

**ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

ĐH * ĐHTT



**BÁO CÁO ĐỒ ÁN CUỐI KỲ
CHỦ ĐỀ: LOG-STRUCTURED MERGE-TREE**

Sinh viên thực hiện:		
STT	Họ và tên	MSSV
1	Vũ Nguyễn Nhật Thanh	19522246
2	Nguyễn Phi Long	19521791

TP. HỒ CHÍ MINH - 07/2021

Mục Lục

PHẦN 1: LOG-STRUCTURED MERGE-TREE	3
1. Lịch sử của Log-Structured Merge-Tree	3
2. Giới thiệu về Log-Structured Merge-Tree:	3
2.1. Log-Structured Merge-Tree (LSM tree) là gì?	3
2.2. Độ khuếch đại đọc – Read Amplification (RA)	4
2.3. Độ khuếch đại ghi – Write Amplification (WA)	4
2.4. Độ khuếch đại không gian – Space Amplification (SA)	4
2.5. Cách thức hoạt động	4
2.5.1. Memtable	5
2.5.2. SSTable	6
2.5.3. Bloom filter	7
2.5.4. Merge và compaction	7
2.5.5. Các thao tác của LSM tree	9
2.6. Ưu và nhược điểm	10
PHẦN 2: THỰC NGHIỆM	11
3. Mô tả thực nghiệm:	11
3.1. Cấu trúc cây thư mục:	11
3.2. Cấu hình máy test	15
3.3. Quá trình thực hiện:	17
3.3.1. Open	18
3.3.2. Insert	18
3.3.3. Search	18
3.3.4. Update	19
3.3.5. Delete	19
4. Kết quả và kết luận:	19
4.1. Open	20
4.2. Insert	22
4.3. Search	24
4.4. Update	26



4.5. Delete.....	28
5. So sánh với Btree.....	30
5.1. Create database.....	31
5.2. Open	33
5.3. Insert	35
5.4. Search.....	37
5.5. Update	39
5.6. Delete.....	41
5.7. Kết luận.....	43
6. Tổng kết	43
TÀI LIỆU THAM KHẢO.....	44



PHẦN 1: LOG-STRUCTURED MERGE-TREE

1. Lịch sử của Log-Structured Merge-Tree

Kiến trúc LSM Tree index lần đầu được công bố bởi Patrick O'Neil và cộng sự vào năm 1996 dưới cái tên Log-Structured Merge-Tree (LSM tree).

LSM tree là cấu trúc dữ liệu cốt lõi đằng sau nhiều cơ sở dữ liệu, bao gồm WiredTiger, BigTable, Apache Cassandra, ScyllaDB và RocksDB, LevelBD, SQLite4.

2. Giới thiệu về Log-Structured Merge-Tree:

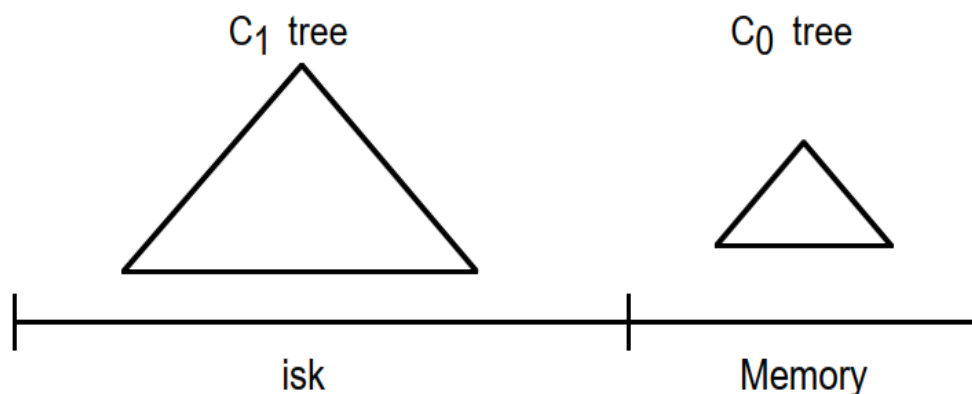
2.1. Log-Structured Merge-Tree (LSM tree) là gì?

Log - Structured Merge - Tree (LSM tree) là một cấu trúc dữ liệu tối ưu hóa hiệu suất ghi vào bộ nhớ ngoài, việc ghi được tối ưu hóa bằng cách chỉ thực hiện ghi tuần tự.

LSM tree không phải là một cấu trúc dữ liệu đơn mà là sự kết hợp của nhiều cấu trúc dữ liệu để tận dụng thời gian đáp ứng của các thiết bị lưu trữ của hệ thống.

LSM tree có đơn giản nhất có 2 level. LSM tree 2 cấp bao gồm 2 cấu trúc tương tự cây (C0, C1):

- C0 bé hơn và nằm hoàn toàn trong RAM (memtable).
- C1 thì nằm trên đĩa cứng (SSTable).



Hình 1: Minh họa C1 và C0



2.2. Độ khuếch đại đọc – Read Amplification (RA)

Độ khuếch đại đọc là số lần đọc trên đĩa cứng cho mỗi truy vấn, việc lưu cache có ảnh hưởng đến độ khuếch đại đọc, có 2 trường hợp quan trọng đó là cold-cache và warm-cache.

Ví dụ: nếu cần đọc 5 page trên đĩa cứng để trả lời một truy vấn, thì độ khuếch đại đọc là 5.

Hệ số đọc được định nghĩa một cách riêng biệt đối với point query và các range query. Đối với các range query, độ dài của phạm vi rất quan trọng (số lượng hàng được tìm nạp).

Lưu ý rằng các đơn vị của độ khuếch đại ghi và độ khuếch đại đọc là khác nhau. Độ khuếch đại ghi đo lượng dữ liệu được ghi nhiều hơn dung lượng dữ liệu mà ứng dụng dự tính ghi bao nhiêu lần, trong khi độ khuếch đại đọc tính số lần đọc đĩa để thực hiện một truy vấn.

2.3. Độ khuếch đại ghi – Write Amplification (WA)

Độ khuếch đại ghi là tỷ lệ giữa lượng dữ liệu được ghi vào thiết bị lưu trữ với lượng dữ liệu được ghi vào cơ sở dữ liệu, nếu WA có giá trị càng nhỏ thì hiệu quả ghi càng cao.

Ví dụ: nếu ghi 10 MB vào cơ sở dữ liệu và tốc độ ghi trên đĩa là 30 MB, thì độ khuếch đại ghi là 3.

2.4. Độ khuếch đại không gian – Space Amplification (SA)

Độ khuếch đại không gian là tỷ lệ giữa lượng dữ liệu trên thiết bị lưu trữ so với lượng dữ liệu trong cơ sở dữ liệu.

Ví dụ: nếu ghi 10MB vào cơ sở dữ liệu và cơ sở dữ liệu sử dụng 100MB trên đĩa, thì độ khuếch đại không gian là 10.

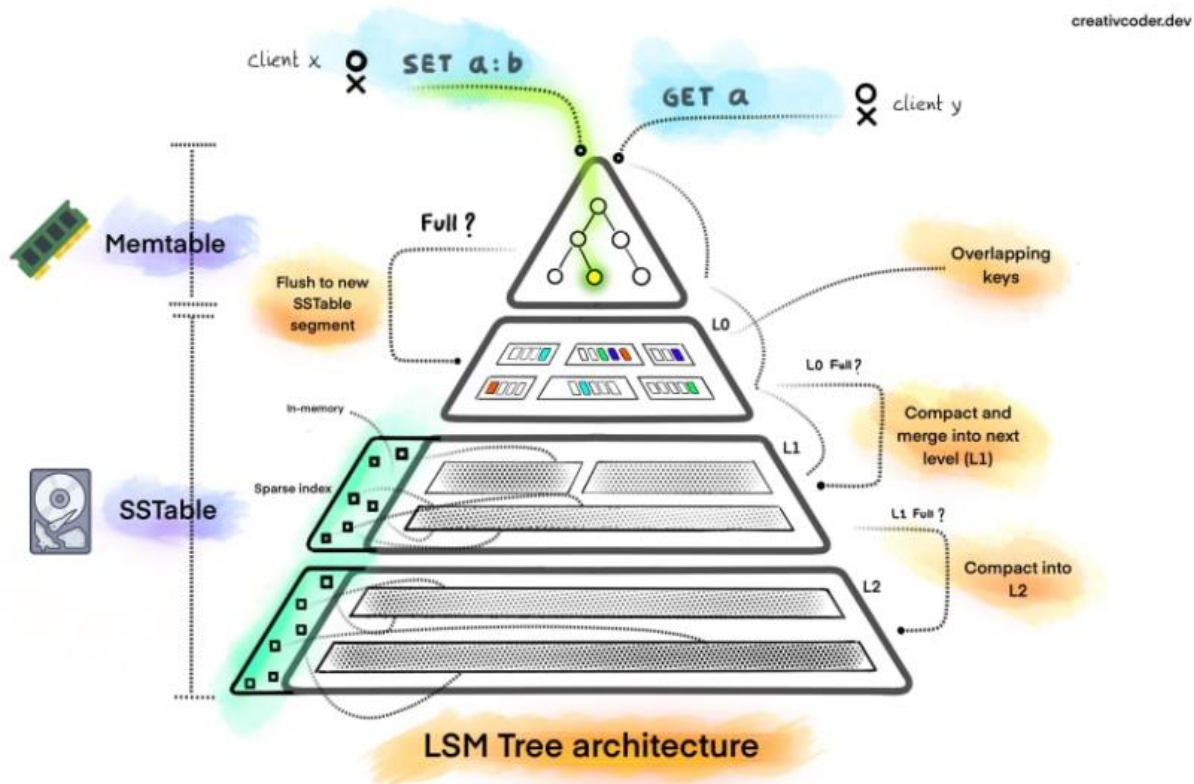
2.5. Cách thức hoạt động

Một cấu trúc dữ liệu có thể tối ưu hóa cho nhiều nhất 2 trong 3 độ khuếch đại đọc, ghi và không gian. LSM tree có độ khuếch đại ghi bé hơn Btree trong khi Btree có độ khuếch đại đọc bé hơn LSM tree.

Do có độ khuếch đại ghi nhỏ, nên LSM tree có cơ chế hoạt động rất khác để tận dụng điểm mạnh đó đồng thời kết hợp với các cấu trúc dữ liệu và giải thuật khác bao gồm:



- Memtable (C0).
- SSTable (C1).
- Bloom filter.
- Merge and Compaction.



Hình 2: Cấu trúc của LSM tree.

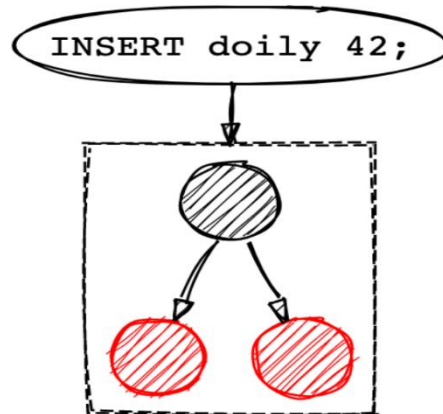
2.5.1. Memtable

Memtable là cấu trúc dữ liệu nằm hoàn toàn trong RAM dùng để lưu giữ các record một cách tạm thời. Memtable không lưu trữ toàn bộ database, nó chỉ lưu giữ tạm thời các record trong RAM để giảm chi phí truy xuất đĩa cứng, toàn bộ memtable sẽ được ghi lên đĩa cứng khi đủ điều kiện.

Memtable là nơi đầu tiên được tìm và kiểm tra mỗi khi có yêu cầu đọc hoặc ghi vào database. Khi có yêu cầu đọc dữ liệu từ database, memtable luôn được đọc và tìm kiếm trước để nếu dữ liệu có trong memtable thì sẽ không tốn chi phí truy xuất dữ liệu từ đĩa cứng. Khi có yêu cầu ghi dữ liệu vào database, dữ liệu sẽ luôn được ghi vào memtable, khi memtable đạt tới một kích thước nhất định nó sẽ được ghi lên đĩa cứng.



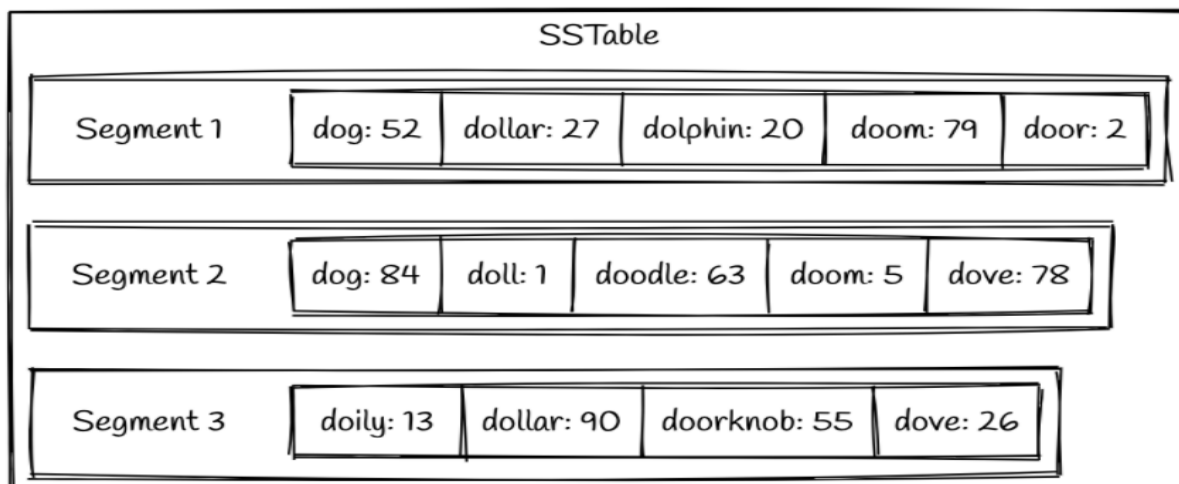
Memtable có cấu trúc tương tự cây được tối ưu cho các mục đích cụ thể nhưng thường là các cấu trúc cây nhị phân tìm kiếm tự cân bằng như 2-3-tree, AVL-tree, Red-Black tree.



Hình 3: Red-Black tree

2.5.2. SSTable

Memtable được ghi ra đĩa cứng bằng file có định dạng Sorted-String Table (SSTable). SSTable lưu trữ các cặp key - value trong đó các key đều chỉ xuất hiện 1 lần duy nhất (không có trùng lặp) và các hàng được sắp xếp theo key. Một SSTable bao gồm nhiều thành phần được sắp xếp gọi là segment và các segment này bất biến khi được ghi lên đĩa cứng. Một database chứa rất nhiều SSTable.



Hình 4: Một SSTable đơn giản.

