 **BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

**------ oOo ------**

BÀI TẬP MÔN: CƠ SỞ DỮ LIỆU NÂNG CAO

**ĐỀ TÀI:**

**Hệ quản trị cơ sở dữ liệu LEVELDB**

**Người hướng dẫn: PGS.TS. Nguyễn Ngọc Hóa**

**Học viên: Nguyễn Công Chiến**

**Lê Văn Trọng**

**Trần Văn Khánh**

**Lớp: Cao học HTTT – K25 (nhóm 5)**

**Hà Nội: 1/2019**

Mục lục

[I. LevelDB là gì? 3](#_Toc535683831)

[1.Features (Các đặc điểm) 3](#_Toc535683832)

[2.Limitations (Các giới hạn) 4](#_Toc535683833)

[3.Performance 4](#_Toc535683834)

[Setup (Thiết lập) 4](#_Toc535683835)

[4.Write performance (Hiệu suất ghi) 5](#_Toc535683836)

[5.Read performance (Hiệu suất đọc) 6](#_Toc535683837)

[II. Mô hình dữ liệu được sử dụng trong DBMS, 7](#_Toc535683838)

[1.Log Structured Merge (LSM) tree 7](#_Toc535683839)

[2.String Sorted Table files (SST) 7](#_Toc535683840)

[3.Table file hierarchy (Phân cấp file Table) 8](#_Toc535683841)

[4.Advanced features (Các tính năng advanced) 10](#_Toc535683842)

[III. Phương pháp và kỹ thuật xử lý các truy vấn/giao tác đồng thời 12](#_Toc535683843)

[1.Opening A Database 12](#_Toc535683844)

[2.Status 12](#_Toc535683845)

[3.Closing A Database 12](#_Toc535683846)

[4.Reads And Writes 13](#_Toc535683847)

[5.Atomic Updates 13](#_Toc535683848)

[6.Synchronous Writes (Ghi đồng bộ) 14](#_Toc535683849)

[7.Concurrency (Tính đồng thời) 15](#_Toc535683850)

[8.Iteration (Tính lặp lại) 15](#_Toc535683851)

[9.Snapshots (Ảnh chụp nhanh) 16](#_Toc535683852)

[10.Slice (Phân mảnh, chia lát) 17](#_Toc535683853)

[11.Comparators (Phần tử so sánh) 18](#_Toc535683854)

[12.Performance (Thực thi, hiệu suất) 20](#_Toc535683856)

[13.Block size (Kích thước Block) 20](#_Toc535683857)

[14.Compression (Nén dữ liệu) 20](#_Toc535683858)

[15.Cache (Bộ nhớ cache) 20](#_Toc535683859)

[16.Key Layout (Bố cục Key) 22](#_Toc535683860)

[17.Filters (Các bộ lọc) 22](#_Toc535683861)

[18.Checksums (Tổng kiểm) 24](#_Toc535683862)

[19.Approximate Sizes (Các kích thước gần đúng) 25](#_Toc535683863)

[20.Environment (Môi trường) 25](#_Toc535683864)

[21.Porting (Chuyển) 26](#_Toc535683865)

[IV. So sánh, đánh giá hiệu năng DBMS đó so với MySQL 27](#_Toc535683866)

**LevelDB**

* + 1. **LevelDB là gì?**

LevelDB là một kho lưu trữ dữ liệu khóa / giá trị (key/value) mã nguồn mở, không phụ thuộc, nguồn mở. Nó được phát triển vào năm 2011 bởi Jeff Dean và Sanjay Ghemawat, các nhà nghiên cứu từ Google. Nó được viết bằng C ++ mặc dù nó có các ràng buộc của bên thứ ba đối với hầu hết các ngôn ngữ lập trình phổ biến. Tất nhiên bao gồm JavaScript / Node.js.

LevelDB dựa trên các ý tưởng trong GoogleTable BigTable nhưng không chia sẻ mã với BigTable, điều này cho phép nó được cấp phép để phát hành nguồn mở. Dean và Ghemawat đã phát triển LevelDB như một sự thay thế cho SQLite như là kho lưu trữ dự phòng cho việc triển khai Chrome Index IndexedDB.

Kể từ đó, nó đã được áp dụng rất rộng rãi trong toàn ngành và đóng vai trò là phần hỗ trợ cho một số cơ sở dữ liệu mới và hiện là phần lưu trữ được đề xuất cho dự phòng của Riak.

## **1.Features (Các đặc điểm)**

• Khóa và giá trị là các mảng byte tùy ý.

• Dữ liệu được lưu trữ được sắp xếp theo khóa.

• Người gọi có thể cung cấp chức năng so sánh tùy chỉnh để ghi đè thứ tự sắp xếp.

• Các thao tác cơ bản là Đặt (khóa, giá trị), Nhận (khóa), Xóa (khóa), Put(key,value), Get(key), Delete(key).

• Nhiều thay đổi có thể được thực hiện trong một đợt nguyên tử (atomic batch).

• Người dùng có thể tạo ảnh chụp nhanh thoáng qua để có chế độ xem dữ liệu nhất quán.

• Lặp lại tiến và lùi được hỗ trợ trên dữ liệu.

• Dữ liệu được tự động nén bằng thư viện nén Snappy.

• Hoạt động bên ngoài (hoạt động của hệ thống tệp, v.v.) được chuyển tiếp qua giao diện ảo để người dùng có thể tùy chỉnh các tương tác của hệ điều hành.

## **2.Limitations (Các giới hạn)**

• Đây không phải là cơ sở dữ liệu SQL. Nó không có mô hình dữ liệu quan hệ, nó không hỗ trợ các truy vấn SQL và nó không hỗ trợ các chỉ mục.

• Chỉ một tiến trình duy nhất (có thể đa luồng) có thể truy cập vào một cơ sở dữ liệu cụ thể tại một thời điểm.

• Không có hỗ trợ máy client-server tích hợp vào thư viện. Một ứng dụng cần sự hỗ trợ như vậy sẽ phải bọc máy chủ của riêng họ xung quanh thư viện.

