пересечения данных:

1. Значение нового элемента дерева пересекается с существующим слева.

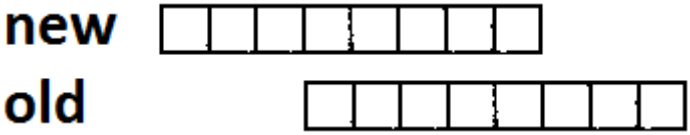

Рисунок 2: Добавление в дерево нового блока (1 случай)

Реализация: Удаляем неактуальные данные у старого элемента и продолжаем добавлять в RBTRee.

2. Значение нового элемента дерева пересекается с существующим справа.

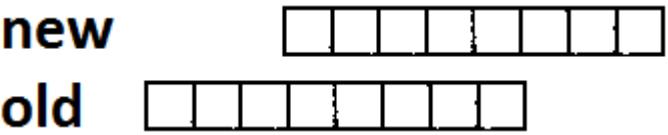

Рисунок 3: Добавление в дерево нового блока (2 случай)

Реализация: Удаляем неактуальные данные у старого элемента и

продолжаем добавлять в RBTRee.

3. Значение нового элемента дерева включается в существующее.

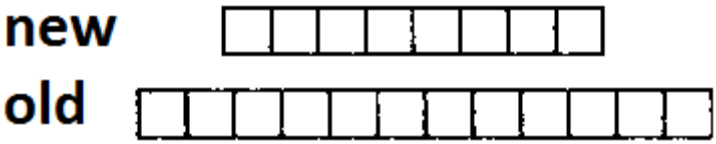

Рисунок 4: Добавление в дерево нового блока (3 случай)

Реализация: Разделяем старый элемент на два, исключая неактуальные данные. Левую часть оставляем на прежней позиции, новый элемент и правую часть продолжаем добавлять в RBTRee.

4. Значение старого элемента дерева включается в новый.

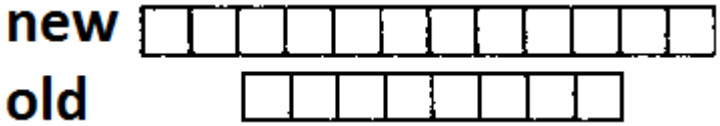

Рисунок 5: Добавление в дерево нового блока (4 случай)

Реализация: Удаляем старый элемент дерева. Новый элемент ставим на данную позицию и корректируем данные в RBTRee.

Вытеснение элемента из RBTree

В данном подходе использовался существующий метод получения первого по ключу элемента из Red-black Tree. Так как ключом при добавлении была позиция блока на основном томе, то вытеснение из дерева будет последовательным.

Корректировка значений в RBTree

Корректировка значений во многом схожа с добавлением. Единственное отличие – отсутствие непосредственного добавления в красно-чёрное дерево.

Сравнение с хэш-таблицей

Для сравнения эффективности работы красно-чёрного дерева и существующей хэш-таблицы был проанализирован набор из 800 000 записей и взята хэш-таблица, имеющая 200 000 ячеек, чтобы количество коллизий было невысоким. Сравнение с существующим решением проводилось по следующим метрикам: скорость добавления, скорость

вытеснения данных, скорость добавления с включенным вытеснением, скорость корректировки.

Для улучшения восприятия информации суммировалось время ста добавлений и десяти вытеснений.

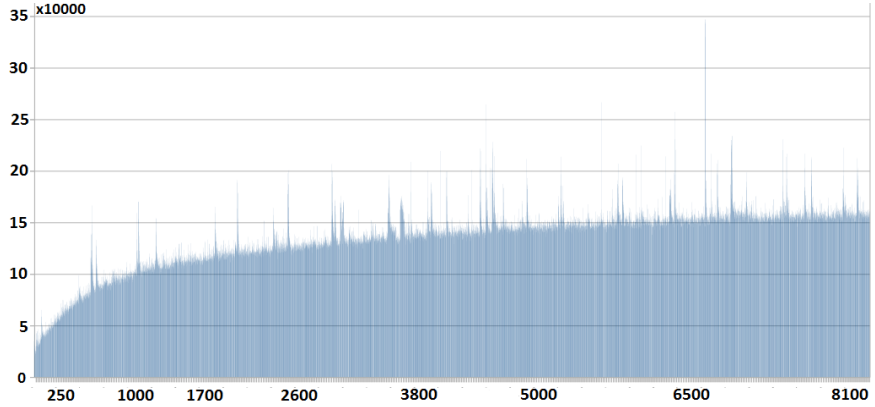

Рисунок 6: Добавление в RBTRee.

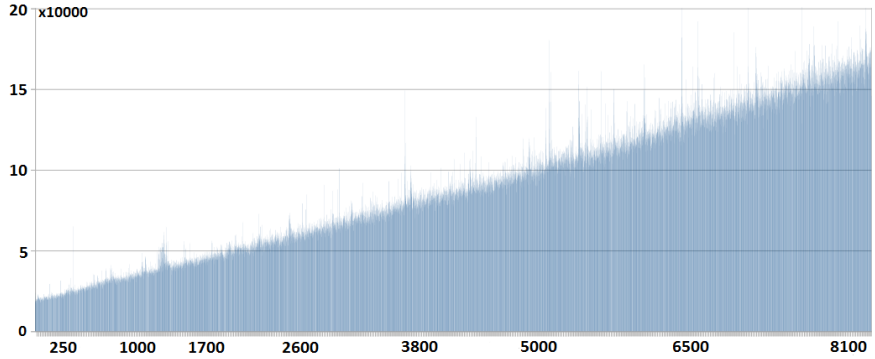

Рисунок 7: Добавление в хэш-таблицу.

