

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



HỆ QUẢN TRỊ CƠ SỞ DỮ LIỆU (CO3021)

FILE STRUCTURE - DATA STORAGE VÀ CONCURRENCY
CONTROL VỚI DBMS MONGODB VÀ MYSQL

GVHD: Võ Thị Ngọc Châu

SV thực hiện: Nguyễn Quang Huy – 1916081
 Nguyễn Đức Hoàng Phú – 2010514
 Lê Đức Thường – 1915442
 Lê Bình Đăng – 1913102

Tp. Hồ Chí Minh, Tháng 12/2022

Mục lục

1 Data Storage và File Structure	6
1.1 Data Storage và File Structure trong InnoDB	6
1.1.1 Kiến trúc tổng quan về InnoDB	6
1.1.2 InnoDB - Cấu trúc In-Memory	6
1.1.2.a Buffer Pool	6
1.1.2.b Change Buffer	11
1.1.2.c Log Buffer	12
1.1.3 InnoDB On-Disk Structures	12
1.1.3.a Table space	12
1.1.3.b Double-write Buffer	15
1.1.4 Innodb Row Format	15
1.1.4.a Định dạng hàng theo kiểu REDUNDANT	16
1.1.4.b Record Header	18
1.1.4.c Hidden Fields	19
1.1.5 Định dạng dòng COMPACT	19
1.1.5.a Variable-length field length list	20
1.1.5.b Record header	21
1.1.5.c Hidden Fields	22
1.1.6 Định dạng hàng DYNAMIC	22
1.1.7 Định dạng hàng COMPRESSED	23
1.1.8 Thực hành định dạng hàng	23
1.1.9 InnoDB Disk I/O and File Space Management	24
1.1.9.a File Space Management	24
1.1.9.b Table Compression	25
1.2 Data Storage và File Structure trong MongoDB	27
1.2.1 WiredTiger Storage Engine	27
1.2.1.a B-Tree based engine	27
1.2.1.b Log-structured Merge Tree Based Engine (LSM Tree) .	28
1.2.1.c Một số tính chất khác của wiredtiger storage engine	30
1.2.1.d Nén dữ liệu (compression)	31
1.2.1.e Tệp lưu dữ liệu cho các collection	32
1.2.1.f Kiểu dữ liệu trong MongoDB	33
1.2.2 MMAPv1 Storage Engine	33
1.2.3 In-memory Storage Engine	35
1.3 So sánh Data Storage và File Structure giữa Mysql(InnoDB) và MongoDB	35
2 Concurrency Control	39
2.1 Đặc điểm của SQL Transaction và tổng quan về Concurrency Control . . .	39
2.2 Mục đích và sự cần thiết của Concurrency Control	39
2.3 Các kỹ thuật sử dụng trong Concurrency Control	45
2.3.1 Two-phase Locking	46
2.3.2 Lock Conversion	46
2.3.3 Kỹ thuật giải quyết Concurrency Control dựa vào trình tự mốc thời gian	48
2.3.3.a The basic timestamp ordering algorithm	49



2.3.3.b	Strict timestamp ordering	51
2.3.3.c	Thomas's Write Rule	51
2.3.3.d	Nhược điểm của Timestamp Ordering Protocol:	51
2.3.4	Kỹ thuật Multiversion Concurrency Control	51
2.3.5	Multiversion technique based on timestamp ordering	52
2.3.6	Multiversion two-phase locking using certify locks	53
2.3.7	Kỹ thuật Validation (Optimistic) Concurrency Control	54
2.3.8	Kỹ thuật Multiple Granularity Locking	56
2.3.8.a	Multiple Granularity Locking Protocol:	58
2.4	Concurrency Control trong MongoDB	59
2.4.1	Global Instance-Wide Lock	59
2.4.2	Lock Request	61
2.4.3	Một số cách để xem trạng thái khóa trong MongoDB	63
2.4.4	Những trường hợp thu hồi khóa trong mongoDB	64
2.4.5	Concurrency Control giữa các Shard và Replication trong MongoDB	65
2.4.6	Lock-free operation	65
2.5	Concurrency Control trong InnoDB	68
2.5.1	Các khái niệm về autocommit, commit, rollback	68
2.5.2	Các kiểu locking của innodb	68
2.5.2.a	Intention Locks	68
2.5.2.b	Record Locks	71
2.5.2.c	Gap Locks	71
2.5.2.d	Next-key Locks	71
2.5.2.e	Insert Intention Locks	72
2.5.2.f	Auto-inc Locks	72
2.5.3	Transaction isolation level	72
2.5.3.a	Repeatable read	73
2.5.3.b	Read committed	73
2.5.3.c	Read uncommitted	74
2.5.3.d	Serializable	74
2.5.3.e	Demo về isolation level giải quyết các vấn đề trong concurrency control	74
2.5.4	Consistent read	78
2.5.5	Locking-reads	80
2.5.6	Locking reads với NOWAIT và SKIP LOCKED	81
2.5.7	Hạn chế deadlock bởi innodb	83
2.6	So sánh giữa MongoDB và InnoDB	84
2.6.1	MVCC	84
2.6.2	Multiple Granularity Locking	85
2.6.3	Locking technique	85
2.6.3.a	Queue locking:	85
2.6.3.b	Release lock process:	87
2.6.3.c	Type lock:	87
2.6.3.d	Optimistic and Pessimistic:	88
2.6.4	Isolation level	88
2.6.5	Consistence non-locking read:	89



3 Thành viên nhóm và vai trò	90
Tài liệu tham khảo	91

Danh sách hình vẽ

1	Lược đồ kiến trúc của InnoDB	6
2	Minh họa thuật toán LRU vùng đệm	7
3	Demo buffer pool	8
4	Buffer pool lúc đầu	9
5	Tạo bảng	9
6	Thông tin buffer pool lúc sau	10
7	Cơ chế change buffer	11
8	Demo File-per-table	12
9	Các thành phần của Data Page	15
10	Sơ đồ mô hình Data Page	16
11	Minh họa định dạng hàng REDUNDANT	17
12	Thông tin của record header	18
13	Minh họa định dạng dòng COMPACT	20
14	Header định dạng dòng COMPACT	22
15	Định dạng hàng COMPRESSED	23
16	Thực hành định dạng hàng COMPRESSED	23
17	Thực hành DYNAMIC và REDUNDANT	24
18	Thông tin bảng sau khi dùng câu lệnh	24
19	Demo table compression	25
20	Demo table compression	26
21	Demo table compression	26
22	Hazard Pointer	27
23	SSTable	28
24	Red-black-tree và SSTable	29
25	Bloom Filter	29
26	Demo việc insert bulk data with WiredTiger	31
27	Demo nén dữ liệu trong MongoDB	32
28	Demo tệp dữ liệu trong MongoDB	33
29	MMAPv1 Storage Engine	34
30	Định nghĩa trường dữ liệu trong bảng MySQL(InnoDB)	36
31	Gọi mô tả lược đồ trong MySQL (InnoDB)	36
32	Cập nhật thông tin lược đồ trong MySQL(InnoDB)	36
33	Lược đồ trong MongoDB	37
34	Nested trong MongoDB	37
35	Kiểu dữ liệu trong MySQL(InnoDB) có thể giới hạn kích thước	38
36	Hỗ trợ tạo khóa chính ngoài khóa có sẵn trong MySQL(InnoDB)	38
37	Trước khi thực hiện query Update	40
38	Kết quả cuối cùng hiện ra là 500 cho bên phía user B nhưng đáng ra phải là 600.	41
39	Như vậy B đã đọc dữ liệu chưa hề tồn tại.	41
40	productID bị thay đổi sau 2 lần truy vấn của userA	42
41	bảng product bị thêm vào các hàng mới sau 2 lần đọc giống nhau bên userA	43
42	2 transaction gây ra deadlock	44
43	Lock conversion	47
44	Lock table	47

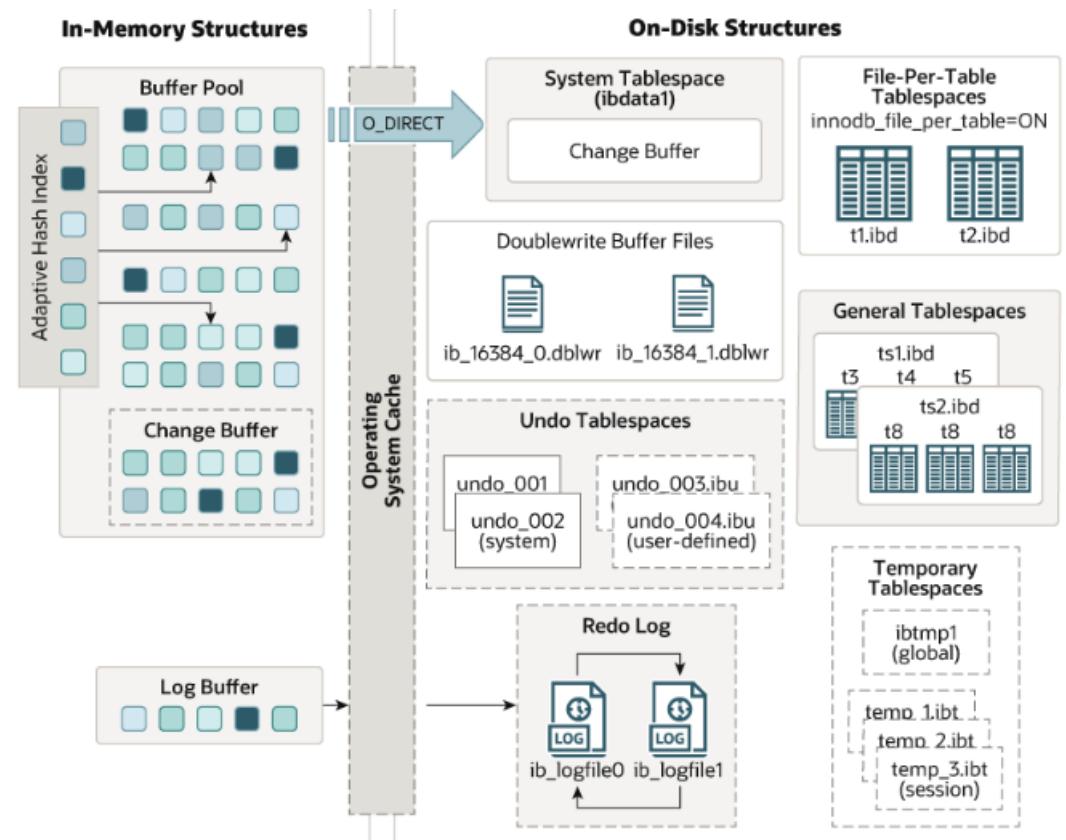


45	deadlock example	48
46	timestamp1	50
47	timestamp2	50
48	52
49	53
50	54
51	54
52	Cây đa độ mịn	57
53	Sơ đồ biểu diễn trạng thái tương thích khóa của kỹ thuật đa độ mịn.	59
54	60
55	60
56	61
57	61
58	62
59	62
60	62
61	63
62	63
63	64
64	66
65	67
66	67
67	lock messages và xem trạng thái khóa: SHARED_READ_ONLY đã được granted	69
68	Ta sử dụng SELECT * FROM messages FOR SHARE, và nhận được khóa IS, khóa tương thích nên nhận ngay được kết quả	70
69	Ta commit khóa IS và thử dùng lệnh FOR UPDATE, lúc này transaction đang đợi khóa	70
70	unlock table thì transaction nhận được khóa X và IX	70
71	transaction B đọc giá trị ở b_id = 2 với name không hề tồn tại.	75
72	userB đã nhận được kết quả chính xác	75
73	userA hai lần đọc kết quả nhận được sự thay đổi khác nhau.	76
74	userA hai lần đọc kết quả nhận được sự thay đổi khác nhau.	77
75	userA hai lần đọc kết quả nhận được sự xuất hiện của một hàng mới	77
76	Không còn bị ảnh hưởng bởi phantom read	78
77	So sánh giữa các mức isolation với các vấn đề mà nó khắc phục được	78
78	t trong quá trình chưa commit bên A vẫn giữ được sự nhất quán.	80
79	session 1 tạo bảng và lock i = 2	82
80	sesion 2 sử dụng option nowait thì ra lỗi vì i = 2 đang bị lock	82
81	session 3 sử dụng option skip locked bỏ qua i = 2 để trả về kết quả	82
82	khi tắt innodb_deadlock_detect	84
83	bật innodb_deadlock_detect và kết quả phản hồi khá nhanh	84
84	Trạng thái hàng đợi khóa khi khóa toàn bộ Instance và dùng lệnh insertOne	85
85	Trạng thái hàng đợi khóa sau khi Unlock Instance	86
86	dù thread_id = 78 đến sau nhưng vẫn thực hiện trước thread_id = 77	86
87	Thông tin khóa của Instance sau khi dùng Instance-Wide Lock	87
88	Innodb kích hoạt các khóa tự động tùy vào điều kiện query	88

1 Data Storage và File Structure

1.1 Data Storage và File Structure trong InnoDB

1.1.1 Kiến trúc tổng quan về InnoDB



Hình 1: Lược đồ kiến trúc của InnoDB

Lược đồ trên biểu thị kiến trúc của InnoDB storage engine bao gồm cấu trúc 2 phần chính là **trên bộ nhớ (in-memory)** và **trên đĩa (on-disk)**.

Ta sẽ lần lượt đi tìm hiểu các thành phần cấu tạo trên.

1.1.2 InnoDB - Cấu trúc In-Memory

1.1.2.a Buffer Pool

Vùng đệm (buffer pool) là một khu vực trong bộ nhớ chính, nơi InnoDB lưu trữ dữ liệu bảng và chỉ mục khi nó được truy cập. Vùng đệm cho phép dữ liệu được sử dụng thường xuyên được truy cập trực tiếp từ bộ nhớ, giúp tăng tốc độ xử lý. Trên các máy chủ chuyên dụng, có tới 80% bộ nhớ vật lý thường được gán cho vùng đệm.

Để tăng hiệu quả cho các hoạt động đọc khối lượng lớn, vùng đệm được chia thành các trang có khả năng chứa nhiều hàng. Để quản lý bộ đệm hiệu quả, vùng đệm được triển khai dưới dạng danh sách các trang được liên kết; dữ liệu hiếm khi được sử dụng sẽ bị xóa khỏi bộ nhớ cache bằng cách sử dụng một biến thể của thuật toán LRU (Least recently used).

Biết cách tận dụng vùng đệm để giữ dữ liệu được truy cập thường xuyên trong bộ

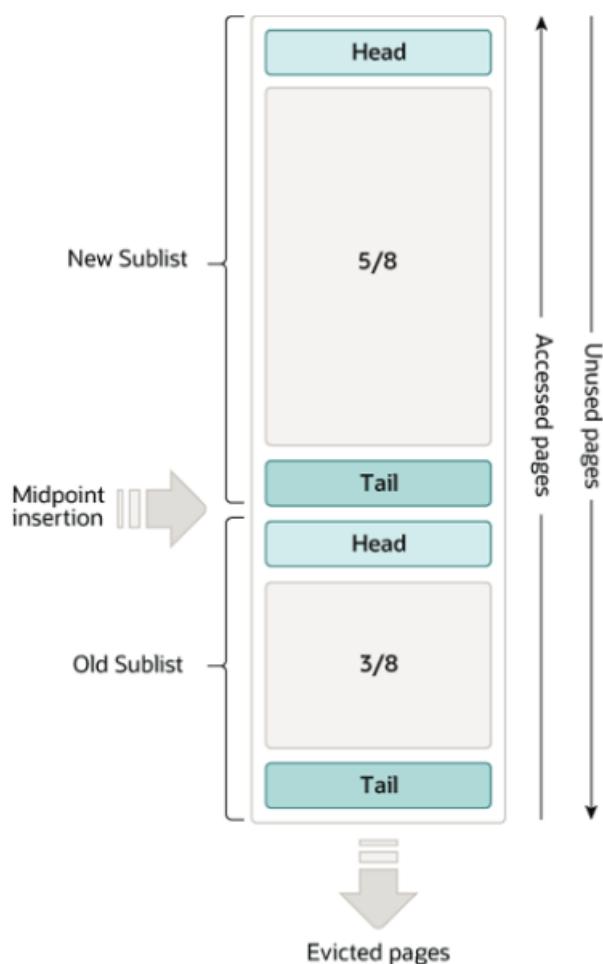
nhớ là một khía cạnh quan trọng của việc điều chỉnh MySQL.

Thuật toán LRU vùng đệm:

Vùng đệm được quản lý dưới dạng danh sách bằng cách sử dụng một biến thể của thuật toán LRU. Khi cần thêm chỗ để thêm trang mới vào vùng đệm, trang ít được sử dụng gần đây nhất sẽ bị loại bỏ và một trang mới được thêm vào giữa danh sách.

Chiến lược chèn điểm giữa này coi danh sách như hai danh sách con:

- Ở phần đầu, một danh sách phụ gồm các trang mới (“young”) được truy cập gần đây
- Ở phía sau, danh sách phụ gồm các trang cũ ít được truy cập gần đây



Hình 2: Minh họa thuật toán LRU vùng đệm

Theo đó 3/8 vùng đệm cho danh sách con cũ, có điểm midpoint là ranh giới của vùng danh sách con mới và danh sách con cũ. 5/8 còn lại là danh sách con mới.

Khi InnoDB đọc một trang vào vùng đệm, ban đầu nó sẽ chèn nó vào điểm giữa (phần đầu của danh sách con cũ). Một trang có thể được đọc vì nó được yêu cầu cho một hoạt động do người dùng khởi tạo, chẳng hạn như truy vấn SQL hoặc như một phần của hoạt động đọc trước được thực hiện tự động bởi InnoDB.

Việc truy cập một trang trong danh sách con cũ làm cho nó trở nên “trẻ”, chuyển nó

đến phần đầu của danh sách con mới. Nếu trang được đọc vì nó được yêu cầu bởi thao tác do người dùng khởi tạo, thì quyền truy cập đầu tiên sẽ xảy ra ngay lập tức và trang được đặt ở dạng trỏ. Nếu trang được đọc do thao tác đọc trước, thì lần truy cập đầu tiên không xảy ra ngay lập tức và có thể hoàn toàn không xảy ra trước khi trang bị loại bỏ.

Khi cơ sở dữ liệu hoạt động, các trang trong vùng đệm không được truy cập thì di chuyển về phía cuối danh sách (được đánh giá như là bắt đầu tuổi bắt đầu già). Các trang trong danh sách phụ mới và cũ đều có tuổi như các trang khác được tạo mới. Các trang trong danh sách con cũ cũng có tuổi khi các trang được chèn ở điểm giữa. Cuối cùng, một trang vẫn chưa được sử dụng khi đến cuối danh sách thì bị loại bỏ.

Theo mặc định, các trang được truy vấn đọc ngay lập tức được chuyển vào danh sách con mới, nghĩa là chúng ở trong vùng đệm lâu hơn. Ví dụ: quét bảng, được thực hiện cho hoạt động mysqldump hoặc câu lệnh SELECT không có mệnh đề WHERE, có thể đưa một lượng lớn dữ liệu vào vùng đệm và loại bỏ một lượng dữ liệu cũ hơn tương đương, ngay cả khi dữ liệu mới không bao giờ được sử dụng nữa. Tương tự, các trang được tải bởi chuỗi nền đọc trước và chỉ được truy cập một lần sẽ được chuyển đến đầu danh sách mới. Những tình huống này có thể đẩy các trang được sử dụng thường xuyên vào danh sách phụ cũ, nơi chúng trở thành đối tượng bị trực xuất.

Lý tưởng nhất là bạn đặt kích thước của vùng đệm thành một giá trị lớn nhất có thể, để lại đủ bộ nhớ cho các quy trình khác trên máy chủ chạy mà không bị phân trang quá mức. Vùng đệm càng lớn, InnoDB càng hoạt động giống như một cơ sở dữ liệu trong bộ nhớ (in-memory), đọc dữ liệu từ đĩa một lần và sau đó truy cập dữ liệu từ bộ nhớ trong các lần đọc tiếp theo.

Mặc định kích thước buffer pool là 128 MB, tức 134217728 bytes. Ta có thể thiết lập kích thước buffer pool offline hay online đều được, ở đây em ví dụ cách sử dụng online.

```
mysql> SELECT @@innodb_buffer_pool_size;
+-----+
| @@innodb_buffer_pool_size |
+-----+
|          134217728 |
+-----+
1 row in set (0,00 sec)

mysql> SET GLOBAL innodb_buffer_pool_size=402653184;
Query OK, 0 rows affected (0,00 sec)

mysql> SELECT @@innodb_buffer_pool_size;
+-----+
| @@innodb_buffer_pool_size |
+-----+
|          402653184 |
+-----+
1 row in set (0,00 sec)

mysql> □
```

Hình 3: Demo buffer pool

Tiếp theo xét một ví dụ khác về việc tốc độ đọc ghi trên của buffer pool:

Hình ảnh phía dưới là một buffer pool lúc đầu:

```
-->-----  
-----  
BUFFER POOL AND MEMORY  
-----  
Total large memory allocated 0  
Dictionary memory allocated 480465  
Buffer pool size 8192  
Free buffers 7234  
Database pages 954  
Old database pages 372  
Modified db pages 0  
Pending reads 0  
Pending writes: LRU 0, flush list 0, single page 0  
Pages made young 0, not young 0  
0.00 youngs/s, 0.00 non-youngs/s  
Pages read 809, created 145, written 199  
0.00 reads/s, 0.00 creates/s, 0.00 writes/s  
No buffer pool page gets since the last printout  
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s  
LRU len: 954, unzip_LRU len: 0  
I/O sum[0]:cur[0], unzip sum[0]:cur[0]  
-----
```

Hình 4: Buffer pool lúc đầu

Tiếp đến, ta bắt đầu tạo một bảng (mặc định ở đây là dùng innodb storage engine).

```
mysql> CREATE DATABASE test;  
Query OK, 1 row affected (0,00 sec)  
  
mysql> use test;  
Database changed  
mysql> CREATE TABLE Persons (  
    ->     PersonID int,  
    ->     LastName varchar(255),  
    ->     FirstName varchar(255),  
    ->     Address varchar(255),  
    ->     City varchar(255)  
    -> );  
Query OK, 0 rows affected (0,10 sec)  
  
mysql> SHOW ENGINE INNODB STATUS\G
```

Hình 5: Tạo bảng



Sau đó ta xem lại thông tin của buffer pool:

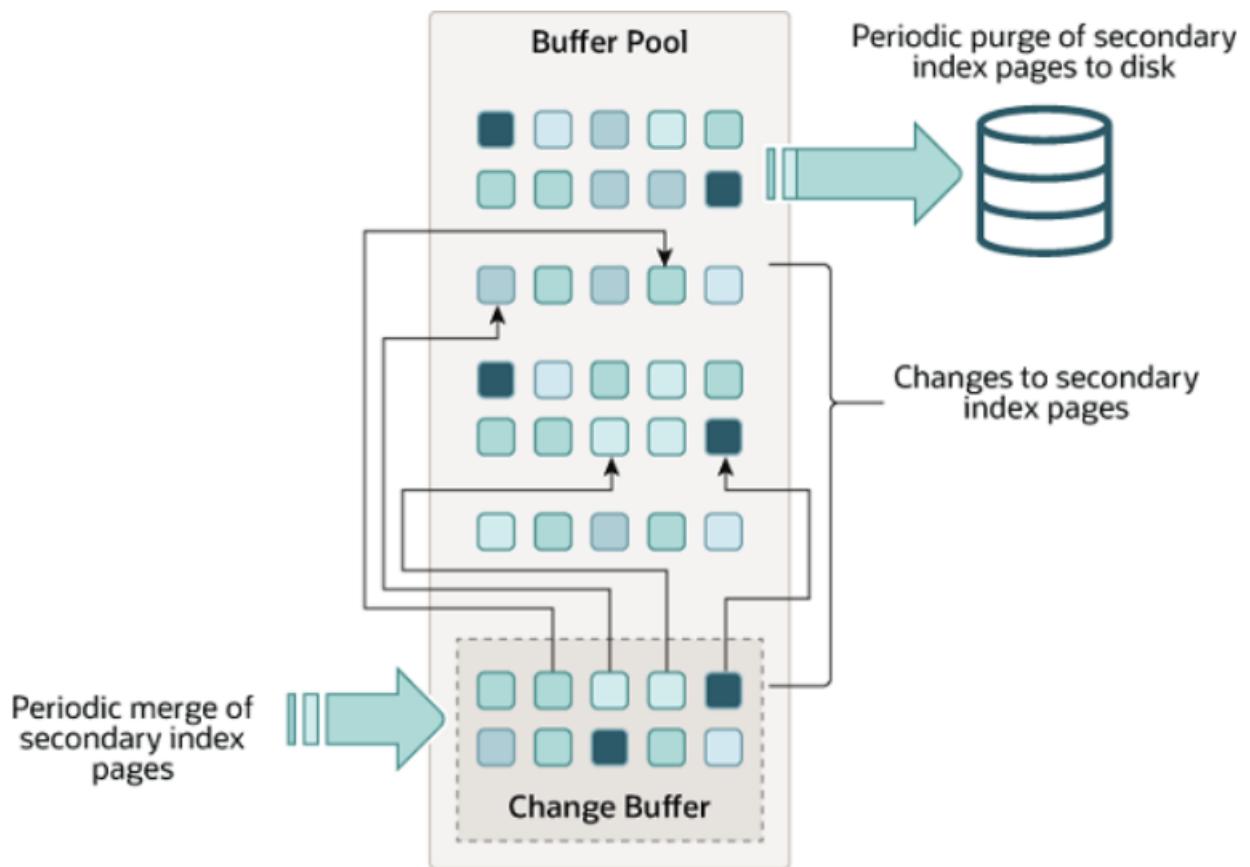
```
-----  
BUFFER POOL AND MEMORY  
-----  
Total large memory allocated 0  
Dictionary memory allocated 511459  
Buffer pool size 8192  
Free buffers 7219  
Database pages 969  
Old database pages 377  
Modified db pages 0  
Pending reads 0  
Pending writes: LRU 0, flush list 0, single page 0  
Pages made young 1, not young 0  
0.00 youngs/s, 0.00 non-youngs/s  
Pages read 817, created 152, written 266  
0.12 reads/s, 0.04 creates/s, 0.27 writes/s  
Buffer pool hit rate 967 / 1000, young-making rate 0 / 1000 not 0 / 1000  
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s  
LRU len: 969, unzip_LRU len: 0  
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
```

Hình 6: Thông tin buffer pool lúc sau

Một số nhận xét: ta thấy vùng buffer trống(free buffers) đã giảm xuống, kích thước trang trong vùng danh sách LRU và LRU sublist (Database pages và Old database pages) đã bắt đầu tăng lên. Thông số Pages made young là 1 là tổng số trang được làm mới trong danh sách LRU vùng đệm (được di chuyển lên đầu danh sách con của các trang “mới”). Các yếu tố như tốc độ đọc, tạo, ghi trang (reads/s, creates/s, writes/s) đã xuất hiện. Và còn nhiều thông số khác mà người quản trị cơ sở dữ liệu DBA có thể khai thác.

1.1.2.b Change Buffer

Bộ đệm thay đổi là một cấu trúc dữ liệu đặc biệt lưu trữ các thay đổi vào các trang chỉ mục phụ khi các trang đó không nằm trong vùng đệm. Các thay đổi trong bộ đệm, có thể là kết quả của các thao tác INSERT, UPDATE hoặc DELETE (DML), được hợp nhất sau đó khi các trang được tải vào vùng đệm bằng các thao tác đọc khác.



Hình 7: Cơ chế change buffer

Không giống như các clustered indexes, các secondary indexes thường không đơn nhất và việc chèn vào các chỉ mục phụ diễn ra theo một thứ tự tương đối ngẫu nhiên. Tương tự, việc xóa và cập nhật có thể ảnh hưởng đến các trang chỉ mục phụ không nằm liền kề trong cây chỉ mục. Việc hợp nhất các thay đổi được lưu trong bộ nhớ cache sau đó, khi các trang bị ảnh hưởng được đọc vào buffer pool bởi các thao tác khác, tránh truy cập I/O ngẫu nhiên đáng kể sẽ được yêu cầu để đọc các trang secondary indexes vào vùng đệm từ đĩa. Định kỳ, hoạt động thanh lọc chạy khi hệ thống hầu như không hoạt động hoặc trong quá trình tắt máy chậm, sẽ ghi các trang index cập nhật vào đĩa. Hoạt động thanh lọc có thể ghi các khối đĩa (disks block) cho một loạt các giá trị chỉ mục hiệu quả hơn nếu mỗi giá trị được ghi vào đĩa ngay lập tức.

1.1.2.c Log Buffer

Log buffer là vùng bộ nhớ chứa dữ liệu được ghi vào log files trên đĩa. Kích thước bộ đệm nhật ký được xác định bởi biến innodb_log_buffer_size. Kích thước mặc định là 16MB. Nội dung Log buffer được chuyển định kỳ vào đĩa. Log buffer lớn cho phép các giao dịch lớn chạy mà không cần phải ghi log data vào đĩa trước khi các giao dịch cam kết. Do đó, nếu bạn có các giao dịch cập nhật, chèn hoặc xóa nhiều hàng, việc tăng kích thước của Log buffer kỹ sẽ tiết kiệm I/O đĩa.

1.1.3 InnoDB On-Disk Structures

1.1.3.a Table space

1. System table space:

System tablespace là vùng lưu trữ cho change buffer. Nó cũng có thể chứa dữ liệu bảng và chỉ mục nếu bảng được tạo trong system tablespace chứ không phải file-per-table hoặc general tablespaces.

System tablespace có thể có một hoặc nhiều data files. Theo mặc định, một tệp dữ liệu trong system tablespace, có tên ibdata1, được tạo trong thư mục dữ liệu. Kích thước và số lượng data files của System tablespace được xác định bởi tùy chọn khởi động innodb_data_file_path.

2. File-per-table tablespace:

File-per-table tablespace trên mỗi bảng chứa dữ liệu và chỉ mục cho một bảng InnoDB và được lưu trữ trên file system trong một data file duy nhất và đây là chế độ mặc định trong InnoDB. Bật tắt chế độ này thì dựa trên biến: innodb_file_per_table

Xét một ví dụ, ta có 2 cách để thay đổi chế độ lưu file có thể trên mysqld hoặc trực tuyến bằng lệnh SET. Ví dụ dưới đây, tạo một table thì trong nơi lưu các table thì đã có file .ibd chứa tên của table đó chứng tỏ 1 table được lưu trong data file duy nhất.

```
mysql>
mysql> SET GLOBAL innodb_file_per_table=ON;
SET GLOBAL innodb_file_per_table=ON;
^C
mysql> use test;
Database changed
mysql> create table test_file_per_table_1(
    -> id INT PRIMARY KEY AUTO_INCREMENT,
    -> name VARCHAR(100)
    -> ) ENGINE = InnoDB;
Query OK, 0 rows affected (0,03 sec)

mysql> !ls -l /var/lib/mysql/test/test_file_per_table_1.ibd
-rw-r----- 1 mysql mysql 114688 Thg 11 12 10:57 /var/lib/mysql/test/test_file_per_table_1.ibd
mysql> [ ]
```

Hình 8: Demo File-per-table



Một file-per-table tablespace được tạo trong một data file .ibd trong thư mục chứa lược đồ của Mysql đặt theo tên của bảng dữ liệu được tạo.

