Performance Improvement Plan for MySQL Insert Buffer

Hwanggyo Lee
Dept of Computer Engineering
Sungkyunkwan University
Suwon, Korea
gt369kr@skku.edu

Sang-Won Lee
Dept of Computer Engineering
Sungkyunkwan University
Suwon, Korea
swlee@skku.edu

ABSTRACT

Using non-clustered secondary index in MySQL database can cause additional and random disk accesses, degrades the performance of storage device. MySQL's storage engine eases that with Insert Buffer, which is used to avoid additional disk accesses. We verified the improvement effect of using Insert buffer in MySQL by Sysbench benchmark. Furthermore, we suggest an idea for improving Insert buffer's performance, and show simple implementation and its experimental result.

CCS Concepts

Information system → Data management systems; Database management system engines

Keywords

MySQL; Insert Buffer; Solid-State Drive; Ramdisk

1. INTRODUCTION

In *InnoDB*, the storage engine of MySQL, a table has one primary index, and zero or more non-clustered secondary indexes. Every index has the B+ tree structure. When a record is inserted in a table, information of the record is inserted in primary index first, in secondary index later. In this process, additional disk accesses, sometimes random also, are occurred because of non-clustered secondary index. We can find similar disk access patterns also in case of updates and deletes. These additional disk accesses cause the inefficient device usage. In InnoDB, the *Insert buffer* is used to resolve the performance degradation.

EDB, October 17-19, 2016, Jeju Island, Republic of Korea
© 2016 ACM. ISBN 978-1-4503-4754-9/16/10…$15.00
DOI: http://dx.doi.org/10.1145/3007818.3007833

2. Insert Buffer

When some entries are inserted into, updated, or deleted from an index, at first, InnoDB checks the requested index root pages is placed in buffer pool. In case of the clustered primary index, the root page is almost always placed in buffer pool, so it does not need any additional disk accesses except to searching internal and leaf pages of the index. In case of the secondary index, however, presence in buffer pool of the root page is not guaranteed, so it may cause more disk accesses. If a table has multiple secondary indexes, the number of additional disk accesses may increase. To resolve these disk accesses, InnoDB uses Insert buffer. Insert buffer is also a system-wide and index-like structure, so it is operated similar to other indexes. The root page is fixed in buffer pool after started-up, so there are no disk accesses for searching Insert buffer's root while InnoDB is running. Insert buffer saves index entries which have to be originally stored in currently not-in-memory secondary index pages, to avoid additional disk accesses [1].

Table 1. Benchmark in SSD and hard disk

Device	SSD		hard disk	
Insert Buffer	On	None	On	None
Transactions per sec.	946.17	792.81	19.22	13.71
R/W requests per sec.	17,031.05	14,270.55	345.98	246.77

We did an experiment to measure the performance improvement effect of using Insert buffer. We planned the experiment for both Solid State Drive (SSD) and hard disk, turn on/off Insert buffer for each device. We used Sysbench, one of the benchmark tool for MySQL. The experiment was performed on 3.4GHz Quad-core processor, 8GB DRAM machine, operating system was Ubunt 14.04.1. We used 256GB commercial SSD, which has about 500~550MB/s bandwidth for sequential access and 90K~100K IOPS for 4K random access, and 7200rpm 1TB hard disk. The Ext4 file system is installed in both devices, without write barrier. The experimental database was the size of 100GB, the size of buffer pool was 1GB, and we ran 50 threads for 3600 seconds. The table 1 is the result of this benchmark.

By the result, the performance of using Insert buffer is 19% better than none case in SSD, 40% better in hard disk.

As we mentioned before, Insert buffer is also a kind of indexes, so its pages are stored in the certain area of disk, named *system tablespace*. The system tablespace is a tablespace to store and manage data structures about whole InnoDB engine. Insert buffer pages occupy some parts of system tablespace pages. Insert buffer's root page resides in memory, but others can be loaded in or evicted from the buffer pool by InnoDB's operation and policy. In addition, Insert buffer has the limited maximum size (by default, 25% of the buffer pool size. In our cases, 256MB), so its contents must be migrated to original indexes sometime during running. Those above mean, Insert buffer makes some amount of disk accesses for operating and managing itself.

To investigate the amount of disk accesses caused by Insert buffer, we run Linux's *Blktrac*e simultaneously to trace disk access patterns, during running Sysbench benchmark mentioned above. The following table is the analysis result of disk access patterns.

Table 2. Blktrace Result Analysis

Device		SSD		hard disk	
Insert Buffer		On	None	On	None
INDEX	Read	52,860,783	45,496,691	1,117,109	700,321
	Write	15,481,472	19,396,386	526,961	249,256
IBUF _INDEX	Read	3,160,746	0	922	0
	Write	3,408,553	0	11,083	0

We could search and classify pages through their page number traced by Blktrace. Table 2 contains only about INDEX and IBUF_INDEX pages, which hold absolute majority and the most important in this result. INDEX means pages for table indexes, and IBUF_INDEX means pages for Insert buffer.

In both cases of SSD and hard disk, the number of reading INDEX pages is much bigger than writing. However, in IBUF_INDEX pages, the number of writing is bigger than reading. Especially, in SSD, 18% of writings are issued to IBUF_INDEX pages.

The result shows that the case of using Insert buffer leads better performance even though it has bigger number of I/Os than 'none' case (using - 74,911,554 in SSD, 1,656,075 in hard disk, none - 64,893,077 in SSD, 949,577 in hard disk).