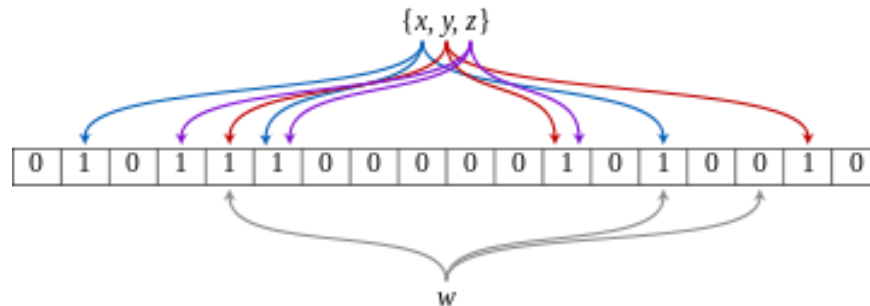


Mỗi SSTable chứa một hash-map, hash-map này đánh dấu các khóa tới các offset để giảm chi phí cho việc tìm kiếm, khi quá trình compaction diễn ra, hash-map cũng sẽ được cập nhật. Do hash-map chỉ lưu offset của key, nên vẫn sẽ có những trường hợp mà key không tồn tại trong database nhưng hash-map vẫn trả về phản hồi là có, việc này dẫn đến lãng phí chi phí truy xuất đĩa cứng cho việc tìm kiếm một dữ liệu không tồn tại trong database, Bloom filter được thêm vào để giải quyết trường hợp này.

2.5.3. Bloom filter

Bloom filter là một cấu trúc dữ liệu xác suất hiệu quả về không gian được sử dụng để kiểm tra xem một phần tử có phải là con của một tập hợp hay không, Bloom filter kiểm tra xem key có tồn tại hay không với độ phức tạp $O(1)$, giúp cải thiện hiệu suất đọc dữ liệu ở một mức độ nào đó, Bloom filter luôn cho kết quả đúng nếu phần tử không tồn tại trong tập hợp.

- Bloom filter(K) trong một SSTable luôn trả về 0 nếu key không tồn tại trong tập hợp.
- Bloom filter(K) trong một SSTable trả về 1 nếu key có thể hoặc không có thể tồn tại trong tập hợp.



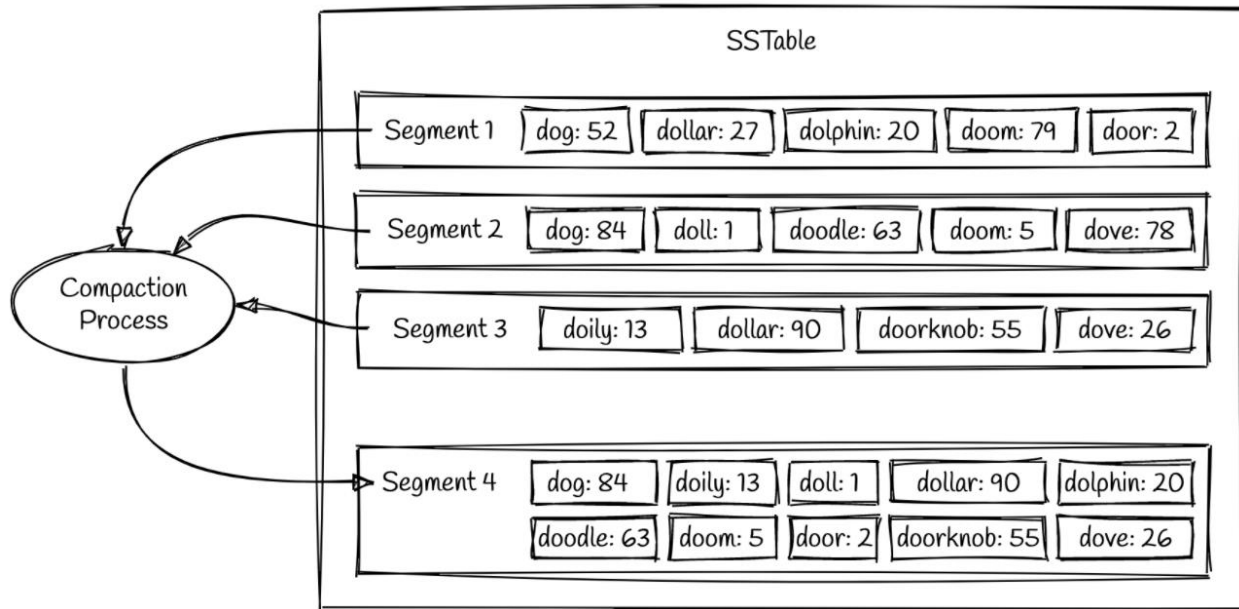
Hình 5: Bloom filter.

2.5.4. Merge và compaction

Theo thời gian, số lượng file được ghi ra đĩa cứng sẽ nhiều lên do memtable liên tục được ghi lên đĩa cứng mỗi khi nó có kích thước đủ lớn, nếu có quá nhiều file chứa các “mảnh dữ liệu” như vậy, việc truy vấn sẽ tốn nhiều chi phí do phải đọc và tìm kiếm dữ liệu trên nhiều “mảnh dữ liệu” từ đĩa cứng. Compaction là một quá trình nền liên tục kết hợp các segment cũ của các SSTable với nhau thành các segment mới trong một SSTable mới.



Quá trình compaction còn giúp merge các file SSTable nhỏ thành một file SSTable có kích thước lớn hơn, thao tác này giúp giảm chi phí truy xuất đĩa cứng và giảm thời gian tìm kiếm dữ liệu khi có yêu cầu.



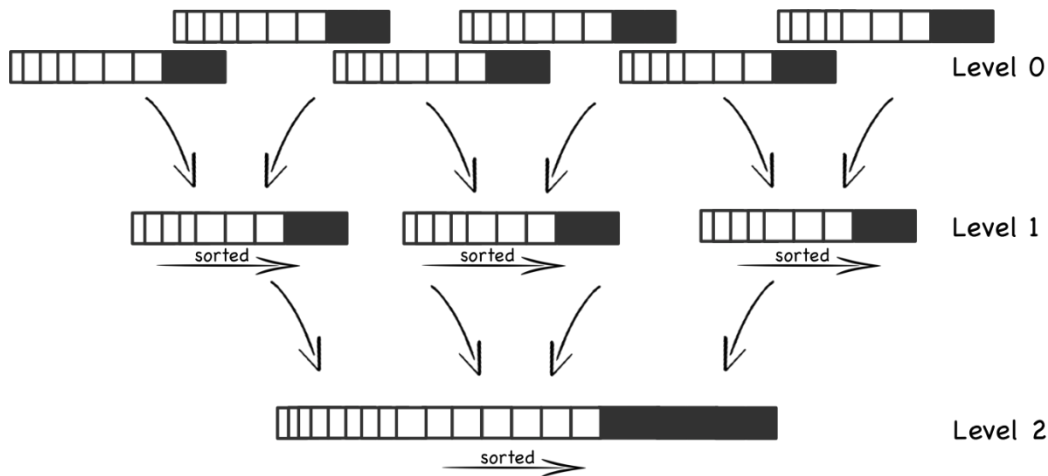
Hình 6 : Mô tả đơn giản quá trình Compaction.

Compaction quét record đầu tiên của từng segment, sau đó chọn ra key nhỏ nhất để ghi vào Merged Segment (trong trường hợp có key cùng có mặt trên nhiều segment, ta chọn record mới nhất và bỏ qua giá trị trên các segment cũ hơn), nếu record hiện tại có key trùng với key đã được ghi vào Merged Segment thì sẽ bỏ qua và quét sang record kế tiếp, quá trình này được lặp lại đến khi merge hết tất cả các SSTable đủ điều kiện.

Có 2 kiểu compaction:

- Compaction theo kích thước (STCS): Ý tưởng đằng sau STCS là merge các SSTable có kích thước nhỏ thành các SSTable có kích thước trung bình khi chúng có đủ số lượng, và merge các SSTable trung bình thành các SSTable lớn hơn khi chúng có đủ số lượng, quá trình này diễn ra liên tục khi kích thước của các SSTable đạt tới một ngưỡng đủ lớn (mặc định là 5GB và lớn nhất là 10TB trong WiredTiger).





Compaction continues creating fewer, larger and larger files

Hình 7: Compaction theo kích thước.

- Compaction theo cấp độ (LBCS): Ý tưởng của LBCS là tổ chức dữ liệu thành các cấp và mỗi cấp đều được sắp xếp. Sau khi một cấp tích lũy đủ dữ liệu, một số dữ liệu ở cấp này sẽ được nén thành cấp cao hơn. Dữ liệu bắt đầu ở cấp độ 0, sau đó được hợp nhất vào lần chạy cấp độ 1. Cuối cùng lần chạy cấp độ 1 được hợp nhất với lần chạy cấp độ 2, v.v. Mỗi cấp độ bị hạn chế về kích thước của nó. Hệ số tăng trưởng k được chỉ định là độ phóng đại của kích thước dữ liệu ở mỗi cấp.

$$level_i = level_{i-1} * k$$

2.5.5. Các thao tác của LSM tree

Đọc và tìm kiếm dữ liệu: Dữ liệu sẽ luôn được đọc và tìm kiếm đầu tiên trong memtable, khi dữ liệu mong muốn không tồn tại trong memtable thì sẽ được tìm kiếm trong các SSTable trên đĩa cứng. Tiến hành duyệt qua các segment trong SSTable và tìm key mong muốn, bắt đầu với segment mới nhất cho đến segment cũ nhất hoặc cho đến khi tìm thấy key, do các segment trong SSTable đã được sắp xếp, nên có thể dừng tìm kiếm nhị phân để việc tìm kiếm các key được nhanh hơn. Một cách tối ưu đơn giản là để một sparse index trong RAM giúp tìm offset cho các value, đồng thời kết hợp với Bloom filter để dừng việc tìm kiếm sớm khi key không tồn tại trong tập hợp.

Chèn dữ liệu: Khi có record mới được chèn, nó sẽ luôn được chèn vào memtable, record luôn được giữ trong RAM đến khi kích thước của memtable đạt tới một ngưỡng kích thước nhất định, toàn bộ memtable sẽ được ghi lên đĩa cứng.



Xóa dữ liệu: Khi một nhận được yêu cầu xóa một record, LSM tree sẽ không xóa bất cứ một thứ gì trong database, LSM tree sẽ ghi thêm một record mới có key chính là key của record được yêu cầu xóa vào memtable, value của record này là một marker độc nhất có tên tombstone. nếu có một lệnh yêu cầu tìm kiếm key này thì sẽ luôn được phản hồi là key không tồn tại do cơ chế luôn tìm kiếm đầu tiên từ memtable và segment mới nhất, record chỉ chính thức bị xóa sau khi đã trải qua quá trình compaction.

Update: Khi một nhận được yêu cầu update một record, LSM tree sẽ không tìm kiếm và update bất cứ một thứ gì trong database, LSM tree sẽ ghi thêm một record mới chứa toàn bộ nội dung cần update vào memtable. Nếu có một yêu cầu tìm kiếm key đó, kết quả của truy vấn sẽ luôn là dữ liệu đã được update do cơ chế luôn tìm kiếm từ memtable và segment mới nhất đầu tiên, record cũ sẽ bị xóa đi và record update sẽ được giữ lại sau khi đã trải qua quá trình compaction.

2.6. Ưu và nhược điểm

Ưu điểm:

- LSM Tree có hiệu năng ghi rất cao.
- Thích hợp với các database có nhiệm vụ chính là ghi thêm dữ liệu.
- Tối ưu mạnh cho các đĩa cứng sử dụng băng từ.

Nhược điểm:

- Quá trình compaction đôi khi ảnh hưởng trực tiếp đến hiệu suất đọc và ghi.
- Nếu thiết lập các thông số liên quan không thích hợp sẽ ảnh hưởng mạnh đến hiệu năng.
- Mỗi khóa có thể tồn tại ở nhiều nơi và do đó để kiểm tra xem key không tồn tại cần quét qua toàn bộ segment nếu không có Bloom filter.



PHẦN 2: THỰC NGHIỆM

3. Mô tả thực nghiệm:

3.1. Cấu trúc cây thư mục:

Nhóm em thực hiện phân chia và tổ chức các thư mục để lưu trữ database, chương trình, mã nguồn và kết quả để tiện cho việc truy xuất, tìm kiếm và thao tác. Tên của các thư mục và tên file chương trình và file kết quả cũng phải phản ánh được vai trò và chức năng của chúng. Do các thư mục, chương trình và mã nguồn đã được tổ chức theo các quy tắc đã được nhóm đặt ra trước, nên việc thực thi tự động các chương trình và tìm kiếm kết quả trở nên dễ dàng hơn.

- **Thư mục bin:** Đây là nơi chứa các file thực thi.
 - **create_btree_table:** Chương trình tạo table dùng BTree.
 - **create_lsm_table:** Chương trình tạo table dùng LSM Tree.
 - **delete:** Chương trình thực hiện thao tác xóa.
 - **insert:** Chương trình thực hiện thao tác chèn.
 - **open:** Chương trình thực hiện thao tác đóng mở database.
 - **print_db:** Chương trình in toàn bộ record.
 - **search:** Chương trình thực hiện thao tác tìm kiếm.
 - **update:** Chương trình thực hiện thao tác cập nhật.
 - **check_and_fix:** Sửa lỗi các file không có kết quả bằng cách chạy lại test đó.
 - **make_csv:** Chương trình tạo file csv tự động.
- **Thư mục btree_db:** Chứa 100 thư mục, mỗi thư mục con chứa 1 database có số record bằng với tên thư mục, các database liên tiếp sẽ lệch nhau 100 000 record.
 - **100000:** Btree database có 100 000 record.
 - ...
 - **10000000:** Btree database có 10 000 000 record.
- **Thư mục lsm_db:** Chứa 50 thư mục, mỗi thư mục chứa 1 database có số record bằng với tên thư mục.
 - **100000:** LSM tree database có 100 000 record.
 - ...
 - **10000000:** LSM tree database có 10 000 000 record.



- **Thư mục result:** Chứa kết quả đo đạc của mọi thao tác trên cả Btree lẫn LSM tree.
 - **btree:** Chứa 6 thư mục tương ứng với kết quả của thao tác tương ứng trên các database sử dụng Btree, bên trong các thư mục con đều chứa 100 thư mục ứng với kết quả của database có số lượng record tương ứng.
 - **delete:** Kết quả của thao tác delete.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **insert:** Kết quả của thao tác insert.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **open:** Kết quả của thao tác open.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **search:** Kết quả của thao tác search.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **update:** Kết quả của thao tác update.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.