Lợi ích của File-Per-Table Tablespace:

So với system tablespace hay general tablespaces thì File-Per-Table Tablespace có một số ưu điểm chính nổi bật được liệt kê như:

- Dung lượng đĩa được trả lại cho hệ điều hành sau khi cắt bớt hoặc xóa một bảng được tạo trong một file-per-table tablespace. Việc cắt bớt hoặc loại bỏ một bảng được lưu trữ trong một shared tablespace sẽ tạo ra không gian trống trong data files của shared tablespace, tệp này chỉ có thể được sử dụng cho dữ liệu InnoDB. Nói cách khác, shared tablespace data file không bị thu nhỏ kích thước sau khi bảng bị cắt bớt hoặc bị loại bỏ.
- Thao tác sao chép bảng như hoạt động ALTER TABLE trên bảng nằm trong shared tablespace có thể làm tăng dung lượng đĩa bị chiếm bởi tablespace. Các hoạt động như vậy có thể yêu cầu nhiều không gian bổ sung như dữ liệu trong bảng cộng với các chỉ mục. Không gian này không được giải phóng trở lại hệ điều hành vì nó dành cho file-per-table tablespaces.
- Hiệu suất của TRUNCATE TABLE tốt hơn khi được thực thi trên các bảng nằm trong file-per-table tablespaces.
- Data files trong file-per-table tablespace có thể được tạo trên các thiết bị lưu trữ riêng biệt cho mục đích tối ưu hóa I/O, quản lý không gian hoặc sao lưu.
- Các bảng được tạo trong file-per-table tablespaces hỗ trợ các tính năng được liên kết với các định dạng hàng DYNAMIC và COMPRESSED
- Các bảng trong shared tablespace bị giới hạn kích thước bởi giới hạn kích thước tablespace là 64TB. Để so sánh, file-per-table tablespace có giới hạn kích thước 64TB, cung cấp nhiều chỗ cho các bảng riêng lẻ tăng kích thước.

Nhược điểm của File-Per-Table Tablespace:

So với system tablespace hay general tablespaces thì File-Per-Table Tablespace có các nhược điểm chính nổi bật được liệt kê như:

- Với k File-Per-Table Tablespace, mỗi bảng có thể có không gian chưa sử dụng mà chỉ có thể được sử dụng bởi các hàng của cùng một bảng, điều này có thể dẫn đến lãng phí không gian nếu không được quản lý đúng cách.
- Hoạt động fsync được thực hiện trên file-per-table data files thay vì shared tablespace data file. Bởi vì các hoạt động fsync là trên mỗi tệp, các hoạt động ghi cho nhiều bảng không thể được kết hợp, điều này có thể dẫn đến tổng số hoạt động fsync cao hơn.
- Mysqld phải giữ một trình xử lý tệp mở cho mỗi vùng bảng tệp trên mỗi bảng, điều này có thể ảnh hưởng đến hiệu suất nếu bạn có nhiều bảng trong file-per-table tablespaces.
- Cần có thêm bộ mô tả tệp khi mỗi bảng có data file riêng.



- Có khả năng bị phân mảnh nhiều hơn, có thể cản trở DROP TABLE và hiệu suất quét bảng. Tuy nhiên, nếu phân mảnh được quản lý, file-per-table tablespaces có thể cải thiện hiệu suất cho các hoạt động này.

3.General Tablespace

Về khả năng của General Tablespace:

Tương tự như system tablespace, general tablespaces là shared tablespaces có khả năng lưu trữ dữ liệu cho nhiều bảng.

General tablespaces có lợi thế về bộ nhớ tiềm năng so với file-per-table tablespaces. Máy chủ lưu giữ siêu dữ liệu của tablespace trong bộ nhớ trong suốt thời gian tồn tại của một tablespace. Nhiều bảng trong ít hơn general tablespaces tiêu tốn ít bộ nhớ hơn cho siêu dữ liệu của general tablespaces so với cùng một số bảng trong separate file-per-table tablespaces.

Các tệp data files của general tablespace có thể được đặt trong một thư mục liên quan hoặc độc lập với thư mục dữ liệu MySQL, cung cấp cho bạn nhiều khả năng quản lý data file và file-per-table tablespaces. Như với file-per-table tablespaces, khả năng đặt data files bên ngoài thư mục dữ liệu MySQL cho phép bạn quản lý hiệu suất của các bảng quan trọng một cách riêng biệt, thiết lập RAID hoặc DRBD cho các bảng cụ thể hoặc liên kết bảng với các đĩa cứng.

General tablespaces hỗ trợ tất cả các định dạng hàng trong bảng và tính năng liên quan.

Về giới hạn của General Tablespace:

- Không thể thay đổi tablespace đã tạo hoặc hiện có thành một general tablespace.
- Không hỗ trợ tạo general tablespaces tạm thời. General tablespaces không hỗ trợ bảng tạm thời.
- Tương tự như system tablespace, việc cắt bớt hoặc bỏ bớt các bảng được lưu trữ trong một general tablespace sẽ tạo ra không gian trống nội bộ trong tệp dữ liệu .ibd của general tablespace mà chỉ có thể được sử dụng cho dữ liệu InnoDB mới. Dung lượng không được giải phóng trở lại hệ điều hành như đối với file-per-table tablespaces.
- Ngoài ra, thao tác ALTER TABLE sao chép bảng trên bảng nằm trong shared tablespace (general tablespace hoặc the system tablespace) có thể làm tăng dung lượng tablespace được sử dụng. Các hoạt động như vậy yêu cầu nhiều không gian bổ sung như dữ liệu trong bảng cộng với các chỉ mục. Không gian bổ sung cần thiết cho thao tác sao chép bảng ALTER TABLE không được giải phóng trở lại hệ điều hành như đối với file-per table tablespaces.
- Các lệnh ALTER TABLE ... DISCARD TABLESPACE và ALTER TABLE ... IMPORT TABLESPACE không được hỗ trợ cho các bảng thuộc general tablespace.

4.Undo Tablespace

Undo Tablespace chứa nhật ký hoàn tác, là tập hợp các bản ghi chứa thông tin về cách hoàn tác thay đổi mới nhất của một giao dịch đối với bản ghi clustered index.

5.Temporary Tablespace

InnoDB sử dụng temporary tablespaces của session và global temporary tablespace.

Session temporary tablespaces lưu trữ các bảng tạm thời do người dùng tạo và internal temporary tables được tạo bởi trình tối ưu hóa khi InnoDB được định cấu hình làm công cụ lưu trữ cho các bảng tạm thời bên trong trên đĩa.

Global Temporary Tablespace lưu trữ các phân đoạn khôi phục cho các thay đổi được thực hiện đối với các bảng tạm thời do người dùng tạo.

1.1.3.b Double-write Buffer

Doublewrite buffer là một vùng lưu trữ nơi InnoDB ghi các trang được đẩy từ buffer pool khi ghi các trang vào vị trí thích hợp của chúng trong data files InnoDB. Nếu có một hệ điều hành, hệ thống con lưu trữ hoặc thoát quá trình mysqld không mong muốn ở giữa quá trình ghi trang, InnoDB có thể tìm thấy một bản sao tốt của trang từ doublewrite buffer trong quá trình khôi phục sự cố.

Mặc dù dữ liệu được ghi hai lần, doublewrite buffer không yêu cầu chi phí I/O gấp đôi hoặc gấp đôi số hoạt động I/O. Dữ liệu được ghi vào doublewrite buffer trong một đoạn tuân tự lớn, với một lệnh gọi fsync () duy nhất tới hệ điều hành (ngoại trừ trường hợp innodb_flush_method được đặt thành O_DIRECT_NO_FSYNC).

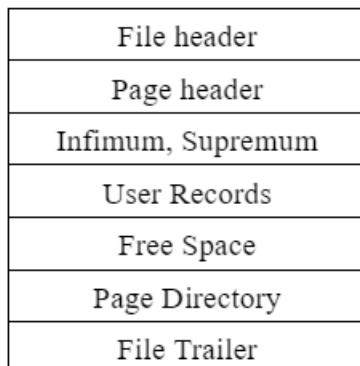
1.1.4 Innodb Row Format

Định dạng hàng của bảng không chỉ xác định cách mỗi hàng dữ liệu được lưu trữ vật lý mà còn ảnh hưởng đến hiệu quả thực thi của các truy vấn và câu lệnh DML. Nếu mỗi trang của đĩa (16K theo mặc định, kích thước trang có thể được điều chỉnh bằng tham số innodb_page_size) có thể lưu trữ nhiều hàng thông tin hơn, nó không chỉ có thể tăng tốc độ truy vấn mà còn giảm số lượng I/O của đĩa trong quá trình cập nhật các hoạt động.

Dữ liệu trong mỗi bảng được lưu trữ riêng biệt trong các trang đĩa, được sắp xếp theo cây B. Trong InnoDB, cả chỉ mục phân cụm (chỉ mục khóa chính) và chỉ mục không phân cụm (chỉ mục bình thường hoặc chỉ mục phụ) đều là dạng cấu trúc dữ liệu của B-tree.

InnoDB storage engine hỗ trợ bốn định dạng hàng: REDUNDANT, COMPACT, DYNAMIC và COMPRESSED. Trước khi đi tìm hiểu định dạng hàng ta sẽ tìm hiểu kiến trúc tổng quan hơn về data page.

Data page chứa 7 thành phần:

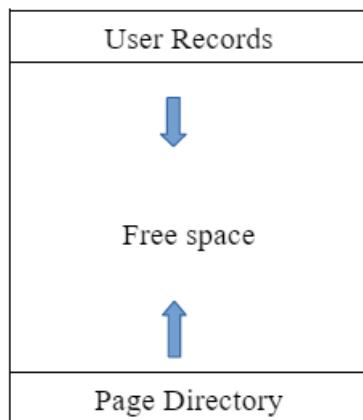


Hình 9: Các thành phần của Data Page

Trong đó:

1. File header: Mô tả về tổng quát về trạng thái trang hiện tại, chiếm 38 bytes cố định
2. Page header: Mô tả thông tin chung của page
3. Infimum, Supremum: Đây là hai bản ghi mà công cụ lưu trữ InnoDB tự động chèn vào trang dữ liệu - bản ghi tối thiểu Infimum và bản ghi tối đa Supremum. Vì hai bản ghi này không được người dùng thêm vào, chúng thường được gọi là "pseudo records"
4. user records: lưu dữ liệu record, phần này sẽ trình bày rõ hơn ở các mục mô tả định dạng hàng
5. free space: Phần này là không gian còn lại trên trang. Cụ thể, phần User Records dùng sử dụng không gian còn lại từ trên xuống, trong khi Page Directory sử dụng không gian còn lại từ dưới lên

Mô hình theo sơ đồ:



Hình 10: Sơ đồ mô hình Data Page

Trong đó:

1. Page Directory: chứa một số vị trí và mỗi vị trí lưu trữ địa chỉ offset của một bản ghi dữ liệu trong trang
2. File Trailer: Dùng để kiểm tra tính integrity của page hiện tại.

1.1.4.a Định dạng hàng theo kiểu REDUNDANT

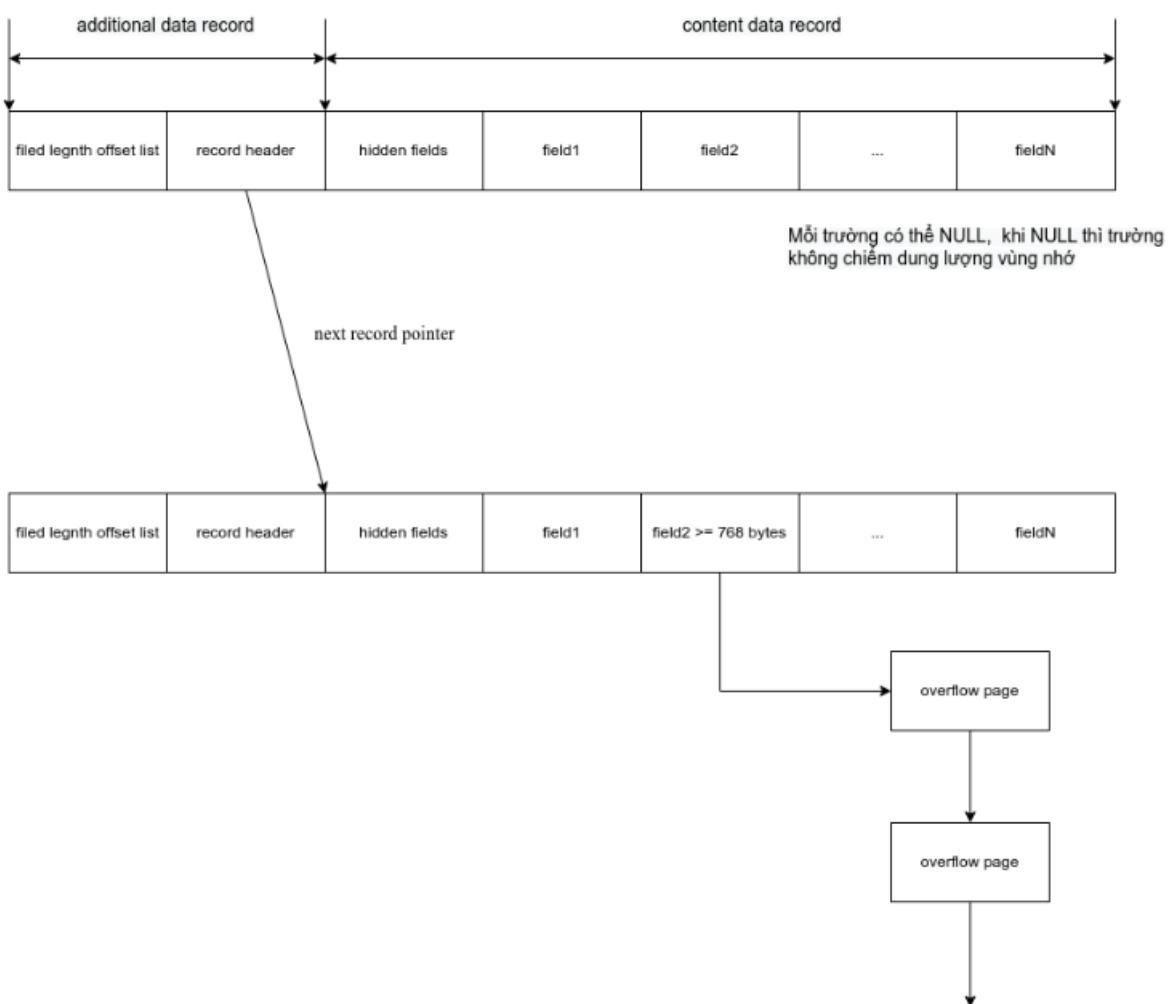
Định dạng hàng REDUNDANT hỗ trợ các phiên bản MySQL cũ hơn và là định dạng hàng được sử dụng trước MySQL 5.0.

Trong định dạng hàng REDUNDANT, chỉ 768 byte đầu tiên của các loại độ dài thay đổi (kiểu dữ liệu VARCHAR, VARBINARY, BLOB và TEXT) được lưu trữ và phần còn lại được lưu trữ trong trang overflow. Đối với các trường có độ dài cố định, nếu giá trị vượt quá hoặc bằng 768 byte, nó cũng sẽ được coi là kiểu có độ dài thay đổi. Ví dụ: trường CHAR (255) có thể vượt quá 768 byte nếu độ dài tối đa của các ký tự trong bộ ký tự

vượt quá 3 (chẳng hạn như tập ký tự utf8mb4).

Nếu độ dài của trường nhỏ hơn hoặc bằng 768 byte, trang overflow sẽ không được sử dụng, trang này thường lưu số lượng hoạt động I/O, vì tất cả dữ liệu được lưu trữ trên nút của cây B. Tuy nhiên, nếu có quá nhiều trường có độ dài thay đổi trong bảng, ngay cả khi các trang overflow không được sử dụng, hiệu quả của chỉ mục sẽ không cao vì số lượng hàng được lưu trữ trên mỗi trang sẽ tương đối nhỏ.

Minh họa:



Hình 11: Minh họa định dạng hàng REDUNDANT

Ở định dạng hàng REDUNDANT, thông tin về độ dài của tất cả các trường (bao gồm cả hidden fields) được lưu trữ thông qua field length offset list. Đầu tiên tính toán thông tin độ dài của từng trường trong bản ghi, sau đó tính giá trị cộng dồn của độ dài theo thứ tự và cuối cùng sắp xếp theo thứ tự ngược lại với thứ tự của từng trường trong bảng dữ liệu, đó là field length offset list được lưu trữ thực tế.

Giả sử một bảng ở định dạng hàng REDUNDANT có ba trường f1, f2 và f3 và độ dài của các trường tương ứng của một bản ghi hàng lần lượt là 20, 3 và 4, khi đó giá trị cộng dồn của độ dài trường là $0 + 20 = 20$, $20 + 3 = 23$, $23 + 4 = 27$, theo thứ tự ngược lại của thứ tự trường, danh sách bù độ dài trường là: 27|23|20

Do đó, để ghi lại và có được độ dài của một trường nào đó, hãy lấy offset value của

trường kế tiếp trừ cho trường hiện tại ví dụ, độ dài của trường f2 là $23 - 20 = 3$.

Cụ thể, không gian bị chiếm bởi mỗi offset trong danh sách offset độ dài trường là 1 byte hoặc 2 byte. Nếu tất cả các trường có độ dài nhỏ hơn 128 byte, mỗi phần bù sẽ chiếm 1 byte, nếu không thì nó chiếm 2 byte. Một số người có thể hỏi tại sao lại là 128 byte? Một byte là 8bit, không phải là 255 byte? Điều này là do cho dù phần bù được lưu trữ trong 1 byte hay 2 byte, bit cao nhất của phần bù được sử dụng để xác định xem nó có phải là NULL hay không. Nếu là NULL, bit cao nhất là 1, ngược lại là 0 (ghi Trường 1byte_offs_flag trong record header cũng xác định số byte được chiếm bởi độ lệch bản ghi). Đối với trường hợp sử dụng 2 byte để lưu giá trị bù, bit cao thứ hai cũng được sử dụng để đánh dấu xem có trang overflow trong giá trị trường này hay không và nếu có overflow thì giá trị đó là 1 (tại thời điểm này, sẽ là một con trỏ đến địa chỉ của dữ liệu còn lại ở cuối dữ liệu trường, chiếm 20 byte, do đó độ dài trường là $768 + 20 = 788$ byte), nếu không thì nó là 0, đây là trường hợp trang overflow trong hình trên.

1.1.4.b Record Header

Record header của định dạng dòng REDUNDANT chiếm 6 byte, tức là 48 bit và được định nghĩa như sau:

tên	kích thước (bits)	mô tả
reserved space	2	Không được sử dụng
deleted_flag	1	Liệu rằng record bị xóa hay chưa
min_rec_flag	1	đánh dấu record tối thiểu không phải là node lá trong cây B
n_owned	4	Số lượng record thuộc sở hữu của vị trí tương ứng với bản ghi
heap_no	13	Số thứ tự của bản ghi trong heap
n_field	10	số lượng trường
1byte_offs_flag	1	bằng 1 nếu độ dài trường offset là 1, bằng 0 nếu độ dài trường offset bằng 2
next_record pointer	16	Vị trí tương đối của bản ghi tiếp theo trên trang

Hình 12: Thông tin của record header

Trong đó:

- Delete_flag: xác định liệu bản ghi có bị xóa hay không. Từ trường này cũng có thể thấy rằng khi bản ghi bị xóa, việc xóa vật lý không được thực hiện ngay lập tức.
- n_field: số trường, vì nó chiếm 10 bit nên số trường tối đa là 1023.
- 1byte_offs_flag, xác định độ dài của trường offset trong field length offset list.
- next_record pointer: trả đến vị trí tương đối của bản ghi tiếp theo trong trang. Qua hình vẽ cũng có thể thấy rằng con trả không trả đến vị trí bắt đầu của bản ghi mà chỉ đến phần bắt đầu của nội dung dữ liệu của bản ghi tiếp theo. Điều này không chỉ giải thích tại sao danh sách bù độ dài trường được sắp xếp theo thứ tự ngược lại của thứ tự trường (vì khoảng cách tương đối giữa trường đầu tiên trong phần nội dung dữ liệu và thông tin độ dài tương ứng của nó gần hơn, điều này có thể cải thiện tỷ lệ truy cập bộ nhớ cache của CPU), và cũng giải thích cách MySQL biết kích thước của không gian bị chiếm bởi field length offset list khi độ dài của danh sách bù độ dài trường không cố định (theo trường thông tin tiêu đề bản ghi 1byte_offs_flag).

1.1.4.c Hidden Fields

| DB_ROW_ID | DB_TRX_ID | DB_ROLL_PTR |

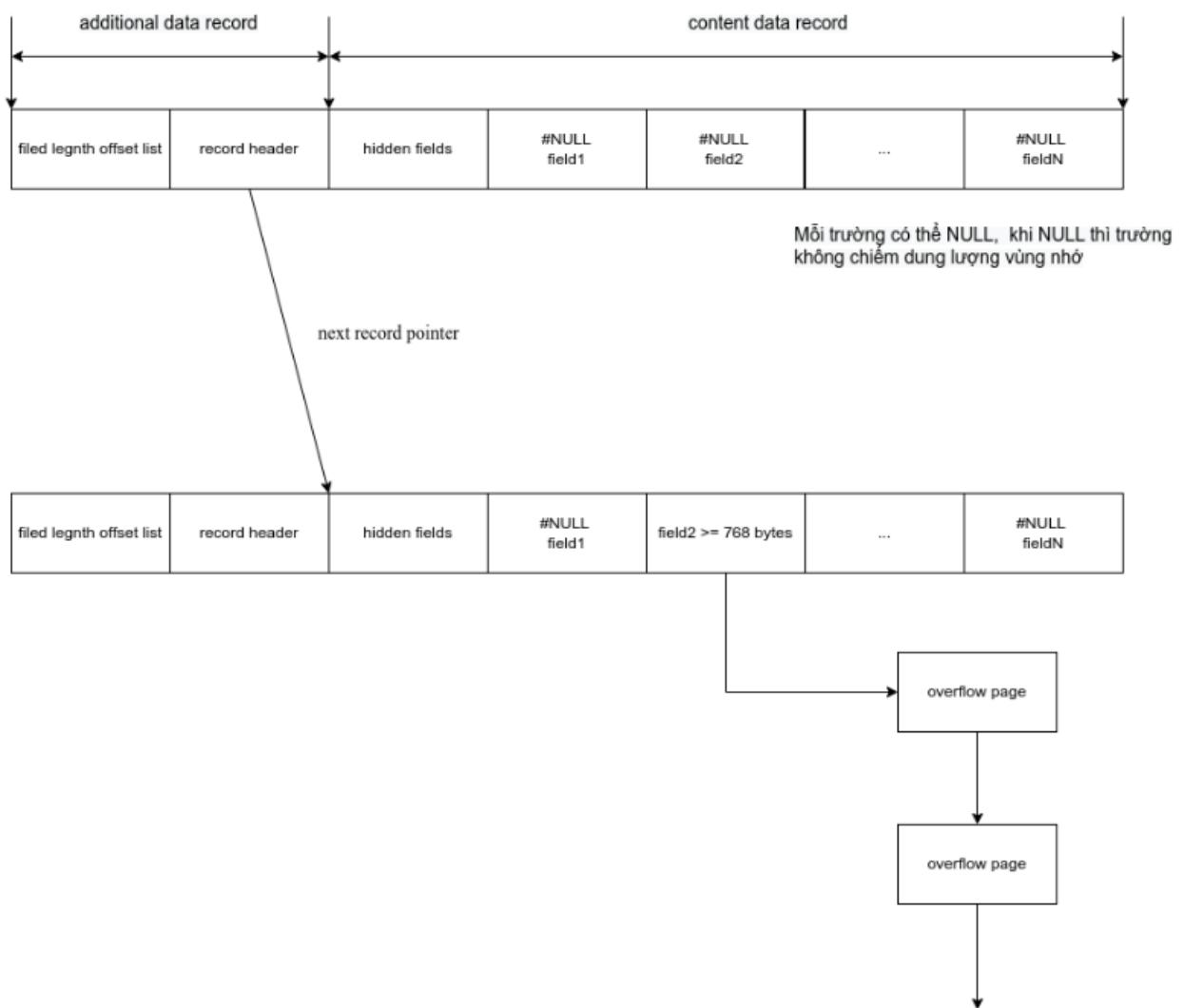
Trong đó:

- DB_ROW_ID là row id và là primary key nếu bảng đó không có primary key, nó chiếm 6 byte.
- DB_TRX_ID là trường transaction ID chiếm 7 bytes. Còn lại là trường DB_ROLL_PTR là rollback index chiếm 6 bytes

1.1.5 Định dạng dòng COMPACT

So với định dạng dòng REDUNDANT, định dạng dòng COMPACT có thể tiết kiệm khoảng 20% dung lượng lưu trữ, nhưng nó cũng tương đối nhiều CPU hơn. Vì vậy, nếu bottleneck ở hiệu suất máy chủ là tốc độ đĩa và tỷ lệ truy cập bộ nhớ cache, COMPACT sẽ nhanh hơn, nhưng nếu bottleneck ở máy chủ là tốc độ CPU, định dạng hàng COMPACT có thể chậm hơn.

Minh họa:



Hình 13: Minh họa định dạng dòng COMPACT

1.1.5.a Variable-length field length list

Đối với các kiểu variable-length như VARCHAR, VARBINARY, TEXT, v.v., nếu chúng ta lưu trữ chúng chiếm bao nhiêu byte, nó có bỏ qua định nghĩa ranh giới giữa các giá trị trường không? Ở định dạng hàng COMPACT, variable-length field length list sẽ lưu trữ độ dài của mỗi kiểu độ dài biến đổi có dữ liệu không phải là NULL và sẽ được sắp xếp theo thứ tự ngược lại của thứ tự trường (ở định dạng REDUNDANT, độ dài của tất cả các trường bao gồm các trường ẩn được lưu trữ., bất kể giá trị của trường có phải là NULL hay không).

Giả sử rằng một bảng ở định dạng hàng COMPACT có ba trường variable-length là f1, f2 và f3 và độ dài của các trường tương ứng của một bản ghi hàng lần lượt là 20, 3 và 4, độ dài trường variable-length danh sách là: | 4 | 3 | 20 |

Cụ thể, không gian được chiếm bởi mỗi trường variable-length được xác định trong variable-length field length list giống như định dạng dòng REDUNDANT, 1 byte hoặc



2 byte, nhưng định dạng dòng REDUNDANT chiếm 1 phần từ hoặc mỗi dòng chiếm 2 byte. Khi số byte tối đa được lưu trữ trong variable-length thay đổi không vượt quá 255 byte, rõ ràng thông tin về độ dài chỉ cần được biểu diễn bằng một byte. Được thảo luận theo từng trường hợp cụ thể:

- Số byte thực tế được sử dụng bởi dữ liệu trường không vượt quá 127 byte: vẫn sử dụng một byte để biểu diễn.
- Nếu không, hãy sử dụng hai byte để biểu diễn.

Tại sao lại là 127 byte, một byte có thể biểu diễn tới 255? Điều này là do khi MySQL thiết kế biểu diễn độ dài, để phân biệt liệu đó có phải là một bit đại diện cho độ dài hay không, người ta quy định rằng nếu bit cao nhất là 1 thì nó là hai byte đại diện cho độ dài, nếu không thì nó là một byte.

Cần lưu ý rằng có thể không có trường variable-length trong bảng và sau đó variable-length field length list. Ngoài ra, nếu giá trị của trường variable-length là NULL, variable-length field length list sẽ không được liệt kê. Trường này được lưu trữ, do không có trường kiểu có độ dài thay đổi hoặc vì giá trị của trường kiểu variable-length là NULL, dẫn đến không có variable-length field length list.

Xét giá trị cờ NULL, cũng có thể có một số trường có thể là NULL trong bảng. Nếu các trường có giá trị NULL được lưu trữ ở định dạng hàng REDUNDANT, nó thực sự lãng phí dung lượng. Ý tưởng về BitMap có thể được sử dụng để đánh dấu các trường nhằm tiết kiệm dung lượng. Cờ NULL là một BitMap.

Cần lưu ý rằng cờ giá trị NULL chỉ dành cho các trường có thể là NULL. Nếu một trường được xác định là không null, thì trường này sẽ không nhập BitMap của cờ giá trị NULL.

Cờ giá trị NULL chiếm bao nhiêu byte? Mỗi trường không phải là NULL chiếm 1 bit. Nếu hơn 8 trường, một byte thêm được thêm vào. Nếu nó nhỏ hơn một byte, bit cao được lấp đầy bằng 0. Giả sử rằng có 9 trường có giá trị có thể là NULL, cờ giá trị NULL chiếm 2 byte. Nếu tất cả các trường trong bảng không rỗng, thì không có cờ giá trị NULL. Tương tự như variable-length field length list, cờ giá trị NULL cũng được sắp xếp theo thứ tự ngược lại.

Nếu có ba trường có giá trị NULL trong bảng: CHAR (10) f1, CHAR (10) f2 và CHAR (10) f3, và giá trị bản ghi của một hàng là '1', '2', '3', thì bit cờ giá trị NULL là 00000000 và giá trị bản ghi hàng khác là '11', NULL, NULL, thì bit cờ giá trị NULL là 00000110.

1.1.5.b Record header

Không giống như định dạng dòng REDUNDANT, record header trong định dạng dòng COMPACT được cố định ở kích thước 5 byte:

Tên	kích thước (bits)	mô tả
reserved space	2	Không được sử dụng
deleted_flag	1	Liệu rằng record bị xóa hay chưa
min_rec_flag	1	đánh dấu record tối thiểu không phải là node lá trong cây B
n_owned	4	Số lượng record thuộc sở hữu của vị trí tương ứng với bản ghi
heap_no	13	Số thứ tự của bản ghi trong heap
record_type	3	Loại bản ghi: bản ghi dữ liệu thông thường là 000, kiểu con trỏ nút là 001, pseudo record - bản ghi đầu tiên hành vi infimum bản ghi 010, pseudo record - bản ghi cuối cùng hành vi tối cao 011, 1xx là một bit dành riêng.
next_record pointer	16	Vị trí tương đối của bản ghi tiếp theo trên trang

Hình 14: Header định dạng dòng COMPACT

So với định dạng hàng REDUNDANT, có nhiều record_type hơn và ít trường n_field và 1byte_flag hơn, điều này cũng khiến định dạng hàng COMPACT biết nhiều thông tin định nghĩa bảng hơn so với định dạng REDUNDANT.

1.1.5.c Hidden Fields

Giống với định dạng REDUNDANT

1.1.6 Định dạng hàng DYNAMIC

Định dạng hàng DYNAMIC gần giống với định dạng hàng COMPACT. Sự khác biệt là DYNAMIC tối ưu hóa các trường variable-length: khi bản ghi quá dài, trường dài nhất sẽ lắn lướt được lưu trữ hoàn toàn ở trang overflow, nghĩa là, nút B-tree chỉ lưu trữ con trỏ có kích thước 20 Byte vào trang overflow cho đến khi kích thước bản ghi là thích hợp. Ngoài ra, liên quan đến định dạng hàng COMPACT, định dạng hàng DYNAMIC hỗ trợ tiền tố chỉ mục lớn và định dạng hàng DYNAMIC có thể hỗ trợ tiền tố chỉ mục 3072 byte.