Как видно на рисунках 6 и 7, на добавление ста элементов в красно-чёрное дерево затрачивается 50 000 наносекунд уже примерно при наличии 25 000 элементов, в то время как хэш-таблица тоже время начинает тратить на добавление только при наличии примерно 200 000 элементов. Но время добавления в хэш-таблицу возрастает линейно, а в RBTree по логарифму и под конец хэш-таблице необходимо примерно 170 000 наносекунд на добавление ста элементов, а красно-чёрному дереву чуть больше 150 000 и разница будет увеличиваться.

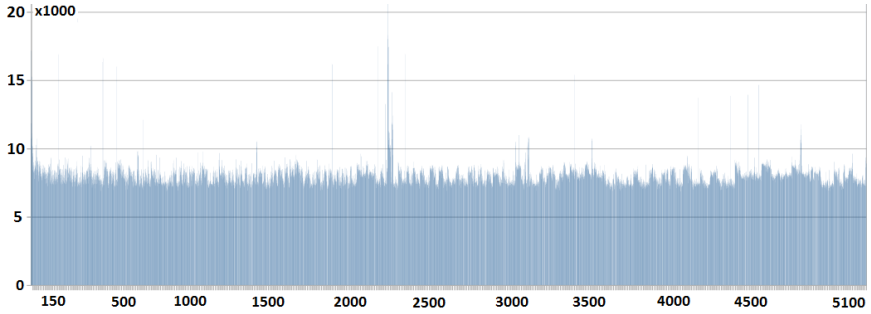

Рисунок 8: Вытеснение из RBTree.

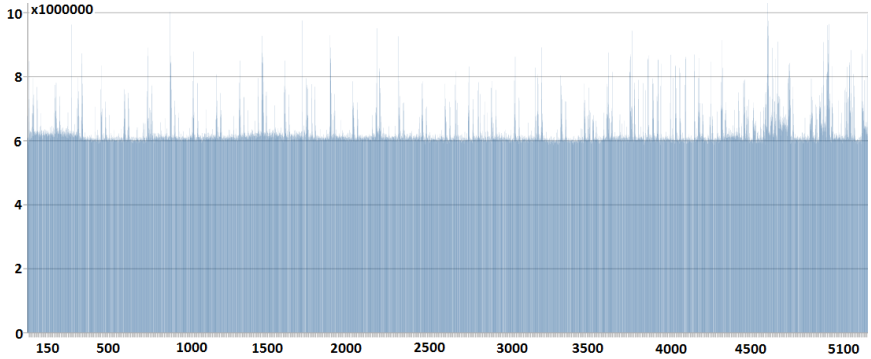

Рисунок 9: Вытеснение из хэш-таблицы.

Из рисунков 8 и 9 следует, что вытеснение из красно-чёрного дерева происходит примерно в 600 раз быстрее. Это вызвано необходимостью сортировать данные в хэш-таблице перед вытеснением.

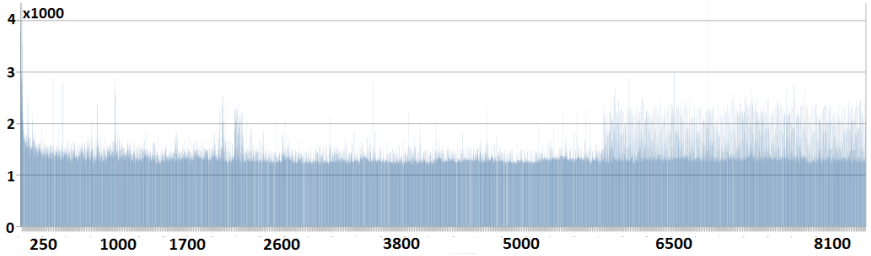

Рисунок 10: Добавление в RBTree с вытеснением.

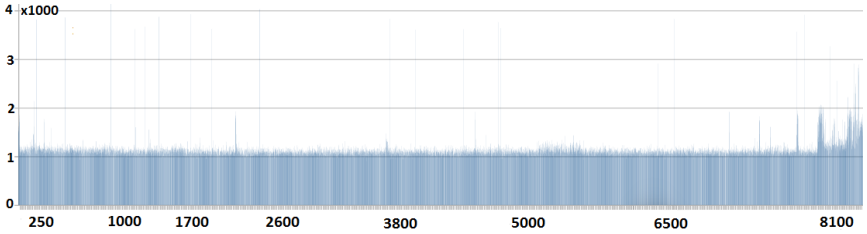

Рисунок 11: Добавление в хэш-таблицу с вытеснением.

На рисунках 10 и 11 видно, что скорость добавления с включенным вытеснением по мере заполнения структуры, различается примерно на 30%.

Заключение

В документе были представлены два варианта реализации вспомогательного тома для случайных записей: на основе хэш-таблицы и красно-чёрного дерева. Было проведено сравнение по следующим метрикам: скорость добавления, скорость вытеснения определённого количества элементов, добавление с вытеснением при достаточном заполнении. В результате оказалось, что RWC основанный на RBTree выигрывает в скорости вытеснения примерно в 600 раз, выигрывает в скорости добавления, когда количество элементов становится достаточно большим, но проигрывает в скорости добавления, с включенным вытеснением примерно на 30%.