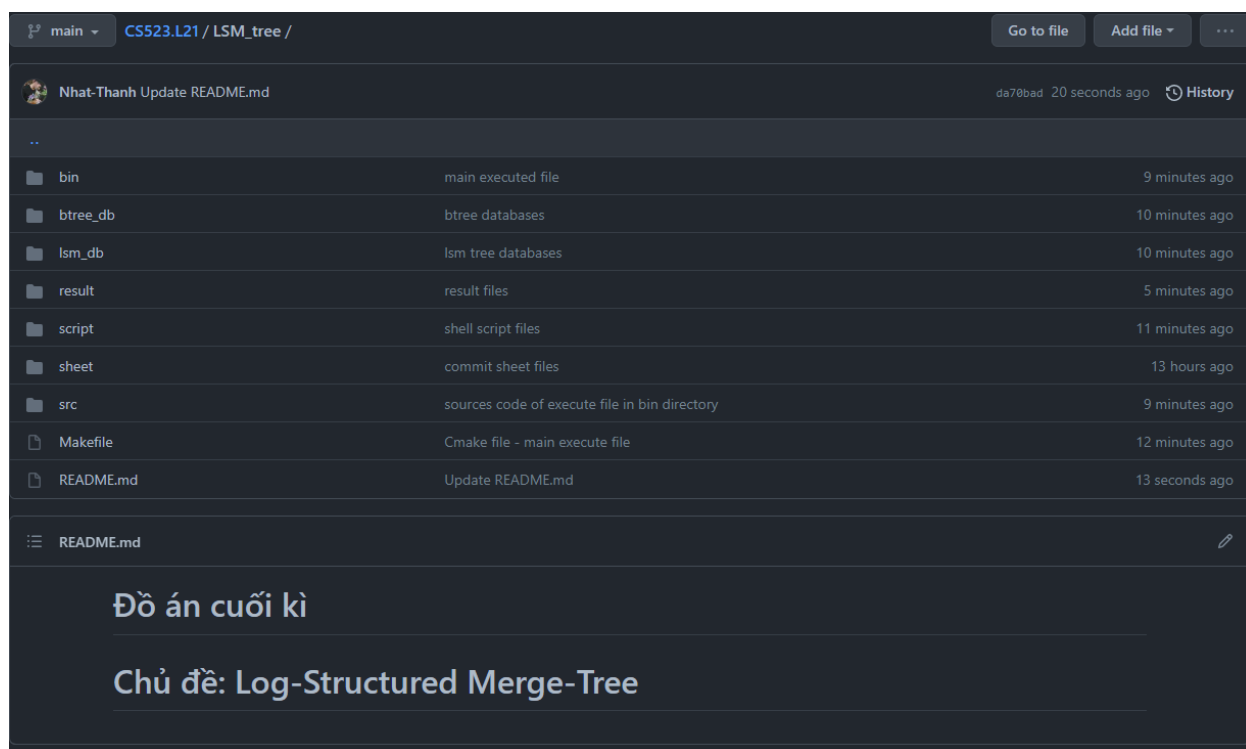
- **lsm:** Chứa 6 thư mục tương ứng với kết quả của thao tác tương ứng trên các database sử dụng LSM tree, bên trong các thư mục con đều chứa 100 thư mục ứng với kết quả của database có số lượng record tương ứng.
 - **delete:** Kết quả của thao tác delete.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **insert:** Kết quả của thao tác insert.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **open:** Kết quả của thao tác open.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **search:** Kết quả của thao tác search.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
 - **update:** Kết quả của thao tác update.
 - **100000:** Kết quả của database 100 000 record.
 - ...
 - **10000000:** Kết quả của database 10 000 000 record.
- **Thư mục script:** Chứa file shell, các file shell này sẽ thực hiện với toàn bộ database, tất cả đều cần truyền một đối số là loại tree mà các database sử dụng.
 - **compact.sh:** merge LSM tree database.
 - **create_database.sh:** Tạo database ứng với từng loại tree.
 - **delete.sh:** Thực hiện xóa 100000 record.



- **init.sh**: Tạo các thư mục lsm_db, btree_db, bin và result.
- **insert.sh**: Thực hiện chèn 100000 record.
- **measure_swinf.sh**: Đo dung lượng RAM và đĩa cứng.
- **open.sh**: Thực hiện mở và đóng database.
- **search.sh**: Thực hiện tìm kiếm một giá trị mong muốn 1000 lần.
- **update.sh**: Thực hiện cập nhật 100000 record.
- **Thư mục sheet**: Chứa bảng kết quả của từng thao tác ứng với từng loại tree.
 - **btree_create_database.csv**
 - **btree_delete.csv**
 - **btree_insert.csv**
 - **btree_open.csv**
 - **btree_search.csv**
 - **btree_update.csv**
 - **lsm_create_database.csv**
 - **lsm_delete.csv**
 - **lsm_insert.csv**
 - **lsm_open.csv**
 - **lsm_search.csv**
 - **lsm_update.csv**
- **Thư mục src**: Nơi chứa mã nguồn của các file thực thi trong **thư mục bin**.
 - **check_and_fix.cpp**
 - **compact.cpp**
 - **create_btree_table.cpp**
 - **create_lsm_table.cpp**
 - **delete.cpp**
 - **insert.cpp**
 - **make_csv.cpp**
 - **open.cpp**
 - **print_db.cpp**
 - **search.cpp**
 - **update.cpp**



- **Makefile:** file tổng hợp các lệnh cần gọi để thực thi, giảm việc gõ lệnh và thực thi tự động, ta sẽ gọi mọi file khác thông qua file này.



Hình 8: Cấu trúc cây thư mục.

3.2. Cấu hình máy test

Hệ điều hành:

- Garuda Linux x86_64
- Linux kernel: 5.12.13
- Desktop: Gnome 40.2

Công cụ: bash, cmake, g++, du, pgrep, ps, wiredtiger.

Cấu hình máy:

- Model: DELL inspiron 3580
- CPU: Intel core i5-8265U (4 cores, 8 threads) @ 3.9GHz
- RAM: 7.63 GB – dual
- GPU: Intel UHD Graphics 620, Radeon™ 520




```

fish /home/thanh

TTTTTTTTTT hhhh                      hhhh      thanh@thanh
T:::TTTTTT h::h                      h::h
T:::TTTTTT h::h                      h::h
T:::TTTTTT h::h                      h::h
TT T::T TT h::h                      aaaaaaaa nnn nnnnnn h::h
T::T h::hhh a:::a n::nn::n h::hhh
T::T h:::hh aaaaaa::a n::nn::n h:::hh
T::T h:::h::h a::a n::nn::n h:::h::h
T::T h::h h::h aaaaaa::a n::nn::n h::h h::h
T::T h::h h::h a:::a n::n n::n h::h h::h
T::T h::h h::h a::a n::n n::n h::h h::h
TT:::TT h::h h::h a::a a::a n::n n::n h::h h::h
T:::TT h::h h::h a:::a n::n n::n h::h h::h
T:::TT h::h h::h a:::a n::n n::n h::h h::h
TTTTTTTT hhhh hhhh aaaaaaa aaa nnnn nnnn hhhh hhhh

[thanh@thanh in ~ via v3.9.5]
[~] sudo inxi --cpu --graphics --machine --memory --system --disk --color=32
[sudo] password for thanh:
System:   Host: thanh Kernel: 5.12.13-zen1-2-zen x86_64 bits: 64 Desktop: GNOME 40.2 Distro: Arch Linux
Machine:  Type: Laptop System: Dell product: Inspiron 3580 v: N/A serial: 5LBKDX2
          Mobo: Dell model: 02G1KD v: A00 serial: /5LBKDX2/CNCMC0096N03CE/ UEFI: Dell v: 1.12.0
          date: 10/28/2020
Memory:   RAM: total: 7.63 GiB used: 2.71 GiB (35.6%)
          Array-1: capacity: 32 GiB slots: 2 EC: None
          Device-1: DIMM A size: 4 GiB speed: spec: 2667 MT/s actual: 2400 MT/s
          Device-2: DIMM B size: 4 GiB speed: spec: 2667 MT/s actual: 2400 MT/s
CPU:      Info: Quad Core model: Intel Core i5-8265U bits: 64 type: MT MCP cache: L2: 6 MiB
          Speed: 3671 MHz min/max: 400/3900 MHz Core speeds (MHz): 1: 3671 2: 3671 3: 3692 4: 3731
          5: 3450 6: 3602 7: 3661 8: 3700
Graphics: Device-1: Intel WhiskeyLake-U GT2 [UHD Graphics 620] driver: i915 v: kernel
          Device-2: AMD Jet PRO [Radeon R5 M230 / R7 M260DX / Radeon 520 Mobile] driver: radeon v: kernel
          Device-3: Microdia Integrated Webcam HD type: USB driver: uvcvideo
          Display: server: X.Org 1.20.11 driver: loaded: ati,intel,radeon unloaded: modesetting
          resolution: 1: 1920x1080~60Hz 2: 1280x1024~75Hz
          OpenGL: renderer: Mesa Intel UHD Graphics 620 (WHL GT2) v: 4.6 Mesa 21.1.3
Drives:   Local Storage: total: 2.04 TiB used: 649.58 GiB (31.1%)
          ID-1: /dev/sda vendor: Western Digital model: WD10SPZX-75Z10T2 size: 931.51 GiB
          ID-2: /dev/sdb vendor: Seagate model: ST1000LM048-2E7172 size: 931.51 GiB
          ID-3: /dev/sdc vendor: Kingston model: SA400M8240G size: 223.57 GiB

[thanh@thanh in ~ via v3.9.5 took 5s]
[~]

```

Hình 9: Thông số chi tiết cấu hình máy test.

Ổ đĩa được sử dụng: Western Digital Blue

- Model: WD10SPZX
- Kích thước: 2.5 inch
- Dung lượng: 1TB
- Cache: 128MB
- Round per minute (RPM): 5400
- Tốc độ đọc: ~100 MB/s
- Tốc độ ghi: ~100 MB/s
- Chuẩn kết nối: SATA3





Hình 10: WD blue 1TB.

3.3. Quá trình thực hiện

Công cụ được sử dụng cho việc thực nghiệm với LSM tree là WiredTiger 10.0.0: WiredTiger là một nền tảng mở rộng NoSQL, mã nguồn mở để quản lý dữ liệu. Phát hành chính thức vào 2012 bởi WiredTiger Inc, theo phiên bản 2 hoặc 3 của GNU General Public License. WiredTiger sử dụng kiến trúc MultiVersion Concurrency Control. WiredTiger là storage engine mặc định của MongoDB.

Tạo 100 database có kích thước từ 100000 → 10000000 record, mỗi database lệch nhau 100000 record. Mỗi database chỉ có một table chứa 2 cột là key, value, cả 2 cột đều có kiểu dữ liệu là integer và giá trị của 2 cột trên từng record đều có giá trị bằng nhau.

Vd: table có định dạng như trên có 3 record là (1 - 1), (2 - 2), (3 - 3) với (a - b) là record có key = a và value = b.

Những thao tác được đo đạc gồm: insert, update, open, search và delete, thứ tự thực hiện các thao tác: *open* → *insert* → *search* → *update* → *delete*. Mọi thao tác được thực hiện bằng các chương trình được viết bằng ngôn ngữ C++, các chương trình đều sử dụng API do WiredTiger cung cấp để mô phỏng lại các thao tác.



Thực hiện đo dung lượng RAM và đĩa cứng trong lúc chạy các thao tác, bỏ qua các file cấu hình và file hệ thống của database trong lúc đo dung lượng đĩa cứng, các file bị bỏ qua:

- WiredTiger: tệp phiên bản WiredTiger
- WiredTiger.wt: tệp siêu dữ liệu WiredTiger
- WiredTiger.turtle: tệp cấu hình công cụ cho tệp WiredTiger.wt
- WiredTiger.config: tệp chứa cấu hình cơ sở dữ liệu
- WiredTiger.basecfg: tệp chứa chuỗi cấu hình được truyền vào wiredtiger_open, nó được đọc bất cứ khi nào cơ sở dữ liệu được mở.
- WiredTiger.lock và : khóa tệp mongod.lock

3.3.1. Open

Thực hiện việc đóng mở database bằng hàm wiredtiger_open() và phương thức close(). wiredtiger_open() thực hiện kết nối tới một database với con trỏ đối tượng connection có kiểu dữ liệu là WT_CONNECTION, close() đóng kết nối của con trỏ connection với database mà nó đang trỏ tới.

3.3.2. Insert

Thực hiện insert 100000 record mới vào từng database, các record này có giá trị key - value nằm trong đoạn $[SIZE_{DB} + 1, SIZE_{DB} + 100000]$ (với $SIZE_{DB}$ là kích thước ban đầu của database)

3.3.3. Search

Thực hiện tìm kiếm một key 1000 lần trên từng database, key được chọn để tìm kiếm có giá trị đúng bằng kích thước ban đầu của database trước lúc thực hiện các thao tác khác.

Vd: một database ban đầu có 100000 record, sau khi insert 100000 record mới thì database sẽ có 200000 record, còn key muốn tìm có giá trị là 100000 (kích thước ban đầu của database).



3.3.4. Update

Thực hiện update toàn bộ record vừa được insert trên từng database, giá trị *value* của các record này được cập nhật theo công thức $value = value + 100000$.

Vd: một database có kích thước ban đầu là 100000 record, một record trong database có giá trị (100001 – 100001) sẽ được update thành (100001 – 200001).

3.3.5. Delete

Thực hiện delete toàn bộ record vừa được update trên từng database, các record sẽ được xóa theo key, thao tác delete này sẽ đưa database về trở về kích thước ban đầu trước lúc insert, việc này sẽ giúp nhóm em thực hiện test lại nhiều lần mà không phải tốn thời gian tạo lại toàn bộ database.

4. Kết quả và kết luận:

Các khái niệm về thông số nhóm em chọn để đo đạc:

- **RSS (Resident Set Size):** là dung lượng RAM mà chương trình đang sử dụng thực tế, dung lượng này không bao gồm dung lượng của các shared library.
- **VSIZ (Virtual Size):** là dung lượng RAM tối đa mà chương trình có thể chiếm dụng trên RAM, có thể hiểu đây là dung lượng RAM mà chương trình yêu cầu khi cấp phát động.

Độ phức tạp thời gian của các thao tác được ước lượng theo 9 công thức sau: $O(n)$, $O(\log n)$, $O(1)$, $O(n \log n)$, $O(\sqrt{n})$, $O(n\sqrt{n})$, $O(n^2)$, $O(n^3)$, $O(2^n)$, với n là kích thước trung bình của các thao tác.

