M.S. THESIS

Towards Instant Database Recovery Using Proactive Log Replay

능동적 로그 재생을 이용한 즉각적인 데이터베이스 복구 방법

FEBRUARY 2016

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Myeongcheol Kim

Abstract

As the scale of database system grows, in addition to runtime performance and reliability, the performance and efficiency of crash recovery keep drawing attention especially in data centers. Most of the today's database systems adopt ARIES-like crash recovery methods which include traditional techniques such as write-ahead logging (WAL) during runtime, forward scanning of the log records and replay per-page chaining of log records during recovery time. In spite of several optimizations, recovery procedure steps, especially redo log replay, take significant amount of time; it may takes minutes or hours. Hereby, we invent a novel technique called proactive background log replay with which log records produced by active transactions are consumed continuously in background so as to reduce recovery time in the face of system failure. This idea assumes the environment where storage exists as a separate computing resource and where this storage is able to parse the accumulated log records and generate page snapshots in background. In this paper, we present the proof of concept design and implementation to demonstrate the feasibility of our approach. We have found by experiment that recovery time is put under control not to exceed some level. And we explain the problem of inevitable write amplification pertaining to log replay.

Keywords: database recovery, log replay, InnoDB

Student Number: 2014-21793

Contents

Abstract **i**

Chapter 1 Introduction **1**

Chapter 2 Background and Motivation **3**

2.1 Write-ahead logging (WAL) 3

2.2 Crash Recovery Procedure 5

2.3 Problem Statement and Motivation 6

Chapter 3 System Architecture **9**

Chapter 4 Implementation **11**

4.1 Change made in InnoDB 11

4.2 Log grinder 12

Chapter 5 Evaluation **16**

5.1 Methodology 16

5.2 Result on recovery time 18

5.3 Result on I/O amplification 18

5.4 Result on transaction throughput 19

Chapter 6 Conclusion **21**

Bibliography **22**

초록 **24**

List of Figures

Figure 2.1 Page update and write-ahead logging (redo logging) . . . 4

Figure 2.2 InnoDB recovery sequence 6

Figure 2.3 CDF of the number of log records of page (system tablespace); x-axis is log scale 7

Figure 2.4 **Storage model**: The interfaces are *write* in log block unit and *read* in page unit. Database process doesn't issue page write. All the pages are managed to be latest by background log reply. 8

Figure 3.1 **System architecture**: Two processes run in parallel while log grinder process runs on dedicated CPU cores. Log files are shared. Records are applied only to shadow tablespace. 10

Figure 4.1 Page hotspot (system tablespace) 13

Figure 4.2 Page hotspot (normal tablespace) 13

Figure 5.1 **Recovery time**: Recovery time is controlled within a range. 17

Figure 5.2 **Write amplification**: Write amplification keeps increasing as log size grows 19
Figure 5.3 **Baseline**: Performance of original system running Sysbench OLTP workloads. 20
Figure 5.4 **Enhancement**: Transaction rate shows good performance when working with log ginder. 20

List of Tables

Chapter 1

Introduction

It is known that data centers undergo lots of failures for various reasons, such as power outage, cooling problem, etc. Data center costs is very expensive from compute and storage equipment to power and cooling [3]. Moreover, such failure becomes norm rather than the exception. Thus, resiliency against failure and continuity of services become a important virtue of data center. Today's database systems adopt recovery methods like [2] which support traditional write-ahead logging (WAL), forward scanning of log records and the replay of per-page chaining of log records at recovery time. With this family of methods, it takes minute or hours for database system to recover form failure. For instance, current implementation of InnoDB scans sequentially all the redo log records accumulated since the last checkpoint at redo stage. The redo stage is the most time-consuming job among recovery steps. Redo log replay includes reading and parsing log records, loading a tremendous amount of pages into memory buffer, and applying changes to memory. Therefore, *replay distance*, which is the amount of records to replay, matters for instant recovery.