**3.Performance** (Hiệu suất)

Đây là một báo cáo hiệu suất (có giải thích) từ hoạt động của chương trình db\_bench đi kèm. Kết quả có phần ồn ào, nhưng đủ để có được ước tính hiệu suất sân bóng.

## **Setup (Thiết lập)**

Chúng tôi sử dụng một cơ sở dữ liệu với một triệu mục. Mỗi mục có khóa 16 byte và giá trị 100 byte. Các giá trị được sử dụng bởi nén chuẩn đến khoảng một nửa kích thước ban đầu của chúng.

|  |
| --- |
| LevelDB: version 1.1  Date: Sun May 1 12:11:26 2011  CPU: 4 x Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz  CPUCache: 4096 KB  Keys: 16 bytes each  Values: 100 bytes each (50 bytes after compression)  Entries: 1000000  Raw Size: 110.6 MB (estimated)  File Size: 62.9 MB (estimated) |

## **4.Write performance (Hiệu suất ghi)**

Các điểm chuẩn " fill " tạo ra một cơ sở dữ liệu hoàn toàn mới, theo thứ tự hoặc theo thứ tự ngẫu nhiên. Điểm chuẩn "fillsync" sẽ xóa dữ liệu từ hệ điều hành vào đĩa sau mỗi hoạt động; các hoạt động ghi khác để lại dữ liệu trong bộ đệm bộ đệm của hệ điều hành trong một thời gian. Điểm chuẩn " overwrite " ghi ngẫu nhiên cập nhật các khóa hiện có trong cơ sở dữ liệu.

|  |
| --- |
| fillseq : 1.765 micros/op; 62.7 MB/s  fillsync : 268.409 micros/op; 0.4 MB/s (10000 ops)  fillrandom : 2.460 micros/op; 45.0 MB/s  overwrite : 2.380 micros/op; 46.5 MB/s |

Mỗi "op" ở trên tương ứng với một lần ghi của một cặp khóa / giá trị (key/value). Tức là, một điểm chuẩn viết ngẫu nhiên đạt khoảng 400.000 lần viết mỗi giây.

Mỗi thao tác "fillsync" có chi phí thấp hơn nhiều (0,3 mili giây) so với tìm kiếm trên đĩa (thường là 10 mili giây). Chúng tôi nghi ngờ rằng điều này là do bản thân đĩa cứng đang đệm bộ cập nhật trong bộ nhớ của nó và phản hồi trước khi dữ liệu được ghi vào đĩa. Điều này có thể an toàn hoặc không dựa trên việc đĩa cứng có đủ năng lượng để lưu bộ nhớ của nó trong trường hợp mất điện hay không.

## **5.Read performance (Hiệu suất đọc)**

Chúng tôi liệt kê hiệu suất của việc đọc tuần tự theo cả hai hướng thuận và ngược, và cả hiệu suất của việc tra cứu ngẫu nhiên. Lưu ý rằng cơ sở dữ liệu được tạo bởi điểm chuẩn là khá nhỏ. Do đó, báo cáo đặc trưng cho hiệu suất của leveldb khi bộ làm việc vừa với bộ nhớ. Chi phí đọc một phần dữ liệu không có trong bộ đệm bộ đệm của hệ điều hành sẽ bị chi phối bởi một hoặc hai lần tìm đĩa cần thiết để lấy dữ liệu từ đĩa. Hiệu suất ghi sẽ hầu như không bị ảnh hưởng bởi liệu bộ làm việc có phù hợp với bộ nhớ hay không.

|  |
| --- |
| readrandom : 16.677 micros/op; (approximately 60,000 reads per second)  readseq : 0.476 micros/op; 232.3 MB/s  readreverse : 0.724 micros/op; 152.9 MB/s |

LevelDB thu nhỏ dữ liệu lưu trữ cơ bản của nó trong nền để cải thiện hiệu suất đọc. Các kết quả được liệt kê ở trên đã được thực hiện ngay lập tức sau khi viết rất nhiều ngẫu nhiên. Kết quả sau khi tính toán (thường được kích hoạt tự động) là tốt hơn.

|  |
| --- |
| readrandom : 11.602 micros/op; (approximately 85,000 reads per second)  readseq : 0.423 micros/op; 261.8 MB/s  readreverse : 0.663 micros/op; 166.9 MB/s |

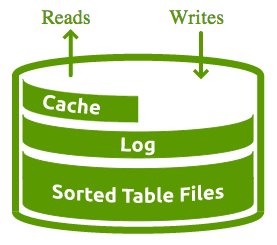
Một số chi phí cao của việc đọc đến từ việc giải nén lặp lại các khối đọc từ đĩa. Nếu chúng tôi cung cấp đủ bộ đệm cho leveldb để nó có thể giữ các khối không nén trong bộ nhớ, hiệu suất đọc sẽ cải thiện lại:

|  |
| --- |
| readrandom : 9.775 micros/op; (approximately 100,000 reads per second before compaction)  readrandom : 5.215 micros/op; (approximately 190,000 reads per second after compaction) |

* + 1. **Mô hình dữ liệu được sử dụng trong DBMS,**

Kiến trúc cơ bản

## **1.Log Structured Merge (LSM) tree**



Tất cả ghi vào một kho LevelDB đi thẳng vào một **log**  và một “memtable” có thể ghi nhớ. Log thường xuyên được xóa (**flushed**) vào các tệp bảng chuỗi được sắp xếp (SST) trong đó dữ liệu có một nơi lưu trữ cố định hơn.

Đọc trên một kho lưu trữ dữ liệu hợp nhất hai cấu trúc dữ liệu riêng biệt này, log và tệp SST. Các tệp SST đại diện cho dữ liệu xử lý xong và log đại diện cho dữ liệu mới, bao gồm các thao tác xóa.