Vd: Thao tác insert k record mới vào database có kích thước ban đầu là $SIZE_{DB}$ record sẽ có kích thước trung bình n là:

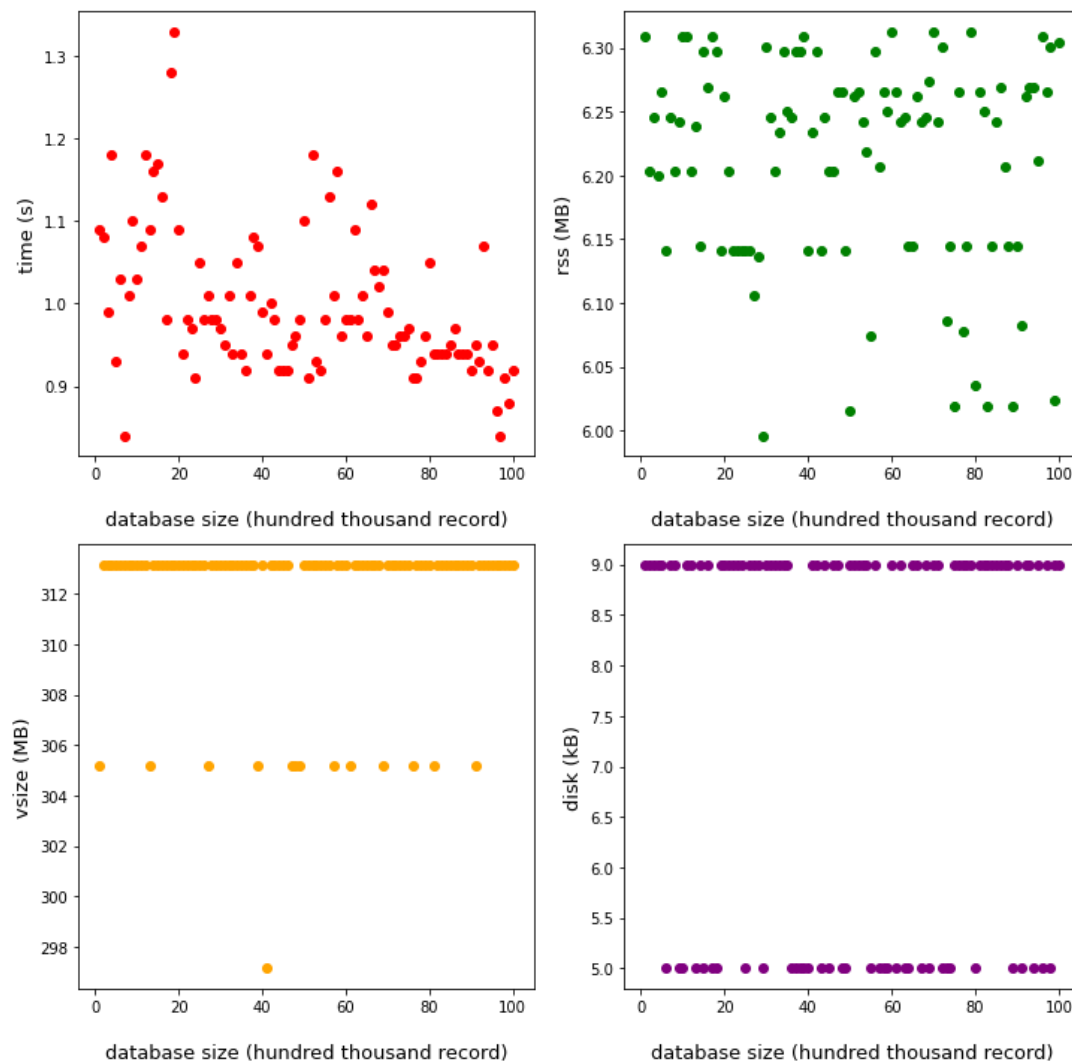
$$\frac{SIZE_{DB} + (SIZE_{DB} + k)}{2} = SIZE_{DB} + \frac{k}{2}$$

Các thao tác sau khi được ước lượng độ phức tạp sẽ được tính độ chênh lệch với thời gian chạy thực tế đã được đi qua một mô hình hồi quy tuyến tính, độ chênh lệch sai số được tính bằng bằng MSE (mean square error), độ phức tạp có sai số nhỏ nhất sẽ là độ phức tạp của thao tác.



4.1. Open

Sau quá trình thực nghiệm thì nhóm em thu được biểu đồ kết quả sau:



Hình 11: Biểu đồ kết quả của thao tác open trên các database sử dụng LSM tree.

Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian open một database sử dụng LSM tree không tăng theo kích thước của database, dẫn chứng là phần lớn các điểm dữ liệu đều tập trung tại khoảng thời gian 0.9 đến 1 giây (độ chênh lệch không đáng kể), và các điểm dữ liệu này trải đều từ database có kích thước 100000 đến 10000000 record.



- Dựa vào biểu đồ RSS, nhóm em nhận thấy các điểm dữ liệu phần lớn phân bố đều nhau trong khoảng 6.13 đến 6.35, những database có kích thước bé (100000 → 2000000 record) cũng chiếm dung lượng RSS bằng với các database có kích thước lớn (8000000 → 10000000 record), một dẫn chứng rõ ràng hơn là biểu đồ VSIZE, phần lớn các điểm dữ liệu đều phân bố theo một đường thẳng nằm ngang. Do đó nhóm em có nhận xét là dung lượng RAM cần cho thao tác open không tăng theo kích thước của database.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là thao tác open có sinh ra các file tạm trong quá trình thực hiện, dung lượng các file tạm này có dung lượng cao nhất là 9KB, và các điểm dữ liệu của biểu đồ phân bố đều từ các database có kích thước nhỏ (100000 record) đến các database có kích thước lớn (10000000 record). Do đó nhóm em có nhận xét là dung lượng file tạm mà thao tác open sinh ra không tăng theo kích thước của database.

Ước lượng độ phức tạp:

- Nhóm em thực hiện ước lượng độ phức tạp của thao tác với 9 độ phức tạp và có được kết quả sau:

```

1 plot_and_show('lsm', 'open', KB_to_MB=True)

O(log(n)): 0.5302474125018415
O(n*log(n)): 11569.634978595373
O(n): 3286.066030121909
O(1): 0.2492301219081908
O(sqrt(n)): 38.28388864935424
O(n*sqrt(n)): 254241.98832399878
O(n^2): 20496818.54443012
O(n^3): 147906653656.76755
O(2^n): 2.142584059011987e+58

```

Hình 12: Độ sai lệch của thời gian thực với thời gian được ước lượng bởi các độ phức tạp.

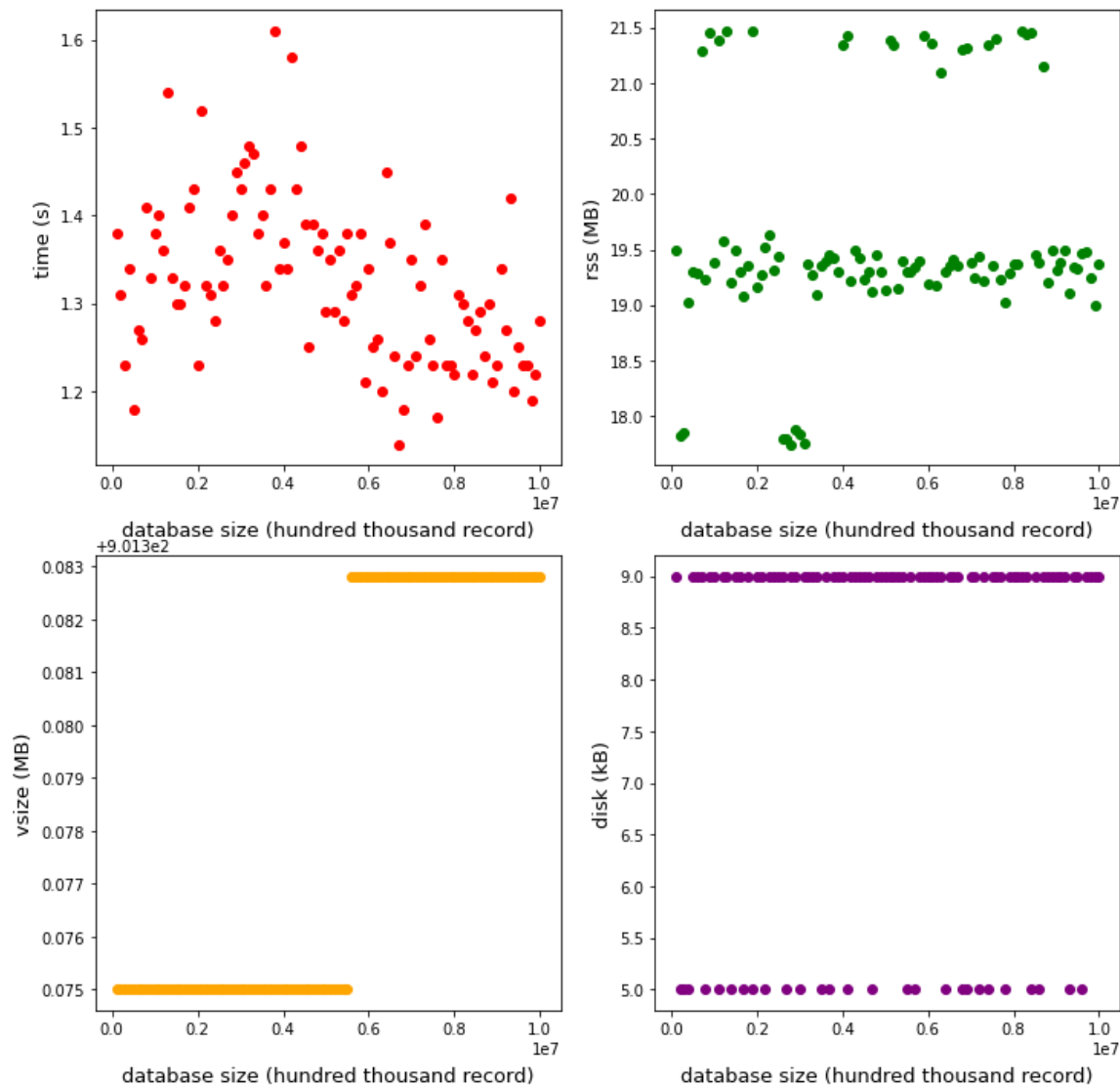
- Độ sai số nhỏ nhất trong 9 công thức trên là 0.2492301 ứng với độ phức tạp $O(1)$, nên thao tác open có độ phức tạp là $O(1)$.

Kết luận: Độ phức tạp của thao tác open của LSM tree là $O(1)$, dung lượng RAM được sử dụng và dung lượng file tạm được sinh ra không phụ thuộc vào kích thước của database.



4.2. Insert

Sau quá trình thực nghiệm nhóm em thu được kết quả như sau:



Hình 13: Biểu đồ biểu diễn kết quả của thao tác insert trên các database sử dụng LSM tree.

Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian chạy của thao tác insert không tăng theo kích thước của database, tại những database có kích thước lớn ($6000000 \rightarrow 10000000$ record), thời gian thực hiện insert lại nhanh hơn so với các database có kích thước nhỏ ($100000 \rightarrow 2000000$ record) vì theo cách thức hoạt



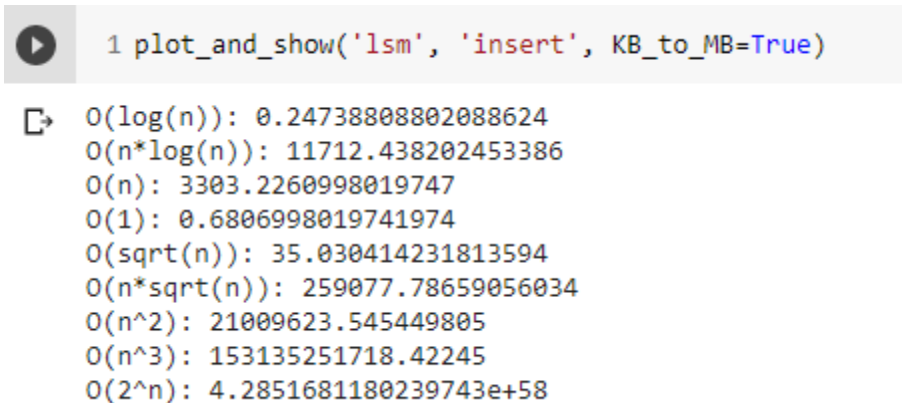
động của LSM tree, các record được insert vào memtable, khi memtable có kích thước đủ lớn sẽ được ghi lên đĩa cứng, sau khi lưu vào đĩa cứng thì quá trình compaction sẽ được thực hiện nếu có đủ điều kiện. Tại các database có kích thước nhỏ, số lượng file database có kích thước nhỏ luôn nhiều hơn do không đủ điều kiện merge dẫn đến việc ghi ra đĩa nhiều lần.

- Dựa vào biểu đồ RSS, nhóm em nhận thấy các điểm dữ liệu phân bố thành 3 khoảng ($0 \rightarrow 18$), ($19 \rightarrow 19.65$) và ($21.25 \rightarrow 21.5$), các điểm dữ liệu phân bố tại các database có kích thước nhỏ ($100000 \rightarrow 2000000$ record) có giá trị tương đương với các điểm dữ liệu tại các database có kích thước lớn ($8000000 \rightarrow 10000000$ record) vì các điểm dữ liệu đều phân bố đều nhau theo từng khoảng. Để bổ sung thêm dẫn chứng trước khi đưa ra kết luận, nhóm em dựa thêm vào dữ liệu của biểu đồ VSIZE, giá trị của các điểm dữ liệu bằng nhau tại các database có kích thước bé ($100000 \rightarrow 5000000$ record), nhưng tăng tại các database có kích thước lớn ($6000000 \rightarrow 10000000$ record). Từ đó, nhóm em có nhận xét là dung lượng RAM mà thao tác insert sử dụng tăng dần theo kích thước của database.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm mà thao tác insert sinh ra trong quá trình chạy phần lớn là 9KB tại các database có kích thước bé và kích thước lớn, nên nhóm em có nhận xét là dung lượng file tạm được sinh ra không tăng theo kích thước của database.

Ước lượng độ phức tạp:

- Nhóm em thực hiện ước lượng độ phức tạp với 9 công thức và có được kết quả sau:

```
1 plot_and_show('lsm', 'insert', KB_to_MB=True)
```



```

O(log(n)): 0.24738808802088624
O(n*log(n)): 11712.438202453386
O(n): 3303.2260998019747
O(1): 0.6806998019741974
O(sqrt(n)): 35.030414231813594
O(n*sqrt(n)): 259077.78659056034
O(n^2): 21009623.545449805
O(n^3): 153135251718.42245
O(2^n): 4.2851681180239743e+58

```

Hình 14: Độ sai lệch của thời gian thực với thời gian được ước lượng bởi các độ phức tạp.