State of the art database systems such as SQL Server, MySQL, and PostreSQL provides log shipping and database mirroring [8] [6] [7] for instant recovery. However, these techniques rely on such a replication that these accompany communication overhead and data redundancy in a distributed environment. Whereas, we consider a new model of storage that keeps doing replay proactively to reduce replay distance. It supports "log-only write" and "page read" interfaces and eliminates the need for page flush at runtime by aid of background replay. We implemented pilot system for verification of our model. Experiment shows the effectiveness of the model.

Chapter 2

Background and Motivation

In this chapter we review basic recovery technique of database systems as a background and discuss the problem that arise from the current design and implementation.

2.1 Write-ahead logging (WAL)

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties which database systems should provide for reliability. Write-ahead logging (WAL) is a family of technique for guaranteeing two properties of ACID, atomicity and durability, and modern database systems rely on it for their crash recovery mechanisms. As its name represents, write-ahead logging writes a log of record update to storage persistently ahead of flushing the data in memory to disk. As a write-ahead logging technique, database systems leaves redo log records which contain information necessary to repeat the same operation during a database recovery. A redo record contains information about the original set of pages and

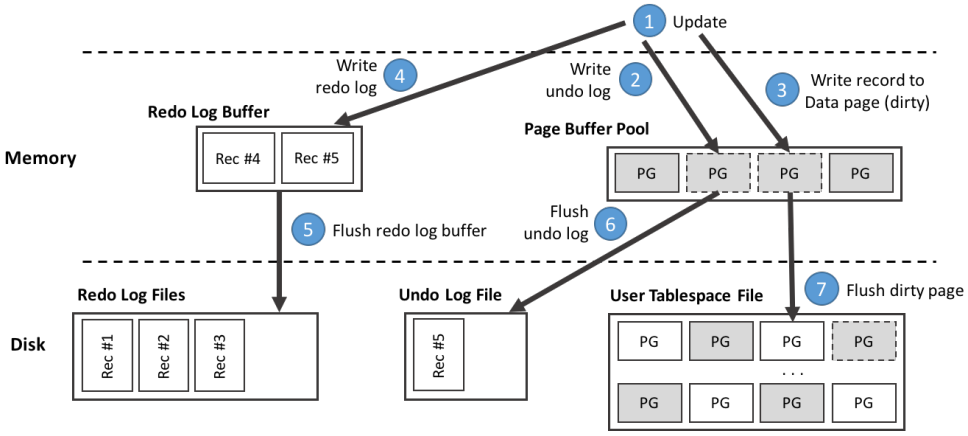

Figure 2.1 Page update and write-ahead logging (redo logging)

what has been changed in them.

Figure 2.1 illustrates what happens when a database page is modified. Record update transaction updates several records of the database pages in memory including table record itself as well as its bookkeeping information such as indexing structure, and undo log to revert uncompleted transaction ((1), (2), and (3)). The pages memory in the page buffer pool is just marked as dirty and then, redo log is written to log buffer which also resides in memory (4). At a later point in time when the period of flushing memory out to disk advents or memory capacity reaches the limit, log buffer and/or the dirty pages are written to disk ((5), (6), and (7)). At this point, write-ahead logging comes into play. Following the write-ahead logging rule, log buffer is output to disk first and then, the dirty pages are output to disk later. This order is very important. By means of this strict order, the set of dirty pages which is possibly lost in the face of unexpected system failure can be re-created.

2.2 Crash Recovery Procedure

In this section, we describe recovery procedure briefly. Explanation is focused on specific storage engine, InnoDB, which is our target of implementation and experiment.