1.1.7 Định dạng hàng COMPRESSED

Định dạng hàng COMPRESSED gần giống với định dạng hàng DYNAMIC. Sự khác biệt là định dạng hàng COMPRESSED thêm chức năng nén dữ liệu bảng và chỉ mục dựa trên định dạng hàng DYNAMIC.

Định dạng hàng	Đặc điểm lưu trữ nhỏ gọn	Lưu trữ nâng cao Variable-Length	Hỗ trợ index key prefix	Hỗ trợ nén	Các loại Tablespace hỗ trợ
REDUNDANT	No	No	No	No	system, file-per-table, general
COMPACT	Yes	No	No	No	system, file-per-table, general
DYNAMIC	Yes	Yes	Yes	No	system, file-per-table, general
COMPRESSED	Yes	Yes	Yes	Yes	file-per-table, general

Hình 15: Định dạng hàng COMPRESSED

1.1.8 Thực hành định dạng hàng

Khi định dạng thì nhớ là định dạng hàng compressed không hỗ trợ ở system tablespace, ví dụ:

```
mysql> SET GLOBAL innodb_file_per_table=OFF;
Query OK, 0 rows affected (0,00 sec)

mysql> SET GLOBAL innodb_default_row_format=COMPRESSED;
ERROR 1231 (42000): Variable 'innodb_default_row_format' can't be set to the value of 'COMPRESSED'
mysql> 
```

Hình 16: Thực hành định dạng hàng COMPRESSED

Có 2 cách để định dạng hàng là dùng biến global hoặc định nghĩa trong từng table riêng, ví dụ ta thử cả 2 cách là ở mức global thì định dạng hàng là DYNAMIC còn ở row_format_table_2 thì override lại là định dạng hàng là REDUNDANT.



```
mysql> SET GLOBAL innodb_default_row_format=DYNAMIC;
Query OK, 0 rows affected (0,00 sec)

mysql> CREATE TABLE row_format_1 (c1 INT);
Query OK, 0 rows affected (0,03 sec)

mysql> CREATE TABLE row_format_2 (c1 INT) ROW_FORMAT=REDUNDANT;
Query OK, 0 rows affected (0,10 sec)
```

Hình 17: Thực hành DYNAMIC và REDUNDANT

Ta thấy khi xem thông tin 2 bảng này:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_TABLES WHERE NAME LIKE 'test/row_format%' \G
***** 1. row *****
    TABLE_ID: 1079
        NAME: test/row_format_1
        FLAG: 33
    N_COLS: 4
        SPACE: 0
    ROW_FORMAT: Dynamic
    ZIP_PAGE_SIZE: 0
    SPACE_TYPE: System
    INSTANT_COLS: 0
TOTAL_ROW_VERSIONS: 0
***** 2. row *****
    TABLE_ID: 1080
        NAME: test/row_format_2
        FLAG: 0
    N_COLS: 4
        SPACE: 0
    ROW_FORMAT: Redundant
    ZIP_PAGE_SIZE: 0
    SPACE_TYPE: System
    INSTANT_COLS: 0
TOTAL_ROW_VERSIONS: 0
2 rows in set (0,00 sec)

mysql> □
```

Hình 18: Thông tin bảng sau khi dùng câu lệnh

Khi mà không được chỉ định rõ định dạng hàng ở table nó sẽ lấy mức định dạng hàng ở global là DYNAMIC như bảng row_format_1. Ở đây cũng có SPACE_TYPE tức là kiểu tablespace là system do đó lúc này không tạo bảng có định dạng hàng kiểu COMPRESSED được

1.1.9 InnoDB Disk I/O and File Space Management

1.1.9.a File Space Management

Mỗi tablespace bao gồm các trang (pages) cơ sở dữ liệu. Mọi tablespace trong một phiên bản MySQL có cùng kích thước trang. Theo mặc định, tất cả các vùng bảng có kích thước trang là 16KB; có thể thay đổi bằng cách: innodb_page_size

Các trang được nhóm thành các extents có kích thước 1MB cho các trang có kích thước lên đến 16KB (64 trang 16KB liên tiếp hoặc 128 trang 8KB hoặc 256 trang 4KB). Đối với kích thước trang là 64KB, kích thước extents là 4MB. Các "tệp" bên trong một không gian bảng được gọi là các segments trong InnoDB.

Khi một segment phát triển bên trong không gian bảng, InnoDB sẽ phân bổ 32 trang đầu tiên cho nó cùng một lúc. Sau đó, InnoDB bắt đầu phân bổ toàn bộ extents cho

segment. InnoDB có thể thêm tối đa 4 extents cùng một lúc vào một phân đoạn lớn để đảm bảo tính tuần tự tốt của dữ liệu.

Hai segments được phân bổ cho mỗi index trong InnoDB. Một dành cho các nút không lá của cây B, nút kia dành cho các nút lá. Giữ các nút lá tiếp giáp nhau trên đĩa cho phép các hoạt động I/O tuần tự tốt hơn, vì các nút lá này chứa dữ liệu bảng thực tế.

1.1.9.b Table Compression

Bởi vì bộ xử lý và bộ nhớ cache có tốc độ đọc nhanh hơn thiết bị lưu trữ đĩa, nhiều khối lượng công việc bị ràng buộc bởi đĩa. Nén dữ liệu cho phép kích thước cơ sở dữ liệu nhỏ hơn, giảm I/O và cải thiện thông lượng, với chi phí nhỏ là tăng hiệu suất sử dụng CPU. Tính năng nén đặc biệt có giá trị đối với các ứng dụng cần đọc nhiều, trên hệ thống có đủ RAM để giữ dữ liệu thường xuyên sử dụng trong bộ nhớ. Bản chất của việc nén tốt là phụ thuộc vào kiểu của dữ liệu, nén hoạt động bằng cách xác định các chuỗi byte lặp lại trong một khối dữ liệu. Các chuỗi ký tự thường nén tốt ví dụ CHAR, VARCHAR, TEXT hoặc BLOB. Mặt khác, các bảng chứa phần lớn dữ liệu nhị phân (số nguyên hoặc số dấu phẩy động) hoặc dữ liệu đã được nén như dạng ảnh thì không nén tốt và nó sử dụng zlib dựa trên LZ77 là thuật toán để nén dữ liệu.

Ví dụ về một table có sẵn trong mysql như sau:

```
mysql> describe information_schema.columns;
+-----+-----+-----+-----+-----+-----+
| Field          | Type           | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| TABLE_CATALOG  | varchar(64)   | NO   |     | NULL    |       |
| TABLE_SCHEMA   | varchar(64)   | NO   |     | NULL    |       |
| TABLE_NAME     | varchar(64)   | NO   |     | NULL    |       |
| COLUMN_NAME    | varchar(64)   | YES  |     | NULL    |       |
| ORDINAL_POSITION | int unsigned | NO   |     | NULL    |       |
| COLUMN_DEFAULT | text          | YES  |     | NULL    |       |
| IS_NULLABLE    | varchar(3)    | NO   |     | NULL    |       |
| DATA_TYPE      | longtext      | YES  |     | NULL    |       |
| CHARACTER_MAXIMUM_LENGTH | bigint        | YES  |     | NULL    |       |
| CHARACTER_OCTET_LENGTH | bigint        | YES  |     | NULL    |       |
| NUMERIC_PRECISION | bigint unsigned | YES  |     | NULL    |       |
| NUMERIC_SCALE  | bigint unsigned | YES  |     | NULL    |       |
| DATETIME_PRECISION | int unsigned | YES  |     | NULL    |       |
| CHARACTER_SET_NAME | varchar(64)   | YES  |     | NULL    |       |
| COLLATION_NAME  | varchar(64)   | YES  |     | NULL    |       |
| COLUMN_TYPE    | mediumtext    | NO   |     | NULL    |       |
| COLUMN_KEY     | enum('','PRI','UNI','MUL') | NO   |     | NULL    |       |
| EXTRA          | varchar(256)  | YES  |     | NULL    |       |
| PRIVILEGES     | varchar(154)  | YES  |     | NULL    |       |
| COLUMN_COMMENT | text          | NO   |     | NULL    |       |
| GENERATION_EXPRESSION | longtext    | NO   |     | NULL    |       |
| SRS_ID         | int unsigned | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
22 rows in set (0,01 sec)

mysql> CREATE TABLE big_table AS SELECT * FROM information_schema.columns;
Query OK, 3605 rows affected (0,65 sec)
Records: 3605  Duplicates: 0  Warnings: 0
```

Hình 19: Demo table compression

Ta nhìn qua thì thấy các kiểu chuỗi xuất hiện khá nhiều, do đó nếu mà nén dữ liệu ở bảng này thì kích thước chắc hẳn sẽ giảm đi đáng kể.

```
Records: 28840 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 57680 rows affected (1,49 sec)
Records: 57680 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 115360 rows affected (9,72 sec)
Records: 115360 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 230720 rows affected (15,36 sec)
Records: 230720 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 461440 rows affected (22,01 sec)
Records: 461440 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 922880 rows affected (28,09 sec)
Records: 922880 Duplicates: 0 Warnings: 0

mysql> INSERT INTO big_table SELECT * FROM big_table;
Query OK, 1845760 rows affected (1 min 6,15 sec)
Records: 1845760 Duplicates: 0 Warnings: 0

mysql> COMMIT;
Query OK, 0 rows affected (0,00 sec)

mysql> ALTER TABLE big_table ADD id int unsigned NOT NULL PRIMARY KEY auto_increment;
Query OK, 0 rows affected (2 min 45,44 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> \! ls -l /var/lib/mysql/test/big_table.ibd
-rw-r----- 1 mysql mysql 788529152 Thg 11 12 14:51 /var/lib/mysql/test/big_table.ibd
```

Hình 20: Demo table compression

Sau khi liên tục chèn dữ liệu của chính nó vào bảng big_table, ta xem thử kích thước của file lưu dữ liệu đó cũng khá lớn giờ ta đi so sánh với việc nén dữ liệu.

Ta tạo một bảng key_block_size_4 để với việc mỗi hàng có định dạng compressed

```
mysql> CREATE TABLE key_block_size_4 LIKE big_table;
Query OK, 0 rows affected (1,82 sec)

mysql> ALTER TABLE key_block_size_4 key_block_size=4 row_format=compressed;
Query OK, 0 rows affected (4,69 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql>
mysql> INSERT INTO key_block_size_4 SELECT * FROM big_table;
Query OK, 3691520 rows affected (4 min 28,51 sec)
Records: 3691520 Duplicates: 0 Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0,00 sec)

mysql> \! ls -l /var/lib/mysql/test/big_table.ibd /var/lib/mysql/test/key_block_size_4.ibd
-rw-r----- 1 mysql mysql 788529152 Thg 11 12 14:51 /var/lib/mysql/test/big_table.ibd
-rw-r----- 1 mysql mysql 184549376 Thg 11 12 14:57 /var/lib/mysql/test/key_block_size_4.ibd
mysql>
```

Hình 21: Demo table compression

với key_block_size_4=4 (đây là tham số về mức độ nén). Nhận thấy rằng kích thước lưu trữ giảm đi hơn 4 lần.

1.2 Data Storage và File Structure trong MongoDB

MongoDB là cơ sở dữ liệu phi quan hệ vì vậy dữ liệu của nó được lưu trữ dưới dạng document (tài liệu). Với dạng lưu trữ này giúp việc lưu trữ dữ liệu linh hoạt hơn và phù hợp hơn với các yêu cầu phát triển dữ liệu trong thực tế.

Storage engine là thành phần chính của MongoDB chịu trách nhiệm trong việc quản lý dữ liệu. MongoDB cung cấp nhiều storage engine. Trong đó có thể kể đến như MMAPv1 Storage Engine, WiredTiger Storage Engine và In-Memory Storage Engine, trong đó In-Memory chỉ có ở phiên bản MongoDB Enterprise.

1.2.1 WiredTiger Storage Engine

Là storage engine mặc định của MongoDB, WiredTiger sử dụng kiến trúc kết hợp bao gồm B-Tree và log-structured merge tree (LSM Tree) để lưu trữ dữ liệu. Kiến trúc kết hợp này bao gồm 2 thành phần chính: In-memory cache với hiện thực là B-Tree và hazard pointer, và disk block manager, các block được quản lý bằng một skip list để phục vụ đánh chỉ mục và cấp phát vùng nhớ.

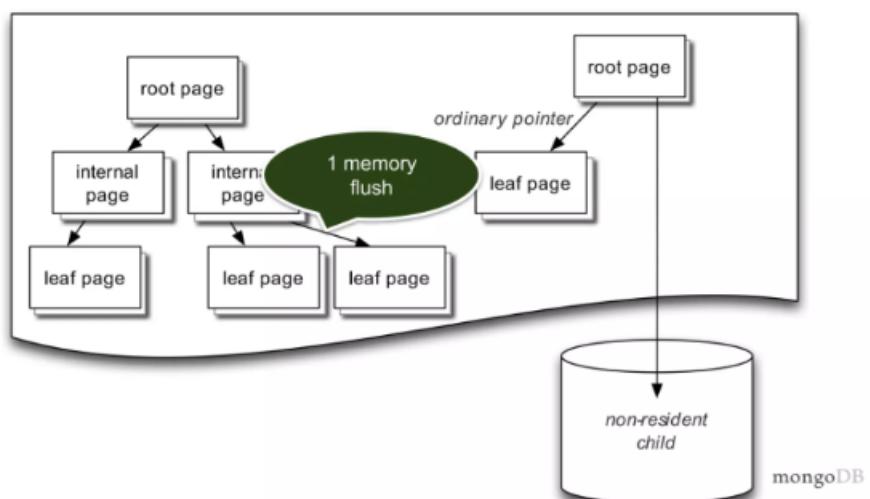
1.2.1.a B-Tree based engine

B-Tree là cấu trúc dữ liệu dạng cây cân bằng dùng để sắp xếp dữ liệu, cho phép tìm kiếm, truy cập tuần tự, thêm và xóa.

Trong đó các nút B-tree được tổ chức theo độ chi tiết của trang và phương pháp thay thế trên bộ nhớ đệm là LRU.

Hazard pointer là con trỏ dùng để trỏ tới file offsets của những disk files.

Hazard Pointers



Hình 22: Hazard Pointer

Các thuận lợi của B-Tree:

- Throughput cao, độ trễ thấp. B-Tree được xây dựng phát triển theo chiều ngang và nông nên lượng ít nodes cần phải đi qua mỗi khi cần truy xuất.
- Các khóa được sắp xếp có thứ tự nên việc truy cập tuần tự và đánh chỉ mục được cân bằng với 1 giải thuật đệ quy.
- Dễ dàng trong việc xử lý lượng lớn thao tác thêm và xóa trong một khoảng thời gian ngắn

Giới hạn của B-Tree:

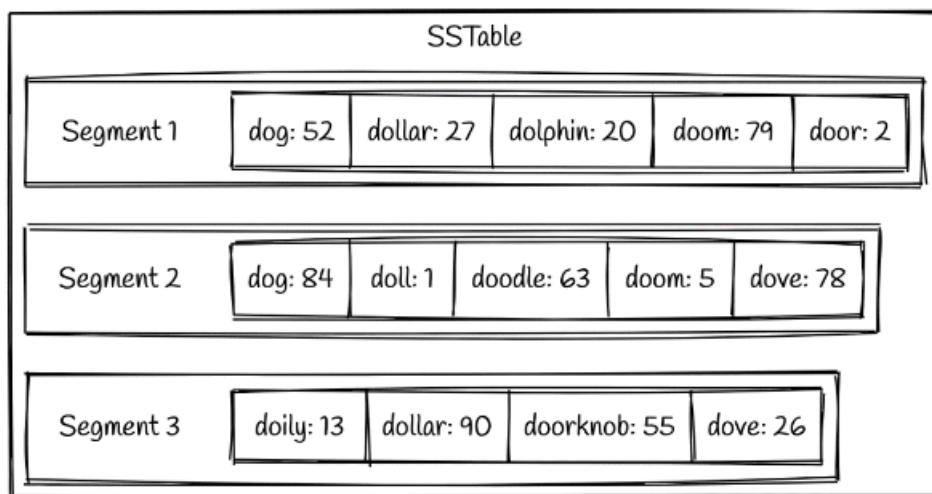
- Khả năng ghi yếu do cần phải đảm bảo dữ liệu được sắp xếp tốt với những lần ghi ngẫu nhiên. Ghi ngẫu nhiên sẽ hao tốn hơn là ghi có tuần tự

1.2.1.b Log-structured Merge Tree Based Engine (LSM Tree)

Vì giới hạn đã đề cập phần trên của B-Tree, để giải quyết với những cơ sở dữ liệu lớn, log-structured merge tree (LSM Tree) ra đời để cải thiện hiệu năng cho các truy cập chỉ mục đến các file với lượng lớn thao tác ghi vào một khoảng thời gian.

LSM Tree là một cấu trúc dữ liệu thường được dùng để giải quyết với những khối lượng thao tác ghi quá lớn. Lúc này cách ghi sẽ được tối ưu thành chỉ hỗ trợ ghi tuần tự.

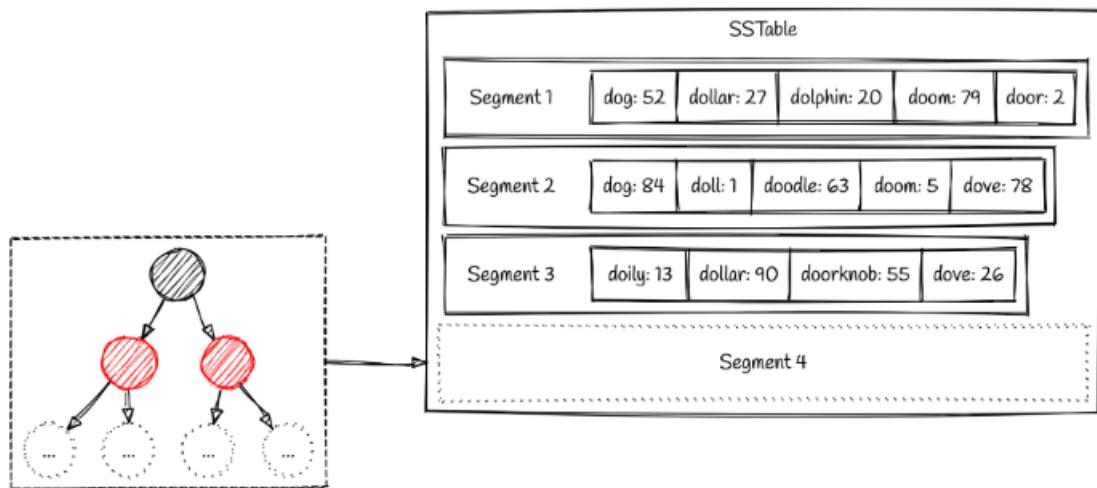
Các LSM Tree được lưu trên đĩa sử dụng định dạng Sorted String table (SSTable). Như tên gọi, SSTable là định dạng lưu trữ kiểu các cặp key-value, trong đó key được lưu theo một thứ tự đã được sắp xếp. Một SSTable bao gồm nhiều file đã được sắp xếp gọi là các segments. Các segment khi được lưu lên đĩa đều khác nhau, ví dụ như hình bên dưới:



Hình 23: SSTable

Như đã trình bày, LSM Tree chỉ thực hiện ghi tuần tự. Để giải quyết việc này, LSM Tree sử dụng thêm 1 cấu trúc in-memory tree, được gọi là memtable, với cấu trúc phía dưới là red-black tree. Khi các thao tác ghi được thực hiện, dữ liệu sẽ được thêm vào red-black tree này cho đến khi cây đạt đến kích thước tối đa được định nghĩa trước. Khi đó, red-black tree sẽ ghi vào đĩa 1 segment đã có sắp thứ tự. Điều này giúp việc ghi các

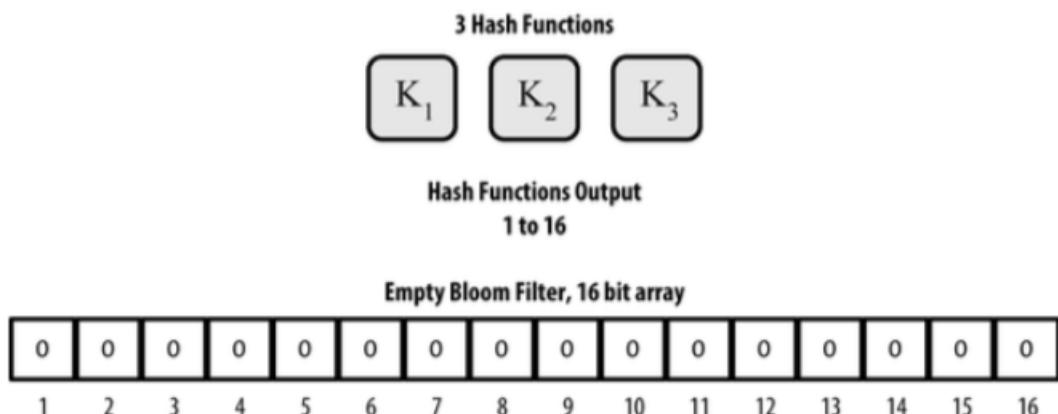
segment file một cách tuần tự mặc dù việc thêm vào không có tính thứ tự.



Hình 24: Red-black-tree và SSTable

Mỗi LSM Tree sử dụng một bloom filter nhằm giải quyết hạn chế về tiêu tốn bộ nhớ nhiều hơn (so với B-Tree) trong quá trình thực hiện các thao tác đọc.

Bloom filter là một cấu trúc dữ liệu xác suất. Bloom filters bao gồm một mảng m phần tử nhị phân (giá trị của mỗi phần tử là 0 hoặc 1), và k hàm băm được thiết kế sao cho output là số nguyên trong khoảng $1 \rightarrow m$. Một ví dụ về bloom filter:



Hình 25: Bloom Filter

Nhờ vào việc sử dụng bloom filter, việc thực hiện những truy cập đĩa không cần thiết có thể bị loại bỏ.

Thuận lợi của LSM Tree:

- Khả năng ghi tuần tự nhanh đảm bảo cho các dữ liệu lớn và đang tăng mạnh.
- Dữ liệu luôn sẵn sàng để truy vấn ngay lập tức



- Các thao tác thêm mới rất nhanh

1.2.1.c Một số tính chất khác của wiredtiger storage engine

Files là một phần của tầng lưu trữ được quản lý bởi storage engine. Đối với WiredTiger, một file sẽ được tạo ra tương ứng với 1 collection và 1 index.

Về giới hạn: 1 collection có thể được định nghĩa giới hạn số lượng document hoặc không. Nếu được định nghĩa giới hạn, tối đa 1 collection có thể chứa 2^{31} documents. Kích thước của một document ở phiên bản MongoDB mới nhất cho phép tối đa 16MB, không có định nghĩa bởi người dùng.

WiredTiger sử dụng unspanned record tuy nhiên lại hỗ trợ record có kích thước động và có thể lưu trữ dạng hàng (row-store) hoặc dạng cột (column-store).

File formats: Row-store và column-store là 2 file format do WiredTiger hỗ trợ, cả 2 đều là dạng key-value

Đối với row-store: Phù hợp cho các truy vấn mà tất cả các cột đều được yêu cầu trả về ở mỗi lần tra cứu (vì chỉ có 1 tập các metadata page để đọc cache và tìm kiếm). Dạng này, cả keys và values đều là dạng chuỗi có kích thước thay đổi.

Đối với column-store: Phù hợp với hầu hết các truy vấn chỉ yêu cầu 1 phần các cột được trả về (vì các cột có thể được chia ra thành nhiều file và chỉ có các cột trả về mới cần nằm trong cache)

Data file format: Trong bộ nhớ, một bảng cơ sở dữ liệu được lưu trữ dưới dạng B-Tree, với các node là các page structure. Root và các internal pages chỉ chứa khóa và các ref tới các page khác, trong khi đó các page ở node lá chứa keys, values hoặc ref tới các overflow pages. Khi các pages này được ghi lên đĩa, mỗi page sẽ tương ứng 1 đơn vị dữ liệu gọi là block. Trên đĩa, 1 WiredTiger data file là một tập các block biểu diễn các page của B-Tree một cách luận lý.

Page types: Các loại page khác nhau tạo các loại cell khác nhau:

Internal Pages: Chứa các key trỏ tới các page khác.

- Row-Store internal page tạo thành từ 1 key cell.
- Column-store internal page chứa ref tới 1 page ngoài.

Leaf Pages: bao gồm 1 page header, theo bởi các key, value hoặc địa chỉ (đều ref đến các overflow page).

- Row-store leaf page tạo thành từ 1 key cell theo bởi 1 value.
- Variable-length Column-store leaf page tạo thành từ các data cells hoặc các deleted cells. Page header chứa một số gọi là số của bản ghi bắt đầu, do đó các key được liên kết đến tất cả các column store values được suy ra từ vị trí của chúng trong trang.
- Fixed-length Column-store leaf page trước hết được tạo thành từ 1 khối dữ liệu bitmap và sau đó là 1 header phụ được theo sau bởi các key cells thay thế với các value cells.

Overflow page: Khi các keys/values quá lớn phải được lưu trữ riêng trong file, khác vị trí mà nó được đặt theo logic. Page sizes và các configuration values như là leaf_key_max và leaf_value_max được dùng để xác định các overflow items.



```
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.serverStatus().storageEngine
{
    name: 'wiredTiger',
    supportsCommittedReads: true,
    oldestRequiredTimestampForCrashRecovery: Timestamp({ t: 1670782554, i: 37 }),
    supportsPendingDrops: true,
    dropPendingIdents: Long("0"),
    supportsSnapshotReadConcern: true,
    readOnly: false,
    persistent: true,
    backupCursorOpen: false,
    supportsResumableIndexBuilds: true
}
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.stats()
{
    db: 'DBMS',
    collections: 2,
    views: 0,
    objects: Long("186"),
    avgObjSize: 81,
    dataSize: Long("15066"),
    storageSize: Long("24576"),
    totalFreeStorageSize: Long("0"),
    numExtents: Long("0"),
    indexes: 2,
    indexSize: Long("28672"),
    indexFreeStorageSize: Long("0"),
    fileSize: Long("0"),
    nsSizeMB: 0,
    ok: 1
}
```

Hình 26: Demo việc insert bulk data with WiredTiger

Trong hình có thể thấy storage engine được sử dụng là wiredTiger. Trong đó:

- Có 2 collections (collections).
- Số lượng document là 186 (objects).
- Kích thước lưu trữ là 20480 bytes (storageSize).
- Tổng kích thước của các dữ liệu chưa nén có trong database là 15066 bytes (dataSize)
- Có 2 index, 1 cho mỗi collection (indexes).
- Kích thước lưu trữ tổng cùn trống cho cả collection và index là 0. (totalFreeStorageSize), trong đó kích thước trống cho index là 0 (indexFreeStorageSize).

1.2.1.d Nén dữ liệu (compression)

Row-store hỗ trợ 4 loại compression bao gồm: key prefix compression, dictionary compression, Huffman encoding và block compression.

- Key prefix compression giúp giảm kích thước yêu cầu của cả đối tượng trong bộ nhớ và trên đĩa bằng cách chỉ lưu các key prefix giống nhau một lần trên mỗi trang. Chi phí phải trả là hao tốn thêm CPU và bộ nhớ khi xử lý trên in-memory tree. Theo mặc định, key prefix compression được vô hiệu.
- Dictionary compression cũng làm giảm kích thước yêu cầu của cả đối tượng trong bộ nhớ và trên đĩa bằng cách chỉ lưu mọi giá trị giống nhau một lần trên mỗi trang. Chi phí cho việc này là thêm một ít hao tốn CPU và bộ nhớ khi ghi các trang vào đĩa. Về mặc định, dictionary compression được vô hiệu hóa.

- Huffman encoding giảm kích thước yêu cầu của cả đối tượng trong bộ nhớ và trên đĩa bằng cách nén các mục giá trị riêng lẻ. Chi phí cho việc này là thêm hao phí cho CPU và bộ nhớ khi ghi các trang vào đĩa. Chi phí này cho CPU có thể cao nên cần được cân nhắc. Về mặc định, Huffman encoding được vô hiệu hóa.
- Block compression giảm kích thước yêu cầu của các đối tượng trên đĩa bằng cách nén các khối của file của đối tượng sao lưu. Các compression engines được WiredTiger hỗ trợ là zlib, snappy và bzip2. Mỗi engine có thể yêu cầu một giá trị kích thước lưu trữ khác nhau nhưng kích thước dữ liệu thì đều giống nhau. Chi phí cho việc này là tăng hao phí cho CPU và bộ nhớ trong việc đọc và ghi các trang lên đĩa. Chi phí này có thể cao nên cần cân nhắc. Về mặc định, Block compression được vô hiệu hóa.

```
// create collections
> db.createCollection('snappy')
> db.createCollection('zlib', {storageEngine: {wiredTiger: {configString: 'block_cc

// insert a compressible document into both collections
> doc = {_id:0, text:<a paragraph of text>}
> db.snappy.insert(doc)
> db.zlib.insert(doc)

// storage size comparison
> db.snappy.stats().storageSize
20480
> db.zlib.stats().storageSize
4096

// data size comparison
> db.snappy.dataSize()
697
> db.zlib.dataSize()
697
```

Hình 27: Demo nén dữ liệu trong MongoDB

Variable-length column-store hỗ trợ 4 loại compression bao gồm: run-length encoding, dictionary compression, Huffman encoding và block compression.

Trong đó: Run-length encoding giảm kích thước yêu cầu của các đối tượng trên đĩa và trong bộ nhớ bằng cách lưu trữ tuần tự, các giá trị trùng chỉ được lưu một lần. Chi phí là một phần tăng nhỏ trong hao phí của CPU và bộ nhớ khi trả về các giá trị từ in-memory trê và khi ghi các trang lên đĩa. Loại compression này luôn được kích hoạt và không thể tắt.

Fixed-length column-store hỗ trợ 1 loại compression: block compression. Nội dung của block compression như phần trên đã trình bày.

1.2.1.e Tệp lưu dữ liệu cho các collection

Với mỗi lần tạo database, mỗi collection sẽ được Mongoddb tạo 1 binary file để lưu trữ. Trong hình lần lượt là trước khi tạo database, tạo 1 database với 1 collection (collection-0-3147...), tạo thêm 1 collection (collection-4-3147...), tương tự collection 7 và 9.