Một bộ đệm (cache) có thể cấu hình được sử dụng để tăng tốc độ đọc phổ biến. Bộ nhớ cache có thể đủ lớn để phù hợp với toàn bộ các working set đang active trong bộ nhớ, tùy thuộc vào ứng dụng.

## **2.String Sorted Table files (SST)**

Mỗi tệp SST được giới hạn ở ~ 2MB, do đó, một kho LevelDB lớn sẽ có nhiều tệp này. Tệp SST được chia nội bộ thành các blocks 4K, mỗi khối có thể được đọc trong một thao tác. Khối cuối cùng là một chỉ mục trỏ đến điểm bắt đầu của mỗi khối dữ liệu và là khóa của mục nhập khi bắt đầu khối. Bộ lọc Bloom được sử dụng để tăng tốc độ tra cứu, cho phép quét nhanh chỉ mục để tìm khối có thể chứa mục nhập mong muốn.

Khóa có thể có tiền tố được chia sẻ (**shared prefixes** ) trong các khối. Bất kỳ tiền tố phổ biến nào cho các khóa trong một khối sẽ được lưu trữ một lần, với các mục tiếp theo chỉ lưu trữ hậu tố duy nhất. Sau một số lượng mục nhập cố định trong một khối, tiền tố được chia sẻ là cài đặt lại cài đặt trước; giống như một khung hình chính trong một codec video. Tiền tố được chia sẻ có nghĩa là không gian tên dài dòng của các khóa không dẫn đến yêu cầu lưu trữ quá mức.

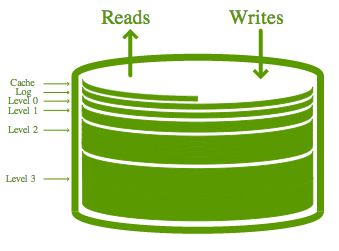
## **3.Table file hierarchy (Phân cấp file Table)**

Các tệp bảng không được lưu trữ theo một trình tự đơn giản, thay vào đó, chúng được tổ chức thành một loạt **levels**. Đây là cấp độ cấp độ cao trong LevelDB.

Các mục nhập trực tiếp từ log được sắp xếp ở Cấp 0, một bộ gồm tối đa 4 tệp. Khi các mục nhập bổ sung buộc Cấp 0 vượt quá tối đa 4 tệp, một trong các tệp SST được chọn và hợp nhất với các tệp SST tạo nên Cấp 1, là một tập hợp tối đa 10 MB tệp. Tiến trình này tiếp tục, với các mức tràn và một tệp tại một thời điểm được hợp nhất với (tối đa 3) tệp SST chồng chéo ở cấp tiếp theo. Mỗi cấp độ vượt cấp 1 là 10 lần kích thước của cấp độ trước.

|  |  |
| --- | --- |
| **Log:** | Max size of 4MB (configurable), then flushed into a set of Level 0 SST files |
| **Level 0:** | Max of 4 SST files, then one file compacted into Level 1 |
| **Level 1:** | Max total size of 10MB, then one file compacted into Level 2 |
| **Level 2:** | Max total size of 100MB, then one file compacted into Level 3 |
| **Level 3+:** | Max total size of 10 x previous level, then one file compacted into next level |

|  |
| --- |
| 0 ↠ 4 SST, 1 ↠ 10M, 2 ↠ 100M, 3 ↠ 1G, 4 ↠ 10G, 5 ↠ 100G, 6 ↠ 1T, 7 ↠ 10T |



Tổ chức này thành các mức tối thiểu hóa việc sắp xếp lại phải diễn ra khi các mục mới được chèn vào giữa một loạt các khóa. Mỗi lần sắp xếp lại, hay phần mềm nén, được giới hạn trong một phần nhỏ của kho lưu trữ dữ liệu. Cấu trúc phân cấp thường dẫn đến dữ liệu ở các cấp cao hơn là dữ liệu trưởng thành nhất, với dữ liệu tươi hơn được lưu trữ trong nhật ký và các cấp ban đầu. Vì các mức ban đầu tương đối nhỏ, việc ghi đè và xóa các mục nhập sẽ tốn ít chi phí hơn so với khi nó xuất hiện ở các cấp cao hơn, nhưng điều này phù hợp với cơ sở dữ liệu điển hình nơi bạn có một bộ dữ liệu trưởng thành lớn và một bộ dữ liệu mới dễ bay hơi hơn (tất nhiên điều này không phải lúc nào cũng đúng, vì vậy hiệu suất sẽ thay đổi đối với các kiểu ghi và truy xuất dữ liệu khác nhau).

Một hoạt động tra cứu (**lookup**) cũng phải đi qua các cấp để tìm mục yêu cầu. Một thao tác đọc yêu cầu một khóa đã cho trước tiên phải tìm trong nhật ký, nếu không tìm thấy ở đó, nó sẽ tìm ở Cấp 0, chuyển lên Cấp 1, v.v. Theo cách này, một thao tác tra cứu phải chịu tối thiểu một lần đọc cho mỗi cấp độ phải được tìm kiếm trước khi tìm thấy mục yêu cầu. Việc tìm kiếm một khóa không tồn tại phải tìm kiếm mọi cấp độ trước khi có thể trả về NotFound ((trừ khi một thao tác Del được ghi lại cho khóa đó trong nhật ký).