Figure 2.2 illustrates crash recovery sequence of InnoDB storage engine. Every time InnoDB starts, it tries to do a recovery even though the system had been shutdown normally for ease of implementation reason. Redo log file stores the last checkpointed log sequence number (LSN) in its dedicated area, checkpoint block, and it is guaranteed that all the dirty pages has been flushed before this LSN. The checkpoint block is managed in the shadow paging fashion for the reliability of the checkpoint itself. Recovery module makes a decision whether recovery actually should be done or not by comparing the last flushed log LSN (L-LSN) with the LSN of the last flushed page (P-LSN). If the flushed page LSN is smaller than the flushed log LSN, recovery procedure needs to be triggered. Recovery steps are comprised of the following.

Forward log scanning Staring from the last checkpoint LSN, recovery modules scans whole redo log records up to the last flushed log LSN. Every time it reads a chunk of blocks it tries to parse log records of the blocks.

Parsing and sorting records Recovery module parses log records one by one and it can know of the space ID and page number to which the record belongs. Then, it appends the record to the address hash table, of which each entry has a chain of records which belongs to same address; space ID and page number. In this way records are sorted in page order and in time order within each page. Address hash table will be used to replay transactions before crash.

Applying records Once building up the address hash table has been done, recovery module does replay of records in page order that is, it load a page

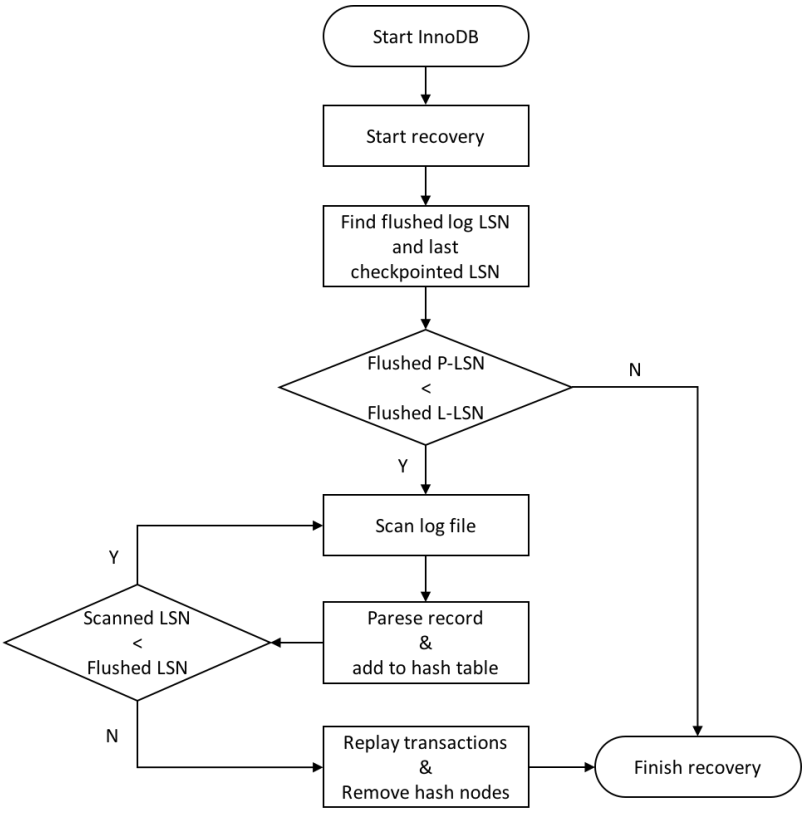

Figure 2.2 InnoDB recovery sequence

from disk and applies records of same address at once and then, goes through the next pages. After traversing all the hash entries, recovery procedure ends and a set of pages remains as dirty, which will be flushed to disk eventually.