The reason of above difference is the burden of random disk accesses. Different from the primary index, the secondary index is neither clustered nor unique. Thus, the insertion of secondary index entries is not guaranteed to have same sequence as the clustered primary index entries. It means that when the entries is directly added to secondary indexes without Insert buffer, those may cause random disk accesses. It is well-known that the random disk access (write for SSD, read and write for hard disk) has much lower performance than the sequential [3][4]. InnoDB solves this problem using Insert buffer, also can gain performance improvement [5]. We did experiments about different cases for equally an hour, so higher performance means more tasks is done during same experimental time. Therefore, the number of disk I/Os increased naturally when Insert buffer is turned on.

3. Improvement Idea for Insert Buffer

3.1 Insert Buffer on Ramdisk

Through the prior experiment, we found that the number of I/Os grows when the Insert buffer is used. The Insert buffer shows good performance with that bigger I/Os, but if the number of I/Os caused by Insert buffer can be reduced, we can get better performance gain. It is our main idea storing the Insert buffer in memory, much faster device than SSD or hard disk, not in system tablespace in disk. We implemented it with Ramdisk, one of the simplest way.

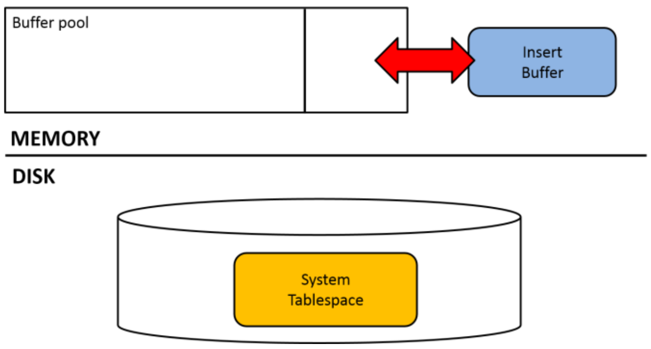

Figure 1. Insert Buffer on Ramdisk

We can consider some part of memory as if a disk partition using Ramdisk, so we can implement simply without any modifications for InnoDB's disk access routines. Evicted Insert buffer pages from buffer pool is stored in the Insert buffer area in Ramdisk, not in system tablespace anymore.

We did same benchmark tests and disk access pattern traces for Insert buffer on Ramdisk. The maximum size of Ramdisk was 2GB, and the other environmental setting were same as the previous experiments.

3.2 Experimental Results

3.2.1 SSD

In SSD, the transaction processing performance of Insert buffer on Ramdisk case is improved about 13% compared with Insert buffer on system tablespace, 35% compared with none case. We cannot see disk accesses for IBUF_INDEX pages in Blktrace result. It means that Insert buffer pages do not stored in system tablespace as we intended. The total number of I/Os in the Ramdisk case is 66,456,266. Comparing with the system tablespace case, the number of I/Os is reduced because I/Os caused by Insert buffer are removed, so the performance can be grown. Next, comparing with none case, the number of I/Os is similar, actually a little bigger in the Ramdisk case, but it shows much better performance. Through this, we can see again that the random disk accesses is much more critical for the performance degradation than the total number of I/Os.

Table 3. Insert Buffer on Ramdisk for SSD

Insert Buffer		None	On SSD	On Ramdisk
Transactions per sec.		792.81	946.17	1071.19
R/W requests per sec.		14,270.55	17,031.05	19,281.51
INDEX	Read	45,496,691	52,860,783	52,295,720
	Write	19,396,386	15,481,472	14,160,546
IBUF _INDEX	Read	0	3160746	0
	Write	0	3408553	0

3.2.2 Hard Disk

Table 4. Insert Buffer on Ramdisk for Hard Disk

Insert Buffer		None	On disk	On Ramdisk
Transactions per sec.		13.71	19.22	19.21
R/W requests per sec.		246.77	345.98	345.81
INDEX	Read	700,321	1,117,109	1,121,181
	Write	249,256	526,961	538,546
IBUF _INDEX	Read	0	922	0
	Write	0	11,083	0

Otherwise in hard disk, we cannot find any remarkable changes in both performance and the number of disk accesses. The original performance of hard disk is much worse than SSD, so hard disk cannot exploit the efficiency of Insert buffer fully under the identical experimental condition. It is shown by the number of I/Os in system tablespace case; the I/Os for Insert buffer page occupies tiny proportion of total I/Os. Therefore, in our experimental case, Insert buffer on Ramdisk solution is not necessary for hard disk.

4. Conclusion and Future Works

MySQL InnoDB uses Insert buffer to resolve the performance degradation caused by non-clustered and non-unique secondary indexes. Through some experiments, we found that using Insert buffer is helpful for device performance, and we can get better performance gain with some modifications. In this paper, we suggested in-memory Insert buffer as the idea, and evaluated the improvement factor with simple implementation using Ramdisk. However, we did experiments using normal volatile DRAM, the durability of in-memory Insert buffer is not guaranteed. Thus, our future works will be that we also can or cannot get same, or better performance gain with non-volatile memory devices.

5. Acknowledgement

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP). (R0126-16-1108, NVRam Based High Performance Open Source DBMS development)

REFERENCES

[1] The InnoDB Change Buffer, http://mysqlserverteam.com/the-innodb-change-buffer/

[2] Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy D. Zawodny, Arjen Lentz, Derek J. Balling, ″High Performance MySQL″, 2nd Ed. O′Reilly Media

[3] DongZhe Ma, JianHua Feng, GuoLiang Li, ″A Survey of Address Translation Technologies for Flash Memories″, ACM Computing Survey, 2014

[4] Sang-won Lee and Bongki Moon, ″Design of Flash-Based DBMS: An In-Page Logging Approach″, ACM SIGMOD, 2007

[5] Hwanggyo Lee and Sang-won Lee, "Performance Evaluation for MySQL Insert Buffer, KCC2016, 2016