## **4.Advanced features (Các tính năng advanced)**

* cung cấp một bộ sưu tập các hoạt động Put và / hoặc Del là nguyên tử; nghĩa là, toàn bộ tập hợp các hoạt động thành công hay thất bại trong một hoạt động Batch duy nhất.
* các trình vòng lặp có thể bắt đầu tại bất kỳ khóa nào trong cửa hàng LevelDB (ngay cả khi khóa đó không tồn tại, nó sẽ chỉ đơn giản chuyển sang khóa từ vựng tiếp theo) và có thể di chuyển tiến và lùi qua cửa hàng.
* một ảnh chụp nhanh (snapsot) cung cấp một tham chiếu đến trạng thái của cơ sở dữ liệu tại một thời điểm. Truy vấn đọc (Nhận và lặp) có thể được thực hiện đối với các ảnh chụp nhanh cụ thể để truy xuất các mục khi chúng tồn tại tại thời điểm ảnh chụp nhanh được tạo. Mỗi trình vòng lặp tạo ra một ảnh chụp nhanh ẩn (trừ khi nó được yêu cầu đối với một ảnh chụp nhanh được tạo rõ ràng). Điều này có nghĩa là bất kể trình lặp và hoạt động trong bao lâu, tập dữ liệu mà nó hoạt động sẽ luôn giống như tại thời điểm trình lặp được tạo

Một số chi tiết về các tính năng nâng cao này sẽ được đề cập trong hai bài viết tiếp theo, khi chúng ta chuyển sang xem cách sử dụng LevelDB để đơn giản hóa việc quản lý dữ liệu trong ứng dụng Node của bạn.

Nếu bạn muốn tìm hiểu thêm và có thể chờ đợi bài viết tiếp theo, hãy xem dự án LevelUP trên GitHub vì đây là trọng tâm của phần lớn hoạt động LevelDB trong cộng đồng Node tại thời điểm này.

* + 1. **Phương pháp và kỹ thuật xử lý các truy vấn/giao tác đồng thời**

## **1.Opening A Database**

Một cơ sở dữ liệu leveldb có một tên tương ứng với một thư mục hệ thống tệp. Tất cả các nội dung của cơ sở dữ liệu được lưu trữ trong thư mục này. Ví dụ sau đây cho thấy cách mở cơ sở dữ liệu, tạo cơ sở dữ liệu nếu cần:

|  |
| --- |
| #include <cassert>  #include "leveldb/db.h"  leveldb::DB\* db;  leveldb::Options options;  options.create\_if\_missing = true;  leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);  assert(status.ok());  ... |

Nếu bạn muốn đưa ra lỗi nếu cơ sở dữ liệu đã tồn tại, hãy thêm dòng sau vào leveldb :: DB :: Mở cuộc gọi:

|  |
| --- |
| options.error\_if\_exists = true; |

## **2.Status**

Bạn có thể đã nhận thấy leveldb :: Kiểu trạng thái ở trên. Các giá trị của loại này được trả về bởi hầu hết các hàm trong leveldb có thể gặp lỗi. Bạn có thể kiểm tra xem kết quả như vậy có ổn không, đồng thời in thông báo lỗi liên quan:

|  |
| --- |
| leveldb::Status s = ...;  if (!s.ok()) cerr << s.ToString() << endl; |

## **3.Closing A Database**

Khi bạn đã hoàn thành với một cơ sở dữ liệu, chỉ cần xóa đối tượng cơ sở dữ liệu. Thí dụ:

|  |
| --- |
| ... open the db as described above ...  ... do something with db ...  delete db; |

## **4.Reads And Writes**

Cơ sở dữ liệu cung cấp các phương thức Đặt, Xóa và Nhận để sửa đổi / truy vấn cơ sở dữ liệu. Ví dụ: đoạn mã sau di chuyển giá trị được lưu trữ dưới key1 sang key2.

|  |
| --- |
| std::string value;  leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);  if (s.ok()) s = db->Put(leveldb::WriteOptions(), key2, value);  if (s.ok()) s = db->Delete(leveldb::WriteOptions(), key1); |

## **5.Atomic Updates**

Lưu ý rằng nếu tiến trình chết sau khi đặt Key2 nhưng trước khi xóa key1, cùng một giá trị có thể được lưu trữ dưới nhiều khóa. Những vấn đề như vậy có thể tránh được bằng cách sử dụng lớp WriteBatch để áp dụng nguyên bản một tập các bản cập nhật:

|  |
| --- |
| #include "leveldb/write\_batch.h"  ...  std::string value;  leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);  if (s.ok()) {  leveldb::WriteBatch batch;  batch.Delete(key1);  batch.Put(key2, value);  s = db->Write(leveldb::WriteOptions(), &batch);  } |

WriteBatch giữ một chuỗi các chỉnh sửa được thực hiện cho cơ sở dữ liệu và các chỉnh sửa này trong lô được áp dụng theo thứ tự. Lưu ý rằng chúng tôi đã gọi Xóa trước khi đặt để nếu key1 giống hệt với key2, chúng tôi sẽ không hoàn toàn bỏ nhầm giá trị.

Ngoài các lợi ích nguyên tử của nó, WriteBatch cũng có thể được sử dụng để tăng tốc các cập nhật hàng loạt bằng cách đặt nhiều đột biến riêng lẻ vào cùng một đợt.

## **6.Synchronous Writes (Ghi đồng bộ)**

Theo mặc định, mỗi lần ghi vào leveldb là không đồng bộ: nó trả về sau khi đẩy ghi từ tiến trình vào hệ điều hành. Việc chuyển từ bộ nhớ hệ điều hành sang bộ lưu trữ liên tục bên dưới xảy ra không đồng bộ. Cờ đồng bộ hóa có thể được bật cho một lần ghi cụ thể để làm cho thao tác ghi không trở lại cho đến khi dữ liệu được ghi đã được đẩy tất cả các cách để lưu trữ liên tục. (Trên các hệ thống Posix, điều này được thực hiện bằng cách gọi fsync (...) hoặc fdatasync (...) hoặc msync (..., MS\_SYNC) trước khi hoạt động ghi trở lại.)

|  |
| --- |
| leveldb::WriteOptions write\_options;  write\_options.sync = true;  db->Put(write\_options, ...); |

Ghi không đồng bộ thường nhanh hơn một nghìn lần so với ghi đồng bộ. Nhược điểm của việc ghi không đồng bộ là sự cố của máy có thể khiến một vài cập nhật cuối cùng bị mất. Lưu ý rằng sự cố chỉ là quá trình viết (tức là không phải khởi động lại) sẽ không gây ra bất kỳ tổn thất nào kể cả khi đồng bộ hóa sai, một bản cập nhật được đẩy từ bộ nhớ quy trình vào hệ điều hành trước khi nó được coi là xong.