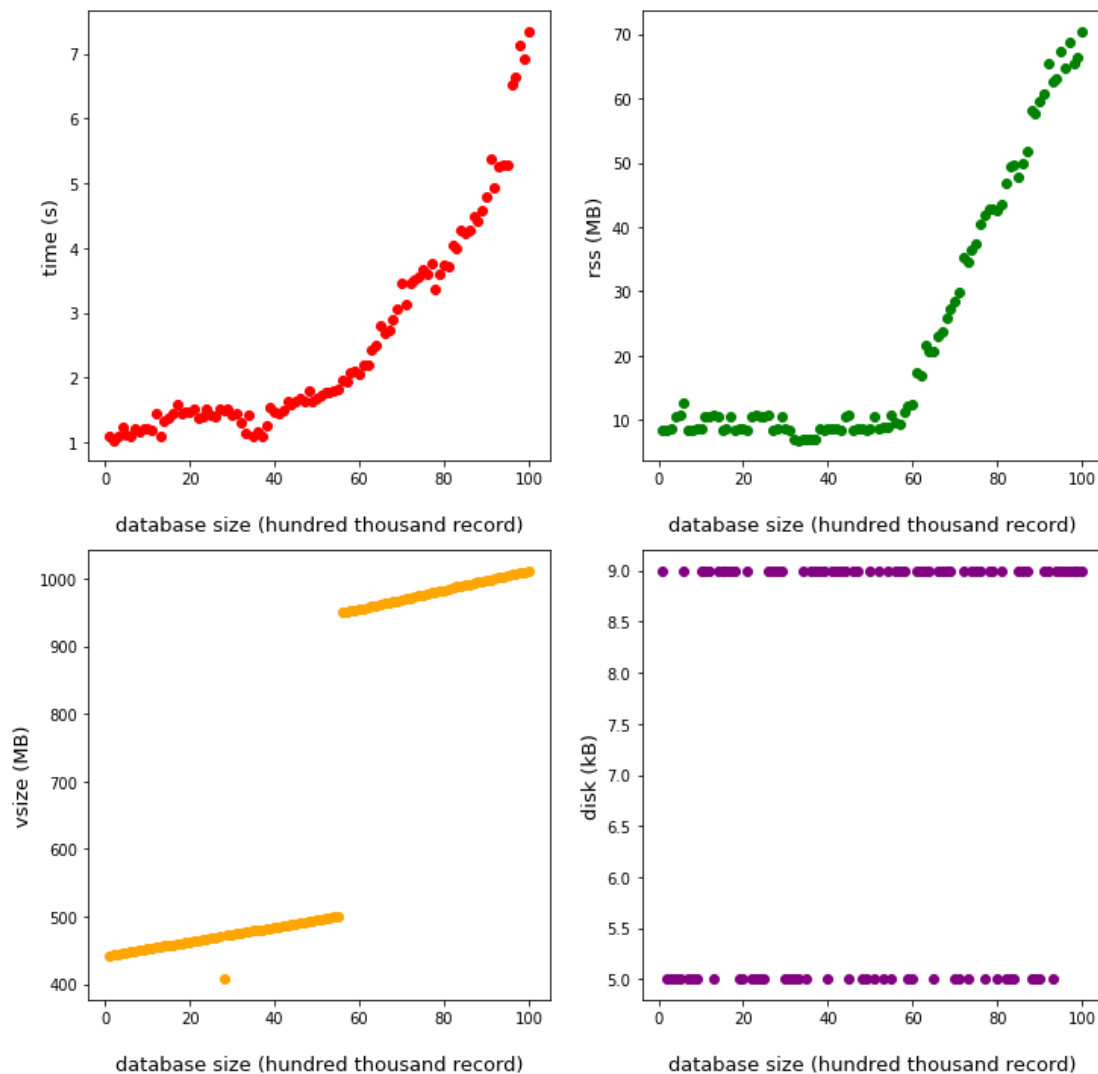
- Độ sai số nhỏ nhất trong 9 công thức trên là 0.2473880 ứng với độ phức tạp $O(\log n)$, nên thao tác insert có độ phức tạp là $O(\log n)$.



Kết luận: Thời gian thực hiện của thao tác insert không tăng theo kích thước của database, tuy nhiên thời gian thực hiện của thao tác sẽ bị ảnh hưởng bởi thao tác merge trong nền, dung lượng RAM mà thao tác insert sử dụng tăng dần theo kích thước của database, dung lượng file tạm được sinh ra trong quá trình thực hiện thao tác không tăng theo kích thước của database.

4.3. Search

Sau quá trình thực nghiệm thì nhóm em có được kết quả như sau:



Hình 15: Biểu đồ biểu diễn kết quả của thao tác search trên các database sử dụng LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian cho thao tác tìm kiếm tăng theo kích thước của database, nếu database càng lớn thì thao tác này sẽ càng tốn nhiều thời gian, dễ nhận thấy nhất là độ dốc của các điểm dữ liệu tăng càng cao khi càng tiến gần về các database kích thước càng lớn.
- Dựa vào biểu đồ RSS, nhóm em có nhận xét là dung lượng RSS mà thao tác này sử dụng tăng theo kích thước của database, vì các điểm dữ liệu có độ dốc tăng cao khi tiến càng gần các database có kích thước lớn, để củng cố và bổ sung thêm nhận xét, nhóm em dựa vào kết quả được biểu diễn bởi biểu đồ VSIZE, kích thước VIZE cần cho thao tác search cũng tăng dần theo kích thước của database, khi tiến về các database lớn (trên 6000000 record), thì kích thước của VSIZE tăng mạnh từ 500MB → 900MB.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là thao tác search cũng sinh ra file tạm trong quá trình thực thi và dung lượng của các file tạm này không tăng theo kích thước của database vì kích thước lớn nhất của file tạm là 9KB và phân bố trải dài từ các database có kích thước bé (100000 → 2000000 record) tới các database có kích thước lớn (8000000 → 10000000 record).

Ước lượng độ phức tạp:

- Nhóm em thực hiện ước lượng độ phức tạp theo 9 công thức và có được kết quả sau:

```

1 plot_and_show('lsm', 'search', KB_to_MB=True)

O(log(n)): 2.0263172022547717
O(n*log(n)): 11126.608611408494
O(n): 3054.7545745971743
O(1): 6.109674597173715
O(sqrt(n)): 18.404518958869275
O(n*sqrt(n)): 252142.42024720655
O(n^2): 20478018.072374597
O(n^3): 147905108791.47733
O(2^n): 2.142584059011987e+58

```

Hình 16: Độ sai lệch của thời gian thực với thời gian được ước lượng bởi các độ phức tạp.

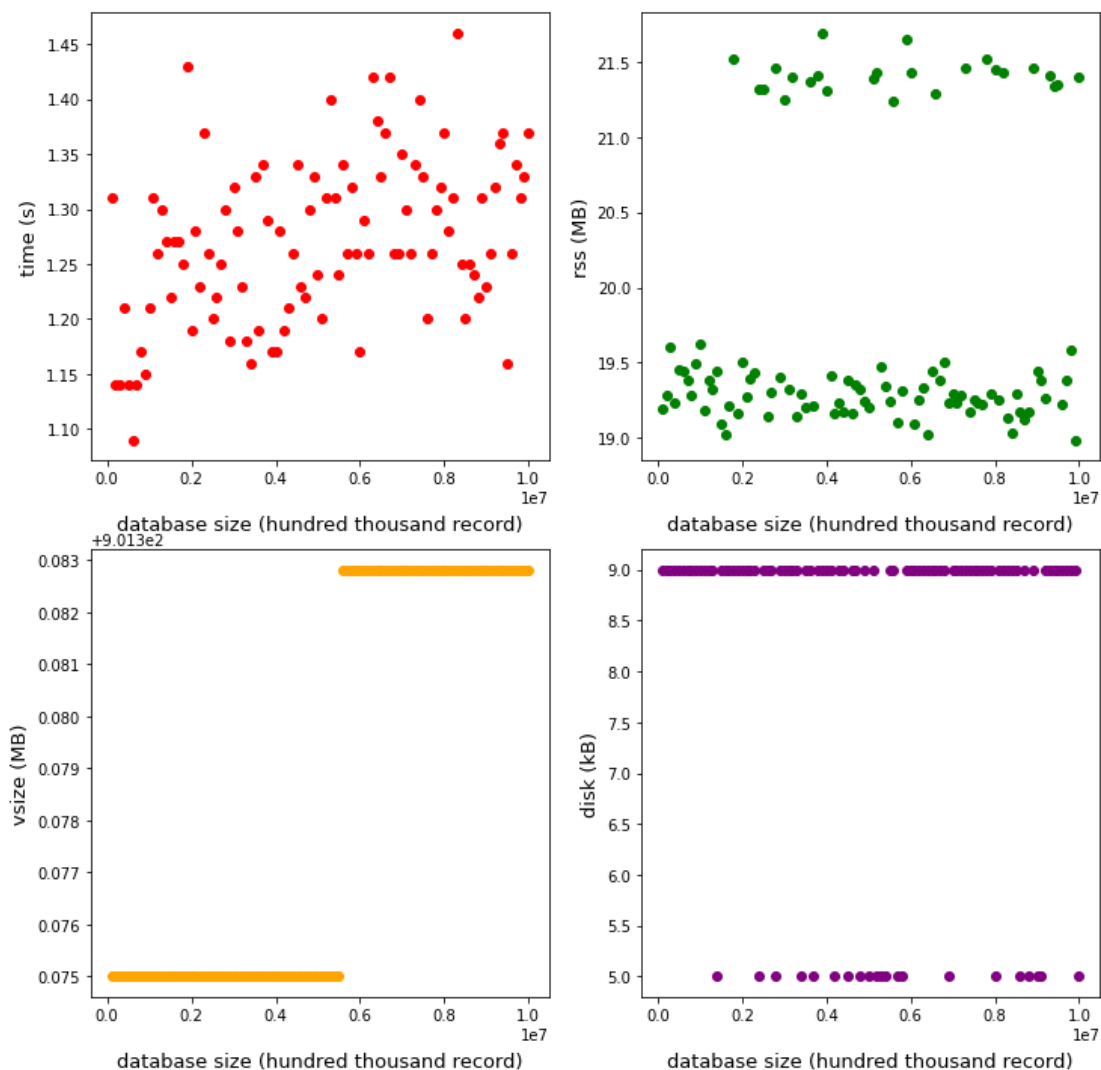
- Độ sai số bé nhất trong 9 công thức là 2.0263172 ứng với độ phức tạp $O(\log n)$, nên độ phức tạp của thao tác search là $O(\log n)$.



Kết luận: Thời gian thực hiện của thao tác search tăng theo kích thước của database, độ phức tạp của thao tác search là $O(\log n)$, dung lượng RAM mà thao tác này sử dụng tăng theo kích thước của database, nếu database có số lượng record càng lớn thì dung lượng RAM và thời gian thực thi sẽ tăng càng cao và không tăng dần đều, thao tác search cũng sinh ra file tạm trong quá trình thực thi nhưng kích thước của file tạm không tăng theo kích thước của database.

4.4. Update

Sau quá trình thực nghiệm nhóm em thu được kết quả sau:



Hình 17: Biểu đồ biểu diễn kết quả của thao tác update trên các database sử dụng LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian thực thi của thao tác update tăng dần theo kích thước của database, các điểm dữ liệu tuy không tăng tuyến tính như thao tác search nhưng dựa vào hình dạng phân bố của các điểm dữ liệu thì chúng đang có xu hướng đi lên.
- Dựa vào biểu đồ RSS, nhóm em nhận thấy dung lượng RSS mà thao tác sử dụng vẫn chưa được biểu diễn một cách rõ ràng, phần lớn các điểm dữ liệu đều tập trung trong khoảng (19 → 19.6) và các điểm dữ liệu này không biểu diễn trạng thái tăng hay giảm một cách rõ ràng vì các giá trị các điểm dữ liệu tại các database có kích thước nhỏ (100000 → 2000000 record) khá tương đồng với giá trị các điểm dữ liệu tại các database có kích thước lớn (8000000 → 10000000 record), để có thể đưa ra kết luận nhóm em dựa thêm vào kết quả của biểu đồ VSIZE, các điểm dữ liệu tại các database có kích thước nhỏ và vừa (100000 → 6000000 record) có giá trị bằng nhau, ngược lại các database có kích thước lớn (6000000 → 10000000 record) thì giá trị tại các điểm dữ liệu lại tăng (901.375 → 901.383 MB), nên nhóm em kết luận là dung lượng RAM cần cho thao tác update tăng theo kích thước của database.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là thao tác update sinh ra file tạm trong quá trình thực thi và dung lượng của các file tạm này không tăng theo kích thước của database vì kích thước lớn nhất của file tạm là 9KB và phân bố trải dài từ các database có kích thước bé (100000 → 2000000 record) đến các database có kích thước lớn (8000000 → 10000000 record).

Ước lượng độ phức tạp:

- Nhóm em thực hiện ước lượng độ phức tạp với 9 công thức và có được kết quả sau:

```
1 plot_and_show('lsm', 'update', KB_to_MB=True)
```

```

O(log(n)): 0.22926602784883826
O(n*log(n)): 11713.600250783884
O(n): 3304.4870590581936
O(1): 0.5950590581938193
O(sqrt(n)): 35.29586878693055
O(n*sqrt(n)): 259079.36057459257
O(n^2): 21009577.754159052
O(n^3): 153135241897.0137
O(2^n): 4.2851681180239743e+58

```

Hình 18: Độ sai lệch của thời gian thực với thời gian được ước lượng bởi các độ phức tạp.

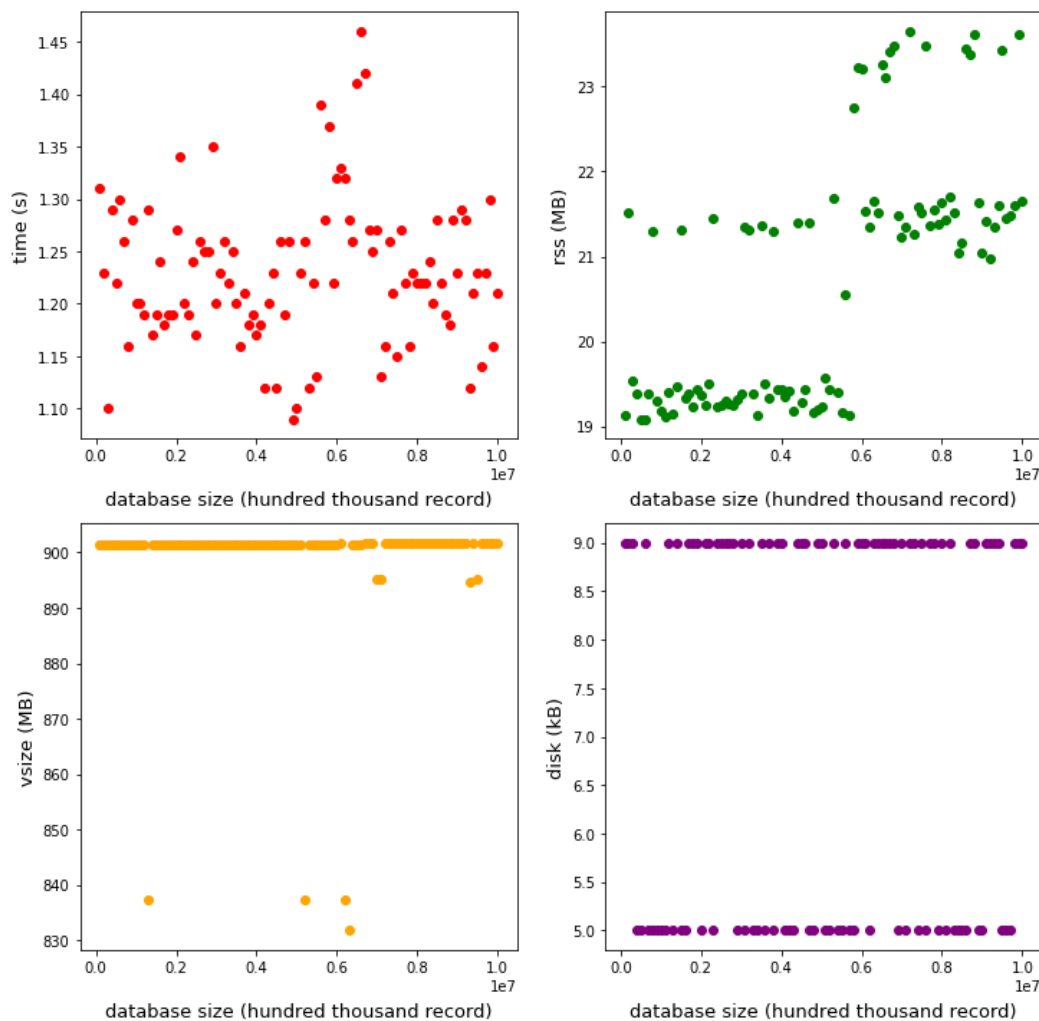


- Độ sai số bé nhất trong 9 công thức là 0.2292660 ứng với độ phức tạp $O(\log n)$, nên độ phức tạp của thao tác update là $O(\log n)$.