2.3 Problem Statement and Motivation

As more pages of memory buffer pool get dirty, the difference between the contents in memory and the ones in persistent storage becomes bigger. But, flushing out the dirty pages frequently so as not to cause such big difference makes the situation worse because the performance of actively running trans-

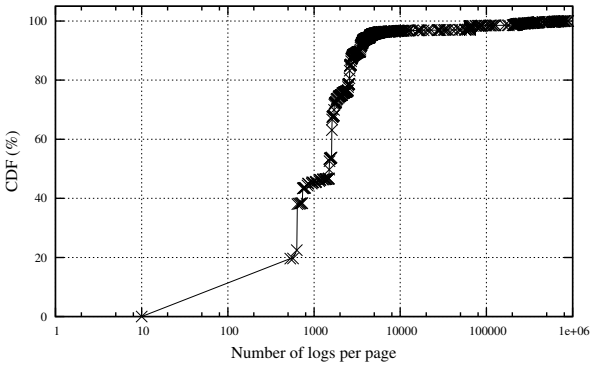

Figure 2.3 CDF of the number of log records of page (system tablespace); x-axis is log scale

actions can be degraded due to the interference of I/O. On the contrary, the longer checkpoint interval becomes, the more log records are accumulated. The amount of log records which need to be replayed during crash recovery is called *replay distance* from checkpoint. If replay distance is very long, a lot of log records could be accumulated for each page, which in turn makes crash recovery time extremely longer. Figure 2.3 shows cumulative frequency distribution of the number of log records of a pages. This result is taken from a snapshot of InnoDB log file in the middle of Sysbench OLTP benchmark test. 55% of pages have more than 1000 log records, total number of bytes of which are much bigger than the page size (4KiB) itself.

There definitely exists the tradeoff between the performance of active transaction and crash recovery time. But, what if we can consume ever-generated log records and apply it to disk pages in background without detriment to the performance of runtime transactions? If it is possible, replay distance can be maintained very short and recovery time comes to be short as well. This technique is called *proactive background log replay.*

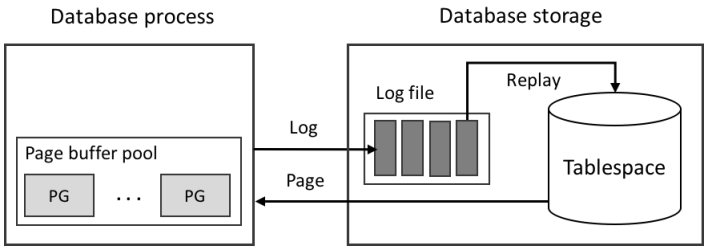

Figure 2.4 **Storage model**: The interfaces are *write* in log block unit and *read* in page unit. Database process doesn't issue page write. All the pages are managed to be latest by background log reply.

Proactive background log replay could degrade the performance of runtime transactions. But, if we eliminate the overhead by offloading it to separate system resources, we could achieve instant crash recovery. [9] shows potential for in-storage processing (ISP). They offload database query execution onto solid state drives (SSDs). We employ the similar approach; we set up the storage model (Figure 2.4) in which key feature is as following.

Interface: Log-only writing/page retrieval This storage supports two major APIs. It accepts only a log block as an input - *write(LSN, log_block)*, and gives the requested page to the host process - *read(page_no)*.

No need of page flush Because all the pages in the tablespace are managed to be latest, there is no need to flush dirty pages. To follow write-ahead logging rule, any updates to page must have been written to the storage as a form of log records using the *write()* API above.

In the next chapter, the details of this model and our pilot system which is implemented as a proof of concept demonstration are discussed.

Chapter 3

System Architecture

This paper focuses on the feasibility of proactive background log replay method before going further into implementation in real system such as flash SSD. In this context, we devise a pilot system which mimics the storage model described in the previous chapter. Figure 3.1 illustrates the architecture of our system. One database process (storage engine) and one log replay process, which we call *log grinder*, run in parallel with the latter running on dedicated cores to minimize computational interference.