Viết không đồng bộ thường có thể được sử dụng một cách an toàn. Ví dụ: khi tải một lượng lớn dữ liệu vào cơ sở dữ liệu, bạn có thể xử lý các cập nhật bị mất bằng cách khởi động lại tải hàng loạt sau khi gặp sự cố. Một sơ đồ kết hợp cũng có thể xảy ra khi mọi lần ghi Nth đều đồng bộ và trong trường hợp xảy ra sự cố, tải hàng loạt được khởi động lại ngay sau khi ghi đồng bộ cuối cùng kết thúc bởi lần chạy trước. (Việc ghi đồng bộ có thể cập nhật điểm đánh dấu mô tả nơi khởi động lại khi gặp sự cố.)

WriteBatch cung cấp một sự thay thế cho việc ghi không đồng bộ. Nhiều bản cập nhật có thể được đặt trong cùng một WriteBatch và được áp dụng cùng nhau bằng cách sử dụng ghi đồng bộ (nghĩa là write\_options.sync được đặt thành đúng). Chi phí bổ sung của ghi đồng bộ sẽ được khấu hao trên tất cả các ghi trong lô.

## **7.Concurrency (Tính đồng thời)**

Một cơ sở dữ liệu chỉ có thể được mở bởi một quá trình tại một thời điểm. Việc thực hiện leveldb có được một khóa từ hệ điều hành để ngăn chặn việc sử dụng sai. Trong một tiến trình đơn lẻ, cùng một đối tượng leveldb :: DB có thể được chia sẻ an toàn bởi nhiều luồng đồng thời. Tức là, các luồng khác nhau có thể ghi vào hoặc tìm nạp các trình lặp hoặc gọi Get trên cùng một cơ sở dữ liệu mà không có bất kỳ đồng bộ hóa bên ngoài nào (việc thực hiện leveldb sẽ tự động thực hiện đồng bộ hóa theo yêu cầu). Tuy nhiên, các đối tượng khác (như Iterator và WriteBatch) có thể yêu cầu đồng bộ hóa bên ngoài. Nếu hai luồng chia sẻ một đối tượng như vậy, chúng phải bảo vệ quyền truy cập vào nó bằng giao thức khóa riêng của chúng. Thêm chi tiết có sẵn trong các tập tin tiêu đề công cộng.

## **8.Iteration (Tính lặp lại)**

Ví dụ sau minh họa cách in tất cả các cặp khóa, giá trị trong cơ sở dữ liệu.

|  |
| --- |
| leveldb::Iterator\* it = db->NewIterator(leveldb::ReadOptions());  for (it->SeekToFirst(); it->Valid(); it->Next()) {  cout << it->key().ToString() << ": " << it->value().ToString() << endl;  }  assert(it->status().ok()); // Check for any errors found during the scan  delete it; |

Biến thể sau đây cho biết cách xử lý chỉ các phím trong phạm vi [bắt đầu, giới hạn) ([start,limit):

|  |
| --- |
| for (it->Seek(start);  it->Valid() && it->key().ToString() < limit;  it->Next()) {  ...  } |

Bạn có thể gặp phải những vấn đề khác (Hồi tính từ: bạn có thể gặp nhau).

|  |
| --- |
| for (it->SeekToLast(); it->Valid(); it->Prev()) {  ...  } |

## **9.Snapshots (Ảnh chụp nhanh)**

Ảnh chụp nhanh cung cấp chế độ xem chỉ đọc nhất quán trên toàn bộ trạng thái của kho lưu trữ khóa-giá trị. ReadOptions :: snapshot có thể không phải là NULL để chỉ ra rằng một lần đọc nên hoạt động trên một phiên bản cụ thể của trạng thái DB. Nếu ReadOptions :: snapshot là NULL, thì read sẽ hoạt động trên một snapshot ngầm của trạng thái hiện tại. Ảnh chụp nhanh được tạo bằng phương thức DB :: GetSnapshot ():

|  |
| --- |
| leveldb::ReadOptions options;  options.snapshot = db->GetSnapshot();  ... apply some updates to db ...  leveldb::Iterator\* iter = db->NewIterator(options);  ... read using iter to view the state when the snapshot was created ...  delete iter;  db->ReleaseSnapshot(options.snapshot); |

Lưu ý rằng khi một ảnh chụp nhanh không còn cần thiết nữa, nó sẽ được giải phóng bằng giao diện DB :: ReleaseSnapshot. Điều này cho phép việc triển khai thoát khỏi trạng thái đang được duy trì chỉ để hỗ trợ việc đọc kể từ ảnh chụp nhanh đó.

## **10.Slice (Phân mảnh, chia lát)**

Giá trị trả về của các lệnh gọi it-> key () và it-> value () ở trên là các thể hiện của leveldb :: loại Slice. Slice là một cấu trúc đơn giản chứa chiều dài và con trỏ tới một mảng byte ngoài. Trả lại một Slice là một cách thay thế rẻ hơn để trả về chuỗi std :: vì chúng ta không cần phải sao chép các khóa và giá trị có khả năng lớn. Ngoài ra, các phương thức leveldb không trả về các chuỗi kiểu C kết thúc null vì các khóa và giá trị leveldb được phép chứa các byte '\ 0'.

Chuỗi C ++ và chuỗi kiểu C kết thúc null có thể dễ dàng chuyển đổi thành Slice:

|  |
| --- |
| leveldb::Slice s1 = "hello";  std::string str("world");  leveldb::Slice s2 = str; |

Một Slice có thể dễ dàng chuyển đổi trở lại chuỗi C ++:

|  |
| --- |
| std::string str = s1.ToString();  assert(str == std::string("hello")); |

Hãy cẩn thận khi sử dụng Slice vì nó phụ thuộc vào người gọi để đảm bảo rằng mảng byte bên ngoài mà các điểm Slice vẫn tồn tại trong khi Slice đang được sử dụng. Ví dụ: lỗi sau:

|  |
| --- |
| leveldb::Slice slice;  if (...) {  std::string str = ...;  slice = str;  }  Use(slice); |

Khi câu lệnh if vượt quá phạm vi, str sẽ bị hủy và bộ lưu trữ dự phòng cho lát sẽ biến mất.