Kết luận: thời gian thực hiện của thao tác update tăng dần theo kích thước của database, độ phức tạp của thao tác update là $O(\log n)$, dung lượng RAM mà thao tác này sử dụng tăng theo kích thước của database nhưng không tăng mạnh như thao tác search, thao tác có sinh ra file tạm trong quá trình thực hiện nhưng kích thước file tạm không tăng theo kích thước của database.

4.5. Delete

Sau quá trình thực nghiệm thì nhóm em thu được kết quả sau:



Hình 19: Biểu đồ biểu diễn kết quả của thao tác delete trên các database sử dụng LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian chạy của thao tác delete không tăng theo kích thước của database, dẫn chứng là giá trị của các điểm dữ liệu tại các database có kích thước bé (100000 → 2000000 record) khá tương đồng với các database có kích thước lớn (8000000 → 10000000 record).
- Dựa vào biểu đồ RSS, nhóm em nhận thấy các điểm dữ liệu không tăng một cách rõ ràng, các điểm dữ liệu tại các database có kích thước nhỏ (100000 → 2000000 record) có giá trị khá tương đồng với các điểm dữ liệu tại các database có kích thước lớn (8000000 → 10000000 record), các điểm dữ liệu phân bố chủ yếu trong đoạn (19 → 19.7), với những điều trên vẫn chưa đủ để nhóm em đưa ra kết luận, nên nhóm em dựa thêm vào kết quả của biểu đồ VSIZE, các điểm dữ liệu của biểu đồ VSIZE có giá trị bằng nhau (900MB) nên nhóm em kết luận dung lượng RAM cần cho thao tác delete không tăng theo kích thước của database.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là thao tác delete có sinh ra file tạm trong quá trình thực thi và dung lượng của các file tạm này không tăng theo kích thước của database vì kích thước lớn nhất của file tạm là 9KB và phân bố trải dài từ các database có kích thước bé (100000 → 2000000 record) tới các database có kích thước lớn (8000000 → 10000000 record).

Ước lượng độ phức tạp:

- Nhóm em thực hiện ước lượng độ phức tạp với 9 công thức và có được kết quả sau:

```
1 plot_and_show('lsm', 'delete', KB_to_MB=True)

O(log(n)): 0.5800411772483279
O(n*log(n)): 11544.855034347764
O(n): 3211.184973141002
O(1): 3.0353731410021005
O(sqrt(n)): 25.75901911199003
O(n*sqrt(n)): 258310.7454713991
O(n^2): 21003150.742973138
O(n^3): 153134758541.8112
O(2^n): 4.2851681180239743e+58
```

Hình 20: Độ sai lệch của thời gian thực với thời gian được ước lượng bởi các độ phức tạp.

- Độ sai số nhỏ nhất trong 9 công thức là 0.5800411 ứng với độ phức tạp là $O(\log n)$, nên thao tác delete có độ phức tạp là $O(\log n)$.



Kết luận: thời gian của thao tác delete tăng theo kích thước của database, độ phức tạp của thao tác delete là $O(\log n)$, dung lượng RAM mà thao tác delete sử dụng không tăng theo kích thước của database, thao tác delete có sinh ra file tạm trong quá trình thực thi nhưng kích thước của file tạm không tăng theo kích thước của database.

5. So sánh với Btree

Nhóm em thực hiện đo đạc và so sánh LSM tree với Btree trên 5 thao tác, create database, open, insert, search, update và delete, trên các database cùng số lượng record với cùng số lần thực thi. Các database chỉ có 1 table, table chỉ chứa 2 cột là key, value, cả 2 cột đều có kiểu dữ liệu là integer.

Các thông số của cả 2 Btree và LSM tree được thiết lập giống nhau, riêng LSM tree sẽ được thiết lập thêm một vài thông số phụ để tránh hiệu năng bị ảnh hưởng do throttle.

Thực hiện so sánh với 100 test case, cách thức thực hiện giống với cách thức đo đạc LSM tree. Trình tự thực hiện các thao tác: *create_database* → *open* → *insert* → *search* → *update* → *delete*, đồng thời trong lúc chạy sẽ đo cả dung lượng RAM và dung lượng file tạm được sinh ra.

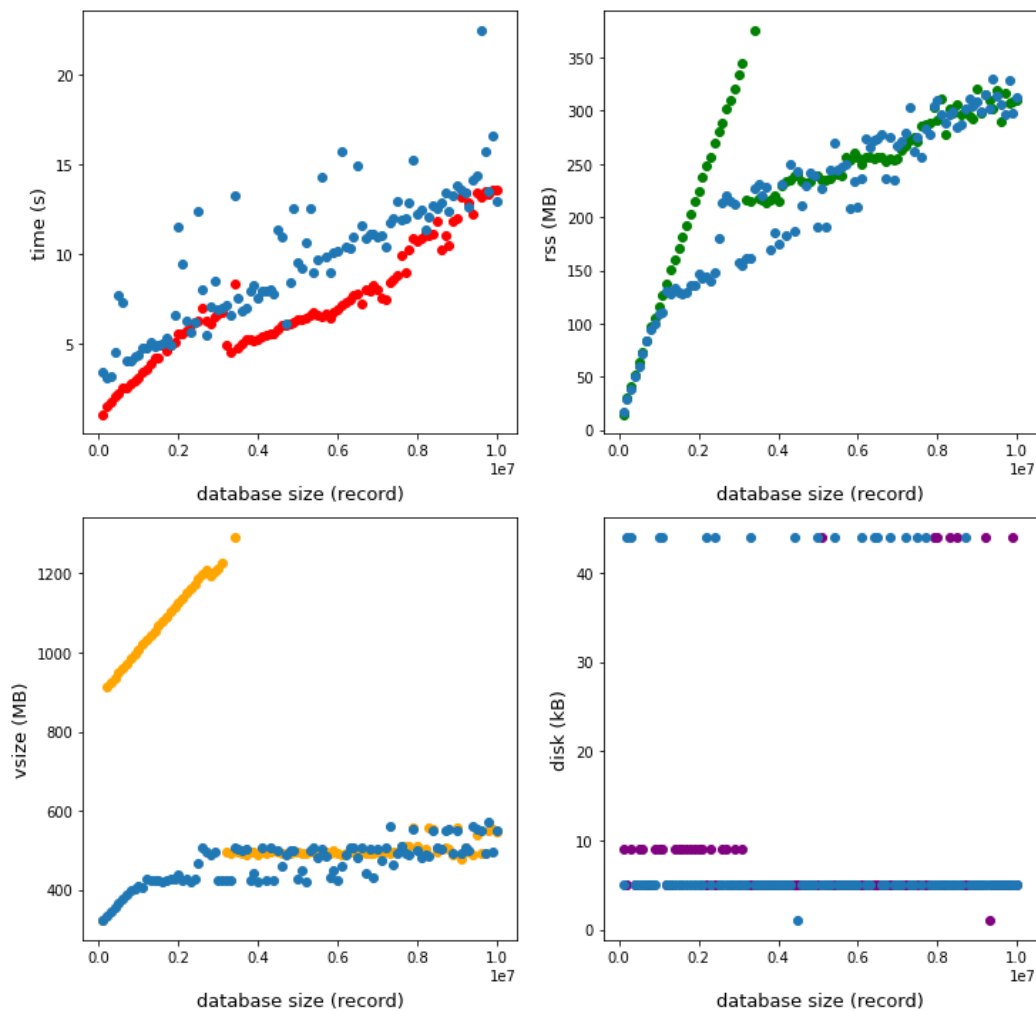


5.1. Create database

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
[ ] 1 lsm_or_btree_better('create_database')
    2 compare('create_database', KB_to_MB=True)
```

lsm's running time is better than btree - 94/100 test cases have running time less then corresponding one
 btree's ram usage is less than lsm - 63/100 test cases have ram usage less then corresponding one
 btree's disk usage is less than lsm - 81/100 test cases have disk usage less then corresponding one



Hình 21: Biểu đồ biểu diễn kết quả của việc tạo database của Btree và LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian tạo database với cùng số lượng record sử dụng LSM tree tốt hơn database sử dụng Btree, dẫn chứng cho điều này là có 94/100 test case mà thời gian thực thi của LSM tree nhanh hơn Btree.
- Dựa vào biểu đồ RSS, nhóm em có nhận xét là dung lượng RSS để tạo database có cùng số lượng record sử dụng Btree tốt hơn so với LSM tree, dẫn chứng cho việc này là có 63/100 test case mà dung lượng RSS của thao tác sử dụng Btree ít hơn so với thao tác sử dụng LSM tree trên cùng một lượng record cho trước.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm được sinh ra trong quá trình tạo database có cùng số lượng record sử dụng Btree tốt hơn so với LSM tree, dẫn chứng là có 81/100 test case mà dung lượng file tạm của thao tác sử dụng Btree ít hơn so với thao tác sử dụng LSM tree với cùng một lượng record cho trước.

Kết luận: Thời gian thực hiện của thao tác tạo database khi sử dụng LSM tree nhanh hơn so với khi sử dụng Btree, nhưng dung lượng RAM và dung lượng file tạm khi sử dụng LSM tree sẽ cao hơn Btree.

Giải thích: Do bản chất thao tác tạo database là thao tác insert, LSM tree được thiết kế để ghi dữ liệu lên đĩa theo cơ chế ghi tuần tự trong khi Btree thì ghi lên đĩa theo cơ chế random, việc ghi random luôn có hiệu năng kém hơn so với việc ghi mở rộng thêm nên thời gian thực hiện của LSM tree nhanh hơn so với Btree. Do LSM tree luôn lưu trong RAM một memtable đồng thời LSM tree cũng có thêm một thao tác merge trong khi Btree chỉ cần load một node vào RAM nên dung lượng RAM mà LSM tree sử dụng luôn cao hơn so với Btree.

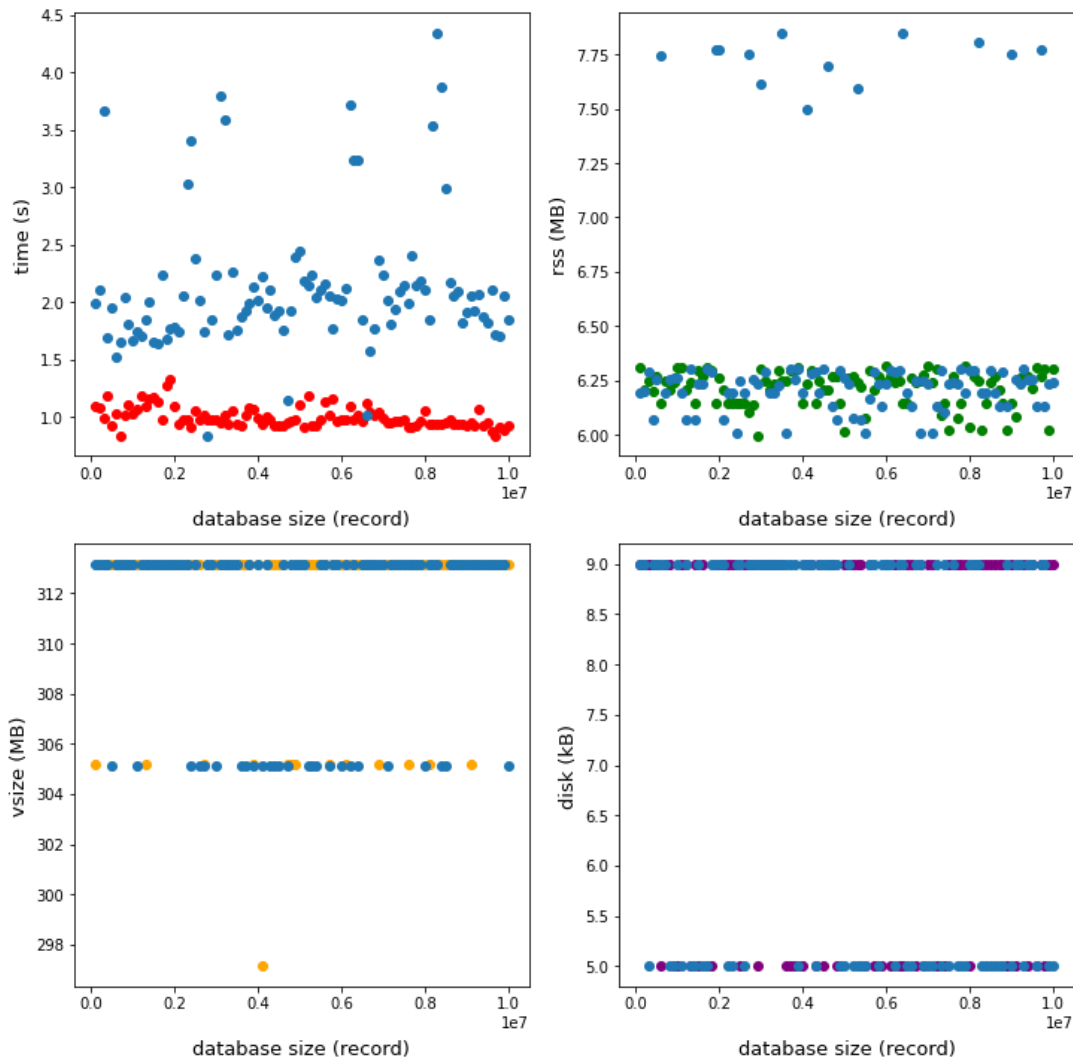


5.2. Open

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
1 lsm_or_btree_better('open')
2 compare('open', KB_to_MB=True)
```

lsm's running time is better than btree - 98/100 test cases have running time less than corresponding one
lsm's ram usage is less than btree - 51/100 test cases have ram usage less than corresponding one
btree's disk usage is less than lsm - 75/100 test cases have disk usage less than corresponding one



Hình 22: Biểu đồ biểu diễn kết quả của thao tác open database của Btree và LSM tree.



Nhận xét:

- Dựa theo biểu đồ TIME, nhóm em có nhận xét là thời gian open một database sử dụng LSM tree tốt hơn so với database sử dụng Btree, dẫn chứng là có 98/100 test case mà thời gian open một database sử dụng LSM tree nhanh hơn so với open một database sử dụng Btree có cùng số record.
- Dựa vào biểu đồ RSS, nhóm em nhận thấy có 51/100, sắp xỉ một nửa test case có dung lượng RSS mà thao tác open một database sử dụng LSM tree ít hơn so với một database sử dụng Btree có cùng số record, nhưng 51/100 vẫn chưa phải là một tỉ lệ đủ để đưa ra kết luận. Các điểm dữ liệu của biểu đồ phân bố phần lớn trong đoạn (5.7 → 6.3) và không tăng theo kích thước của database, các điểm dữ liệu cũng có sự chênh lệch không quá đáng kể thậm chí là có những điểm dữ liệu có giá trị bằng nhau, thêm vào đó chỉ có 51/100 (~50%) test case mà các điểm dữ liệu của LSM tree tốt hơn Btree nên kết luận cuối của nhóm em là dung lượng RAM mà thao tác open của database sử dụng cả LSM tree và Btree là bằng nhau.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm sinh ra khi thực hiện thao tác open một database sử dụng Btree ít hơn, dẫn chứng là có 75/100 test case mà dung lượng file tạm được sinh ra khi open một database sử dụng Btree ít hơn so với một database sử dụng LSM tree.