Log files are shared between two processes. Data file, which is called tablespace file in InnoDB storage engine, is replicated and each process interacts with its own data file. The one log grinder is using acts as a *shadow copy of data file*. To mimic "log-only write" interface, storage engine writes records to log file in accordance with its own log flush/checkpoint interval and the log grinder scans, parses, and applies records to the pages loaded from shadow tablespace. We let the storage engine do page I/O as it always does but, give it a very large page buffer enough to prevent page flush during test. This artificial environment

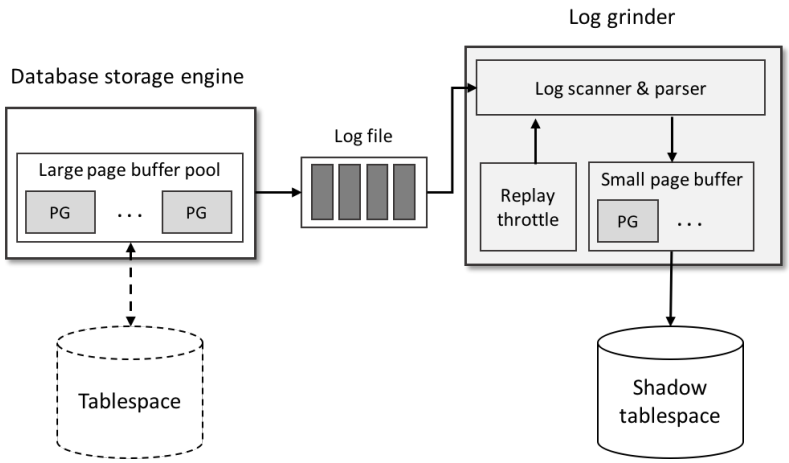

Figure 3.1 **System architecture**: Two processes run in parallel while log grinder process runs on dedicated CPU cores. Log files are shared. Records are applied only to shadow tablespace.

is for mimicking "no flush" feature.

Log grinder is composed of three main components inside; log scanner/parser, page buffer, and replay throttle. Log scanner is notified of the event for log flush by storage engine so that log grinder can know of the log sequence number (LSN) up to which records are written safely. Then, it parses the records to append the records to the list which is referred to for replay later. Page buffer needs to be smaller than one of storage engine because real storage might have the limited amount of memory. Replay throttle decides when to apply the parsed records to pages and flush the dirty pages to persistent data file.

Chapter 4

Implementation

Our implementation is based on the latest stable version of MySQL at the moment, v5.6.27.

4.1 Change made in InnoDB

Disabling file locking Whole region of database files is locked with *fcntl()* system call by storage engine at the start of database server to prevent another server process from being instantiated. Nevertheless, we remove the locking because log grinder needs to share log file with storage engine. It will not be the problem in real system which has the log file inside.

Inter-process communication Log grinder needs to be informed of the last flush of log records because there is no information about the last flushed LSN in the log file header. This is necessary not to exceed the boundary of log which is managed in a circular fashion. Communication channel is set up with UNIX domain socket and storage engine sends event of $\langle command, LSN \rangle$ tuple

to log grinder every few MiBs of new log records flush.

Disabling log checkpoint Because InnoDB touches checkpoint area and log grinder also makes checkpoint after page flush, that area possibly corrupted unless we provide synchronization on it. Whereas, our system assumes checkpoint is done only for log replay. We restrain InnoDB from checkpoint modification.

4.2 Log grinder

The basic functionality of log grinder such as file I/O, buffer pool management, and double write buffer is same as that of InnoDB. Thus, it is implemented by modifying and adding new functionality to InnoDB source code. There are several things we need to consider for our pilot system.

Replay throttle and page hotness We conduct runtime profiling with MySQL server to see if there exists page hotness. Figure 4.1 and Figure 4.2 shows the page access pattern. As time goes by (the progress of LSN in x-axis), accessed page numbers are plotted. For normal tablespace, user data file, page accesses are evenly distributed because that is the characteristics of OLTP workloads [5] [4]. But, system tablespace has several hotspots as it contains bookkeeping information for every updates such as change buffer, double write buffer, and undo logs [6]. We need to consider this characteristic to reduce flush overhead.