## **11.Comparators (Phần tử so sánh)**

Các ví dụ trước đã sử dụng hàm thứ tự mặc định cho khóa, thứ tự các byte theo từ vựng. Tuy nhiên, bạn có thể cung cấp một bộ so sánh tùy chỉnh khi mở cơ sở dữ liệu. Ví dụ: giả sử mỗi khóa cơ sở dữ liệu bao gồm hai số và chúng ta nên sắp xếp theo số thứ nhất, phá vỡ các mối quan hệ theo số thứ hai. Đầu tiên, xác định một lớp con thích hợp của leveldb :: Trình so sánh biểu thị các quy tắc sau:

|  |
| --- |
| class TwoPartComparator : public leveldb::Comparator {  public:  // Three-way comparison function:  // if a < b: negative result  // if a > b: positive result  // else: zero result  int Compare(const leveldb::Slice& a, const leveldb::Slice& b) const {  int a1, a2, b1, b2;  ParseKey(a, &a1, &a2);  ParseKey(b, &b1, &b2);  if (a1 < b1) return -1;  if (a1 > b1) return +1;  if (a2 < b2) return -1;  if (a2 > b2) return +1;  return 0;  }  // Ignore the following methods for now:  const char\* Name() const { return "TwoPartComparator"; }  void FindShortestSeparator(std::string\*, const leveldb::Slice&) const {}  void FindShortSuccessor(std::string\*) const {}  }; |

Bây giờ tạo cơ sở dữ liệu bằng cách sử dụng bộ so sánh tùy chỉnh này:

|  |
| --- |
| TwoPartComparator cmp;  leveldb::DB\* db;  leveldb::Options options;  options.create\_if\_missing = true;  options.comparator = &cmp;  leveldb::Status status = leveldb::DB::Open(options, "/tmp/testdb", &db);  ... |

### Backwards compatibility (Khả năng tương thích ngược)

Kết quả của phương thức Tên của bộ so sánh được đính kèm với cơ sở dữ liệu khi nó được tạo và được kiểm tra trên mỗi cơ sở dữ liệu tiếp theo mở. Nếu tên thay đổi, leveldb :: DB :: Cuộc gọi mở sẽ thất bại. Do đó, thay đổi tên khi và chỉ khi định dạng khóa và chức năng so sánh mới không tương thích với cơ sở dữ liệu hiện có và bạn có thể loại bỏ nội dung của tất cả các cơ sở dữ liệu hiện có.

Tuy nhiên, bạn vẫn có thể dần dần phát triển định dạng khóa của mình theo thời gian với một chút lập kế hoạch trước. Ví dụ: bạn có thể lưu trữ số phiên bản ở cuối mỗi khóa (một byte sẽ đủ cho hầu hết các mục đích sử dụng). Khi bạn muốn chuyển sang định dạng khóa mới (vd để nó sử dụng số phiên bản được tìm thấy trong các phím để quyết định cách diễn giải chúng.

## **12.Performance (Thực thi, hiệu suất)**

Hiệu suất có thể được điều chỉnh bằng cách thay đổi các giá trị mặc định của các loại được xác định trong tệp bao gồm / leveldb / tùy chọn.h.

## **13.Block size (Kích thước Block)**

leveldb nhóm các khóa liền kề với nhau thành cùng một khối và một khối như vậy là đơn vị chuyển đến và lưu trữ liên tục. Kích thước khối mặc định là khoảng 4096 byte không nén. Các ứng dụng chủ yếu thực hiện quét số lượng lớn nội dung của cơ sở dữ liệu có thể muốn tăng kích thước này. Các ứng dụng thực hiện nhiều lần đọc các giá trị nhỏ có thể muốn chuyển sang kích thước khối nhỏ hơn nếu các phép đo hiệu suất cho thấy sự cải thiện. Không có nhiều lợi ích trong việc sử dụng các khối nhỏ hơn một kilobyte hoặc lớn hơn vài megabyte. Cũng lưu ý rằng nén sẽ hiệu quả hơn với kích thước khối lớn hơn.

## **14.Compression (Nén dữ liệu)**

Mỗi khối được nén riêng trước khi được ghi vào bộ lưu trữ liên tục. Nén theo mặc định vì phương pháp nén mặc định rất nhanh và tự động bị vô hiệu hóa đối với dữ liệu không thể nén. Trong các trường hợp hiếm hoi, các ứng dụng có thể muốn tắt hoàn toàn việc nén, nhưng chỉ nên làm như vậy nếu điểm chuẩn cho thấy sự cải thiện hiệu suất:

|  |
| --- |
| leveldb::Options options;  options.compression = leveldb::kNoCompression;  ... leveldb::DB::Open(options, name, ...) .... |

## **15.Cache (Bộ nhớ cache)**

Nội dung của cơ sở dữ liệu được lưu trữ trong một tập hợp các tệp trong hệ thống tệp và mỗi tệp lưu trữ một chuỗi các khối được nén. Nếu tùy chọn.block\_cache không phải là NULL, nó được sử dụng để lưu trữ nội dung khối không nén được sử dụng thường xuyên.

|  |
| --- |
| #include "leveldb/cache.h"  leveldb::Options options;  options.block\_cache = leveldb::NewLRUCache(100 \* 1048576); // 100MB cache  leveldb::DB\* db;  leveldb::DB::Open(options, name, &db);  ... use the db ...  delete db  delete options.block\_cache; |

Lưu ý rằng bộ đệm chứa dữ liệu không nén, và do đó, nó phải có kích thước theo kích thước dữ liệu cấp ứng dụng, mà không giảm bất kỳ sự nén nào. (Bộ nhớ đệm của các khối được để lại cho bộ đệm bộ đệm của hệ điều hành hoặc bất kỳ triển khai Env tùy chỉnh nào được cung cấp bởi máy khách.)