Kết luận: các database sử dụng LSM tree có thời gian open database nhanh hơn so với database sử dụng Btree có cùng số record, dung lượng RAM cần cho thao tác open của cả 2 database là bằng nhau nhưng dung lượng file tạm mà database sử dụng LSM tree sinh ra cao hơn so với database sử dụng Btree.

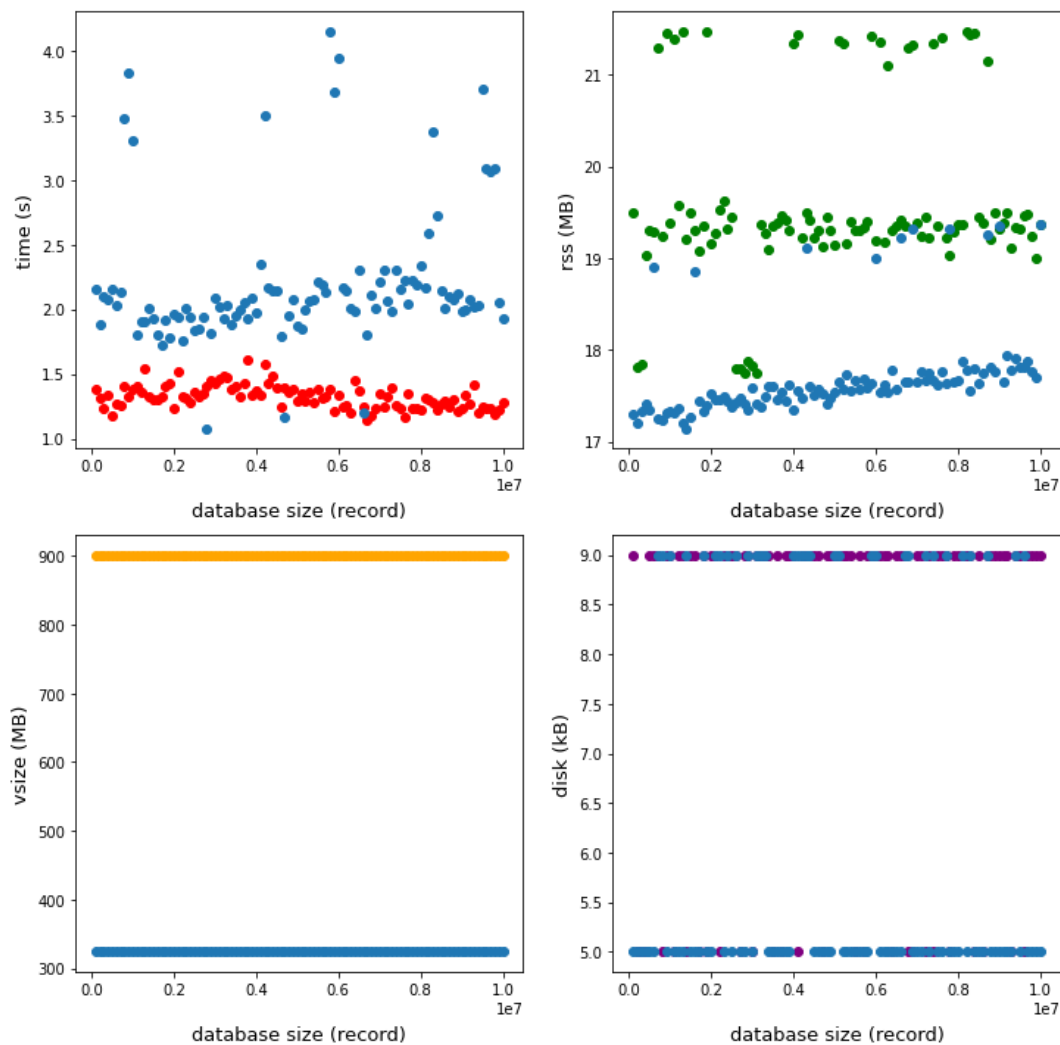


5.3. Insert

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
1 lsm_or_btree_better('insert')
2 compare('insert', KB_to_MB=True)
```

lsm's running time is better than btree - 97/100 test cases have running time less than corresponding one
btree's ram usage is less than lsm - 98/100 test cases have ram usage less than corresponding one
btree's disk usage is less than lsm - 92/100 test cases have disk usage less than corresponding one



Hình 23: Biểu đồ biểu diễn kết quả của thao tác insert trên Btree và LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em nhận xét là thời gian để insert 100000 record vào các database sử dụng LSM tree ít hơn các database sử dụng Btree, dẫn chứng cho điều này là có 97/100 test case mà thời gian thực thi của thao tác insert trên các database sử dụng LSM tree nhanh hơn database sử dụng Btree.
- Dựa vào biểu đồ RSS nhóm em nhận xét là dung lượng RSS được sử dụng bởi thao tác insert trên các database sử dụng Btree ít hơn so với khi thực thi trên các database sử dụng LSM tree, dẫn chứng là có 98/100 test case mà dung lượng RSS được dùng bởi các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số record, thêm vào đó là trên biểu đồ VSIZE, các điểm dữ liệu của các database sử dụng LSM tree đều có giá trị cao hơn các database sử dụng Btree.
- Dựa vào biểu đồ DISK, nhóm em nhận xét là dung lượng file tạm được sinh ra trong quá trình thực thi của thao tác insert trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree, dẫn chứng là có 92/100 test case mà dung lượng file tạm được sinh ra của các database sử dụng Btree ít hơn so với các database sử dụng LSM tree.

Kết luận: Các database sử dụng LSM tree có thời gian insert nhanh hơn so với các database sử dụng Btree có cùng số record, dung lượng RAM cần cho thao tác và dung lượng file tạm được sinh ra của các database sử dụng LSM tree cao hơn so với các database sử dụng Btree.

Giải thích: LSM tree có thời gian insert nhanh hơn Btree do cơ chế ghi nối tiếp vào các sector vừa được ghi trước đó, còn Btree ghi lên đĩa tại các vị trí ngẫu nhiên trong đĩa cứng, việc ghi nối tiếp sẽ nhanh hơn việc ghi vào các vị trí ngẫu nhiên, cơ chế ghi nối tiếp luôn phát huy được lợi thế của nó trên các HDD. Do LSM tree luôn lưu một memtable trong RAM, còn Btree chỉ load và RAM một node của chính nó từ đĩa cứng, không gian bộ nhớ để lưu một memtable sẽ luôn nhiều hơn lưu một node, nên dung lượng RAM mà LSM tree sử dụng luôn cao hơn Btree.

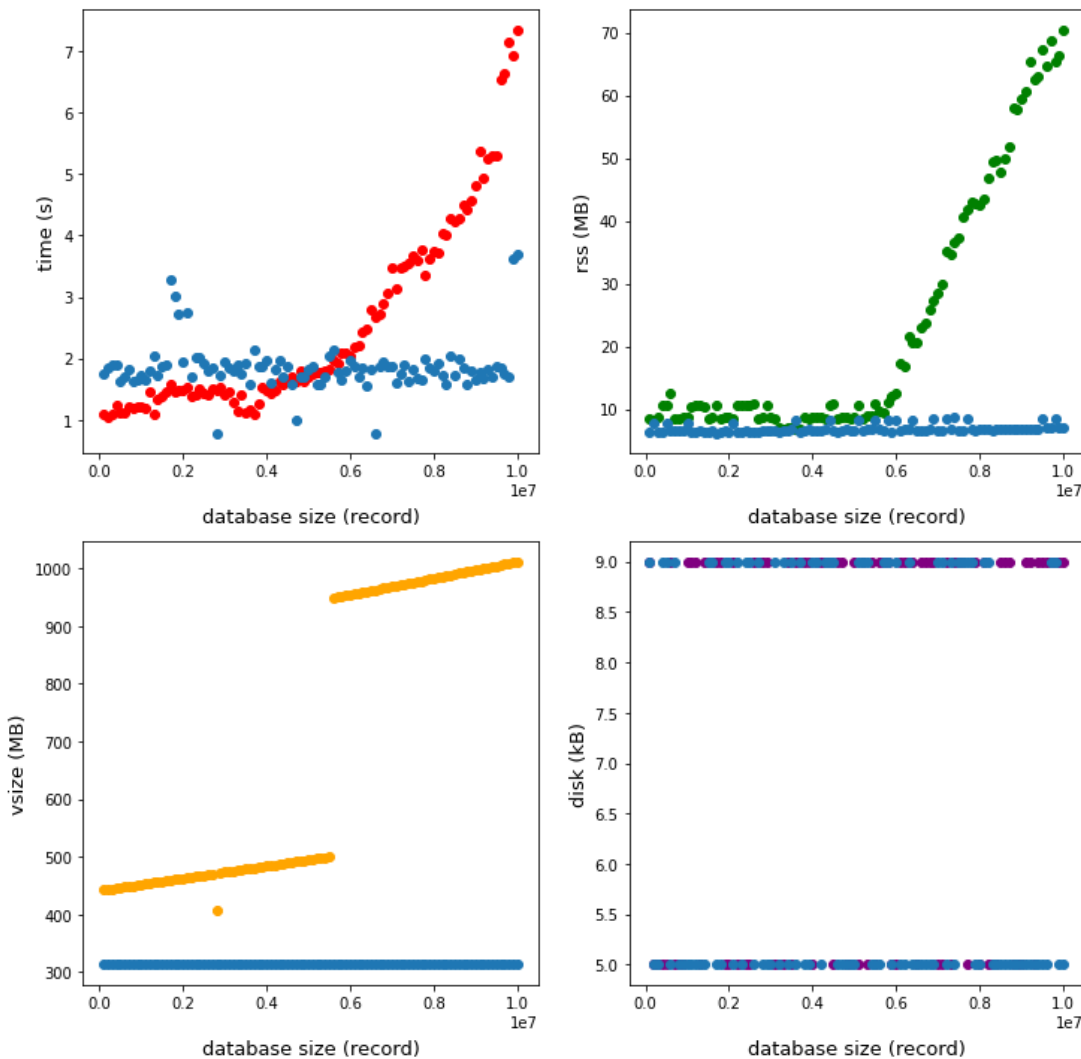


5.4. Search

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
[18] 1 lsm_or_btree_better('search')
      2 compare('search', KB_to_MB=True)
```

btree's running time is better than lsm - 51/100 test cases have running time less than corresponding one
btree's ram usage is less than lsm - 99/100 test cases have ram usage less than corresponding one
btree's disk usage is less than lsm - 79/100 test cases have disk usage less than corresponding one



Hình 24: Biểu đồ biểu diễn kết quả của thao tác search trên Btree và LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em nhận thấy có 51/100 test case mà thời gian thực thi thao tác search trên các database sử dụng Btree nhanh hơn so với các database sử dụng LSM tree, nhưng tỉ lệ 51/100 vẫn chưa đủ thuyết phục để đưa ra kết luận cuối cùng. Các điểm dữ liệu đại diện cho thời gian chạy của các database sử dụng LSM tree (màu đỏ) ít hơn so với Btree tại các database có kích thước vừa và nhỏ (100000 → 4000000 record), nhưng tại các data base có kích thước lớn thì thời gian chạy của các database này lại tăng rất nhiều, trong khi thời gian chạy của các database sử dụng Btree vẫn gần như không hay đổi so với các database có kích thước vừa và nhỏ, điều đó chứng minh rằng các database sử dụng Btree có thời gian thực hiện thao tác search ít hơn so với các database sử dụng LSM tree có cùng số record.
- Dựa vào biểu đồ RSS, nhóm em có nhận xét là dung lượng RSS cần cho thao tác search trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree, dẫn chứng là có 99/100 test case mà dung lượng RSS cần cho thao tác search trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số lượng record, thêm vào đó là trên biểu đồ VSIZE, các điểm dữ liệu của các database sử dụng LSM tree đều có giá trị cao hơn các database sử dụng Btree.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm được sinh ra trong quá trình search trên các database sử dụng Btree ít hơn các database sử dụng LSM tree, dẫn chứng là có 79/100 test case mà dung lượng file tạm được sinh ra trong quá trình search trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số lượng record.

Kết luận: Các database sử dụng Btree có thời gian search ít hơn các database sử dụng LSM tree, dung lượng RAM cần cho thao tác và dung lượng file tạm được sinh ra trong lúc thực thi của các database sử dụng Btree cũng ít hơn so với các database sử dụng LSM tree có cùng số lượng record.

Giải thích: LSM tree sẽ tìm kiếm record mong muốn trong memtable trước tiên, khi memtable không chứa record mong muốn, thì record sẽ được tìm trên đĩa cứng, quá trình này sẽ qua Bloom filter để kết luận là key của record mong muốn có trong SSTable hay không, nếu có thì việc tìm kiếm sẽ được thực hiện trên từng segment từ mới nhất đến cũ nhất trong các SSTable cho đến khi tìm được record mong muốn, việc truy vấn này rườm rà và phức tạp hơn Btree do phải thực hiện tìm kiếm trên nhiều segment.