```
[lebinhdang@BinhDang mongodb % ls
WiredTiger
WiredTiger.lock
WiredTiger.turtle
WiredTiger.wt
WiredTigerHS.wt
_mdb_catalog.wt
collection-0-3147794937929069605.wt
collection-0-7287819206806730876.wt
collection-2-3147794937929069605.wt
collection-2-7287819206806730876.wt
collection-4-3147794937929069605.wt
collection-4-7287819206806730876.wt
collection-7-3147794937929069605.wt
collection-9-3147794937929069605.wt
diagnostic.data

index-1-3147794937929069605.wt
index-1-7287819206806730876.wt
index-10-3147794937929069605.wt
index-3-3147794937929069605.wt
index-3-7287819206806730876.wt
index-5-3147794937929069605.wt
index-5-7287819206806730876.wt
index-6-3147794937929069605.wt
index-6-7287819206806730876.wt
index-8-3147794937929069605.wt
journal
mongod.lock
sizeStorer.wt
storage.bson
```

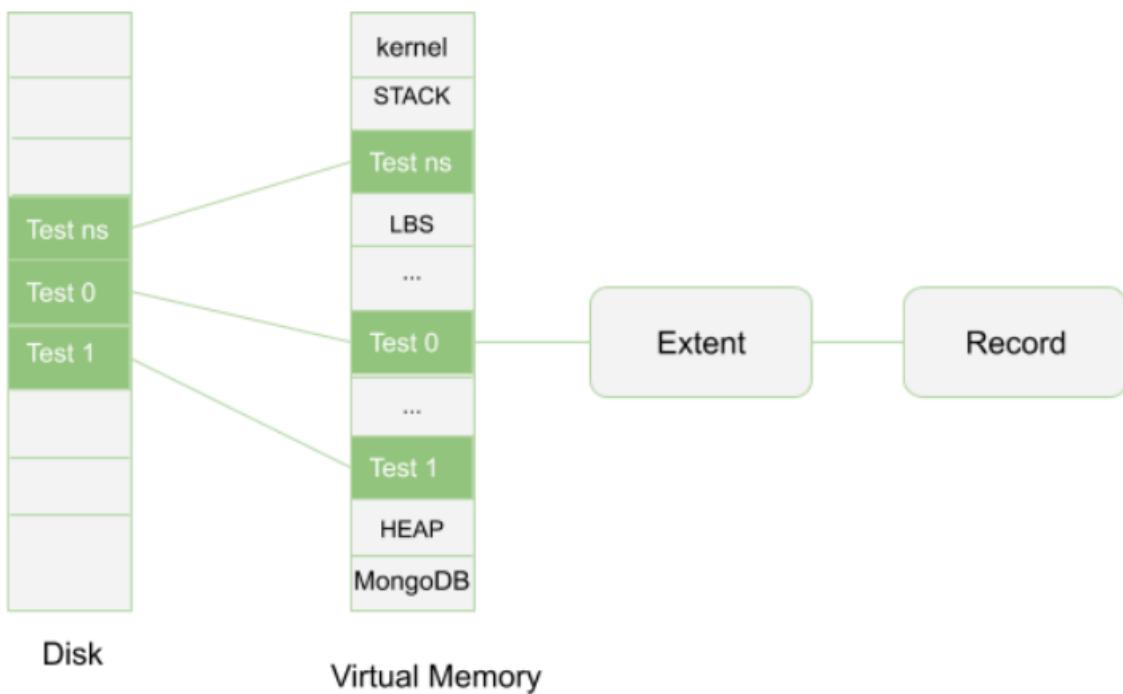
Hình 28: Demo tệp dữ liệu trong MongoDB

1.2.1.f Kiểu dữ liệu trong MongoDB

MongoDB hỗ trợ nhiều loại kiểu dữ liệu, có thể kể đến:

- String
- Integer
- Boolean
- Double
- Array
- Timestamp
- Object
- Null
- Date
- Binary data
- Regular Expression

1.2.2 MMAPv1 Storage Engine



Hình 29: MMAPv1 Storage Engine

Trước khi sự xuất hiện của WiredTiger, MMAPv1 là storage engine mặc định của MongoDB.

MMAPv1 được xây dựng dựa trên B-Tree, cung cấp nhiều chức năng như hỗ trợ lưu trữ và quản lý bộ nhớ. Đối với MongoDB, tất cả các records được bố trí liền kề trên đĩa và trong trường hợp một document nếu cập nhật và phát triển lớn hơn so với kích thước record được cấp phát, lúc đó MongoDB sẽ cấp phát 1 record mới để lưu trữ (gọi là tái cấp phát - reallocation). Đối với MMAPv1 đây là một thuận lợi cho việc truy cập các dữ liệu tuần tự, nhưng cùng với đó, một hạn chế gập phải là hao tốn thời gian bởi tất cả các chỉ mục tài liệu cần được cập nhật và nó có thể gây ra phân mảnh vùng nhớ. Để tránh việc tái cấp phát này, MMAPv1 tận dụng cấp phát Power of 2 Sized Allocation để mọi document được lưu trong một record (có chứa thêm một số vùng được gọi là padding). Padding được dùng để cho phép các cập nhật của document làm tăng kích thước document nhưng không vượt quá kích thước record, giảm nguy cơ phải tái cấp phát.

Power of 2 sized allocation: Mỗi record có kích thước đơn vị byte là lũy thừa của 2 (ví dụ như 32, 64, 128, 256,...). 2MB là giới hạn mặc định, đối với bất kỳ tài liệu nào vượt quá giới hạn này, bộ nhớ của nó được làm tròn thành bội số gần nhất của 2MB.

Ví dụ: Một document kích thước 200MB, kích thước được làm tròn đến 256MB và 56MB không gian trống này được để sẵn cho các việc cập nhật làm tăng thêm kích thước của document. Điều này đảm bảo cho document có thể phát triển thay vì hệ thống phải tái cấp phát khi các document đạt giới hạn không gian lưu trữ trong record.

Về sử dụng bộ nhớ: Tất cả vùng nhớ trống đều trở thành vùng nhớ đệm của MMAPv1. Các bộ hoạt động với kích thước chính xác và đạt được hiệu suất tối ưu thông qua một bộ hoạt động phù hợp với bộ nhớ. Bên cạnh đó, cứ mỗi 60 giây, MMAPv1 ghi xuống đĩa các thay đổi trên dữ liệu do vậy tiết kiệm vùng nhớ đệm.

1.2.3 In-memory Storage Engine

Bắt đầu từ MongoDB Enterprise phiên bản 3.2.6 MongoDB cung cấp storage engine in-memory. Ngoài các metadata và diagnostic data, in-memory storage engine không quản lý bất cứ dữ liệu trên đĩa nào khác, bao gồm data, indexes, user credentials, etc.

Bằng việc bỏ qua truy xuất I/O trên đĩa, in-memory storage engine chấp nhận độ trễ trong mức dự báo của các thao tác cơ sở dữ liệu.

In-memory storage engine yêu cầu tất cả dữ liệu phải phù hợp với những mô tả cài đặt trong configuration file .

Về mặc định, in-memory storage engine sử dụng (50% RAM - 1GB).

1.3 So sánh Data Storage và File Structure giữa Mysql(InnoDB) và MongoDB

Trong quá trình so sánh thì có nhiều phần giới thiệu đã nói ở phần lý thuyết và có hiện thực.

Giống nhau:

- Cả 2 đều sử dụng thuật toán LRU cho vùng đệm, dữ liệu ít khi được sử dụng thì bị xoá khỏi vùng đệm
- Đều có hỗ trợ việc lưu dữ liệu trên đĩa theo dạng cây B
- Đều có hỗ trợ việc lưu trữ dữ liệu theo định dạng hàng, phục vụ cho các hệ thống xử lý giao dịch OLTP (Online transaction processing)
- Cả 2 đều sử dụng unspanned records tuy nhiên record lớn quá thì phải hỗ trợ trang tràn (overflow page) khi mỗi node cấp phát nội dung không đủ, nhưng vẫn đảm bảo phần dư của mỗi node thì record khác không chèn vào.
- Đều hỗ trợ các khoá chính mặc định: trong innoDB là DB_ROW_ID, còn trong mongoDB là trường _id
- Việc phân cấp trong Mysql (innodb) và MongoDb cũng tương đồng tương ứng như sau:
Trong Mysql (innodb): Database->Table->Row/Column
Trong MongoDb: Database->Collection->Document/Field
- Cùng hỗ trợ các kiểu dữ liệu trong Mysql và MongoDB đặc trưng như kiểu nguyên, số thập phân, chuỗi, kiểu luận lý, kiểu dãy (array), kiểu ngày, giờ
- Đều hỗ trợ việc nén dữ liệu để tiết kiệm kích thước lưu trữ

Khác nhau:

Khác nhau về sự linh hoạt của lược đồ:

Mysql (InnoDB) sử dụng lược đồ phải định nghĩa trước các trường vào kiểu dữ liệu, có thể thay đổi lược đồ sau khi định nghĩa. MongoDB thì không yêu cầu phải định nghĩa lược đồ

Với MySQL:

```
mysql> CREATE TABLE Persons (
    ->     PersonID int,
    ->     LastName varchar(255),
    ->     FirstName varchar(255),
    ->     Address varchar(255),
    ->     City varchar(255)
    -> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0,05 sec)
```

Hình 30: Định nghĩa trường dữ liệu trong bảng MySQL (InnoDB)

```
mysql> DESCRIBE Persons;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| PersonID | int      | YES  |   | NULL    |   |
| LastName | varchar(255) | YES  |   | NULL    |   |
| FirstName | varchar(255) | YES  |   | NULL    |   |
| Address | varchar(255) | YES  |   | NULL    |   |
| City | varchar(255) | YES  |   | NULL    |   |
+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Hình 31: Gọi mô tả lược đồ trong MySQL (InnoDB)

```
mysql> alter table Persons
    -> drop column City ;
Query OK, 0 rows affected (0,03 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> alter table Persons
    -> add column Email varchar(255);
Query OK, 0 rows affected (0,04 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> describe Persons;
+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| PersonID | int      | YES  |   | NULL    |   |
| LastName | varchar(255) | YES  |   | NULL    |   |
| FirstName | varchar(255) | YES  |   | NULL    |   |
| Address | varchar(255) | YES  |   | NULL    |   |
| Email | varchar(255) | YES  |   | NULL    |   |
+-----+-----+-----+-----+-----+
5 rows in set (0,00 sec)
```

Hình 32: Cập nhật thông tin lược đồ trong MySQL (InnoDB)

Với MongoDB:

Lược đồ dữ liệu quyết định cách thức mà ứng dụng làm việc với dữ liệu của nó. Với loại cơ sở dữ liệu quan hệ truyền thống, lược đồ phải được định nghĩa trước khi thêm bất cứ dữ liệu nào vào và không thể thay đổi lược đồ nếu có các yêu cầu mới hay các mở rộng liên quan đến ngữ cảnh của ứng dụng (strict schema). Trái ngược điều đó, MongoDB hay các cơ sở dữ liệu NoSQL nói chung đã phát triển để giải quyết hạn chế này bằng cách cho phép người dùng thêm dữ liệu mà không cần lược đồ định nghĩa trước (dynamic schema). MongoDB cung cấp mô hình dữ liệu tài liệu cho phép người dùng lưu trữ và gộp các loại dữ liệu, đồng thời cung cấp khả năng xử lý các truy cập dữ liệu phức tạp và các tính năng lập chỉ mục phong phú.

```
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.createCollection("items2")
{ ok: 1 }
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.items2.insert({})
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
{
  acknowledged: true,
  insertedIds: { '0': ObjectId("639d4dcc59718139c0a1fb60") }
}
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.items2.insertOne({ "name": "ABC", "category": "notebook" })
{
  acknowledged: true,
  insertedId: ObjectId("639d4df659718139c0a1fb61")
}
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.items2.insertOne({ "item": "miscelaneaus", "name": "book" })
{
  acknowledged: true,
  insertedId: ObjectId("639d4e4759718139c0a1fb62")
}
[Atlas atlas-ch7sa4-shard-0 [primary] DBMS> db.items2.find()
[
  { _id: ObjectId("639d4dcc59718139c0a1fb60") },
  {
    _id: ObjectId("639d4df659718139c0a1fb61"),
    name: 'ABC',
    category: 'notebook'
  },
  {
    _id: ObjectId("639d4e4759718139c0a1fb62"),
    item: 'miscelaneaus',
    name: 'book'
  }
]
```

Hình 33: Lược đồ trong MongoDB

Khác nhau về việc MongoDB có hỗ trợ Nested Collection mà Mysql không hỗ trợ Nested Row. Ví dụ trong MongoDB, thuộc tính address là một Collection nằm trong Collection ban đầu, nó gồm 2 thuộc tính là province và district

```
_id: ObjectId('639797bf8a72807b7798f3ce')
name: 209
address: Object
  province: "Ho Chi Minh"
  district: "Thu Duc city"
```

Hình 34: Nested trong MongoDB

Khác nhau về việc Mysql (InnoDB) có hỗ trợ Double-write buffer để ghi dữ liệu từ buffer pool vào file ở đĩa còn MongoDB thì không.

Khác nhau về việc MongoDB có hỗ trợ kiến trúc cây LSM để hỗ trợ ghi lượng lớn dữ liệu còn Mysql(InnoDB) thì không.

Khác nhau về việc MongoDB còn hỗ trợ kiến trúc In-memory còn Mysql(InnoDB) thì không.

Kiểu dữ liệu trong Mysql(InnoDB) có thể giới hạn được kích thước ví dụ như varchar(20).

```
mysql> create table Class(
    -> name varchar(20)
    -> ) ENGINE=InnoDB;
Query OK, 0 rows affected (0,05 sec)

mysql> □
```

Hình 35: Kiểu dữ liệu trong MySQL(InnoDB) có thể giới hạn kích thước

Khác nhau về việc MongoDB có hỗ trợ file format theo kiểu column-store còn Mysql(MongoDB) thì không.

Khác nhau về việc MongoDB không hỗ trợ việc tạo khóa chính ngoài khóa có sẵn là “_id” như Mysql (innodb) thì có:

```
mysql> DESCRIBE Student;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id    | int  | NO   | PRI | NULL   |       |
+-----+-----+-----+-----+-----+
1 row in set (0,00 sec)

mysql> □
```

Hình 36: Hỗ trợ tạo khóa chính ngoài khóa có sẵn trong MySQL(InnoDB)

Khác nhau về cách nén dữ liệu: Mongo sử dụng Zlip và Snappy để nén dữ liệu còn MongoDB chỉ sử dụng Zlip.

Khác nhau về các phân bổ các record: Bên Mysql(InnoDB) thì sử dụng phân bổ liên kết, mỗi block dùng pointer để liên kết với block tiếp theo. còn MongoDB thì sử dụng phân bổ một cách liên tục các block (contiguous allocation) trên đĩa.

Với việc các DBMS hiện nay đều hỗ trợ việc tự tổ chức cấu trúc file và hệ thống lưu trữ của riêng nó, việc demo chủ đề này thông qua một application gần như là không thể đối với nhóm chúng em. Thay vào đó, nhóm em đã demo kỹ hơn về các trường hợp cụ thể, cũng như ưu nhược điểm của từng DBMS thông qua các ví dụ trong phần trình bày của nhóm ở phần trước.

2 Concurrency Control

2.1 Đặc điểm của SQL Transaction và tổng quan về Concurrency Control

Để có thể hiểu rõ hơn về sự ra đời của khái niệm concurrency control và sự cần thiết của khái niệm này, ta cần đào sâu hơn về những đặc điểm cơ bản của SQL Transaction và những vấn đề xoay quanh nó.

Như ta đã biết, transaction là 1 tập các câu lệnh thực thi tới CSDL, các câu lệnh được thực thi 1 cách tuần tự. Transaction có 4 tính chất là:

- **Atomicity (Tính nguyên tố):** nếu có bất kỳ lệnh nào bị lỗi, transaction sẽ dừng lại và rollback tất cả các câu lệnh đã thực thi, trả lại cơ sở dữ liệu như lúc vừa bắt đầu transaction, vậy nên khi thực hiện 1 transaction, hoặc là tất cả các thay đổi dữ liệu được thực thi, hoặc không thay đổi nào được thực hiện.
- **Consistency (Tính nhất quán):** Một transaction sau khi hoàn thành phải để tất cả dữ liệu ở trạng thái nhất quán. Trong CSDL quan hệ, dữ liệu sau khi được thêm/xóa/cập nhật bởi các transaction sẽ phải tuân thủ đúng các luật lệ ban đầu được đặt ra cho cơ sở dữ liệu để đảm bảo tính toàn vẹn dữ liệu. Tất cả cấu trúc dữ liệu, như index phải đúng khi kết thúc một transaction.
- **Isolation (Tính cách ly):** Các transaction sẽ không tác động vào nhau, sự thay đổi mà chúng tạo nên phải độc lập với nhau. Một transaction chỉ được lấy dữ liệu ở trạng thái trước hoặc sau khi dữ liệu này bị 1 transaction khác thay đổi chứ không phải ở 1 trạng thái trung gian.
- **Durability (Tính bền vững):** Sau khi 1 transaction thực hiện thành công, các thay đổi của nó sẽ trở thành chính thức và bền vững, không bị rollback.

Nếu lưu lượng transaction được thực hiện trong 1 khoảng thời gian là ít, thì việc đảm bảo 4 tính chất này có vẻ đơn giản. Tuy nhiên trong thực tế, lượng transaction được thực hiện trong 1 DBMS là vô cùng lớn và liên tục và ở nhiều thời điểm khác nhau, vì vậy ta cần có 1 hệ thống để đảm bảo 4 tính chất này được bảo toàn.

Sự yếu kém trong việc quản lý các thao tác với dữ liệu thường dẫn tới các vấn đề về xung đột và hiệu năng trong hệ thống có nhiều users. Khi số lượng users thao tác với dữ liệu ngày một tăng, việc quản lý thao tác dữ liệu hiệu sao cho quả là vô cùng quan trọng.

Vì vậy, để đảm bảo tính chất isolation và consistency của các transaction, khái niệm concurrency control (kiểm soát tính đồng thời) được đưa ra.

Không chỉ vậy, việc kiểm soát tính đồng thời còn giúp giải quyết các mâu thuẫn giữa các thao tác đọc - ghi (read - write) và ghi - ghi (write - write), giúp bảo đảm khả năng tuần tự hóa của dữ liệu.

Do đó, kiểm soát tính đồng thời là một yếu tố quan trọng đối với việc hoạt động bình thường của một hệ thống khi có hai hoặc nhiều giao dịch yêu cầu quyền truy cập vào cùng một cơ sở dữ liệu cùng một lúc.

2.2 Mục đích và sự cần thiết của Concurrency Control

Nếu ta không duy trì được tính nhất quán trong Conccurrency Control, sẽ dễ dẫn đến hiện tượng sau khiến database của chúng ta không còn mang tính ACID nữa. Để

denmo ta sử dụng bảng update có các trường sau

```
1 CREATE TABLE product(
2     productID VARCHAR(4) NOT NULL,
3         quantity INT NOT NULL
4 );
```

- **Lost update:** Trường hợp này xảy ra khi 2 hoặc nhiều transaction cùng update 1 row từ giá trị ban đầu của nó, Update cuối cùng sẽ ghi đè các update bởi các transaction khác dẫn đến mất dữ liệu. đặt trường hợp, 2 user cùng lúc muốn thay đổi thay đổi giá trị ‘quantity’ của product ‘1001’ khi đều muốn update giá trị giảm đi trừ 100. Hai người đều sử dụng câu lệnh bên dưới với mục đích cuối cùng của 2 user đều là đưa quantity = 600 với productId = ‘1001’:

```
1 update product set quantity = quantity - 100 where productID = 1001;
2
```

The screenshot shows two side-by-side MySQL Command Line Client windows. Both windows have the title 'MySQL, 8.0 Command Line Client - Unicode'.
Left Window (User A):
- Shows the creation of the 'product' table.
- Displays the initial data for the 'product' table:

productID	quantity
1001	700
1002	600
1003	500
1004	400
1005	300

Right Window (User B):
- Shows the same initial data for the 'product' table:

productID	quantity
1001	700
1002	600
1003	500
1004	400
1005	300

Both windows show the execution of the update query 'update product set quantity = quantity - 100 where productID = 1001;' followed by a commit command.

Hình 37: Trước khi thực hiện query Update

- **Dirty Read (Uncommitted dependency):** là tình huống khi một transaction đọc dữ liệu chưa được cam kết. Ví dụ: Giả sử transaction 1 cập nhật một hàng và để nó chưa commit, trong khi đó, transaction 2 đọc hàng đã cập nhật. Nếu transaction 1 khôi phục thay đổi, transaction 2 sẽ đọc dữ liệu được coi là chưa tồn tại. Cùng bảng dữ liệu đã demo với lost update, ta sẽ set bên user B đọc dữ liệu trước khi user A roll back.

```
1 #user B set level first
2 SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
```

- **Inconsistent analysis (nonrepeatable read):** xảy ra khi một transaction đọc cùng một hàng hai lần và mỗi lần nhận được một giá trị khác nhau. Ví dụ, giả sử

The screenshot shows two MySQL command-line clients running simultaneously. Both clients are connected to the same database and are performing the same sequence of SQL queries:

```

mysql> use dbms_assignment;
Database changed
mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    700 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql> update product set quantity = quantity - 100 where productID = 1001;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> commit
->;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    600 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql> update product set quantity = quantity - 100 where productID = 1001;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> commit
->;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    500 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql>
  
```

The right client shows the final state of the database after both transactions have been committed. Product ID 1001 now has a quantity of 500, while all other products remain at their initial values.

Hình 38: Kết quả cuối cùng hiện ra là 500 cho bên phía user B nhưng đáng ra phải là 600.

This screenshot shows two MySQL command-line clients. The left client performs the following sequence of queries:

```

mysql> use dbms_assignment;
Database changed
mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    700 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql> update product set quantity = quantity - 100 where productID = 1001;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    600 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql> rollback;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from product;
+-----+-----+
| productID | quantity |
+-----+-----+
| 1001      |    700 |
| 1002      |    600 |
| 1003      |    500 |
| 1004      |    400 |
| 1005      |    300 |
+-----+-----+
5 rows in set (0.00 sec)

mysql>
  
```

The right client shows the MySQL monitor interface, which displays the current connection information and the MySQL version.

Hình 39: Như vậy B đã đọc dữ liệu chưa tồn tại.

transaction T1 đọc dữ liệu. Do đồng thời, một transaction khác T2 cập nhật cùng một dữ liệu và commit, Bây giờ nếu transaction T1 đọc lại cùng một dữ liệu, nó sẽ thấy một giá trị khác. Ở phần demo ta thực hiện query 2 transaction cho 2 user khác nhau:

```

1   #user A
2   #=====
3   SET autocommit = 0;
4   SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
5
6   START TRANSACTION;
  
```

```

7      SELECT * FROM PRODUCT;
8      #do anything here
9      DO SLEEP(10);
10     SELECT * FROM PRODUCT;
11     COMMIT;
12     =====
13
14     #user B
15     =====
16     SET autocommit = 0;
17
18     START TRANSACTION;
19     UPDATE product SET quantity = quantity - 100 where productID = 1001;
20     COMMIT;
21     =====
22

```

The screenshot shows two side-by-side MySQL Command Line Client windows. Both windows have identical command-line histories, demonstrating a phantom read scenario.

```

Enter password: *****
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 31
Server version: 8.0.31 MySQL Community Server - GPL

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> source C:\Users\LENOVO\OneDrive\Desktop\mysql_helper\Inconsistent analysis\userA.sql
Database changed
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

+-----+-----+
| productID | quantity |
+-----+-----+
| 1001 | 600 |
| 1002 | 600 |
| 1003 | 500 |
| 1004 | 400 |
| 1005 | 300 |
+-----+
5 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

+-----+-----+
| productID | quantity |
+-----+-----+
| 1001 | 500 |
| 1002 | 600 |
| 1003 | 500 |
| 1004 | 400 |
| 1005 | 300 |
+-----+
5 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>

```

Hình 40: *productID bị thay đổi sau 2 lần truy vấn của userA*

- Phantom Read:** xảy ra khi hai truy vấn giống nhau được thực thi, nhưng các hàng được hai truy vấn truy xuất lại khác nhau. Ví dụ: giả sử transaction T1 truy xuất một tập hợp các hàng thỏa mãn một số tiêu chí tìm kiếm. Bây giờ, transaction T2 tạo ra một số hàng mới phù hợp với tiêu chí tìm kiếm cho transaction T1. Nếu transaction T1 thực hiện lại câu lệnh đọc các hàng, thì lần này nó sẽ nhận được một tập hợp các hàng khác.

```

1      #user A
2      =====
3      SET autocommit = 0;
4      SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
5
6      START TRANSACTION;
7      SELECT * FROM PRODUCT;

```

```
8      #do anything here
9      DO SLEEP(10);
10     SELECT * FROM PRODUCT;
11     COMMIT;
12     =====
13
14     #user B
15     =====
16     SET autocommit = 0;
17     SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
18
19     START TRANSACTION;
20     INSERT INTO product(productID,quantity) VALUES ('1010',1000);
21     COMMIT;
22     =====
23
```

The screenshot shows two separate MySQL command-line sessions running simultaneously. Both sessions are executing the same SQL script, `userA.sql`, which contains a sleep operation and a select query. The left session, labeled 'MySQL 8.0 Command Line Client', has completed its execution and displays the results of the select query:

productID	quantity
1001	700
1002	600
1003	500
1004	400
1005	300

The right session, also labeled 'MySQL 8.0 Command Line Client', is still executing the script and has not yet reached the point where it would display the results of the select query.

Hình 41: bảng product bị thêm vào các hàng mới sau 2 lần đọc giống nhau bên userA

- **Deadlock:** là trạng thái mà mỗi transaction trong một tập hợp chứa nhiều hơn 2 transaction đang đợi tới lượt sử dụng dữ liệu mà dữ liệu đó đang bị khóa bởi transaction khác trong tập hợp vừa nêu. Demo về hiện tượng deadlock khi transaction B sử dụng dữ liệu transaction A đang lock và ngược lại.

```
1      #user A
2      =====
3      SET autocommit = 0;
4      START TRANSACTION;
5      BEGIN;
6      INSERT INTO table1(name, value) VALUES('trans1', 100);
7      DO SLEEP(10);
8      INSERT INTO table2(name, value) VALUES('trans2', 100);
```

```
9      COMMIT;
10     #=====
11
12     #user B
13     #=====
14     SET autocommit = 0;
15     START TRANSACTION;
16     BEGIN;
17     UPDATE table2 set value = 300;
18     DO SLEEP(10);
19     UPDATE table1 set value = 300;
20     COMMIT;
21     #=====
```

The screenshot shows two separate MySQL command-line sessions running simultaneously. Both sessions are executing the same SQL script, which attempts to update two different tables (table1 and table2) in a specific sequence. The deadlock occurs because both sessions are holding locks on the tables while waiting for the other session's transaction to commit or roll back. The logs from both sessions show the execution of the script and the resulting deadlock state.

Hình 42: 2 transaction gây ra deadlock

- **Starvation:** là tình huống khi một transaction phải đợi trong một khoảng thời gian không xác định để có được khóa. DBMS thường có cách xử lý cách trường hợp này bằng cách tăng cấp độ ưu tiên priority (đợi lâu sẽ có độ ưu tiên cao hơn), FCFS hoặc wait-die.
- **Cascading rollback:** Nếu trong một scheduling, lỗi của một transaction khiến một số transaction phụ thuộc khác bị lùi lại hoặc hủy bỏ. Nó chỉ đơn giản là dẫn đến lãng phí thời gian của CPU. Các Cascading Rollback này xảy ra do các sự cố Dirty Read.

Trong bất kỳ database nào, sự yếu kém trong việc quản lý các thao tác với dữ liệu thường dẫn tới các vấn đề về xung đột và hiệu năng trong hệ thống có nhiều users. Khi số lượng users thao tác với dữ liệu ngày một tăng, việc quản lý thao tác dữ liệu hiệu sao cho quả và kiểm soát concurrency control là vô cùng quan trọng và cần thiết.

2.3 Các kỹ thuật sử dụng trong Concurrency Control

Trước khi đi sâu hơn về kỹ thuật Two - Phase Locking, ta cần tìm hiểu một khái niệm quan trọng liên quan đến chủ đề này.

Đó là khái niệm Locking (khóa). Locking là cơ chế được sử dụng để đảm bảo tính toàn vẹn của các transaction và duy trì tính nhất quán của CSDL khi nhiều người thao tác trên dữ liệu cùng lúc.

- Khóa là 1 thao tác đảm bảo quyền được đọc hoặc ghi dữ liệu cho 1 transaction. Ví dụ khi ta khóa dữ liệu X (Lock (X)), các transaction khác khi cần tới X sẽ không truy cập được dữ liệu này.
- Trái ngược với khóa, mở khóa (Unlocking) là thao tác gỡ bỏ những quyền đọc/ghi của transaction với dữ liệu được nhắm tới. Ví dụ khi ta mở khóa dữ liệu X (Unlock(X)) thì tất cả các transaction sẽ đều có quyền truy cập và thao tác trên dữ liệu X.

Khóa và mở khóa là các thao tác mang tính atomic (atomic operation), có nghĩa là chúng là những thao tác không thể bị gián đoạn và chia thành các thao tác khác nhau, tương tự như các thao tác đọc và ghi.

Điều này giúp cho 2 hoặc nhiều transaction khác nhau không thể có được cùng 1 khóa độc quyền cho cùng 1 dữ liệu. Bởi vì nếu ta giả sử Lock(X) hoặc Unlock(X) không phải là những atomic operation, thì có khả năng 1 transaction sau khi gọi hàm Lock(X) để lấy khóa độc quyền cho dữ liệu X, giá trị khóa của X chưa kịp thay đổi từ không khóa sang khóa (giả sử như chưa chạy tới dòng lệnh đó), thì một transaction khác cũng gọi hàm Lock(X) cho cùng dữ liệu mà transaction đầu tiên gọi, tạo ra tình huống 2 transaction có cùng khóa độc quyền cho cùng 1 dữ liệu (vì transaction thứ 2 tưởng rằng dữ liệu này chưa bị khóa bởi transaction đầu). Điều này sẽ dễ dàng khiến cho tính toàn vẹn và nhất quán của dữ liệu bị lung lay, vì vậy việc các thao tác khóa và không khóa phải là các atomic operation là cực kỳ quan trọng.