Khi thực hiện đọc hàng loạt, ứng dụng có thể muốn tắt bộ đệm ẩn để dữ liệu được xử lý bởi đọc hàng loạt không kết thúc thay thế hầu hết các nội dung được lưu trong bộ nhớ cache. Tùy chọn per-iterator có thể được sử dụng để đạt được điều này:

|  |
| --- |
| leveldb::ReadOptions options;  options.fill\_cache = false;  leveldb::Iterator\* it = db->NewIterator(options);  for (it->SeekToFirst(); it->Valid(); it->Next()) {  ...  } |

## **16.Key Layout (Bố cục Key)**

Lưu ý rằng đơn vị truyền đĩa và bộ nhớ đệm là một khối. Các khóa liền kề (theo thứ tự sắp xếp cơ sở dữ liệu) thường sẽ được đặt trong cùng một khối. Do đó, ứng dụng có thể cải thiện hiệu suất của nó bằng cách đặt các khóa được truy cập gần nhau và đặt các phím được sử dụng không thường xuyên trong một vùng riêng của không gian khóa.

Ví dụ: giả sử chúng tôi đang triển khai một hệ thống tệp đơn giản trên đầu trang leveldb. Các loại mục chúng tôi có thể muốn lưu trữ là:

|  |
| --- |
| filename -> permission-bits, length, list of file\_block\_ids  file\_block\_id -> data |

Chúng tôi có thể muốn tiền tố các khóa tên tệp bằng một chữ cái (giả sử '/') và các tệp file\_block\_id bằng một chữ cái khác (giả sử '0') để quét qua siêu dữ liệu không buộc chúng tôi tìm nạp và lưu trữ nội dung tệp cồng kềnh.

## **17.Filters (Các bộ lọc)**

Do cách tổ chức dữ liệu leveldb trên đĩa, một lệnh gọi Get () có thể liên quan đến nhiều lần đọc từ đĩa. Cơ chế FilterPolicy tùy chọn có thể được sử dụng để giảm đáng kể số lượng đĩa đọc.

|  |
| --- |
| leveldb::Options options;  options.filter\_policy = NewBloomFilterPolicy(10);  leveldb::DB\* db;  leveldb::DB::Open(options, "/tmp/testdb", &db);  ... use the database ...  delete db;  delete options.filter\_policy; |

Mã trước liên kết chính sách lọc dựa trên bộ lọc Bloom với cơ sở dữ liệu. Lọc dựa trên bộ lọc Bloom phụ thuộc vào việc giữ một số bit dữ liệu trong bộ nhớ cho mỗi khóa (trong trường hợp này là 10 bit cho mỗi khóa vì đó là đối số chúng tôi đã chuyển cho NewBloomFilterPolicy). Bộ lọc này sẽ giảm số lần đọc đĩa không cần thiết cho các cuộc gọi Get () theo hệ số xấp xỉ 100. Việc tăng số bit trên mỗi phím sẽ dẫn đến việc giảm nhiều hơn với chi phí sử dụng bộ nhớ nhiều hơn. Chúng tôi khuyên các ứng dụng có bộ làm việc không phù hợp với bộ nhớ và nhiều lần đọc ngẫu nhiên sẽ đặt chính sách bộ lọc.

Nếu bạn đang sử dụng một bộ so sánh tùy chỉnh, bạn nên đảm bảo rằng chính sách bộ lọc bạn đang sử dụng tương thích với bộ so sánh của bạn. Ví dụ, hãy xem xét một bộ so sánh bỏ qua khoảng trắng ở cuối khi so sánh các khóa. NewBloomFilterPolicymust không được sử dụng với một bộ so sánh như vậy. Thay vào đó, ứng dụng nên cung cấp chính sách bộ lọc tùy chỉnh cũng bỏ qua các khoảng trắng ở cuối. Ví dụ:

|  |
| --- |
| class CustomFilterPolicy : public leveldb::FilterPolicy {  private:  FilterPolicy\* builtin\_policy\_;  public:  CustomFilterPolicy() : builtin\_policy\_(NewBloomFilterPolicy(10)) {}  ~CustomFilterPolicy() { delete builtin\_policy\_; }  const char\* Name() const { return "IgnoreTrailingSpacesFilter"; }  void CreateFilter(const Slice\* keys, int n, std::string\* dst) const {  // Use builtin bloom filter code after removing trailing spaces  std::vector<Slice> trimmed(n);  for (int i = 0; i < n; i++) {  trimmed[i] = RemoveTrailingSpaces(keys[i]);  }  return builtin\_policy\_->CreateFilter(&trimmed[i], n, dst);  }  }; |

Các ứng dụng nâng cao có thể cung cấp chính sách bộ lọc không sử dụng bộ lọc nở nhưng sử dụng một số cơ chế khác để tóm tắt một bộ khóa. Xem leveldb / filter\_policy.h để biết chi tiết.

## **18.Checksums (Tổng kiểm)**

leveldb liên kết tổng kiểm tra với tất cả dữ liệu mà nó lưu trữ trong hệ thống tệp. Có hai kiểm soát riêng biệt được cung cấp về mức độ tích cực của các tổng kiểm tra này:

ReadOptions :: verify\_checksums có thể được đặt thành true để buộc xác minh tổng kiểm tra tất cả dữ liệu được đọc từ hệ thống tệp thay mặt cho một lần đọc cụ thể. Theo mặc định, không có xác minh như vậy được thực hiện.