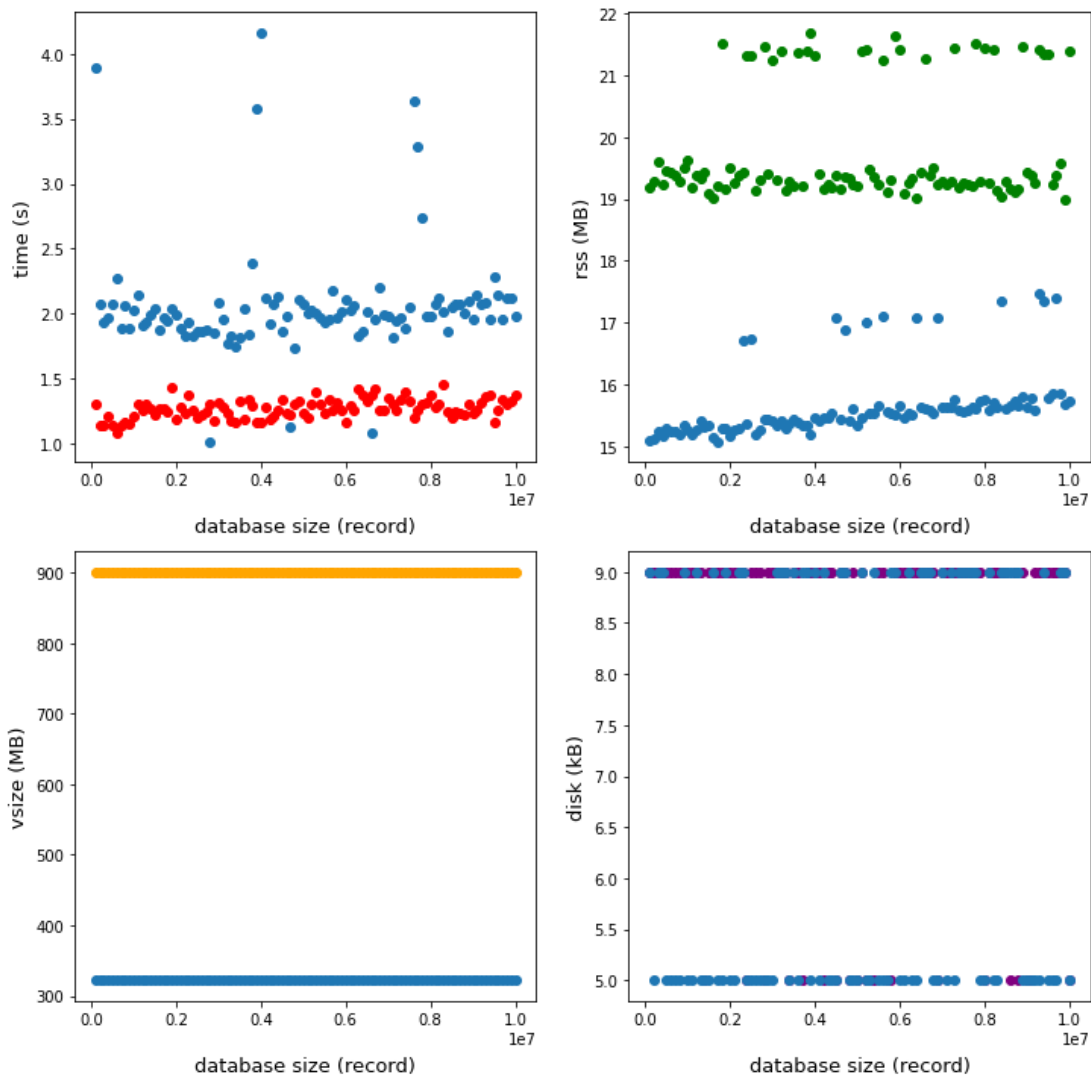


5.5. Update

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
1 lsm_or_btree_better('update')
2 compare('update', KB_to_MB=True)
```

- lsm's running time is better than btree - 97/100 test cases have running time less than corresponding one
- btree's ram usage is less than lsm - 100/100 test cases have ram usage less than corresponding one
- btree's disk usage is less than lsm - 94/100 test cases have disk usage less than corresponding one



Hình 25: Biểu đồ biểu diễn kết quả của thao tác update trên Btree và LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian thực hiện thao tác update trên các database sử dụng LSM tree nhanh hơn các database sử dụng Btree, dẫn chứng là các điểm dữ liệu của các database sử dụng LSM tree (màu đỏ) hầu hết đều nằm dưới so với các điểm dữ liệu của các database sử dụng Btree, tổng cộng có 97/100 test case mà thời gian thực thi của thao tác update trên các database sử dụng LSM tree nhanh hơn so với các database sử dụng Btree với cùng số lượng record.
- Dựa vào biểu đồ RSS nhóm em có nhận xét là dung lượng RSS được sử dụng bởi thao tác update trên các database sử dụng Btree ít hơn so với khi thực thi trên các database sử dụng LSM tree, dẫn chứng là có 100/100 test case có dung lượng RSS được dùng bởi các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số record, thêm vào đó là trên biểu đồ VSIZE, tất cả các điểm dữ liệu của các database sử dụng LSM tree (màu cam) đều có giá trị cao hơn các database sử dụng Btree.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm được sinh ra trong lúc thực hiện update của các database sử dụng Btree ít hơn các database sử dụng LSM tree, dẫn chứng là có 94/100 test case mà dung lượng file tạm được sinh ra trong lúc thực hiện update trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số lượng record.

Kết luận: thời gian chạy của thao tác update trên các database sử dụng LSM tree nhanh hơn so với các database sử dụng Btree, dung lượng RAM cần dùng và dung lượng file tạm sinh ra trong lúc chạy đều lớn hơn so với các database sử dụng Btree.

Giải thích: LSM tree có cơ chế thực hiện update khác so với Btree, LSM tree ghi trực tiếp record vừa được update vào memtable, khi memtable đạt tới một kích thước nhất định thì sẽ được ghi lên đĩa cứng, khi đó trên đĩa cứng sẽ đều chứa record cũ và record mới. Giá trị cũ của record sẽ chỉ được loại bỏ sau khi thao tác merge, trước lúc merge, nếu có bất kỳ yêu cầu nào cần truy xuất tới record chứa khóa vừa được update, thì record mới nhất sẽ luôn được trả về do LSM tree luôn đọc từ segment mới nhất. Btree phải truy xuất và tìm kiếm địa chỉ của node chứa giá trị cần update trên đĩa cứng và ghi đè giá trị lên vùng nhớ đó, nên sẽ chậm hơn LSM tree.

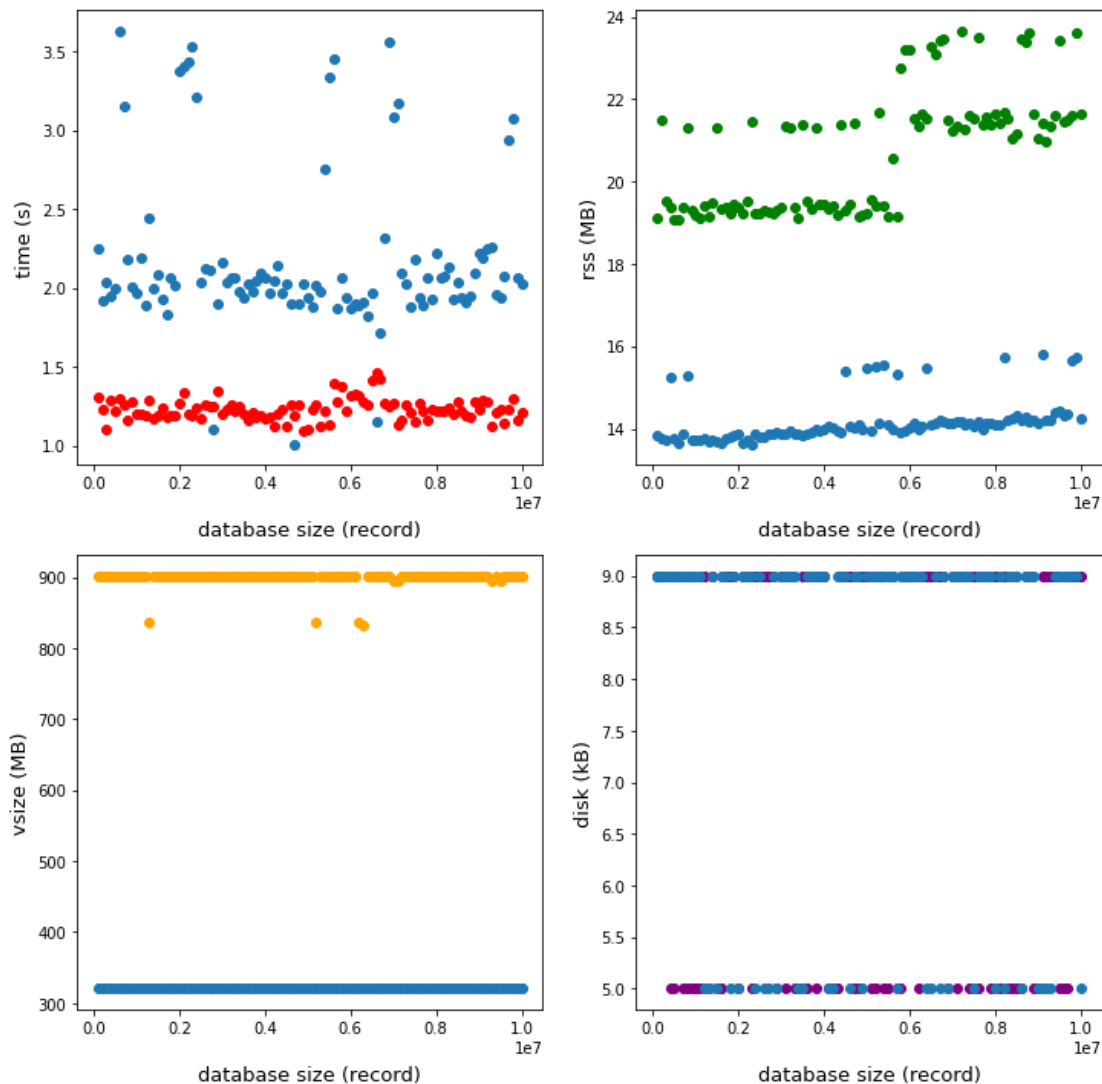


5.6. Delete

Sau quá trình thực nghiệm nhóm em thu được kết quả và biểu diễn thành các biểu đồ sau, kết quả của Btree là những điểm dữ liệu màu xanh dương và các màu còn lại là của LSM tree:

```
1 lsm_or_btree_better('delete')
2 compare('delete', KB_to_MB=True)
```

lsm's running time is better than btree - 97/100 test cases have running time less than corresponding one
btree's ram usage is less than lsm - 100/100 test cases have ram usage less than corresponding one
btree's disk usage is less than lsm - 66/100 test cases have disk usage less than corresponding one



Hình 26: Biểu đồ biểu diễn kết quả của thao tác delete trên Btree và LSM tree.



Nhận xét:

- Dựa vào biểu đồ TIME, nhóm em có nhận xét là thời gian chạy của thao tác delete trên các database sử dụng LSM tree nhanh hơn so với các database sử dụng Btree, dẫn chứng là có 97/100 test case mà thời gian chạy của thao tác delete trên các database sử dụng LSM tree nhanh hơn so với các database sử dụng Btree có cùng số lượng record.
- Dựa vào biểu đồ RSS, nhóm em có nhận xét là dung lượng RSS cần cho thao tác delete trên các database sử dụng Btree ít hơn các database sử dụng LSM tree, điều dễ nhận thấy nhất là phần lớn các điểm dữ liệu của các database sử dụng LSM tree (màu xanh lục) đều nằm trên các điểm dữ liệu của các database sử dụng Btree, dẫn chứng thuyết phục nhất là có 100/100 test case mà dung lượng RSS cần cho thao tác delete trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số lượng record.
- Dựa vào biểu đồ DISK, nhóm em có nhận xét là dung lượng file tạm được sinh ra trong lúc thực hiện delete trên các database sử dụng Btree ít hơn các database sử dụng LSM tree, dẫn chứng là có 66/100 test case mà dung lượng file tạm được sinh ra trong lúc thực hiện delete trên các database sử dụng Btree ít hơn so với các database sử dụng LSM tree có cùng số lượng record.

Kết luận: Thời gian chạy của thao tác delete trên các database sử dụng LSM tree nhanh hơn so với các database sử dụng Btree, dung lượng RAM cần thiết và dung lượng file tạm sinh ra của thao tác delete trên các database sử dụng Btree cũng ít hơn so với các database sử dụng LSM tree.

Giải thích: LSM tree có quá trình delete một record khác với Btree, thay vì tìm và delete trực tiếp như Btree thì LSM tree ghi thêm một record mới vào memtable, record này có giá trị key chính là key của record muốn delete, value của record này là một giá trị được gọi là tombstone, khi memtable đạt tới một kích thước nhất định thì sẽ được ghi lên đĩa cứng, khi đó trên đĩa cứng sẽ đều chứa record cũ và record mới. Giá trị cũ của record sẽ chỉ được loại bỏ sau quá trình compaction, trước lúc đó, nếu có bất kì truy vấn nào tới record chứa key vừa bị delete, thì sẽ luôn nhận được phản hồi là không có khóa do LSM tree luôn đọc từ segment mới nhất và record chứa key vừa bị delete có value là tombstone luôn nằm trên các segment phía sau (segment mới hơn). Btree phải truy xuất và tìm kiếm địa chỉ của node chứa record cần delete trên đĩa cứng và xóa record đó, nên sẽ chậm hơn LSM tree.



5.7. Kết luận

LSM tree được tối ưu cho việc ghi dữ liệu vào đĩa cứng bằng việc ghi tuần tự thay vì ghi ngẫu nhiên như Btree. Để tận dụng lợi thế là có cơ chế ghi đĩa cứng nhanh, các thao tác như update, delete có cách thức hoạt động khác với Btree, dẫn đến các thao tác này có thời gian thực thi nhanh hơn so với Btree, nhưng LSM tree tỏ ra không hiệu quả khi thực hiện các việc truy xuất dữ liệu, điển hình là search, do các record được chia thành các segment, nên việc truy vấn mất nhiều thời gian đọc các segment và tìm kiếm record khớp với yêu cầu, dù có sự trợ giúp của Bloom filter nhưng LSM tree vẫn có thời gian truy xuất dữ liệu chậm hơn so với Btree.

Dù có lợi thế về cơ chế ghi dữ liệu lên đĩa nhanh nhưng LSM tree lại tốn nhiều dung lượng RAM hơn Btree do phải lưu một memtable trong RAM, đồng thời dung lượng file tạm được sinh ra trong lúc chạy cũng lớn hơn Btree khi thực hiện cùng một thao tác.

6. Tổng kết

LSM tree được tối ưu cho việc ghi đĩa cứng bằng cơ chế ghi tuần tự thay vì ghi ngẫu nhiên như Btree, mọi thao tác của LSM tree có cách thức hoạt động khác với Btree để tận dụng lợi thế ghi đĩa cứng nhanh.

LSM tree lưu các record trên đĩa cứng theo định dạng là SSTable, trong các SSTable chứa các thành phần được gọi là segment, các phần tử trong segment này được sắp xếp để tối ưu việc tìm kiếm. LSM tree sử dụng Bloom filter để hỗ trợ việc tìm kiếm, truy xuất dữ liệu nhưng vẫn không hiệu quả bằng Btree.

LSM tree luôn lưu một memtable trong RAM nên dung lượng RAM cần cho các thao tác luôn lớn hơn Btree, các thao tác sẽ luôn được thực hiện trên memtable trước khi thực hiện với đĩa cứng. Khi dung lượng của memtable trong RAM đạt một ngưỡng nhất định thì nó sẽ được ghi lên đĩa cứng, khi số lượng SSTable đủ lớn thì quá trình compaction sẽ được thực hiện trong nền để merge các file SSTable.

LSM tree không thích hợp với các database có nhu cầu truy xuất dữ liệu với tần suất cao, LSM tree chỉ thích hợp với các database có nhiệm vụ chủ yếu là ghi và lưu dữ liệu, điển hình là ghi log.

Link github chứa toàn bộ mã nguồn, bảng tính và toàn bộ kết quả: [tại đây](#).

Link colab dùng để vẽ biểu đồ và mô phỏng kết quả: [tại đây](#)

Link mã nguồn của Wiredtiger: [tại đây](#)



TÀI LIỆU THAM KHẢO

- [1]. [The Log-Structured Merge-Tree \(LSM-Tree\)](#)
- [2]. [Official Wiredtiger document](#)
- [3]. [Official Wiredtiger API document](#)
- [4]. [B-Tree vs Log-Structured Merge-Tree](#)
- [5]. [Hiểu về mongodb wiredtiger](#)
- [6]. [Log Structured Merge Trees – medium](#)
- [7]. [Understanding LSM Trees: What Powers Write-Heavy Databases](#)
- [8]. [Bloom filter](#)
- [9]. [AVL tree](#)
- [10]. [2–3 tree](#)
- [11]. [Red–black tree](#)
- [12]. [Write amplification](#)
- [13]. [Log-structured merge-tree](#)
- [14]. [What is a LSM Tree?](#)