Những loại khóa phổ biến thường xuất hiện trong DBMS là:

- **Binary Lock (Khóa nhị phân):** Khóa nhị phân có 2 giá trị là 0 và 1, trong đó 1 là giá trị thể hiện dữ liệu đang bị khóa và 0 là thể hiện dữ liệu đang không bị khóa. Khóa nhị phân giúp dữ liệu giữ được tính loại trừ lẫn nhau (Mutual Exclusion) vì nó giúp cho tối đa chỉ có 1 transaction có thể có được khóa cho dữ liệu cần sử dụng.
- **Shared/Exclusive (Read/Write) Lock (Khóa chia sẻ/độc quyền hay còn gọi là Khóa đọc/ghi):** Khác với khóa nhị phân, loại khóa này có 3 trạng thái là read-locked (đọc/chia sẻ khóa), write-locked (ghi/độc quyền khóa) và unlocked (không khóa). Ở trạng thái read-locked (hay còn gọi là shared-lock), các transaction khác có quyền để đọc dữ liệu này. Còn ở trạng thái Write-Locked hay còn gọi là Exclusive Locked, chỉ có 1 transaction có quyền giữ khóa cho dữ liệu đó. Có thể thấy rằng, khóa đọc/ghi có ít ràng buộc hơn khóa nhị phân. Ngoài ra, khi 1 transaction đang giữ khóa cho 1 dữ liệu cụ thể nào đó, transaction đó có thể yêu cầu chuyển từ trạng thái khóa này sang trạng thái khóa khác (lock conversion) với điều kiện phải thỏa mãn 1 số ràng buộc cho trước.
- **Certify Lock (Khóa chứng nhận):** Khóa chứng nhận có 4 trạng thái, ngoài 3 trạng thái Read/Write-Locked và Unlocked giống của khóa chia sẻ/độc quyền thì



khóa chứng nhận còn có 1 trạng thái khác là trạng thái Certify-Locked. Đây là trạng thái khóa độc quyền, trạng thái khóa này xuất hiện khi có 2 transaction khác nhau muốn có quyền đọc-ghi theo thứ tự với dữ liệu X. Để điều này được diễn ra, X tạo ra 2 phiên bản là phiên bản committed và phiên bản local. Ở phiên bản committed, các transaction có khóa đọc cho dữ liệu X thì sẽ đều được đọc tại phiên bản committed, còn ở phiên bản local thì chỉ có những transaction có khóa ghi mới được thao tác trên phiên bản này. Sau khi các thao tác cập nhật hoặc ghi được thực hiện bởi transaction T trên dữ liệu X, T sẽ có một certify-lock để cập nhật những thao tác mình vừa làm lên phiên bản limited của dữ liệu X, sau đó phiên bản local sẽ biến mất. Bằng việc xử lý các transaction thông qua certify lock, hiệu năng của DBMS sẽ được cải thiện phần nào.

2.3.1 Two-phase Locking

Two-Phase Locking (2PL) Protocol hay còn gọi là giao thức khóa 2 pha là một giao thức mà với mọi thao tác khóa của 1 transaction (như read_lock, write_lock) đều phải được thực hiện trước thao tác mở khóa (unlock). Giao thức này được dùng để đảm bảo tính tuần tự của các transaction. Vì khi 1 transaction đang thao tác trên 1 tập dữ liệu, nó sẽ khóa dữ liệu đó lại và không cho các transaction khác đồng thời can thiệp và thao tác với dữ liệu ban đầu mà transaction đầu đang thực hiện, chỉ khi nào thực hiện xong thì nó mới mở khóa. Để hiện thực được giao thức khóa 2 pha, mọi transaction đều phải chia thành 2 giai đoạn:

- Growing (Expanding) Phase (Pha phát triển): Đây là giai đoạn mà transaction lấy khóa cho những dữ liệu cần thiết khi thao tác, khi đang trong giai đoạn này transaction sẽ không giải phóng bất kỳ khóa nào.
- Shrinking Phase (Pha thu hẹp): Đây là giai đoạn sau khi transaction thực hiện thao tác thành công, ở giai đoạn này transaction sẽ giải phóng toàn bộ những khóa nó cần khi thực hiện thao tác và không yêu cầu thêm bất kỳ khóa nào.

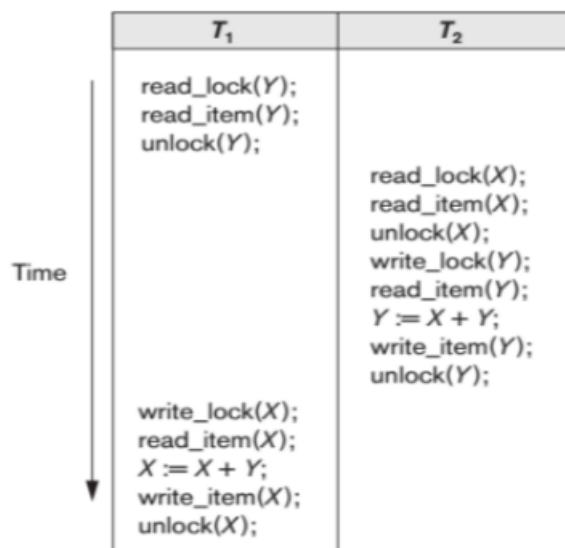
Hai pha này sẽ loại trừ nhau, pha phát triển sẽ đi trước pha thu hẹp.

2.3.2 Lock Conversion

Nếu trong quá trình thực hiện những pha này mà transaction có nhu cầu chuyển đổi khóa, thì việc nâng cấp khóa (từ khóa đọc thành khóa ghi) phải được thực hiện trong pha phát triển, ngược lại việc giáng cấp khóa (từ khóa ghi thành khóa đọc) phải được thực hiện trong pha thu hẹp. Ví dụ về một trường hợp mà 2 transaction đều không tuân thủ giao thức khóa 2 pha: Trong trường hợp này transaction unlock(Y) trước khi thực hiện thao tác chính, điều này làm cho transaction 2 có quyền thao tác trên dữ liệu Y, làm mất đi tính tuần tự giữa 2 transaction này.

Để áp dụng được kỹ thuật khóa 2 pha trong cụ thể 1 DBMS nào đó, ta cần có 2 components sau:

- **Lock Manager (Trình quản lý khóa):** Trình quản lý khóa có chức năng chính là quản lý khóa cho từng dữ liệu cụ thể.



Hình 43: Lock conversion

- **Lock Table (Bảng trạng thái khóa):** Bảng trạng thái khóa lưu trữ chi tiết khóa của từng transaction (lưu thông tin dữ liệu bị khóa, loại khóa của dữ liệu đó và con trỏ trả tới dữ liệu tiếp theo bị khóa). Một cách đơn giản để hiện thực Lock Table là qua cấu trúc dữ liệu Linked Lists.

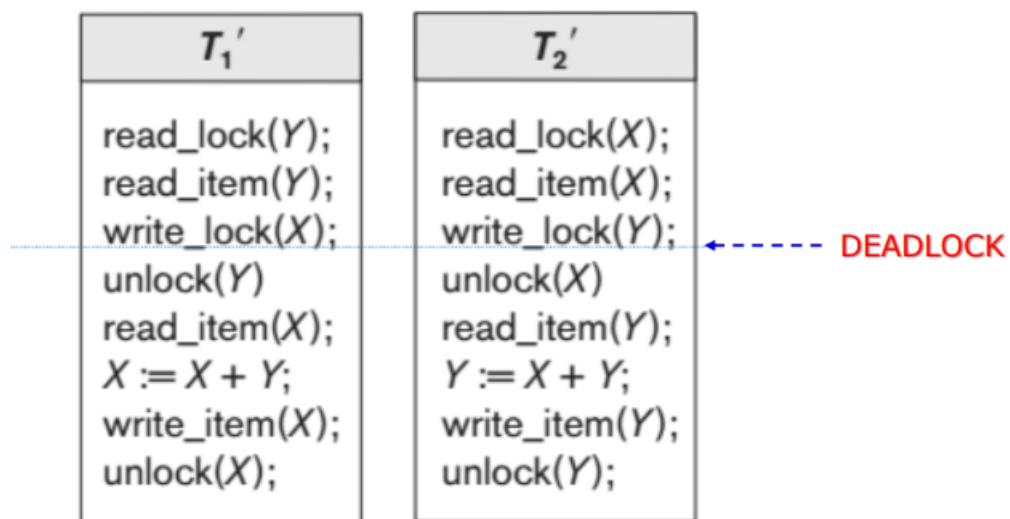
Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Hình 44: Lock table

Ví dụ về thông tin 1 node nếu hiện thực Lock Table bằng Linked List. Kỹ thuật 2PL có thể giải quyết được vấn đề tuần tự của các transaction khác nhau nhưng không thể giải quyết được vấn đề Deadlock.

Ví dụ cụ thể về 1 trường hợp 2 transaction gặp trạng thái deadlock. Ngoài kỹ thuật 2PL thông thường, ta còn có những biến thể khác của kỹ thuật này, ta có thể kể đến một vài như:

- **Strict 2PL:** Strict 2PL sẽ đảm bảo rằng các transaction sẽ không mở khóa bất kỳ 1 khóa độc quyền nào mà nó đang nắm trước khi transaction đó commits hoặc aborts. Chính vì vậy, không một transaction nào khác có thể thao tác lên dữ liệu mà transaction đầu đang ghi trước khi nó commit, làm tăng tính recoverability của transaction. Kỹ thuật này vẫn có khả năng gây ra trạng thái deadlock giữa các transaction.
- **Conservative 2PL (static 2PL):** Kỹ thuật này tương tự như kỹ thuật 2PL thông thường, chỉ khác việc trước khi transaction thực thi, nó sẽ tạo ra 1 tập read-set và write-set chứa những data mà nó sẽ thực hiện các thao tác đọc - ghi lên, nếu có bất kỳ dữ liệu nào transaction cần mà chưa lấy được khóa, transaction sẽ không khóa



Hình 45: deadlock example

bất kỳ dữ liệu nào. Bằng cách này, static 2PL sẽ giúp hệ thống tránh khỏi trạng thái deadlock.

- **Rigorous 2PL:** Tương tự như kỹ thuật Strict 2PL, tuy nhiên kỹ thuật này đảm bảo rằng các transaction sẽ không mở bất kỳ khóa (đọc/ghi) nào, trước khi transaction đó commit hoặc abort. Kỹ thuật này vẫn có thể khiến hệ thống bị trạng thái deadlock.

Lưu ý: Ta có thể sử dụng kỹ thuật 2PL với khóa nhị phân, lúc này khóa sẽ chỉ còn có 2 trạng thái là Lock và Unlock, với trạng thái Lock kể cả là thao tác đọc hay ghi thì nó cũng sẽ không cho bất kỳ các thao tác khác truy cập dữ liệu khi thao tác đó có khóa. Nhưng bên cạnh đó two phase locking vẫn có một số nhược điểm như sau đây:

- Deadlock (như đã ví dụ ở trên)
- Starvation
- Dirty Read và Irrecoverable
- Cascading Rollback Problem: xảy ra khi việc hệ thống rollback 1 transaction và kéo theo nhiều transaction khác buộc phải rollback theo

2.3.3 Kỹ thuật giải quyết Concurrency Control dựa vào trình tự mốc thời gian

Kỹ thuật này dùng trình tự mốc thời gian của transaction (Timestamp Ordering) để lên lịch và sắp xếp thứ tự thực hiện của chúng, đảm bảo tính tuần tự giữa những transaction.

Chính vì dựa vào mốc thời gian, nên mỗi transaction đều chắc chắn sẽ được thực thi tuần tự, không transaction nào phải đợi. Vì vậy, kỹ thuật này đảm bảo việc hệ thống sẽ không bị deadlock.

Ngoài ra, kỹ thuật này còn cần đảm bảo rằng nếu 1 dữ liệu mà nhiều transaction

vừa có nhu cầu đọc/ghi với nó (conflicting operations) thì thứ tự thực hiện của những thao tác trên cũng sẽ phải không làm ảnh hưởng tới tính tuần tự của hệ thống.

Để hiện thực kỹ thuật này, ta cần biết đến những khái niệm sau:

Transaction Timestamp - TS(T): là mốc thời gian dùng để xác định một transaction T, giá trị của TS(T) được gán theo thứ tự mà transaction T được gửi tới hệ thống.

Read Timestamp of item X - read_TS(X): là mốc thời gian sớm nhất giữa các transaction mà đọc thành công dữ liệu X, hoặc là mốc thời gian của transaction mới nhất vừa được thêm vào hệ thống mà đọc thành công dữ liệu X.

Write Timestamp of item X - write_TS(X): là mốc thời gian sớm nhất giữa các transaction mà ghi thành công dữ liệu X, hoặc là mốc thời gian của transaction mới nhất vừa được thêm vào hệ thống mà ghi thành công dữ liệu X.

2.3.3.a The basic timestamp ordering algorithm

Dựa vào những giá trị trên, người ta đã xây dựng ra 1 giải thuật cơ bản để đảm bảo các thao tác giữa những transaction không gây ra xung đột (conflict) như sau:

- Nếu 1 transaction T yêu cầu ghi lên dữ liệu X (write_item(X)) mà $\text{read_TS}(X) > \text{TS}(T)$ hoặc $\text{write_TS}(X) > \text{TS}(T)$ thì hệ thống sẽ abort, thực hiện rollback (bởi vì điều này chứng tỏ, đã có 1 transaction nào đó ở sau T trong trình tự mốc thời gian nhưng lại thực hiện thao tác đọc/ghi trước T, làm vi phạm quy tắc về trình tự mốc thời gian). Sau đó, hệ thống sẽ cập nhật lại các timestamp của $\text{write_TS}(X)$ hoặc $\text{read_TS}(X)$ và thực hiện lại thao tác theo đúng trình tự quy ước. Trong những trường hợp còn lại, hệ thống sẽ thực hiện thao tác $\text{write_item}(X)$ của transaction T và cập nhật giá trị $\text{write_TS}(X) = \text{TS}(T)$
- Nếu 1 transaction T yêu cầu đọc dữ liệu X (read_item(X)) mà $\text{write_TS}(X) > \text{TS}(T)$ thì hệ thống sẽ abort, thực hiện rollback (bởi vì điều này chứng tỏ, đã có 1 transaction nào đó ở sau T trong trình tự mốc thời gian nhưng lại thực hiện thao tác ghi lên X trước khi T đọc X, làm vi phạm quy tắc về trình tự mốc thời gian). Sau đó, hệ thống sẽ cập nhật lại các timestamp của $\text{write_TS}(X)$ và thực hiện lại thao tác theo đúng trình tự quy ước.
- Trong những trường hợp còn lại, hệ thống sẽ thực hiện thao tác $\text{read_item}(X)$ của transaction T và cập nhật giá trị $\text{read_TS}(X)$ bằng với timestamp lớn nhất giữa $\text{read_TS}(X)$ và $\text{TS}(T)$.

Bản chất của thuật toán này là để kiểm tra xem các thao tác gây xung đột (read-write) có đang vi phạm trình tự mốc thời gian không, và nếu có thì sẽ rollback và sắp xếp lại thứ tự thực hiện để tuần tự hóa các xung đột này.

Một ví dụ về một thao tác bị từ chối vì vi phạm trình tự mốc thời gian. Trong ví dụ này, Transaction T1 có $\text{TS} = 1$ tới trước transaction T2 có $\text{TS} = 2$, tuy nhiên $\text{Write_item}(X)$ của T1 nếu được thực hiện thông thường thì sẽ bị thực hiện sau $\text{Read_item}(X)$ của T2, điều này vi phạm quy ước trình tự mốc thời gian, làm mất tính tuần tự. Vì vậy, hệ thống sẽ từ chối thao tác $\text{Read_item}(X)$ của T2, thực hiện rollback và cập nhật lại mốc thời gian của các thao tác vi phạm trình tự, sau đó thực hiện lại những

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
	Write_item(X)	Read_item(X)	Read_TS = 2	
	Read_item(Y)			
		Write_item(X)	Write_TS = 2	
	Write_item(Y)			

Hình 46: timestamp1

thao tác đó.

Bằng cách trên, các xung đột đã được tuần tự hóa.

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
	Write_item(X)		Write_TS = 1	
		Read_item(X)	Read_TS = 2	
		Write_item(X)	Write_TS = 2	
	Read_item(Y)			Read_TS = 1
	Write_item(Y)			Write_TS = 1

Hình 47: timestamp2

2.3.3.b Strict timestamp ordering

Tương tự như timestamp ordering, tuy nhiên kỹ thuật này quản lý các tác vụ theo thời gian chặt chẽ hơn (để thuận lợi hơn trong việc khôi phục) với mục đích chính vẫn là tuân tự hóa các xung đột.

Điểm khác với kỹ thuật timestamp ordering thông thường là: Khi một transaction T đọc hoặc ghi lên 1 dữ liệu X cụ thể mà có $TS(T) > write_TS(X)$ thì thay vì thực hiện liền, nó sẽ đợi sao cho cái transaction T' mà có $TS(T') = write_TS(X)$ ghi lên dữ liệu X và commit hoặc abort, thì transaction T ban đầu mới bắt đầu thao tác với dữ liệu X.

Kỹ thuật này tuy làm giảm tốc độ thực hiện của từng transaction nhưng sẽ tránh được nhiều trường hợp phải abort và rollback liên tục của kỹ thuật timestamp ordering thông thường.

2.3.3.c Thomas's Write Rule

Thomas's Write Rule là 1 biến thể của thuật toán basic timestamp ordering algorithm nhưng không hoàn toàn tập trung vào việc tuân tự hóa các thao tác gây xung đột, từ chối ít thao tác ghi hơn.

Thomas's Write Rule được mô tả như sau Với mọi thao tác ghi dữ liệu Write_item(X):

- Nếu $read_TS(X) > TS(T)$ thì hệ thống sẽ abort hoặc rollback transaction T và từ chối thao tác ghi.
- Nếu $write_TS(X) > TS(T)$ thì hệ thống sẽ không thực thi thao tác ghi ngay mà tiếp tục xử lý các thao tác khác. Bởi vì điều này chứng tỏ, đã có những transaction khác đứng sau T trong thứ tự mốc thời gian thực hiện thao tác ghi lên dữ liệu X. Vì vậy, ta chỉ cần bỏ qua chứ không cần rollback để thực hiện thao tác ghi này.
- Trong những trường hợp còn lại, ta sẽ thực thi lệnh ghi Write_item(X) như bình thường, và đặt lại mốc thời gian $write_TS(X) = TS(T)$.

2.3.3.d Nhược điểm của Timestamp Ordering Protocol:

Tuy có nhiều biến thể khác nhau nhưng nhược điểm chung của kỹ thuật này là dễ dẫn đến việc cascading rollback, starvation (khi phải đợi những transaction khác rollback liên tục) và unrecoverable.

2.3.4 Kỹ thuật Multiversion Concurrency Control

Kỹ thuật này cho phép một dữ liệu sẽ có nhiều phiên bản khác nhau.

Khi một transaction ghi vào 1 dữ liệu cụ thể, transaction đó sẽ ghi ra 1 phiên bản mới và giữ lại phiên bản cũ của dữ liệu đó. Trong trường hợp một transaction khác có nhu cầu thao tác với dữ liệu trên, hệ thống sẽ chọn 1 phiên bản phù hợp sao cho không làm mất đi tính tuân tự rồi cấp quyền sử dụng cho transaction cần. Cụ thể hơn, một vài thao tác đọc sẽ được thực thi lên phiên bản cũ để dữ liệu để đảm bảo tính tuân tự của hệ thống.

2.3.5 Multiversion technique based on timestamp ordering

Mỗi khi có 1 transaction nào đó thực hiện ghi lên dữ liệu X cụ thể, hệ thống sẽ tạo ra 1 phiên bản mới của X (tạm gọi là X') với hai mốc thời gian $\text{read_TS}(X')$ và $\text{write_TS}(X')$ (giống trong kỹ thuật dùng thứ tự mốc thời gian) để lưu. Sẽ có nhiều phiên bản của cùng 1 dữ liệu được lưu trong hệ thống.

Đặc biệt, yêu cầu đọc sẽ không bao giờ bị từ chối trong kỹ thuật sử dụng đa phiên bản (vì sẽ luôn chọn được 1 phiên bản để đọc). Để đảm bảo tính tuần tự của hệ thống, kỹ thuật này quy ước như sau:

- Nếu một transaction T yêu cầu thao tác ghi dữ liệu ($\text{write_item}(X)$), trong trường hợp phiên bản được ghi đè lên mới nhất của X có $\text{write_TS}(X) \leq \text{TS}(T)$ và $\text{read_TS}(X) > \text{TS}(T)$ thì hệ thống sẽ abort và thực hiện rollback ($\text{write_TS}(X) \leq \text{TS}(T)$ là để tìm phiên bản mới nhất của X khi thực hiện transaction T và $\text{read_TS}(X) > \text{TS}(T)$ biểu hiện dữ liệu này đã được đọc sau khi T ghi đè lên, nên đã vi phạm trình tự mốc thời gian). Trong những trường hợp còn lại, hệ thống sẽ tạo 1 phiên bản mới của X và gán các giá trị $\text{read_TS}(X) = \text{write_TS}(X) = \text{TS}(T)$.
- Nếu một transaction T yêu cầu thao tác đọc dữ liệu ($\text{read_item}(X)$), hệ thống sẽ tìm phiên bản mới nhất của X vừa được ghi đè lên ($\text{write_TS}(X) \leq \text{TS}(T)$) sau đó cho phép T đọc giá trị của phiên bản dữ liệu vừa tìm được. Cuối cùng, hệ thống sẽ cập nhật giá trị $\text{read_TS}(X)$ của phiên bản đó bằng với giá trị mốc thời gian lớn nhất giữa $\text{TS}(T)$ và giá trị $\text{read_TS}(X)$ hiện tại.

Để hiểu rõ hơn ví dụ này, em sẽ đi qua 1 ví dụ cụ thể như sau:

T1	T2	T3	X ₀	Y ₀	Z ₀
TS = 20	TS = 25	TS = 15	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
R1(X)	R2(Z)	R3(Y)			
W1(X)	R2(Y)	R3(Z)			
R1(Y)	W2(Y)	W3(Y)			
W1(Y)	R2(X)	W3(Z)			
		W2(X)			

Hình 48

Trong ví dụ này ta xét 3 transaction T1, T2, T3 có trình tự mốc thời gian như trong hình và 3 dữ liệu Xo, Yo và Zo.

Ban đầu, T3 thực hiện thao tác đọc Y và Z, làm thay đổi giá trị read_TS của Y và Z thành 15. Tiếp đến, T3 ghi lên Y và Z, tạo ra phiên bản mới và cập nhật các giá trị $R = W = 15$ cho cả 2 dữ liệu này. Sau đó, T1 thực hiện ghi và đọc X, cập nhật giá trị read_TS của X lên 20 và tạo ra 1 phiên bản mới của X, với giá trị R và W của phiên bản này bằng 20. Thao tác đọc Y của R2 sẽ bị delay hoặc abort/rollback cho tới khi T1 thực hiện thao tác W1(Y). Thao tác W1(Y) sẽ tiếp tục tạo ra 1 phiên bản mới của Y, cập nhật giá trị $R = W = 20$ cho phiên bản mới này. Vì $\text{TS}(T1) < \text{TS}(T2)$ nên thao tác R2(Y) của T2 sẽ bị delay cho tới khi T1 thực hiện xong W1(Y), điều này làm cho Y ở

phiên bản vừa cập nhật sẽ thay đổi giá trị read_TS thành 25. Cuối cùng, các thao tác W2(Y), R2(X), W2(X) sẽ tạo ra 2 phiên bản mới của X và Y và cập nhật theo những quy luật trên, R2(X) sẽ làm thay đổi giá trị read_TS phiên bản gần nhất của nó thành 25. Cuối cùng, ta được bảng như sau:

T1	T2	T3	X ₀	Y ₀	Z ₀	X ₂₀	Y ₁₅	Z ₁₅	Y ₂₀	Y ₂₅	X ₂₅
TS = 20	TS = 25	TS = 15	R=0 W=0	R=0 W=0	R=0 W=0	R=20 W=20	R=15 W=15	R=15 W=15	R=20 W=20	R=25 W=25	R=25 W=25
		R3(Y)		R=15							
		R3(Z)			R=15						
R1(X)			R=20								
W1(X)							created				
		W3(Y)					created				
		W3(Z)						created			
	R2(Z)								R=25		
R1(Y)							R=20				
W1(Y)									created		
	R2(Y)								R=25		
	W2(Y)									created	
	R2(X)					R=25					
	W2(X)										created

Hình 49

2.3.6 Multiversion two-phase locking using certify locks

Khác với kỹ thuật two-phase locking bình thường, kỹ thuật này sử dụng thêm 1 loại khóa khác là certify-locks. Điều đó kéo theo dữ liệu khi sử dụng kỹ thuật này sẽ có đến 4 trạng thái: read-locked, write-locked, certify-locked và unlocked. Kỹ thuật này được hiện thực như sau:

Khác với 2PL thông thường, multiversion 2PL cho phép 1 transaction T đọc dữ liệu X kể cả khi một transaction khác T' đang giữ khóa độc quyền cho X.

Khi một transaction T có khóa đọc cho dữ liệu X, hệ thống sẽ tạo 1 phiên bản mới cho dữ liệu này (X'), tạo nên 2 phiên bản là 1 phiên bản đã được commit từ trước và phiên bản mới tạo như trên.

Những transactions khác T có thể tiếp tục đọc dữ liệu X mà hệ thống đã commit trong khi T đang giữ khóa.

Một khi transaction T thực hiện ghi xong và muốn commit lên hệ thống, T cần phải có certify lock của tất cả các dữ liệu mà T đang giữ khóa đọc. Chỉ khi nào dữ liệu không bị transaction nào giữ khóa đọc ghi, nó mới có quyền cấp khóa certify cho transaction khác.

Cuối cùng, khi toàn bộ certify locks đã được cấp cho transaction T, phiên bản committed của dữ liệu X sẽ được cập nhật và phiên bản cũ sẽ được thay thế, certify locks cũng sẽ được bỏ.

Kỹ thuật này giúp đảm bảo dữ liệu không bị abort một cách kéo theo, tuy nhiên deadlocks vẫn có thể xảy ra trong quá trình nâng cấp khóa.

Dể hiểu rõ hơn về kỹ thuật này, ta quan sát một ví dụ cụ thể như sau.

T1'	T2'	X	Y
Read_lock(Y)	Read_lock(X)		
Read_item(Y)	Read_item(X)		
Read_lock(X)	Read_lock(Y)		
Read_item(X)	Read_item(Y)		
Write_lock(X)	Write_lock(Y)		
Write_item(X)	Write_item(Y)		
Unlock(Y)	Unlock(X)		
Unlock(X)	Unlock(Y)		

Hình 50

Sau khi thực hiện kỹ thuật Multiversion 2PL, ta được kết quả sau:

T1'	T2'	X _{committed}	Y _{committed}	X _{T1'}	Y _{T2'}
Read_lock(Y)			Read_locked (T1')		
Read_item(Y)			T1' reads		
Write_lock(X)		Write_locked (T1')			
	Read_lock(X)	Read_locked (T2')			
	Read_item(X)	T2' reads			
	Write_lock(Y)		Write_locked (T2')		
Read_item(X)		T1' reads			
Write_item(X)				created	
Certify_lock(X)		X _{T1'} → X _{committed}		discarded	
	Read_item(Y)		T2' reads		
	Write_item(Y)				created
	Certify_lock(Y)		Y _{T2'} → Y _{committed}		discarded

Hình 51

Có thể thấy, hệ thống thực hiện các transaction theo quy tắc tương tự như kỹ thuật 2PL thông thường, tự nhiên sau khi các transaction ghi lên dữ liệu (X hoặc Y) tương ứng, phiên bản tạm thời (X-T1' và Y-T2') sẽ bị discarded và thay vào đó sẽ được commit vào phiên bản chính của dữ liệu.

2.3.7 Kỹ thuật Validation (Optimistic) Concurrency Control

Kỹ thuật này có những đặc điểm chính như sau:

Khi transaction đang thực hiện, sẽ không có bất kỳ hành động nào kiểm tra concurrency control.

Những thao tác cập nhật của transaction sẽ không được áp dụng trực tiếp vào dữ



liệu database cho tới khi transaction hoàn thành.

Vào giai đoạn trước khi transaction hoàn thành, một pha kiểm định (validation phase) sẽ kiểm tra xem những thao tác của transaction có làm vi phạm tính tuần tự của hệ thống hay không.

Trong trường hợp tính tuần tự không bị vi phạm, transaction sẽ được commit và database sẽ cập nhật những thay đổi đó. Ở trường hợp còn lại, transaction sẽ bị abort và được khởi chạy lại sau.

Sở dĩ kỹ thuật này có tên gọi là Optimistic (lạc quan) là vì kỹ thuật này cho rằng hệ thống sẽ hoạt động khá trơn tru và xuất hiện ít xung đột, nên sẽ không kiểm tra khi các transaction đang thực thi.

Khi sử dụng kỹ thuật này, 1 transaction sẽ trải qua 3 pha:

- **Read Phase (Pha đọc):** Trong pha này, transaction có thể đọc những dữ liệu đã được commit ở trong database. Những cập nhật của transaction sẽ chỉ được cập nhật trong phiên bản local của dữ liệu.
- **Validation Phase (Pha kiểm định):** Tiếp đến, hệ thống sẽ kiểm tra xem các thay đổi vừa rồi có làm ảnh hưởng đến tính tuần tự của hệ thống hay không.
- **Write Phase (Pha ghi):** Cuối cùng, nếu kiểm định thành công, hệ thống sẽ cập nhật những thao tác của transaction với dữ liệu vào database, nếu không hệ thống sẽ cho hủy transaction và cho chạy lại ở lần sau.

Điểm đặc trưng ở kỹ thuật này là ở pha kiểm định, vậy nên em sẽ trình bày cụ thể hơn trong pha này, hệ thống sẽ thực hiện những gì.

Trong pha kiểm định của 1 transaction T1, hệ thống sẽ sử dụng kỹ thuật trình tự mốc thời gian để kiểm tra xem T1 có gây xung đột hoặc làm ảnh hưởng tới những transaction đã được commit hoặc những transaction T2 khác ($TS(T2) > TS(T1)$) đang trong pha kiểm định hay không.

Để nhằm đảm bảo tính tuần tự và tính nhất quán, hệ thống thực hiện những bước kiểm tra sau:

1. Kiểm tra xem T2 có thực hiện pha ghi trước khi T1 bắt đầu pha đọc hay không.
2. Kiểm tra xem có đúng là T1 thực hiện pha ghi trước khi T2 hoàn thành pha ghi hay không, và kiểm tra xem nếu tập dữ liệu mà T1 đọc có điểm gì trùng với tập dữ liệu ghi của T2.
3. Kiểm tra xem nếu T2 hoàn thành pha đọc trước T1, và kiểm tra tập dữ liệu mà T1 đọc và ghi xem có trùng với tập dữ liệu mà T2 ghi hay không.

Chỉ khi nào mà kiểm định (1) trả về false, thì hệ thống mới tiếp tục thực hiện kiểm định điều kiện (2), và chỉ khi nào (2) trả về false, hệ thống mới kiểm định điều kiện (3).

Chỉ cần 1 trong 3 kiểm định trên đúng, chứng tỏ hệ thống vẫn đảm bảo được tính tuần tự và nhất quán của dữ liệu. Trong những trường hợp còn lại, pha kiểm định của transaction T sẽ trả về thất bại và hệ thống sẽ cho abort sau đó khởi chạy lại T sau.

Nhược điểm lớn nhất của kỹ thuật này đó là hệ thống sẽ dễ gây nên hiện tượng starvation trong trường hợp xuất hiện nhiều transaction ngắn nhưng thường xuyên gây conflict.

2.3.8 Kỹ thuật Multiple Granularity Locking

Để hiểu kỹ thuật này, đầu tiên ta cần biết về khái niệm Granularity (Độ mịn). **Granularity** được quy ước là kích thước của dữ liệu được phép bị khóa. Ta lưu ý rằng, kích thước một dữ liệu càng lớn, thì việc kiểm soát tính đồng thời sẽ càng làm chậm hệ thống. Ví dụ như khi ta khóa một dữ liệu có kích thước ngang với một khối trong đĩa, ta cần phải khóa cả khối mà chứa dữ liệu đó, khiến cho nhiều transaction khác khi muốn thao tác với dữ liệu bên trong khối đó sẽ phải đợi. Còn nếu kích thước của dữ liệu chỉ bằng một record, thì trường hợp trên sẽ không xảy ra mà các transaction khác sẽ dễ dàng thực thi hơn.