Tùy chọn :: paranoid\_checks có thể được đặt thành true trước khi mở cơ sở dữ liệu để thực hiện cơ sở dữ liệu gây ra lỗi ngay khi phát hiện tham nhũng nội bộ. Tùy thuộc vào phần nào của cơ sở dữ liệu đã bị hỏng, lỗi có thể được đưa ra khi cơ sở dữ liệu được mở hoặc sau đó bởi một hoạt động cơ sở dữ liệu khác. Theo mặc định, kiểm tra hoang tưởng bị tắt để cơ sở dữ liệu có thể được sử dụng ngay cả khi các phần lưu trữ liên tục của nó bị hỏng.

Nếu cơ sở dữ liệu bị hỏng (có lẽ không thể mở được khi bật tính năng kiểm tra hoang tưởng), leveldb :: RepairDBf có thể được sử dụng để khôi phục càng nhiều dữ liệu càng tốt

## **19.Approximate Sizes (Các kích thước gần đúng)**

Phương thức GetApproximateSizes có thể được sử dụng để lấy số byte gần đúng của không gian hệ thống tệp được sử dụng bởi một hoặc nhiều phạm vi khóa.

|  |
| --- |
| leveldb::Range ranges[2];  ranges[0] = leveldb::Range("a", "c");  ranges[1] = leveldb::Range("x", "z");  uint64\_t sizes[2];  leveldb::Status s = db->GetApproximateSizes(ranges, 2, sizes); |

Cuộc gọi (preceding call ) trước sẽ đặt kích thước [0] thành số byte gần đúng của không gian hệ thống tệp được sử dụng bởi phạm vi khóa [a..c) và kích thước [1] thành số byte gần đúng được sử dụng bởi phạm vi khóa [x .. z).

## **20.Environment (Môi trường)**

Tất cả các hoạt động tệp (và các cuộc gọi hệ điều hành khác) do triển khai leveldb được định tuyến thông qua một đối tượng leveldb :: Env. Các khách hàng tinh vi có thể muốn cung cấp triển khai Env của riêng họ để kiểm soát tốt hơn. Ví dụ, một ứng dụng có thể đưa ra độ trễ nhân tạo trong đường dẫn IO của tệp để hạn chế tác động của leveldb đối với các hoạt động khác trong hệ thống.

|  |
| --- |
| class SlowEnv : public leveldb::Env {  ... implementation of the Env interface ...  };  SlowEnv env;  leveldb::Options options;  options.env = &env;  Status s = leveldb::DB::Open(options, ...); |

## **21.Porting (Chuyển)**

leveldb có thể được chuyển sang một nền tảng mới bằng cách cung cấp các triển khai cụ thể của nền tảng của các loại / phương thức / hàm được xuất bởi leveldb / port / port.h. Xem leveldb / port / port\_example.h để biết thêm chi tiết.

Ngoài ra, nền tảng mới có thể cần một triển khai leveldb :: Env mặc định mới. Xem leveldb / produc / env\_poseix.h để biết ví dụ.

* + 1. **So sánh, đánh giá hiệu năng LevelDB so với MySQL**
  + DB MySQL
    - Cấu trúc bảng Product: product\_id, product\_code, status. Trong đó khởi tạo sẵn 20 bản ghi có product\_id có giá trị từ 1-20
    - Cấu trúc bảng Subscriber: isdn, status, product\_id, status, create\_datetime, create\_username. Trong đó isdn là khoá chính, sắp xếp theo giá trị tăng dần; product\_id là khoá ngoại của bảng Product.
    - Insert 200 triệu bản ghi từ vào bảng Subscriber (sử dụng thread để chạy insert song song), trong đó Subscriber.status sinh ngẫu nhiên từ 0-3, Subscriber.product\_id sinh ngẫu nhiên từ 1-20.
    - Query 100.000 bản ghi là 100.000 số sinh ngẫu nhiên trong khoảng 200 triệu số từ 800.000.000 đến 999.999.999.
  + Test DB LevelDB:
    - Cấu trúc bảng dữ liệu Subscriber: isdn, status
    - Cấu trúc bảng dữ liệu Sub\_Product: isdn, product\_id
    - Insert 100 triệu bản ghi vào 2 bảng dữ liệu Subscriber, Sub\_Product (mỗi bảng 100 triệu bản ghi, số isdn từ 900.000.000 đến 999.999.999), trong đó Subscriber.status sinh ngẫu nhiên từ 0-3, Sub\_Product.product\_id sinh ngẫu nhiên từ 1-20.
    - Query 1.000.000 bản ghi là 1 triệu số sinh ngẫu nhiên trong khoảng 100 triệu số từ 900.000.000. đến 999.999.999, với điều kiện status=2 và product\_id in (5,10,15)
  + Kết quả:

+ Sử dụng MySQL:

++ Thời gian insert: 3600s/ 100 triệu bản ghi

++ Thời gian query: chậm, không ra kết quả

+ Sử dụng LevelDB:

++ Thời gian insert: 813s/ 100 triệu bản ghi

++ Thời gian query: 3452s/ 1 triệu bản ghi

* + 1. **Kết luận**
* LevelDB với cách lưu trữ đơn giản phù hợp với các ứng dụng đòi hỏi truy vấn nhanh như phiên giao dịch hay giỏ hàng…
* Việc đọc tuần tự (hoặc traversal) của LevelDB là cực kỳ hiệu quả, và nó gần như gần với thứ tự đọc hệ thống tập tin.
* Viết tuần tự, hiệu năng cao (không đồng bộ), bị giới hạn bởi tốc độ đĩa, ghi ngẫu nhiên, hiệu suất hơi kém, nhưng hiệu năng vẫn là một lợi thế lớn so với các DB khác. Cho dù viết tuần tự hay viết ngẫu nhiên, hiệu suất nhanh hơn nhiều so với BTree.
* Không hỗ trợ transaction, chỉ có thể truy vấn dữ liệu thông qua KEY, hỗ trợ hàng loạt đọc và ghi các hoạt động.
* Kích thước dữ liệu Key-Value trong LevelDB không thể quá lớn. Nếu lưu trữ một khóa hoặc giá trị lớn, nó sẽ có tác động lớn hơn đến hiệu suất đọc và ghi của LevelDB