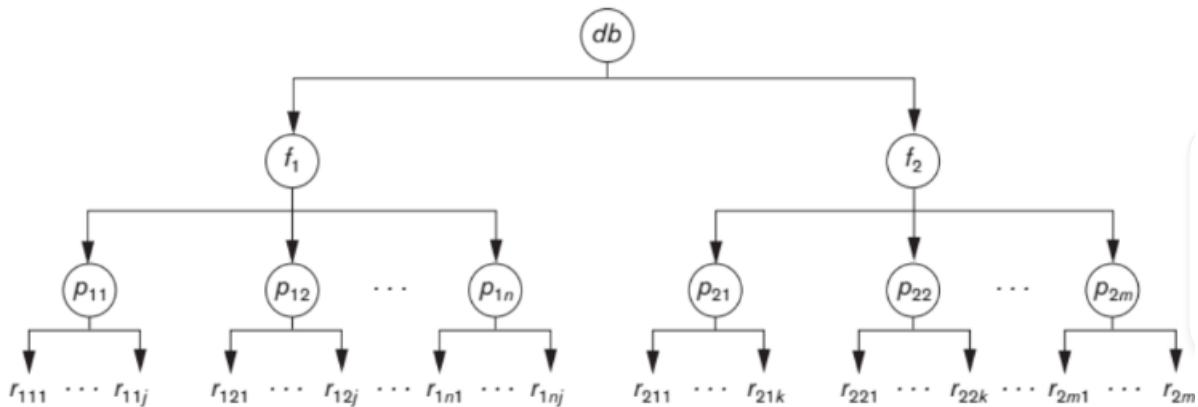
Không chỉ dữ liệu có kích thước lớn gặp vấn đề, dữ liệu có kích thước nhỏ cũng vậy. Khi kích thước của dữ liệu càng nhỏ, database sẽ càng chứa nhiều dữ liệu hơn, kéo theo việc phải kiểm soát nhiều khóa hơn, khiến cho hệ thống dễ bị quá tải khóa (lock overhead), làm cho hệ thống bị dính trường hợp storage dùng cho lưu trữ dữ liệu thì ít, nhưng storage dùng cho kiểm soát khóa, kiểm soát đồng thời thì lại quá nhiều.

Nếu nhỏ quá cũng không tốt mà lớn quá cũng không tốt, vậy thì kích thước của một mục như thế nào là hợp lý?

Câu trả lời chính là dựa vào loại của từng transaction. Nếu transaction chỉ thực hiện truy cập lên một số lượng nhỏ records, thì kích thước của một dữ liệu tốt nhất nên ngang với một record. Trong trường hợp còn lại, khi mà transaction truy cập nhiều dữ liệu trong cùng 1 file, thì kích thước của 1 mục nên ngang với block hoặc file để transaction có thể coi toàn bộ những record mà nó muốn truy xuất là một hoặc một vài mục dữ liệu. Những vấn đề và ý tưởng giải quyết trên đã tạo ra khái niệm về đa độ mịn.

Multiple Granularity (đa độ mịn) là thuật ngữ liên quan đến việc chia nhỏ cơ sở dữ liệu thành các khối có thể khóa được. Việc sử dụng đa độ mịn sẽ giúp hệ thống kiểm soát được những dữ liệu nào cần được khóa và khóa như thế nào, giúp hệ thống dễ dàng quyết định khi nào thì nên khóa hoặc mở khóa 1 dữ liệu cụ thể. Trong cấu trúc đa độ mịn, mức độ mịn (kích thước của một dữ liệu) có thể được thay đổi linh hoạt. Chính vì những lý do trên, giao thức sử dụng đa độ mịn sẽ làm tăng tính đồng thời và giảm thiểu việc quá tải khóa (lock overhead). Cấu trúc này có thể được biểu diễn dưới dạng cây. Cụ thể như sau, giả sử một cây có 4 bậc:

- Bậc 1 chứa node đại diện cho cả database.
- Bậc 2 chứa node đại diện cho hệ thống files.
- Bậc 3 chứa node đại diện cho hệ thống các pages.
- Bậc 4 là các node lá đại diện cho các records.



Hình 52: Cây đa độ mịn

Có tổng cộng 5 loại khóa khi áp dụng kỹ thuật này, các loại khóa đó là:

- **Shared Lock (S) trên node N:** Dữ liệu trong N sẽ bị khóa bằng khóa chia sẻ.
- **Exclusive Lock (X) trên node N:** Dữ liệu trong N sẽ bị khóa bằng khóa độc quyền.
- **Intention-shared Lock (IS) trên node N:** Khóa chia sẻ sẽ được gửi tới một vài node con của N.
- **Intention-exclusive Lock (IX) trên node N:** Khóa độc quyền sẽ được gửi tới một vài node con của N
- **Shared-intention-exclusive Lock (SIX) trên node N:** Dữ liệu trong N sẽ bị khóa bằng khóa chia sẻ nhưng dữ liệu trong một vài node con của N sẽ bị khóa bởi khóa độc quyền.

Để tìm hiểu tại sao khi phân bậc như cấu trúc trên lại đem đến nhiều hiệu quả, ta tìm hiểu thông qua ví dụ sau

Giả sử trường hợp có 1 transaction T1 muốn update toàn bộ record trong file f1, T1 sẽ có khóa độc quyền cho toàn bộ các trang trong file f1 và các record nằm trong các trang đó. Điều này thuận tiện cho T1 hơn vì chỉ cần giữ 1 khóa là khóa của file f1, thay vì giữ khóa của rất nhiều các trang hoặc rất nhiều record nằm trong file đó.

Tiếp đến, ta giả sử có 1 transaction khác là T2 chỉ muốn đọc 1 dữ liệu cụ thể nằm trong 1 trang thuộc file f1 đó (giả sử là dữ liệu r1nj), điều đó kéo theo T2 phải có được khóa đọc cho dữ liệu r1nj. Việc kiểm tra xem T2 muốn khóa đọc cho r1nj có hợp lệ hay không cũng sẽ hiệu quả hơn nhiều khi sử dụng cách phân cấp này, bởi vì lúc này ta chỉ cần duyệt ngược từ node r1nj đến node root (database) của cây trên, trong quá trình duyệt mà hệ thống báo có conflicting lock thì ta chỉ cần block T2 lại và để T2 đợi.

Tuy nhiên, trong trường hợp mà T2 đến trước T1, khóa ghi dữ liệu r1nj sẽ được trao cho T2 trước, và khi T1 cần khóa ghi cho file f1, lock manager sẽ gặp nhiều vấn đề hơn trong việc kiểm tra xem những node con của file f1 (page và record) có gây ra lock conflict hay không. Điều này sẽ làm mất đi độ hiệu quả khi sử dụng kỹ thuật khóa đa độ mịn.

Chính vì để khắc phục vấn đề này, **Intention Lock (IS, IX, SIX)** được tạo ra. Intention Lock sẽ giúp transaction xác định được các loại khóa mà nó cần (shared hoặc



exclusive) cho những node con khi đi từ root tới node dữ liệu mà nó cần thao tác. Intention Lock hoạt động theo kiểu nếu như có 1 transaction muốn thao tác tới 1 dữ liệu X cụ thể, nó sẽ từ từ báo cho các node cha của X là nó muốn thao tác với X bằng việc xin khóa intention phù hợp cho từng node cha, khi đã đủ khóa để đi được tới X thì hệ thống mới cho phép transaction này thao tác với X. Có nghĩa khi 1 transaction đang nắm khóa intention cho 1 node, transaction đó sẽ muốn thao tác với dữ liệu của node con của node đó.

2.3.8.a Multiple Granularity Locking Protocol:

Giao thức khóa đa độ mịn được quy ước như sau

1. Tính tương thích của khóa (lock compatibility) phải được tuân thủ.
2. Gốc (root) của cây phải được khóa đầu tiên, trong bất kỳ chế độ nào.
3. Một transaction T chỉ được cấp khóa S hoặc IS cho 1 node N chỉ khi transaction T đó đã có khóa IS hoặc IX của các node cha của N.
4. Một transaction T chỉ được cấp khóa X, IX hoặc SIX cho 1 node N chỉ khi transaction T đó đã có khóa IX hoặc SIX của các node cha của N.
5. Một transaction T có thể khóa một node chỉ khi nó chưa mở khóa node nào (để tuân thủ giao thức 2PL trong quá trình khóa/mở khóa).
6. Một transaction T có thể mở khóa một node N, chỉ khi không có node con nào của N đang bị khóa bởi T.

Quy tắc 2, 3, 4 nêu điều kiện để một transaction được cấp khóa cho 1 node trong các trường hợp cụ thể. Quy tắc 5, 6 áp dụng giao thức 2PL vào hệ thống nhằm đảm bảo tính tuân tự.

Giao thức khóa đa độ mịn được dùng rất phù hợp trong trường hợp nhiều transaction thao tác với nhiều kiểu dữ liệu với những kích thước khác nhau, làm giảm tải locking overhead và transaction blocking so với những cấu trúc đơn độ mịn.

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Hình 53: Sơ đồ biểu diễn trạng thái tương thích khóa của kỹ thuật đa độ mịn.

2.4 Concurrency Control trong MongoDB

MongoDB chủ yếu sử dụng kỹ thuật khóa đa độ mịn để giải quyết vấn đề kiểm soát tính đồng thời. Vì phần kỹ thuật khóa đa độ mịn nhóm em đã trình bày ở phần trước rồi, nên trong phần này sẽ chỉ nêu những điểm đặc trưng của MongoDB khi sử dụng kỹ thuật này.

Cụ thể hơn, MongoDB áp dụng kỹ thuật này cho 3 cấp dữ liệu là cấp global, cấp database và cấp collection. Trong một vài trường hợp ở dưới cấp collection (cấp document), mongoDB sẽ cho phép storage engine cá nhân tự giải quyết vấn đề concurrency control.

Ngoài những khóa shared/exclusive lock phổ biến, MongoDB còn sử dụng các loại khóa intention lock khác mà ta thường thấy trong kỹ thuật khóa đa độ mịn như Intention-shared Lock, Intention-Exclusive Lock.

Giả sử như một transaction yêu cầu khóa ghi (X) cho 1 dữ liệu cụ thể ở cấp document, transaction này sẽ cần thêm khóa Intention-exclusive Lock (IX) ở cả cấp database và collection trong quá trình duyệt tới dữ liệu cần khóa.

Một database có thể bị khóa đồng thời bởi khóa IS và IX, nhưng nếu đã bị khóa bởi khóa X thì sẽ không thể bị khóa bởi các khóa khác được nữa. Ngoài ra, nếu một dữ liệu bị khóa bởi khóa S, thì nó chỉ có thể bị khóa thêm bởi khóa IS và ngược lại. MongoDB đã tận dụng điều này để xử lý Lock Request một cách hiệu quả hơn.

2.4.1 Global Instance-Wide Lock

Trước phiên bản 2.2, MongoDB chỉ cho phép 1 khóa ở cấp “global” đối với 1 đối tượng mongod (mongod instance). Từ phiên bản 2.2 trở đi, MongoDB cho phép khóa ở từng database cho các tác vụ đọc ghi.

Ví dụ như trong trường hợp ta có 6 database và 1 database yêu cầu khóa ghi, thì ở phiên bản 2.2, cả 5 database còn lại vẫn sẽ sẵn sàng cho việc đọc ghi.

Tuy nhiên, đối với những tác vụ ở cấp global, mà phổ biến là những thao tác ngắn nhưng liên quan đến nhiều database, thì hệ thống vẫn sẽ dùng instance-wide lock (lock



cả 1 instance) chứ không lock lần lượt từng database. Vì lock từng database sẽ làm tốn rất nhiều tài nguyên, trong khi thao tác mình đang thực hiện thì lại ngắn.

Dầu tiên, ta demo bằng việc lock toàn bộ instance qua lệnh db.fsyncLock()

```
27017> db.fsyncLock()
{
  info: 'now locked against writes, use db.fsyncUnlock() to unlock',
  lockCount: Long("1"),
  seeAlso: 'http://dochub.mongodb.org/core/fsynccommand',
  ok: 1
}
```

Hình 54

Sau đó dùng lệnh db.adminCommand(lockInfo:1) để xem trạng thái khóa ở các cấp global.

```
27017> db.adminCommand(
...   {
...     lockInfo: 1
...   }
...
{
  lockInfo: [
    {
      resourceId: '{2305843009213693952: Global, 0}',
      granted: [
        {
          mode: 'IS',
          convertMode: 'NONE',
          enqueueAtFront: false,
          compatibleFirst: false,
          debugInfo: '',
          clientInfo: { desc: 'fsyncLockWorker', opid: 596505 }
        }
      ],
      pending: []
    },
    {
      resourceId: '{2305843009213693953: Global, 1}',
      granted: [
        {
          mode: 'IS',
        }
      ],
      pending: []
    }
  ]
}
```

Hình 55

```
resourceId: '{2305843009213693955: Global, 3}',  
granted: [  
  {  
    mode: 'S',  
    convertMode: 'NONE',  
    enqueueAtFront: true,  
    compatibleFirst: true,  
    debugInfo: '',  
    clientInfo: { desc: 'fsyncLockWorker', opid: 596505 }  
  },  
  ],  
  pending: []  
],  
ok: 1
```

Hình 56

Như ta đã thấy ở trên, toàn bộ các cấp dữ liệu ở instance này đã bị lock, để mở khóa ta dùng lệnh db.fsyncUnlock() sau khi Unlock, các resource trên sẽ được mở khóa như sau.

```
27017> db.fsyncUnlock()  
{ info: 'fsyncUnlock completed', lockCount: Long("0"), ok: 1 }  
27017> db.adminCommand({lockInfo:1})  
{ lockInfo: [], ok: 1 }
```

Hình 57

Lưu ý: lockInfo sẽ chỉ cho người dùng biết về khóa trong phạm vi 1 instance nào đó.

2.4.2 Lock Request

Mỗi yêu cầu cho khóa (lock request) đọc, ghi đều theo thứ tự được đưa vào hàng đợi. Tuy nhiên, để tối ưu hóa throughput, khi hệ thống cấp khóa cho 1 lock request, những lock request tương thích khác cũng sẽ được chấp nhận, hoặc thậm chí nếu có 1 lock request gây xung đột, hệ thống sẽ thực hiện giải phóng một số khóa để giải quyết xung đột.

Để hiểu rõ hơn về cách MongoDB giải quyết lock request, ta xét ví dụ sau:

Giả sử một collection vừa giải phóng khóa X và hàng đợi cấp khóa của collection này là như sau:

IS -> IS -> X -> X -> S -> IS

Trong trường hợp thông thường, nếu hàng đợi này hoạt động tuân thủ nghiêm ngặt theo nguyên tắc FIFO (First-In, First-Out), collection này sẽ chỉ cấp 2 khóa IS và đợi cho đến khi nào 2 khóa IS này được giải phóng thì nó mới có thể cấp được khóa X. Tuy nhiên, MongoDB sẽ không hoạt động vậy mà trong khi cấp 2 khóa IS đầu hàng đợi, nó sẽ cấp luôn những khóa S và IS ở sau dù cho 2 khóa này có vào hàng đợi sau X (vì khóa S có thể cùng tồn tại với khóa IS được nên việc cấp khóa này chắc chắn không gây xung đột). Việc MongoDB tối ưu hóa theo cách này sẽ giúp hệ thống hạn chế việc phải giải quyết hiện tượng starvation rất nhiều.

Đối với những tác vụ đọc/ghi thông thường, WiredTiger sử dụng kỹ thuật optimistic concurrency control (nhóm em có trình bày ở phần trước) để đảm bảo tính đồng thời và nhất quán của các thao tác này. Khi storage engine phát hiện ra 2 thao tác gây xung đột, một trong chúng sẽ báo là có xung đột ghi và MongoDB sẽ thông báo và cho thử lại thao

tác đó sau.

Để demo phần này, đầu tiên em sẽ khóa toàn bộ instance của mình bằng lệnh db.fsyncLock(), sau đó sử dụng lệnh insertOne.

Hiện tượng xảy ra tương tự như trên là lệnh này sẽ đợi khóa cho tới khi em mở

```
27017> db.demo2.insertOne( { name: "a10", age: 22 } )
```

Hình 58

khóa toàn bộ instance này. Tuy nhiên, để kiểm tra hàng đợi khóa của operation sử dụng database này, em tiếp tục sử dụng lệnh db.adminCommand(lockInfo:1)

```
resourceId: '{2305843009213693955: Global, 3}',  
granted: [  
  {  
    mode: 'S',  
    convertMode: 'NONE',  
    enqueueAtFront: true,  
    compatibleFirst: true,  
    debugInfo: ''  
    clientInfo: { desc: 'fsyncLockWorker', opid: 628374 }  
  },  
],  
pending: [  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: { desc: 'LogicalSessionCacheRefresh', opid: 630626 }  
  },  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: { desc: 'LogicalSessionCacheRefresh', opid: 630626 }  
  },  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: { desc: 'LogicalSessionCacheRefresh', opid: 630626 }  
  },  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: {  
      desc: 'conn62',  
      connectionId: 62,  
      client: '127.0.0.1:50901',  
      opid: 630718  
    }  
  }  
]  
],  
ok: 1  
27017> |
```

Hình 59

```
mongosh mongodb://127.0.0.1:27017> | + - v  
  }  
],  
pending: [  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: {  
      desc: 'LogicalSessionCacheRefresh', opid: 630626 }  
  },  
  {  
    mode: 'IX',  
    convertMode: 'NONE',  
    enqueueAtFront: false,  
    compatibleFirst: false,  
    debugInfo: ''  
    clientInfo: {  
      desc: 'conn62',  
      connectionId: 62,  
      client: '127.0.0.1:50901',  
      opid: 630718  
    }  
  }  
]  
],  
ok: 1  
27017> |
```

Hình 60

Như ta đã thấy, khác với phần trên, ngoài việc collection này đang bị khóa, hàng đợi khóa của collection này tăng lên vì lệnh insertOne em vừa sử dụng.

Sau khi giải phóng khóa cho instance bằng lệnh db.fsyncUnlock(), hiện tượng đầu tiên xảy ra là lệnh insertOne trước đó được thực hiện ngay, vì lúc này nó không còn phải đợi khóa.

```
27017> db.demo2.insertOne( { name: "a10", age: 22 } )
{
  acknowledged: true,
  insertedId: ObjectId("639df21266188cade4fef8a2")
}
```

Hình 61

2.4.3 Một số cách để xem trạng thái khóa trong MongoDB

Ngoài ra, để xem thông tin khóa trên từng operations ta có thể xem qua lệnh db.currentOp(). Ta có thể dùng lệnh db.currentOp() để xem thông tin khóa của những Operation đang sử dụng database.

Lệnh db.currentOp() sẽ hiển thị thông tin của từng operation sử dụng database,

```
27017> db.currentOp()
{
  inprog: [
    {
      type: 'op',
      host: 'DESKTOP-IVPPMP4:27017',
      desc: 'JournalFlusher',
      active: true,
      currentOpTime: '2022-12-17T23:08:39.700+07:00',
      opid: 603034,
      op: 'none',
      ns: '',
      command: {},
      numYields: 0,
      locks: {},
      waitingForLock: false,
      lockStats: {},
      waitingForFlowControl: false,
      flowControlStats: {}
    },
    {
      type: 'op',
      host: 'DESKTOP-IVPPMP4:27017',
      desc: 'conn44',
      connectionId: 44,
      client: '127.0.0.1:49822',
      appName: 'mongosh 1.6.1',
      clientMetadata: {
        $clusterTime: {
          clusterTime: ISODate("2022-12-17T23:08:39.700+07:00"),
          signature: [Object]
        }
      }
    }
  ]
}
```

Hình 62

trong đó có trường locks để xem khóa, numYields là số lần thu hồi khóa ngoài ra còn có các thông số khác như waitingForLock để xem thử operation này có đang đợi khóa hay không, lockStats để xem chi tiết thông số khóa.

Trong trường hợp sau khi khóa instance bằng lệnh db.fsyncLock() và dùng db.currentOp() để xem thông tin, ta được output như sau

```
{  
    type: 'op',  
    host: 'DESKTOP-IVPPMP4:27017',  
    desc: 'fsyncLockWorker',  
    active: true,  
    currentOpTime: '2022-12-17T23:10:41.738+07:00',  
    opid: 604585,  
    op: 'none',  
    ns: '',  
    command: {},  
    numYields: 0,  
    locks: {  
        ParallelBatchWriterMode: 'r',  
        FeatureCompatibilityVersion: 'r',  
        ReplicationStateTransition: 'w',  
        Global: 'R'  
    },  
    waitingForLock: false,  
    lockStats: {  
        ParallelBatchWriterMode: { acquireCount: { r: Long("1") } },  
        FeatureCompatibilityVersion: { acquireCount: { r: Long("1") } },  
        ReplicationStateTransition: { acquireCount: { w: Long("1") } },  
        Global: { acquireCount: { R: Long("1") } }  
    },  
    waitingForFlowControl: false,  
    flowControlStats: {}  
},  
],  
fsyncLock: true,
```

Hình 63

Sau khi khóa, ta thấy trường thông tin fsyncLock đã chuyển thành true, thông tin khóa (locks) đã được cập nhật.

Ta còn có lệnh db.serverStatus() để xem thông tin server, output của lệnh này chứa rất nhiều trường thông tin khác của server trong đó có những thông tin về khóa tương tự như lệnh db.currentOp() vừa dùng.

2.4.4 Những trường hợp thu hồi khóa trong MongoDB

Trong nhiều tình huống, các tác vụ truy vấn, cập nhật và xóa dữ liệu khi thực hiện quá lâu sẽ bị thu hồi khóa để các tác vụ khác có thể thao tác với dữ liệu đó.

Ở những phiên bản đầu (VD như phiên bản 2.0) hệ thống sẽ dựa vào time slice và số lượng thao tác đang đợi để được thực hiện để quyết định có nên thu hồi khóa hay không.

Trong những phiên bản sau (từ phiên bản 2.2) trở đi, nhiều thuật toán dự đoán truy xuất trên đĩa (vd như page fault) xuất hiện, cải thiện quyết định thu hồi khóa của hệ thống. Cụ thể hơn, MongoDB có thêm một trạng thái là “yield for page fault” (tạm dịch là thu hồi khóa vì gặp page fault). MongoDB sẽ dựa vào thông tin truy xuất dữ liệu trong bộ nhớ và dự đoán dữ liệu nào có sẵn trước khi thực hiện 1 thao tác đọc. Nếu MongoDB dự đoán được là dữ liệu này sẽ không có sẵn với thao tác đọc này, MongoDB sẽ báo “yield for page fault” và tạm thời thu hồi khóa của thao tác này, sau khi load dữ liệu cần đọc vào memory, hệ thống sẽ trả lại khóa và thao tác tiếp tục được thực thi.

Nếu có một tác vụ ghi dài mà ảnh hưởng đến nhiều documents khác (VD như update() nhiều dữ liệu) thì khóa của thao tác này sẽ bị thu hồi một cách định kỳ, để các thao tác đọc khác có thể được thực hiện trong quá trình thao tác ghi này diễn ra. Tương tự đối với những tác vụ đọc dài (đọc nhiều dữ liệu).



Tóm lại trong đa số trường hợp, việc thu hồi khóa là không cần thiết vì các intent lock ở cấp global, database hoặc collection sẽ không cản trở các thao tác đọc/ghi. Tuy nhiên, việc thu hồi khóa định kỳ sẽ được diễn ra cho 1 thao tác vì những mục đích sau:

- Để tránh các transaction lưu trữ thực hiện quá lâu vì những transaction này có khả năng yêu cầu một lượng lớn dữ liệu trong bộ nhớ.
- Để đóng vai trò làm 1 điểm thao tác gián đoạn, giúp hệ thống có thể loại bỏ những thao tác thực hiện quá lâu trong hệ thống.
- Để cho phép các thao tác quan trọng cần khóa độc quyền cho collection (vd như drop collection/index hoặc các thao tác tạo) được thực hiện.

Vì MongoDB không có câu lệnh hỗ trợ việc khóa thủ công các collection nên việc demo thu hồi khóa gần như là không thể, em xin được bỏ qua phần này.

2.4.5 Concurrency Control giữa các Shard và Replication trong MongoDB

Như ta đã biết, MongoDB có hỗ trợ scale theo chiều ngang (Horizontal Scaling), tức là sẽ hỗ trợ chia nhỏ dữ liệu của hệ thống và tải chúng lên nhiều server. Điều này kéo theo việc MongoDB cũng sẽ phải đưa ra giải pháp giải quyết tính đồng thời giữa các sharded server được chia nhỏ này.

Với mỗi sharded cluster, khóa sẽ chỉ được áp dụng với một shard cụ thể nào đó thôi, chứ không được áp dụng cho toàn bộ cluster. Điều này kéo theo mỗi mongod instance sẽ độc lập với các mongod instance khác trong sharded cluster và sẽ chỉ sử dụng khóa của riêng instance đó.

Replication trong MongoDB là một chức năng cho phép nhiều database server chia sẻ cùng 1 tập dữ liệu nhằm giảm tải dư thừa dữ liệu và giúp cải thiện hiệu năng khi load.

Trong replica set, secondary là tập dữ liệu sao chép từ dữ liệu gốc (master) trong database. Tập secondary chỉ chấp nhận những tác vụ đọc, ngược lại tập primary sẽ tập trung xử lý những tác vụ ghi. Secondary sẽ dựa vào sổ ghi nhật ký để thực hiện song song những tác vụ đọc.

Bắt đầu từ phiên bản 4.0, các tác vụ đọc trong secondary sẽ đọc một snapshot của dữ liệu (snapshot là bản copy hoàn chỉnh của dữ liệu trong 1 mongod instance trong 1 thời gian xác định) nếu trong quá trình secondary đang trải qua quá trình sao chép. Điều này cho phép những tác vụ đọc được diễn ra đồng thời trong quá trình sao chép nhưng vẫn đảm bảo tính consistency của dữ liệu.

Đối với những phiên bản trước 4.0, tác vụ đọc trong secondary sẽ bị chặn cho tới khi việc sao chép hoàn thành.

2.4.6 Lock-free operation

MongoDB sẽ hỗ trợ những thao tác lock-free (không yêu cầu khóa) cho một vài câu lệnh như:



- find
- count
- distinct
- aggregate
- mapReduce
- listCollections
- listIndexes

Dể kiểm thử trường hợp này trong mongoDB, đầu tiên em sử dụng lệnh db.fsyncLock() để khóa toàn bộ instance em đang dùng, sau đó dùng lệnh insertOne để ghi vào collection demo2. Hiện tượng xảy ra là hệ thống bị kẹt chứ không thực thi, vì em đã khóa toàn bộ instance nên giao tác này sẽ chờ mãi mà không có khía cạnh phục vụ cho việc thực thi.

```
27017> db.fsyncLock()
{
  info: 'now locked against writes, use db.fsyncUnlock() to unlock',
  lockCount: Long("1"),
  seeAlso: 'http://dochub.mongodb.org/core/fsynccommand',
  ok: 1
}
27017> db.demo2.insert( { name: "a9", age: 22 } )
DeprecationWarning: Collection.insert() is deprecated. Use insert
Stopping execution...
27017> db.demo2.insert( { name: "a9", age: 22 } )

27017>

27017> db.demo2.insertOne( { name: "a9", age: 22 } )
|
```

Hình 64

Tuy nhiên, khi em dùng lệnh find() thì kết quả được trả về một cách nhanh chóng



```
27017> db.fsyncLock()
{
  info: 'now locked against writes, use db.fsyncUnlock() to unlock',
  lockCount: Long("1"),
  seeAlso: 'http://dochub.mongodb.org/core/fsynccommand',
  ok: 1
}
27017> db.demo2.insert( { name: "a9", age: 22 } )
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or
Stopping execution...
27017> db.demo2.insert( { name: "a9", age: 22 } )

27017>
27017> db.demo2.insertOne( { name: "a9", age: 22 } )
Stopping execution...
27017>

>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
27017> db.demo2.find()
[
  { _id: ObjectId("638dbf7db0ce7102d5499c6e"), name: 'a3', age: 21 },
  { _id: ObjectId("638dbf82b0ce7102d5499c6f"), name: 'a2', age: 21 },
  { _id: ObjectId("639dcc1cc0835999417d174"), name: 'a1', age: 21 },
  { _id: ObjectId("639dccf5cc0835999417d179"), name: 'a4', age: 21 },
  { _id: ObjectId("639dccf7cc0835999417d17a"), name: 'a5', age: 21 }
]
```

Hình 65

Để kiểm thử tiếp, em sẽ unlock toàn bộ instance và dùng lại lệnh InsertOne

```
mongosh mongodb://127.0.0.1:27017 | + | ~
}
27017> db.demo2.insert( { name: "a9", age: 22 } )
DeprecationWarning: Collection.insert() is deprecated. Use insertOne, insertMany, or bulkWrite.
Stopping execution...
27017> db.demo2.insert( { name: "a9", age: 22 } )

27017>
27017> db.demo2.insertOne( { name: "a9", age: 22 } )
Stopping execution...
27017>

>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
27017> db.demo2.find()
[
  { _id: ObjectId("638dbf7db0ce7102d5499c6e"), name: 'a3', age: 21 },
  { _id: ObjectId("638dbf82b0ce7102d5499c6f"), name: 'a2', age: 21 },
  { _id: ObjectId("639dcc1cc0835999417d174"), name: 'a1', age: 21 },
  { _id: ObjectId("639dccf5cc0835999417d179"), name: 'a4', age: 21 },
  { _id: ObjectId("639dccf7cc0835999417d17a"), name: 'a5', age: 21 }
]
27017> db.fsyncUnlock()
{ info: 'fsyncUnlock completed', lockCount: Long("0"), ok: 1 }
27017> db.demo2.insertOne( { name: "a9", age: 22 } )
{
  acknowledged: true,
  insertedId: ObjectId("639deef166188cade4fef8a1")
}
27017> |
```

Hình 66

Lần này, em đã insert thành công

2.5 Concurrency Control trong InnoDB

2.5.1 Các khái niệm về autocommit, commit, rollback

Trong innodb, tất cả các hoạt động của người dùng dùng đều được đóng gói trong transaction. autocommit là một chế độ mà ở đó:

- Nếu người dùng bật autocommit thì mỗi câu lệnh sql thông thường nếu không bắt đầu bởi START TRANSACTION đều là một transaction, nếu có bất cứ sự thay đổi transaction nào thay đổi trên database, thì những người dùng khác lập tức nhìn thấy sự thay đổi này.
- Còn nếu người dùng tắt autocommit tùy vào mức transaction isolation level, thì câu lệnh vẫn được thực hiện nhưng chỉ trong phiên của người dùng đó mới thấy được sự thay đổi còn người dùng khác có thể không thấy sự thay đổi này. Để có thể áp dụng được sự thay đổi trên toàn database, người dùng phải dùng lệnh commit. Còn nếu có bất cứ sai sót nào trong lúc thực thi, người dùng phải dùng lệnh rollback để chỉ định database quay về trạng thái trước khi transaction gây ra lỗi được xử lý.

Mặc định của Innodb là autocommit luôn được bật, để đảm bảo chắc chắn trước khi thực hiện các transaction, ta nên tắt autocommit đi, một số lệnh thường được thao tác liên quan autocommit:

```
1 set autocommit = 0; //turn off autocommit
2 select @autocomit; //show mode autocommit
```

Ngoài ra, cả hai lệnh COMMIT và ROLLBACK đều giải phóng tất cả các khóa InnoDB đã được đặt trong giao dịch hiện tại. Và một trong hai lệnh này đều bắt buộc ở dưới cùng transaction nếu ta muốn áp dụng sự thay đổi “thật sự” lên database.

2.5.2 Các kiểu locking của innodb

2.5.2.a Intention Locks

Đối với mức row-level locking, innodb sử dụng 2 loại khóa exclusive (x) lock và shared lock (s). Bên cạnh đó để xử lý mức table-level locking, Innodb còn hỗ trợ thêm multiple granularity locking khi mà cho phép tồn tại cả row-level locks và table-level locks trong cùng một bảng. Để hiện thực hóa điều đó, InnoDB cũng sử dụng intention locks, mà ở đây intention lock này là một table-level lock với 2 dạng khóa chính cũng là exclusive và shared, nó sẽ cho biết được loại yêu cầu mà các transaction khác muốn sử dụng đối với một hàng trong bảng cụ thể 2 loại này như sau:

- Intention shared lock (IS) sẽ chỉ ra rằng có một transaction muốn hướng đến việc sử dụng shared lock trên một hàng của bảng. Câu lệnh để tạo ra IS lock đó là: SELECT ... FROM ... FOR SHARE.
- Intention exclusive lock (IX) sẽ chỉ ra rằng có một transaction muốn hướng đến việc sử dụng exclusive lock trên một hàng của bảng. Câu lệnh để tạo ra IS lock đó là: SELECT ... FROM ... FOR UPDATE.

Intention locking cũng phụ thuộc vào sơ đồ biểu diễn trạng thái của kỹ thuật đa độ mịn đã trình bày trước đó. Một khóa sẽ được gán cho một transaction đang yêu

câu một tài nguyên nào đó nếu nó tương thích với khoá đã tồn tại và bắt buộc không bị xảy ra xung đột với các loại khóa đã tồn tại khác. Nếu có tình trạng conflict xảy ra. Transaction sẽ chờ cho tới khi khóa gây ra xung đột được giải phóng, nếu khóa này không được giải phóng sẽ xảy ra tình trạng deadlock lúc này sẽ dễ dàng có lỗi xảy ra.

Trước khi transaction có thể được xác nhận các lock trên một hàng, chúng cần phải được được xác nhận intention lock trên bảng của hàng đó trước. Mục đích chính của intention lock là để cho biết được ai đó đang muốn khóa một hàng hoặc “sẽ” khóa một hàng của bảng. Ta sẽ demo locking với intention lock share và read. Các bước demo lần lượt thực hiện theo thứ tự sau đây. Để demo ta sẽ sử dụng bảng messages có các trường sau

```
1 CREATE TABLE messages(
2     id INT NOT NULL,
3         message VARCHAR(50) NOT NULL
4 );
```

- **LOCK TABLE messages READ:** Các transaction khác nếu có lock hàng trong table này để đọc đều sẽ tương thích và nhận được kết quả ngay lập tức, nếu lock hàng để UPDATE thì phải đợi.
- Transaction B thử SELECT * FROM messages FOR SHARE thì nhận được kết quả ngay lập tức. Nhưng đã có khóa nhận share nhận được.
- Transaction B thử SELECT * FROM messages FOR UPDATE thì phải đợi TABLE release lock, lúc này Transaction B sẽ nhận được khóa Intension lock X.
- Ta xem bảng khóa sau khi các hành động trên.

```
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> LOCK TABLE messages READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT ENGINE_TRANSACTION_ID as trx_id,
-> OBJECT_NAME as 'table',
-> INDEX_NAME,
-> LOCK_DATA,
-> LOCK_MODE,
-> LOCK_STATUS
-> FROM performance_schema.data_locks;
+-----+-----+-----+-----+-----+
| trx_id | table | INDEX_NAME | LOCK_DATA | LOCK_MODE | LOCK_STATUS |
+-----+-----+-----+-----+-----+
| 284109823678656 | messages | NULL | NULL | S | GRANTED |
+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Hình 67: lock messages và xem trạng thái khóa: SHARED_READ_ONLY đã được granted

```

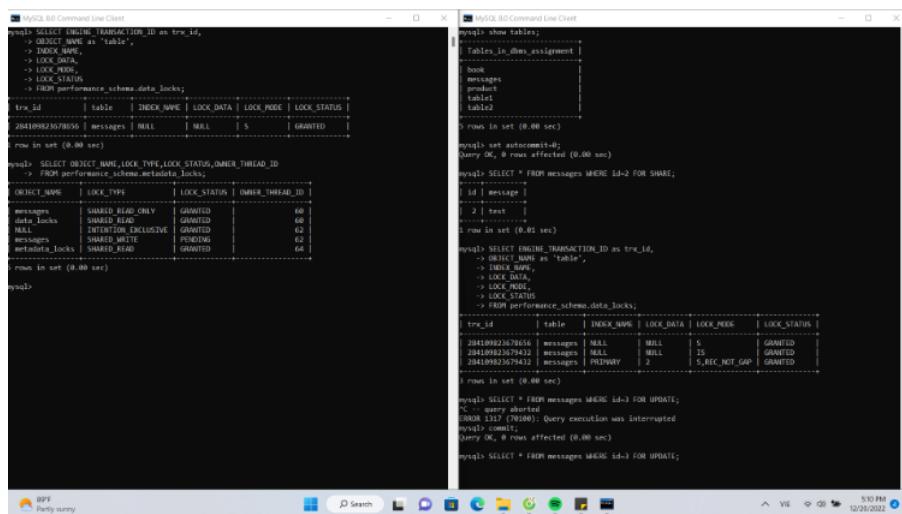
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM messages WHERE id=2 FOR SHARE;
+----+-----+
| id | message |
+----+-----+
| 2  | test   |
+----+-----+
1 row in set (0.01 sec)

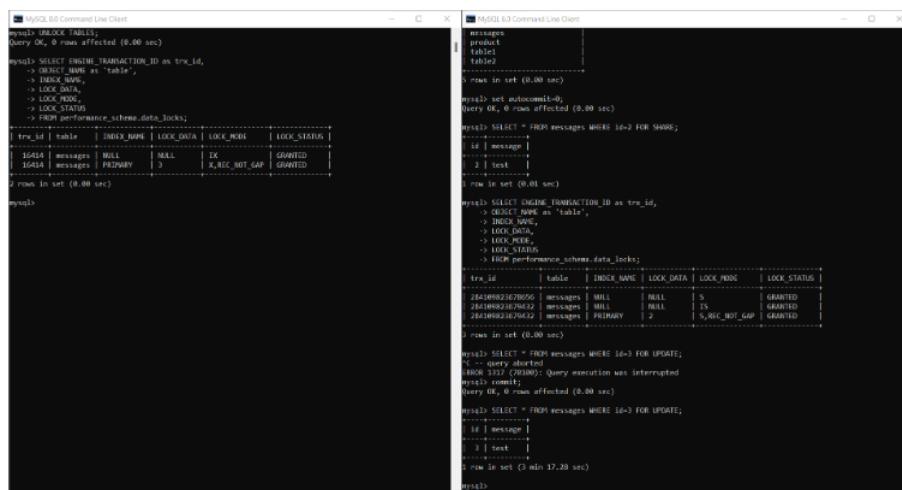
mysql> SELECT ENGINE_TRANSACTION_ID as trx_id,
-> OBJECT_NAME as 'table',
-> INDEX_NAME,
-> LOCK_DATA,
-> LOCK_MODE,
-> LOCK_STATUS
-> FROM performance_schema.data_locks;
+-----+-----+-----+-----+-----+-----+
| trx_id | table | INDEX_NAME | LOCK_DATA | LOCK_MODE | LOCK_STATUS |
+-----+-----+-----+-----+-----+-----+
| 284109823678656 | messages | NULL      | NULL     | S          | GRANTED    |
| 284109823679432 | messages | NULL      | NULL     | IS         | GRANTED    |
| 284109823679432 | messages | PRIMARY   | 2        | S,REC_NOT_GAP | GRANTED |
+-----+-----+-----+-----+-----+-----+

```

Hình 68: Ta sử dụng `SELECT * FROM messages FOR SHARE`, và nhận được khóa IS, khóa tương thích nên nhận ngay được kết quả



Hình 69: Ta commit khóa IS và thử dùng lệnh `FOR UPDATE`, lúc này transaction đang đợi khóa



Hình 70: unlock table thì transaction nhận được khóa X và IX

Dựa vào bảng ta có thể cấp thêm các khóa như REC_NOT_GAP và GAP nếu ta mở rộng phạm vi tìm kiếm.

2.5.2.b Record Locks

Khóa này xảy ra khi chúng ta thực hiện các tác vụ trên index của record tương ứng. Lấy ví dụ transaction 1 đang hiện thực lệnh `SELECT * FROM ... WHERE index = 1 FOR SHARE`, lúc này khi các transaction khác nếu đụng đến index 1 sẽ bị lock hết và chờ đến khi transaction 1 commit. Nếu bảng không có index, innodb sẽ tự tạo ra các hidden clustered index và dùng chúng để xử lý record locking. Nói một chút về clustered index, đây là một loại khóa đặc biệt được sử dụng để lưu trữ được tạo ra từ innodb nhằm tăng tốc độ tìm kiếm cũng như các tác vụ DML, nếu table được định nghĩa có PRIMARY KEY thì innodb sẽ sử dụng nó như là clustered index, nếu không được định nghĩa PRIMARY KEY thì innodb sẽ tìm kiếm cột đầu tiên có giá trị phân biệt làm clustered index, nếu không tồn tại cột như vậy thì innodb sẽ dựa vào thứ tự các row và gán chúng thành clustered index.

2.5.2.c Gap Locks

Khóa này xảy ra khi ta thực hiện các tác vụ tìm kiếm hoặc DML trên một đoạn trên index record hoặc phần trước hoặc sau index record đó. Để hình dung rõ ra, các dạng câu lệnh gây ra gap locks.

- `SELECT * FROM ... WHERE index BETWEEN 10 and 20 FOR UPDATE;` (1)
- `SELECT * FROM ... WHERE index >= 20 FOR SHARE;` (2)
- `SELECT * FROM ... WHERE index <= 10 FOR UPDATE;` (3)

Nói về công dụng của gap locks, lấy ví dụ khi transaction thực hiện câu lệnh (1), nó sẽ ngăn không cho bất cứ transaction nào khác có thể insert, update vào đoạn index từ 10 đến 20, “gap” này dù trong đó đã có đầy đủ dữ liệu hay chưa. “gap” này là một phần của sự đánh đổi giữa hiệu suất và tính đồng thời, đồng thời được sử dụng ở một số “transaction level” chứ không phải các cấp độ khác.

Gap lock sẽ không được sử dụng cho các transaction liên quan đến xử lý một hàng với unique index, nếu hàng đó không phải unique index thì gap lock sẽ được thực thi với khoảng cách từ đầu tới giá trị của index đó. Gap lock còn cho phép tồn tại hai khóa conflict với nhau ở hai transaction khác nhau ở trong cùng một “gap”. Bởi vì nếu có một record liên quan tới khóa nằm trong gap bị trực xuất, thì gap lock giữa các transaction khác nhau đều sẽ được hợp nhất.

Gap lock có thể không được thực hiện nếu ta thay đổi transaction isolation level được xác định là `READ_COMMITTED` hoặc là ta bật lên `innodb_locks_unsafe_for_binding`.

2.5.2.d Next-key Locks

Next-key locks là sự kết hợp của record lock trên trường index và gap lock trên đoạn “gap” phía trước index đó, đây cũng là loại khóa row-level. Lấy ví dụ nếu ta cần query một mẫu dữ liệu có index = 20, thì next-key locks sẽ được áp dụng trên đoạn $(a, 20]$ với a là giá trị index nhỏ hơn gần index 20 nhất, nếu a không tồn tại thì gap sẽ là toàn bộ dữ liệu trước 20. Vì thế Next-key locks có công dụng là trong cùng một bảng dữ liệu nếu có



một session đang có khóa S hoặc X trên một index R nào đó thì session khác sẽ không thể insert ngay tức thì vào đoạn gap trước R.

Mặc định của mức isolation innodb là REPEATABLE _ READ, vì thế vẫn sẽ bị ảnh hưởng bởi phantom row trong các tác vụ tìm kiếm dữ liệu , cho nên ta có thể bật next-key locks để ngăn chặn ảnh hưởng này.

2.5.2.e Insert Intention Locks

Đây là một lock thuộc gap lock được sinh ra cho các tác vụ liên quan đến thứ tự thêm một hàng, đây cũng là loại khóa row-level. Lock này được kích hoạt dựa theo nguyên lý, nếu transaction thêm vào cùng một gap index, chúng sẽ không phải đợi nhau nếu đã biết được rằng chúng hoàn toàn không liên quan đến vị trí insert với nhau. Những transaction đến trước sẽ nếu mà cần insert vào gap đang bị block chúng sẽ nhận được khóa này, các transaction đến sau khi insert nếu gặp khóa này sẽ phải đợi nếu không sẽ nhận được các khóa intention khác cho riêng index cần

2.5.2.f Auto-inc Locks

Là một khóa đặc biệt thuộc table-level. Khoá được kích hoạt khi chỉ khi transaction thêm vào bảng các thông tin liên quan đến cột mang tính chất AUTO _ INCREMENT. Với một trường hợp đơn giản, nếu một transaction đang chèn các giá trị vào bảng, thì bất kỳ transaction nào khác phải đợi để thực hiện thao tác chèn của riêng chúng vào bảng đó, sao cho các hàng được chèn bởi transaction đầu tiên nhận được các giá trị khóa chính liên tiếp.

Biến innodb_autoinc_lock_mode sẽ kiểm soát thuật toán được sử dụng cho việc auto_increment locking để tránh các trường hợp các cột mang tính chất AUTO _ INCREMENT sau khi thêm vào bảng bị sai thứ tự. Ta có thể điều khiển biến này để chọn giữa tính đúng đắn về thứ tự và tối ưu việc concurrency cho các tác vụ insert.

2.5.3 Transaction isolation level

Isolation level sẽ điều chỉnh mức độ cân bằng giữa performance và tính đúng đắn của kết quả khi mà nhiều transaction thay đổi cùng một dữ liệu vào cùng một thời điểm. Repeatable read là chế độ mặc định của innodb. Nhưng ngoài ra Innodb còn hỗ trợ thêm một số tùy chỉnh isolation level khác nữa chẳng hạn như: read uncommitted và read committed nhưng người dùng chỉ có thể thay đổi isolation trong session của riêng mình, các session khác sẽ không bị ảnh hưởng. Bạn có thể nói lỏng các quy tắc nhất quán với READ COMMITTED hoặc thậm chí READ UNCOMMITTED, trong các tình huống mà việc nhất quán hay lặp lại tính nhất quán ít quan trọng hơn việc giảm thiểu số lượng chi phí khóa. Serializable sẽ đảm bảo rằng một luật nghiêm ngặt Repeatable read khi nó hạn chế được phantom read hay đảm bảo giải quyết tốt deadlock nhưng thời gian và chi phí khóa sẽ lớn hơn. Các tác vụ consistence read và locking read sẽ được nhắc đến ở phần này, còn giải thích sâu hơn sẽ ở mục tiếp theo.

2.5.3.a Repeatable read

Với các tác vụ consistent read, tất cả các query trong cùng một transaction đều đọc snapshot chụp nhanh được thiết lập bởi query đầu tiên được đọc trong transaction đó. Ta có thể nhận được ảnh chụp nhanh mới hơn cho các query của mình bằng cách thực hiện giao dịch hiện tại và sau đó đưa ra các truy vấn mới. Với chế độ này với mỗi transaction, InnoDB sẽ tự động gửi cho transaction time point mà query của nó được xem xét trong database, với các câu lệnh insert, update, delete sau timepoint được thực hiện từ các transaction khác, transaction hiện tại sẽ không thấy bất cứ thay đổi gì.

Với các tác vụ liên quan đến việc locking reads (chẳng hạn như SELECT ..FOR UPDATE hay SELECT ..FOR SHARE), các câu lệnh UPDATE, DELETE được sử dụng, việc locking sẽ phụ thuộc vào phạm vi tìm kiếm chẳng hạn: tìm với chỉ mục duy nhất dựa theo một điều kiện duy nhất hoặc theo tìm kiếm theo khoảng với điều kiện tìm kiếm tương ứng (chẳng hạn SELECT .. WHERE id > 100)

- Với tìm kiếm các chỉ mục duy nhất theo điều kiện tìm kiếm duy nhất (chẳng hạn id là primary index, SELECT ... WHERE id = 10): innodb sẽ sử dụng record lock và không cần dùng gap lock với đoạn trước đó (không cần lo việc insert lúc này).
- Với các điều kiện tìm kiếm còn lại, InnoDB sẽ dùng gap lock hoặc next-key lock cho các tác vụ này vào đoạn gap được bao phủ bởi phạm vi tìm kiếm (điều này để giữ tính nhất quán tránh xuất hiện non repeatable read).

2.5.3.b Read committed

Với tác vụ consistent read, các query trong cùng một giao dịch lần lượt đọc snapshot mới nhất được tạo ra từ query thực hiện trước.

Còn với cách mà mức này xử lý ở locking reads, nó sẽ chỉ lock index record, không cần lock phần gap trước đó điều này vô tình cho phép có thể thêm vào các record mới kế bên index đã bị lock. Trong mức này gap lock chỉ được sử dụng với việc kiểm tra các thao tác liên qua khóa ngoại hay việc kiểm tra khóa bị trùng lặp. Do gap locking hầu hết sẽ bị vô hiệu nên sẽ xảy ra hiện tượng phantom read. Tuy nhiên khi gây ra bất lợi nhưng chúng vẫn có ít trong một số trường hợp:

- Với những câu lệnh UPDATE hoặc DELETE, InnoDB sẽ chỉ lock những row liên quan đến thao tác này, không cần quan tâm đến các row khác (các row không matching theo điều kiện WHERE), điều này sẽ hạn chế khả năng xảy ra của deadlock.
- Với những câu lệnh UPDATE, nếu hàng đã bị lock, InnoDB sẽ thực hiện việc kiểm tra trước bằng cách lấy về phiên bản commit mới nhất để xem liệu hàng đó có khớp với điều kiện WHERE hay không. Nếu không khớp hàng sẽ bị bỏ qua, nếu khớp transaction sẽ đợi đến lượt lock của mình. Như vậy có thể tiết kiệm thêm một chút thời gian.

Read committed có thể được set khi mới khởi tạo hoặc trong thời gian động, mức cũng có thể set ảnh hưởng đến toàn bộ session hoặc chỉ riêng mình đó.

2.5.3.c Read uncommitted

Đây là mức giúp hệ thống xử lý nhanh nhất khi đây thuộc kiểu nonlocking, sử dụng mức này sẽ khiến cho việc đọc không được nhất quán gây ra hiện tượng DIRTY READ, còn lại nó sẽ hoạt động giống read committed.

2.5.3.d Serializable

Mức này có cơ chế tương tự như REPEATABLE READ, nhưng innodb sẽ chủ động chuyển các câu lệnh SELECT về thành SELECT ... FOR SHARE nếu autocommit bị tắt, điều này sẽ giúp cho kết quả được nhất quán và chính xác nhưng sẽ mất nhiều thời gian để đợi hơn. Nếu autocommit được bật thì việc chuyển đổi này sẽ không được thực hiện. Nếu ta kích hoạt mức này, chúng ta sẽ không cần phải khóa nhưng vẫn đảm bảo được nhất quán trong kết quả của việc đọc.

2.5.3.e Demo về isolation level giải quyết các vấn đề trong concurrency control

Cả 4 mức đã đề cập đều giải quyết được lost update. Để demo ta sử dụng bảng book có các trường sau

```
1 CREATE TABLE book(
2     b_id INT PRIMARY KEY,
3     name VARCHAR(50) NULL,
4     author VARCHAR(120) NULL,
5     category_id INT NULL
6 );
```

Đối với mức Read Uncommited mức này sẽ bị ảnh hưởng bởi các vấn đề còn lại, lấy ví dụ demo về dirty read.

```
1 #userA
2 SET autocommit = 0;
3
4 START TRANSACTION;
5 UPDATE book SET name = 'expired' WHERE b_id=2;
6 DO SLEEP(10);
7 ROLLBACK;
8 SELECT * FROM book;
9
10 #userB
11 SET autocommit = 0;
12
13 SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
14 SELECT * FROM book WHERE b_id=2;
15
```

The screenshot shows two MySQL command-line clients side-by-side. Both clients are running the same SQL script: `source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userA.sql`. The left client is connected as userA, and the right client is connected as userB.

UserA (Left Client):

```

mysql> source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userA.sql
Database changed
Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Query OK, 0 rows affected (0.01 sec)
Query OK, 10 rows affected (0.01 sec)
Rows matched: 10  Changed: 10  Warnings: 0

+ b_id | name   | author | category_id |
| 1    | MySQL  | HP1   | 1           |
| 2    | mongDB | HS    | 2           |
| 3    | couchDB | HS   | 2           |
| 4    | redis   | HS    | 3           |
| 5    | noSQL   | HS    | 2           |
| 6    | SQL Server | HS  | 1           |
| 7    | xDB    | HP2   | 3           |
+----+
7 rows in set (0.00 sec)

mysql>

```

UserB (Right Client):

```

mysql> source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userB.sql
Database changed
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

+ b_id | name   | author | category_id |
| 2    | mongDB | HS    | 2           |
+----+
1 row in set (0.00 sec)

mysql>

```

Hình 71: transaction B đọc giá trị ở $b_id = 2$ với name không hề tồn tại.

Đối với mức Read committed: Nó khắc phục được dirty read nhưng lại bị ảnh hưởng bởi unrepeatable read và phantom read. Để khắc phục dirty read của mức trên ta chỉnh level lại ở transaction B như sau:

```

1 SET autocommit = 0;
2
3 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
4 SELECT * FROM book WHERE b_id=2;

```

Lúc này kết quả mà ta nhận được sẽ được nhất quán

Ta sẽ thiết lập 2 transaction để trường hợp unrepeatable read xảy ra với mức này.

The screenshot shows two MySQL command-line clients side-by-side. Both clients are running the same SQL script: `source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userA.sql`. The left client is connected as userA, and the right client is connected as userB.

UserA (Left Client):

```

mysql> source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userA.sql
Database changed
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

Query OK, 0 rows affected (0.01 sec)
Query OK, 10 rows affected (0.01 sec)
Rows matched: 10  Changed: 10  Warnings: 0

+ b_id | name   | author | category_id |
| 1    | MySQL  | HP1   | 1           |
| 2    | mongDB | HS    | 2           |
| 3    | couchDB | HS   | 2           |
| 4    | redis   | HS    | 3           |
| 5    | noSQL   | HS    | 2           |
| 6    | SQL Server | HS  | 1           |
| 7    | xDB    | HP2   | 3           |
+----+
7 rows in set (0.00 sec)

mysql>

```

UserB (Right Client):

```

mysql> source C:/Users/LENOVO/OneDrive/Desktop/dbs/innodb/isolation_level/read_uncommitted/userB.sql
Database changed
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

+ b_id | name   | author | category_id |
| 2    | mongDB | HS    | 2           |
+----+
1 row in set (0.00 sec)

mysql>

```

Hình 72: userB đã nhận được kết quả chính xác

```

1 #userA
2 SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

```

```

3
4     START TRANSACTION;
5     SELECT * FROM book;
6     DO SLEEP(10);
7     SELECT * FROM book;
8     COMMIT;
9
10    #userB
11    SET autocommit = 0;
12
13    START TRANSACTION;
14    UPDATE book SET name="Test change" WHERE b_id=2;
15    SELECT * FROM book;
16    COMMIT;

```

b_id	name	author	category_id
1	MySQL	H1	1
2	MongoDB	H1	2
3	couchDB	H2	2
4	oracle	H3	3
5	Redis	H3	3
6	noSQL	H4	2
7	SQL_Server	H5	1

b_id	name	author	category_id
1	MySQL	H1	1
2	Test change	H1	2
3	couchDB	H2	2
4	oracle	H3	3
5	noSQL	H4	2
6	SQL_Server	H5	1
7	Redis	H2	3

Hình 73: userA hai lần đọc kết quả nhận được sự thay đổi khác nhau.

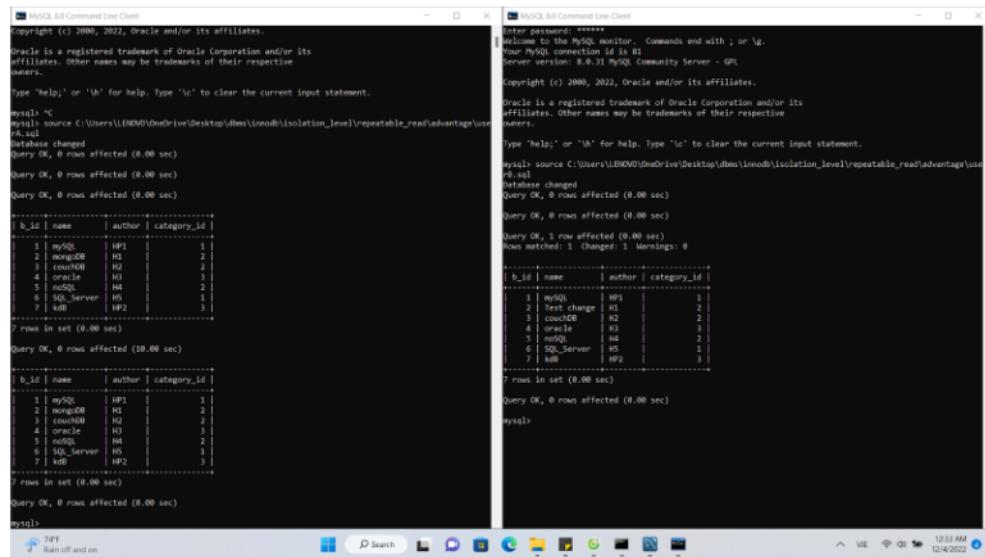
Đối với mức Repeatable Read nó hạn chế được điểm yếu về unrepeatable read nhưng lại bị vấn đề phantom read. Đối với hạn chế điểm yếu, nhóm chỉnh level của transaction A lại là REPEATABLE READ, và đã khắc phục được lỗi demo bên trên.

Ta sẽ tạo ra 2 transaction khác để mô phỏng hiện tượng phantom read.

```

1    #userA
2    =====
3    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
4
5    START TRANSACTION;
6    SELECT * FROM book;
7    DO SLEEP(10);
8    SELECT * FROM book;
9    COMMIT;
10   =====
11
12   #userB
13   =====
14   SET autocommit = 0;
15
16   START TRANSACTION;

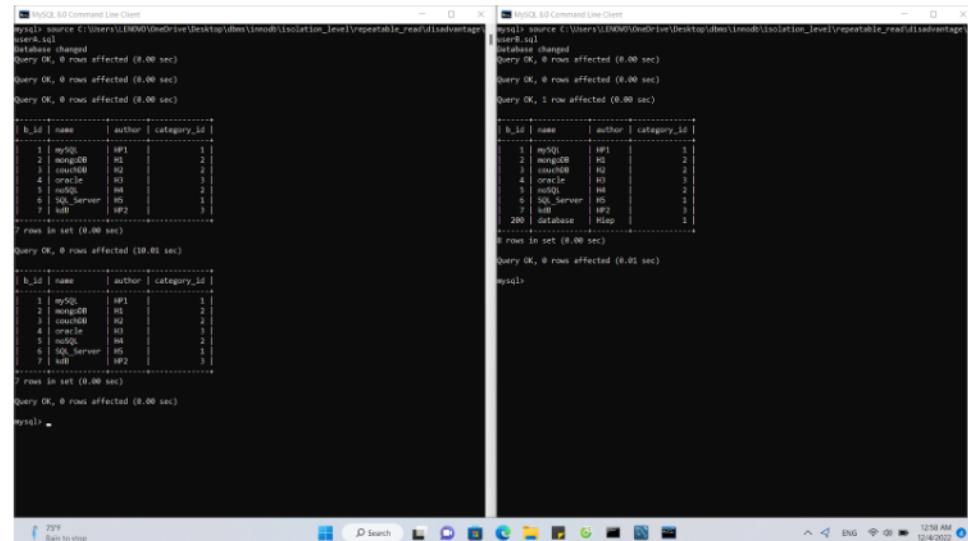
```



Hình 74: userA hai lần đọc kết quả nhận được sự thay đổi khác nhau.

```

17 INSERT INTO book VALUES(200, "database", "Hiep", 1);
18 SELECT * FROM book;
19 COMMIT;
20 #=====
  
```



Hình 75: userA hai lần đọc kết quả nhận được sự xuất hiện của một hàng mới

Mức cuối cùng là Serializable: Nó khắc phục được hết các hiện tượng như dirty read, unrepeatable read, phantom read nhưng đánh đổi đó là thời gian chờ khóa khá lâu dẫn đến hiệu năng hệ thống bị chậm. Ta khắc phục demo ở phía trên bằng cách chỉnh lại level của transaction A thành Serializable.

The screenshot shows two side-by-side MySQL Command Line Client windows. Both windows have identical command-line histories and display the same data from a table named 'books'.

```

mysql> source C:\Users\LENOVO\OneDrive\Desktop\dmcs\innoDB\isolation_level\Serializable\advantage\userB.sql
Database changed
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)
Query OK, 0 rows affected (0.00 sec)

+----+ name + author + category_id +
| 1 | mySQL | HP1   | 1      |
| 2 | mongoDB | HS    | 2      |
| 3 | cassDB | HS    | 2      |
| 4 | oracle | HS    | 3      |
| 5 | nosQL | HS    | 2      |
| 6 | sqlServer | HS  | 1      |
| 7 | kdb | MP2   | 3      |
+----+
7 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

+----+ name + author + category_id +
| 1 | mySQL | HP1   | 1      |
| 2 | mongoDB | HS    | 2      |
| 3 | cassDB | HS    | 2      |
| 4 | oracle | HS    | 3      |
| 5 | nosQL | HS    | 2      |
| 6 | sqlServer | HS  | 1      |
| 7 | kdb | MP2   | 3      |
| 200 | database | Hiep | 3      |
+----+
8 rows in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

mysql>
  
```

Hình 76: Không còn bị ảnh hưởng bởi phantom read

Isolation level	Dirty read	Nonrepeatable read	Phantom
Read uncommitted	yes	yes	yes
Read committed	no	yes	yes
Repeatable read	no	no	yes
Serializable	no	no	no

Hình 77: So sánh giữa các mức isolation với các vấn đề mà nó khắc phục được

2.5.4 Consistent read

Có ý nghĩa là InnoDB sẽ sử dụng multi-version để có thể query trên một snapshot của cơ sở dữ liệu trong một thời điểm nào đó. Ta sẽ gọi thời điểm này là t, câu lệnh query sẽ chỉ nhìn thấy những thay đổi bởi transaction đã commit trước thời điểm t, như vậy transaction mới thực hiện lệnh query này sẽ không chịu bất cứ tác động nào từ transaction commit sau thời điểm t hoặc transaction chưa commit. Ngoại lệ duy nhất ở đây tức là câu truy vấn sẽ nhìn thấy được sự thay đổi bởi những câu lệnh đã thực hiện trước trong cùng một transaction, điều này có thể gây ra việc các hàng trước và sau khi update đều sẽ cùng tồn tại, sẽ gây ra trạng thái chưa từng có trong database.

Consistent read là chế độ mặc định trong các tiến trình liên quan đến việc sử dụng câu query SELECT trong cả hai isolation là read committed và read repeatable. Khi sử dụng chế độ này sẽ không dùng lock để khóa bất cứ table nào mà nó sử dụng và do đó, các phiên khác có thể tự do sửa đổi các bảng đó cùng lúc với việc đọc nhất quán đang được thực hiện trên bảng.

Nhưng ta có một lưu ý snapshot chỉ áp dụng cho các câu lệnh SELECT trong transaction và không áp dụng cho các câu lệnh DML, vì thế sẽ xảy ra trường hợp:



- Một transaction cập nhật hoặc xóa các hàng được commit bởi một transaction khác, thì những thay đổi đó sẽ hiển thị đối với transaction hiện tại. Trong ví dụ trên ta có thể thấy unrepeatable read xảy ra

```
1   SELECT COUNT(c2) FROM t1 WHERE c2 = 'abc';
2   -- Returns 0: no rows match.
3   UPDATE t1 SET c2 = 'cba' WHERE c2 = 'abc';
4   -- Affects 10 rows: another txn just committed 10 rows with 'abc' values.
5   SELECT COUNT(c2) FROM t1 WHERE c2 = 'cba';
6   -- Returns 10: this txn can now see the rows it just updated.
7
```

- Một transaction với việc thêm hoặc tự chỉnh sửa các thông tin trong nó sau đó commit, một session khác theo mức REPEATABLE READ sẽ ảnh hưởng đến transaction trong session đó dù không đúng đến bất cứ thông tin nào trong đây. Ví dụ này sẽ nói lên là hiện tại thông tin c1 đã bị xóa hết bởi transaction khác mặc dù transaction hiện tại chưa làm gì.

```
1   SELECT COUNT(c1) FROM t1 WHERE c1 = 'xyz';
2   -- Returns 0: no rows match.
3   DELETE FROM t1 WHERE c1 = 'xyz';
4   -- Deletes several rows recently committed by other transaction.
5
```

Có một cách khắc phục hiện tượng này đó là khi commit transaction thì thực hiện thêm START TRANSACTION WITH CONSISTENT SNAPSHOT. để bật chế độ multi-version concurrency control (MVCC).

Demo ở dưới đây sẽ thể hiện điều trên, với thứ tự commit của transaction B được thực hiện trước, và transaction A không thấy điều này trong quá trình thực hiện. Đến khi commit mới thấy được sự thay đổi. Code để demo transaction.

```
1 #transaction A
2 set autocommit = 0;
3 START TRANSACTION WITH CONSISTENT SNAPSHOT;
4 SELECT * FROM t;
5 SELECT * FROM t;
6 commit;
7 SELECT * FROM t;
8
9 #transaction B
10 SET autocommit=0;
11 INSERT INTO t VALUES (1, 2);
12 COMMIT;
```

The screenshot shows two separate MySQL command-line sessions running simultaneously. Both sessions are connected to the same database and show the same sequence of SQL commands being executed.

Session 1 (Left):

```
mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION WITH CONSISTENT SNAPSHOT
-> ;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t;
+----+----+
| id | val |
+----+----+
| 1  | 2  |
+----+----+
1 row in set (0.00 sec)

mysql>
```

Session 2 (Right):

```
mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t VALUES (1, 2);
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql>
```

Hình 78: *t* trong quá trình chưa commit bên A vẫn giữ được sự nhất quán.

Consistent Read không hoạt động trên các câu lệnh DDL điển hình với ví dụ DROP TABLE, MySQL sẽ không sử dụng bản đã xóa rồi và InnoDB sẽ hủy luôn nó, vì thế ta không thể tạo ra snapshot.

2.5.5 Locking-reads

Chế độ này chỉ được kích hoạt nếu autocommit bị tắt. Nếu bạn truy vấn dữ liệu và sau đó chèn hoặc cập nhật dữ liệu liên quan trong cùng một transaction, câu lệnh SELECT thông thường sẽ không cung cấp đủ khả năng bảo vệ. Các transaction khác có thể cập nhật hoặc xóa các hàng mà bạn vừa mới truy vấn. InnoDB hỗ trợ hai loại khóa đọc mang lại sự an toàn hơn đó là:

- **SELECT ... FOR SHARE:** Đặt khóa chế độ share trên bất kỳ hàng nào được đọc. Các session khác có thể đọc các hàng nhưng không thể sửa đổi chúng cho đến khi transaction của đang giữ khóa được thực hiện. Nếu bất kỳ hàng nào trong số này bị thay đổi bởi một transaction khác chưa được commit, thì query sẽ đợi cho đến khi giao dịch đó kết thúc rồi sử dụng các giá trị mới nhất.
- **SELECT ... FOR UPDATE:** Đối với index record xuất hiện trong phạm vi khi tìm kiếm, innodb sẽ lock hàng tương ứng với index record đó, điều này tương tự cho câu lệnh UPDATE. Các transaction khác bị chặn không cho phép update các hàng đó, nếu yêu cầu cho FOR SHARE thì phải đợi.

Tất cả các khóa được đặt bởi các query FOR SHARE và FOR UPDATE được giải phóng khi transaction được commit hoặc rollback. Bên cạnh đó, ta có thể thực hiện locking với subquery thông qua các cách sau:



- Chỉ thực hiện locking read trên t1

```
1   SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2) FOR UPDATE;
2
```

- Thực hiện locking read trên t1 và t2

```
1   SELECT * FROM t1 WHERE c1 = (SELECT c1 FROM t2 FOR UPDATE) FOR UPDATE;
2
```

2.5.6 Locking reads với NOWAIT và SKIP LOCKED

Xét trường hợp hàng đã bị lock bởi transaction, câu lệnh “SELECT ... FOR UPDATE” hoặc “SELECT ... FOR SHARE” cùng yêu cầu lock hàng đó, thì phải đợi cho đến khi transaction blocking hủy bỏ khóa hàng đó. Hành vi này ngăn các transaction cập nhật hoặc xóa các hàng được yêu cầu cập nhật bởi các giao dịch khác. Tuy nhiên việc chờ các khóa hàng bị hủy bỏ là điều không cần thiết nếu bạn muốn query trả về kết quả ngay lập tức khi mà hàng được yêu cầu đã bị khóa, hoặc bạn chấp nhận được kết quả trả về sẽ bao gồm việc loại trừ các hàng bị khóa khỏi tập hợp.

Để thực hiện hóa điều trên, InnoDB hỗ trợ thêm 2 tùy chọn NOWAIT và SKIP LOCKED được sử dụng dùng câu lệnh SELECT ... FOR UPDATE hoặc SELECT ... FOR SHARE. 2 tùy chọn này chỉ áp dụng trên row-level locking.

- **Với tùy chọn NOWAIT:** việc locking read sử dụng chỉ thị này sẽ không bao giờ chờ nhận được quyền khóa hàng mà thay vào đó câu query sẽ thực hiện ngay tức khắc, sẽ báo lỗi nếu hàng đã bị locked.
- **Với tùy chọn SKIP LOCKED:** locking read sẽ không chờ khóa, query sẽ thực thi ngay lập tức bằng cách loại bỏ các hàng đang bị khóa bởi tập kết quả. Các query bỏ qua các hàng bị khóa gây ra dạng xem dữ liệu không nhất quán. SKIP LOCKED do đó không phù hợp với công việc giao dịch nói chung. Tuy nhiên, nó có thể được sử dụng để tránh tranh chấp khóa khi nhiều session truy cập vào cùng một bảng tạo thành một hàng đợi.

Demo về sử dụng 2 option này : với một session tạo ra bảng thử nghiệm, 2 session còn lại một chọn option NOWAIT, một chọn option SKIP LOCKED

```
1 #session 1:
2 set autocommit=0;
3 CREATE TABLE t (i INT, PRIMARY KEY (i)) ENGINE = InnoDB;
4 commit;
5 START TRANSACTION;
6 SELECT * FROM t WHERE i = 2 FOR UPDATE;
7
8 #session 2:
9 set autocommit=0;
10 START TRANSACTION;
11 SELECT * FROM t WHERE i = 2 FOR UPDATE NOWAIT;
12
13 #session 3:
```

```
14    set autocommit=0;
15    START TRANSACTION;
16    SELECT * FROM t FOR UPDATE SKIP LOCKED;
```

```
mysql> INSERT INTO t (i) VALUES(1),(2),(3);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> commit;
Query OK, 0 rows affected (0.01 sec)

mysql> select * from tables t;
ERROR 1146 (42S02): Table 'dbms_assignment.tables' doesn't exist
mysql> select * from t;
+---+
| i |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows in set (0.00 sec)

mysql> Start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE;
+---+
| i |
+---+
| 2 |
+---+
1 row in set (0.00 sec)

mysql>
```

Hình 79: session 1 tạo bảng và lock i = 2

```
mysql> use dbms_assignment;
Database changed
mysql> set autocommit = 0;
Query OK, 0 rows affected (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t WHERE i = 2 FOR UPDATE NOWAIT;
ERROR 3572 (HY000): Statement aborted because lock(s) could not be acquired immediately and NOWAIT is set.
mysql>
```

Hình 80: session 2 sử dụng option nowait thì ra lỗi vì i = 2 đang bị lock

```
mysql> use dbms_assignment;
Database changed
mysql> set autocommit=0;
Query OK, 0 rows affected (0.00 sec)

mysql> start transaction;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t FOR UPDATE SKIP LOCKED;
+---+
| i |
+---+
| 1 |
| 3 |
+---+
2 rows in set (0.00 sec)

mysql> +---
```

Hình 81: session 3 sử dụng option skip locked bỏ qua i = 2 để trả về kết quả



2.5.7 Hạn chế deadlock bởi innodb

Deadlock có thể xảy ra khi các transaction khóa các hàng trong nhiều bảng (thông qua các câu lệnh như UPDATE hoặc SELECT FOR ... UPDATE), nhưng theo thứ tự ngược lại. Deadlock cũng có thể xảy ra khi các câu lệnh như vậy khóa phạm vi bản ghi chỉ mục và gap. Nếu phần LATEST DETECTED DEADLOCK của monitor OUTPUT trả về có đoạn tin nhắn sau: “TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH, WE WILL ROLL ROLLBACK FOLLOWING TRANSACTION” điều này có nghĩa là số lượng transaction chờ đã đạt tới giới hạn 200, innodb cũng coi điều này là deadlock và sẽ khôi phục các giao dịch trong danh sách giờ, điều này cũng coi là 1 deadlock khi xảy ra hơn 1000000 khóa thuộc sở hữu các transaction trong danh sách chờ.

Trong innodb db có hỗ trợ cơ chế tìm kiếm deadlock khi deadlock detection được bật, InnoDB tự động phát hiện các deadlock và rollback một hoặc nhiều transaction để phá vỡ bế deadlock. InnoDB cố gắng chọn các transaction với kích thước nhỏ hoặc phạm vi tìm kiếm nhỏ để rollback, trong đó kích thước của transaction được xác định bởi số lượng hàng được chèn, cập nhật hoặc xóa.

Innodb sẽ nhận thức được khóa bảng nếu innodb_table_locks = 1 (mặc định) và autocommit = 0. Bằng không thì, InnoDB không thể phát hiện được deadlock nếu mà table đã bị lock bởi câu lệnh MySQL LOCK TABLES hoặc lock bởi storage engine khác mà Innodb có liên quan đến. Có một cách để giải quyết tình huống này bằng cách đặt biến hệ thống innodb_lock_wait_timeout, nếu các transaction đợi lâu hơn khoảng thời gian này thì sẽ tự động rollback.

Trong một hệ thống đòi hỏi tính concurrency, deadlock detection đôi khi sẽ rất mất thời gian và sẽ làm chậm hệ thống khi nhiều thread chờ cùng một khóa. Ta có thể tắt deadlock detection thông qua biến innodb_deadlock_detect, và set thời gian chờ thông qua biến đã nói ở ngay phía trên. Ở ví dụ về deadlock ta đã tắt đi innodb_deadlock_detect và kết quả là sau 50s (mặc định default timeout) transaction A đã bị rollback (do A đến trước), vì thế giờ ta sẽ bật innodb_deadlock_detect và transaction A đã được thực thi thành công và transaction B đã bị rollback (do B chứa nhiều thao tác DML dễ gây ra deadlock hơn)



The screenshot shows two separate MySQL command-line sessions running simultaneously. Both sessions are executing the same SQL script, which includes a source command for 'transaction1.sql' and 'transaction2.sql'. The left session shows a deadlock occurring, with the error message 'ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction'. The right session also shows the same sequence of queries and ends with a successful update. This demonstrates that deadlock detection is enabled by default.

Hình 82: khi tắt innodb_deadlock_detect

The screenshot shows two MySQL command-line sessions. In both sessions, the 'innodb_deadlock_detect' variable is explicitly set to 0 using the 'SET GLOBAL' command. Both sessions then execute the same SQL script from 'transaction1.sql' and 'transaction2.sql'. The right session successfully completes all operations, while the left session fails due to a deadlock, as indicated by the error message 'ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction'. This illustrates that deadlock detection is disabled when this variable is set to 0.

Hình 83: bật innodb_deadlock_detect và kết quả phản hồi khá nhanh

2.6 So sánh giữa MongoDB và InnoDB

2.6.1 MVCC

MongoDB có hỗ trợ MVCC cho những transaction ở cấp document.

Mục đích của việc hỗ trợ đa phiên bản trong MongoDB là để hỗ trợ tính atomic cho những tác vụ đọc/ghi liên quan đến nhiều documents khác nhau (ở 1 hoặc nhiều collection khác nhau). Vì tính chất phân tán, transaction có thể được dùng ở nhiều collections, databases, documents và cả ở các shard khác nhau nên đa phiên bản là rất cần thiết trong MongoDB.

InnoDB có hỗ trợ MVCC nhưng trong các tác vụ yêu cầu consistent-read, ta cần bật chế độ này lên thông qua câu lệnh: START TRANSACTION WITH CONSISTENT SNAPSHOT; nếu không InnoDB chỉ đọc dựa vào snapshot thông qua Multi-version mà

không có sự đồng bộ giữa các snapshot.

2.6.2 Multiple Granularity Locking

Giống nhau: đều phân mức dữ liệu thành các cấp và sử dụng chung cơ chế intention lock và ngoài ra không hỗ trợ thêm SIX lock (Shared intention exclusive).

Khác nhau về level và cách chia độ mịn:

- MongoDB hỗ trợ đa độ mịn ở 3 cấp là cấp global, database và collection, có hỗ trợ ở cấp document trong một vài trường hợp tùy chỉnh bởi WiredTiger. Vì MongoDB hỗ trợ mạnh cho mỗi trường phân tán nên cần thêm mức database để phân tán những dữ liệu liên quan với nhau ra nhiều database.
- Innodb Chỉ hỗ trợ ở 2 cấp là row và table tương ứng cho (document và collection trong mongodb), Innodb không hỗ trợ mỗi trường phân tán nên không cần tập trung chú ý nhiều vào lock database.

Vậy tùy vào thiết kế database, thiết kế theo hướng phân tán thì nên sử dụng mongoDB, còn thiết kế theo hướng tập trung thì nên sử dụng innodb.

2.6.3 Locking technique

2.6.3.a Queue locking:

Các transaction trong MongoDB sẽ có hàng đợi cấp khóa riêng biệt, hoạt động theo nguyên tắc FIFO (First-in First-out). Tuy nhiên, hàng đợi này hoạt động linh hoạt hơn ở chỗ khi cấp khóa, nó sẽ cấp 1 lúc nhiều khóa đọc hoặc khóa dự định đọc (S và IS) cho nhiều thao tác cần trong hàng đợi, giúp tăng cường hiệu năng trong quy trình xử lý khóa.

Để demo, đầu tiên em dùng lệnh db.fsyncLock() để khóa toàn bộ instance, sau đó dùng lệnh insertOne để thêm vào 1 collection ở instance đó.

```
mongosh mongodb://127.0.0.1:27017> db.fsyncLock()
{
  "op": "lock",
  "mode": "full"
}
27017> db.insertOne({ _id: 1 })
{
  "op": "insertOne",
  "ns": "test.collection"
}
27017> db.fsyncUnlock()
{
  "op": "unlock"
}
27017>
```

Hình 84: Trạng thái hàng đợi khóa khi khóa toàn bộ Instance và dùng lệnh insertOne

Sau khi dùng lệnh db.adminCommand({lockInfo:1}) để quan sát trạng thái khóa của instance, ta thấy hàng đợi khóa tăng lên. Tiếp đến, em unlock instance bằng lệnh db.fsyncUnlock(), lúc này các khóa trong hàng đợi lần lượt được cấp, lệnh insertOne được hiện thực và hàng đợi khóa cũng như thông tin khóa của instance trở nên rỗng.

```
27017> db.adminCommand({lockInfo:1})
{ lockInfo: [], ok: 1 }
27017>
```

Hình 85: Trạng thái hàng đợi khóa sau khi Unlock Instance

Lúc này, hàng đợi khóa đang có một khóa ghi (do em sử dụng lệnh InsertOne), khóa ghi này đang đợi được cấp vì hiện giờ em đã khóa toàn bộ Instance. Sau khi unlock instance bằng lệnh db.fsyncUnlock(), lệnh insertOne được thực thi và hàng đợi khóa quay trở về trạng thái rỗng.

Các transaction trong InnoDB sẽ sử dụng thuật toán CATS (Contention-Aware Transaction Scheduling) để ưu tiên các giao dịch đang chờ khóa. Khi nhiều giao dịch đang chờ khóa trên cùng một đối tượng, thuật toán CATS sẽ xác định giao dịch nào nhận được khóa trước. Ở đây không hỗ trợ lock-request như phía MongoDB. CATS sẽ không cho phép transaction thực hiện tuần tự theo thời gian nó đến mà còn dựa vào weight được gán cho mỗi transaction (weight ở đây được tính dựa vào nội tại của transaction có thể kể đến như, thời gian ước tính xử lý transaction, số lượng table khác được dùng,...), nếu 2 transaction cùng weight thì sử dụng thời gian nó đợi lock, transaction nào lâu hơn sẽ nhận được lock trước. Bên cạnh đó cũng hỗ trợ thêm mode FIFO hoặc auto (tự thay đổi tùy vào điều kiện thực thi), nhiều bài báo công bố hiệu suất của CATS tốt hơn nhiều so với FIFO.

```
mysql> SELECT OBJECT_NAME,LOCK_TYPE,LOCK_STATUS,OWNER_THREAD_ID
   -> FROM performance_schema.metadata_locks
   -> WHERE OBJECT_SCHEMA='dbms_assignment' AND OBJECT_TYPE='TABLE';
+-----+-----+-----+-----+
| OBJECT_NAME | LOCK_TYPE      | LOCK_STATUS | OWNER_THREAD_ID |
+-----+-----+-----+-----+
| messages    | SHARED_READ_ONLY | GRANTED    |          76      |
| messages    | SHARED_WRITE     | PENDING    |          77      |
| messages    | SHARED_WRITE     | PENDING    |          78      |
| messages    | SHARED_WRITE     | PENDING    |          80      |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT OBJECT_NAME,LOCK_TYPE,LOCK_STATUS,OWNER_THREAD_ID
   -> FROM performance_schema.metadata_locks
   -> WHERE OBJECT_SCHEMA='dbms_assignment' AND OBJECT_TYPE='TABLE';
+-----+-----+-----+-----+
| OBJECT_NAME | LOCK_TYPE      | LOCK_STATUS | OWNER_THREAD_ID |
+-----+-----+-----+-----+
| messages    | SHARED_WRITE    | GRANTED    |          77      |
| messages    | SHARED_WRITE    | GRANTED    |          80      |
+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Hình 86: dù thread_id = 78 đến sau nhưng vẫn thực hiện trước thread_id = 77

2.6.3.b Release lock process:

Đối với MongoDB những tác vụ làm ảnh hưởng đến nhiều documents khác (VD như update() nhiều dữ liệu) thì khóa của thao tác này sẽ bị MongoDB thu hồi một cách định kỳ (thu hồi trong 1 khoảng thời gian sau đó trả lại) để các thao tác đọc khác có thể được thực hiện trong quá trình thao tác khác được diễn ra.

Còn đối với Innodb, khóa chỉ được thu hồi nếu ở cuối mỗi transaction ta chỉ định lệnh commit hoặc rollback hoặc các trường hợp có lỗi xảy ra mà dbms phát hiện được như trường hợp deadlock, không có việc chuyển tiếp trao khóa định kỳ ở đây.

2.6.3.c Type lock:

Giống nhau: đều sử dụng X và S lock cũng như cơ chế intention lock.

Khác nhau:

- MongoDB có Instance-Wide Lock, có thể lock 1 lúc nhiều database trong 1 instance, sử dụng thường xuyên khi 1 thao tác tuy ngắn nhưng thực thi liên quan đến nhiều database.

```
{  
    type: 'op',  
    host: 'DESKTOP-IVPPMP4:27017',  
    desc: 'fsyncLockWorker',  
    active: true,  
    currentOpTime: '2022-12-20T17:06:20.439+07:00',  
    opid: 1694751,  
    op: 'none',  
    ns: '',  
    command: {},  
    numYields: 0,  
    locks: {  
        ParallelBatchWriterMode: 'r',  
        FeatureCompatibilityVersion: 'r',  
        ReplicationStateTransition: 'w',  
        Global: 'R'  
    },  
    waitingForLock: false,  
    lockStats: {  
        ParallelBatchWriterMode: { acquireCount: { r: Long("1") } },  
        FeatureCompatibilityVersion: { acquireCount: { r: Long("1") } },  
        ReplicationStateTransition: { acquireCount: { w: Long("1") } },  
        Global: { acquireCount: { R: Long("1") } }  
    },  
    waitingForFlowControl: false,  
    flowControlStats: {}  
},
```

Hình 87: Thông tin khóa của Instance sau khi dùng Instance-Wide Lock

- InnoDB có hỗ trợ thêm record-lock và gap-lock để bảo vệ dữ liệu đang được truy xuất khỏi hiện tượng race-condition, bên cạnh đó còn có sự xuất hiện của insert intention lock, next-key lock, auto-increment lock để tăng thêm độ an toàn tránh hiện tượng phantom read hay thiếu nhất quán khi đọc dữ liệu. Các khóa này tùy trường hợp sẽ tự động xuất hiện kèm với các khóa X và S. Đổi lại độ an toàn là thời gian thực hiện transaction sẽ bị kéo dài làm giảm performance của hệ thống. Ta có thể tùy chỉnh tắt các khóa tự động này đi bằng cách chỉ thị hoặc đổi traslation_isolation.

```

+-----+
| OBJECT_NAME | LOCK_TYPE      | LOCK_STATUS | OWNER_THREAD_ID |
+-----+
| messages    | SHARED_READ_ONLY | GRANTED     | 82          |
| messages    | SHARED_READ     | GRANTED     | 83          |
| NULL        | INTENTION_EXCLUSIVE | GRANTED     | 84          |
| messages    | SHARED_WRITE    | PENDING      | 84          |
| data_locks   | SHARED_READ     | GRANTED     | 83          |
| metadata_locks | SHARED_READ    | GRANTED     | 83          |
+-----+
6 rows in set (0.00 sec)

mysql> SELECT ENGINE_TRANSACTION_ID as trx_id,
       -> OBJECT_NAME as 'table',
       -> INDEX_NAME,
       -> LOCK_DATA,
       -> LOCK_MODE,
       -> LOCK_STATUS
       -> FROM performance_schema.data_locks;
+-----+
| trx_id    | table      | INDEX_NAME | LOCK_DATA      | LOCK_MODE | LOCK_STATUS |
+-----+
| 284109823678656 | messages | NULL      | NULL          | S         | GRANTED    |
| 284109823679432 | messages | NULL      | NULL          | IS        | GRANTED    |
| 284109823679432 | messages | PRIMARY   | supremum pseudo-record | S         | GRANTED    |
| 284109823679432 | messages | PRIMARY   | 5              | S         | GRANTED    |
| 284109823679432 | messages | PRIMARY   | 6              | S         | GRANTED    |
| 284109823679432 | messages | PRIMARY   | 7              | S         | GRANTED    |
| 284109823679432 | messages | PRIMARY   | 2              | S,REC_NOT_GAP | GRANTED    |
+-----+
7 rows in set (0.00 sec)

```

Hình 88: Innodb kích hoạt các khóa tự động tùy vào điều kiện query

2.6.3.d Optimistic and Pessimistic:

WiredTiger dùng Optimistic Locking Technique cho cấp document. Còn InnoDB sử dụng Pemisstic cho cấp row-level, cả hai cơ chế hạn chế được race-condition nhưng yêu cầu của Pemisstic nghiêm ngặt hơn

2.6.4 Isolation level

Giống nhau: Cả hai đều hỗ trợ 5 mức isolation level cơ bản đó, snapshot concurrency control thường chỉ được hỗ trợ nếu ta bật lên

Khác nhau:

- Các isolation level trong InnoDB là: read uncommitted, read committed, repeatable read, serializable và snapshot.
- Trong khi đó, MongoDB hỗ trợ 5 isolation level là: local, available, majority, linearizable và snapshot cho các tác vụ đọc/ghi thông qua các lệnh read/write concern.
- Default Isolation Level trong MongoDB là ReadUncommitted (tương ứng với lệnh readconcern đặt ở mức local hoặc available) còn trong Innodb là Repeatable Read, cả hai đều tránh được hiện tượng lost update nhưng chúng ta có thể dễ thấy Mức độ an toàn của isolation level của innodb cao hơn khi chỉ bị hiện tượng phantom read còn MongoDB bị ảnh hưởng tất cả các hiện tượng nhưng độ an toàn về cô lập dữ liệu của MongoDB vẫn được đảm bảo do hỗ trợ sẵn MVCC nên tốc độ thực thi nhanh hơn, nhưng đòi hỏi tốn nhiều tài nguyên để lưu trữ hơn. Ta có thể tùy chỉnh mức độ isolation level ở cả 2 dbms.
- Còn các mức trong InnoDB tùy trường hợp có thể thay đổi trong "runtime", có nhiều chế độ hỗ trợ tùy mục đích khi đi kèm với các câu lệnh mang tính chất consistence read hay locking read.



- MongoDB có hỗ trợ read/write concern, hỗ trợ cho cơ chế giúp người dùng kiểm tra thao tác mình vừa làm có đúng như nhu cầu bản thân hay không. Trong khi innodb không hỗ trợ cơ chế isolation này

Ta có thể thấy nếu dung lượng của ứng dụng không quá lớn nên sử dụng innodb để đảm bảo độ an toàn thông tin cao hơn, còn nếu dung lượng lớn và muốn ứng dụng có tốc độ thực thi cao thì nên sử dụng mongodb.

2.6.5 Consistence non-locking read:

Nếu ta chấp nhận transaction có thể gây ra kết quả sai hoặc lỗi nhưng đổi lại thời gian thực thi nhanh chóng không phải đợi thì ta có thể sử dụng các lệnh hoặc chỉ thị để câu lệnh ta có thể kích hoạt chế độ này. Cả mongoDB và innodb đều hỗ trợ chế độ này nhưng theo những cách khác nhau.

MongoDB sẽ hỗ trợ những thao tác lock-free (không yêu cầu khóa) thông qua vài câu lệnh như ở mức độ từ collection trở xuống

Còn Innodb hỗ trợ thông qua các chỉ thị trực tiếp ở phía cuối câu nhưng chỉ ở mức row-level ví dụ như câu lệnh dưới đây, nếu dùng NOWAIT thì có thể báo lỗi không trả về nếu hàng đã bị khóa, nếu dùng Skip-locked có thể bỏ qua hàng đã bị khóa mà trả về kết quả cần tìm.

¹ SELECT * FROM t WHERE i = 2 FOR UPDATE NOWAIT;

Vì nhóm em trước giờ khi hiện thực ứng dụng cũng chỉ làm những trang dashboard, những tính năng đi kèm chứ chưa Deploy hoàn chỉnh 1 application trong lĩnh vực thương mại điện tử (Ecommerce) nên chưa mô phỏng được việc nhiều người dùng sử dụng cơ sở dữ liệu cùng một lúc, dẫn đến việc Demo Concurrency Control qua 1 application cụ thể là rất khó khăn. Vì vậy thay vào đó, nhóm em đã thay thế bằng việc demo kỹ hơn về các trường hợp, cũng như thế mạnh của từng DBMS mà nó hỗ trợ.



3 Thành viên nhóm và vai trò

Họ và tên	MSSV	Phân công công việc	Hoàn thành
Nguyễn Quang Huy	1916081	Tìm hiểu tổng quan, các kỹ thuật sử dụng trong Concurrency Control, Concurrency Control trong MongoDB, tham gia trình bày phần so sánh Concurrency Control giữa InnoDB và MongoDB trong MongoDB, đánh báo cáo	100%
Nguyễn Đức Hoàng Phú	2010514	Tìm hiểu mục đích và sự cần thiết của Concurrency Control, Concurrency Control trong InnoDB, tham gia trình bày phần so sánh Concurrency Control giữa InnoDB và MongoDB, đánh báo cáo	100%
Lê Bình Đăng	1913102	Tìm hiểu phần Data Storage và File Structure trong MongoDB, tham gia trình bày phần so sánh Data Storage và File Structure giữa MySQL(InnoDB) và MongoDB	100%
Lê Đức Thường	1915442	Tìm hiểu phần Data Storage và File Structure trong InnoDB, tham gia trình bày phần so sánh Data Storage và File Structure giữa MySQL(InnoDB) và MongoDB	100%

Link github của phần demo [tại đây](#)



Tài liệu

- [1] R. Elmasri, S. R. Navathe, Fundamentals of Database Systems, 7th Edition, Pearson-Addison Wesley, 2016.
- [2] H. G. Molina, J. D. Ullman, J. Widom, Database System Implementation, Prentice-Hall, 2000.
- [3] H. G. Molina, J. D. Ullman, J. Widom, Database Systems: The Complete Book, Prentice-Hall, 2002.
- [4] A. Silberschatz, H. F. Korth, S. Sudarshan, Database System Concepts, 7th Edition, McGraw-Hill, 2019.
- [5] R. Ramakrishnan, J. Gehrke, Database Management Systems, 4th edition, McGraw-Hill 2018.
- [6] Specifications of today's database management systems in practice: Oracle, MS SQL Server, MySQL, PostgreSQL, Versant, Apache Cache, MongoDB, Neo4J, etc.
- [7] Marco Antonio Casanova, The Concurrency Control Problem for Database Systems
- [8] Alexander Thomasian, Database Concurrency Control: Methods, Performance, and Analysis
- [9] Florin Balasa, Data Storage
- [10] Kristina Chodorow, Michael Dirolf, MongoDB: The Definitive Guide
- [11] Charles Bell, Introducing InnoDB Cluster: Learning the MySQL High Availability Stack