Parallelized page flush Log replay incurs inevitable write amplification while trying to maintain the latest image of pages. So, massive write operations should be parallelized not to make whole procedure lag behind. Moreover, we can enhance throughput and exploit the rich internal parallelism of flash storage.

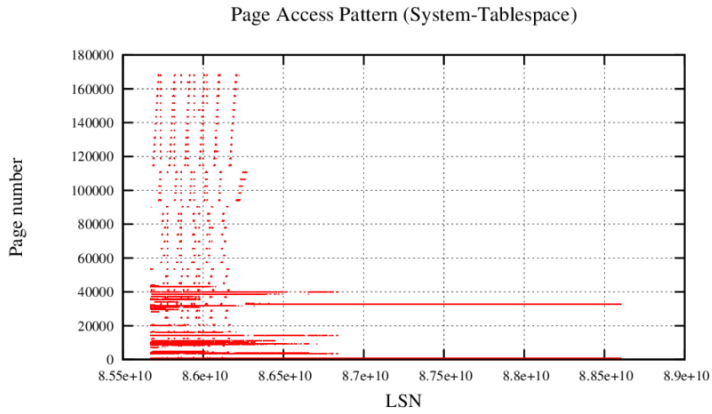

Figure 4.1 Page hotspot (system tablespace)

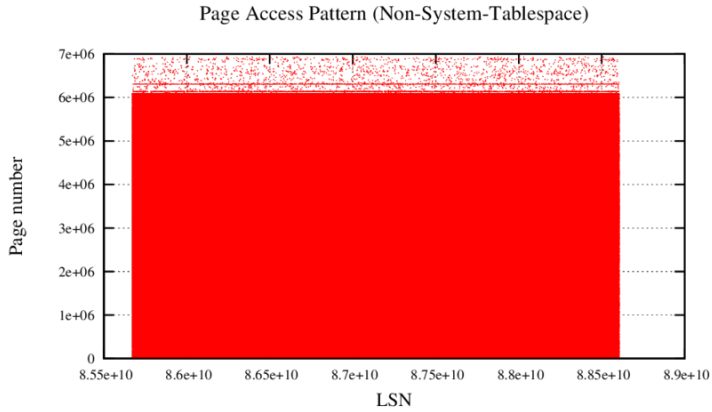

Figure 4.2 Page hotspot (normal tablespace)

Overall replay sequence are shown in the Algorithm 1. On receiving log flush event which contains the last flushed LSN, records are scanned and parsed up to the LSN. After parsing, these are added to address hash table. This step is exactly the same as that of InnoDB except that it scans up to the end of logs while our log grinder scans up to the point storage engine notifies of. Next, it

checks statistics of the total number of dirty pages and page hotness for *replay throttle* decision. This is for controlling replay distance as short as possible while preventing log replay from lagging behind log flush events. The concept behind this is that.

- The hotter a dirty page is, the less it needs to be flushed. But, more cold pages piles up along with it.
- Total number of dirty pages should not across a threshold because the more dirty pages we wait for without page flush, the more flush overhead arises later.

Procedure call to **THROTTLE-SHOULD-FIRE** in Algorithm 1 makes decision whether to start to apply records to pages and to flush them based on the following equations. It figures out how much overloaded dirty pages are for flush. Page hotness is defined as total sum of record size (RS) of the page. And the load is calculated by multiplying the number of dirty pages (ND) and the sum of the multiplicative inverse of each page's hotness. If new cold page comes in, both ND and the sum of inverse increase with resulting higher *Load* value. On the contrary, *Load* decreases as pages are accessed more frequently because *Hotness* increases. If *Load* crosses a threshold, log replay comes into operation.

$$Hotness = \sum RS \tag{4.1}$$

$$Load = ND * \sum \frac{1}{Hotness} \tag{4.2}$$

Algorithm 1 Log replay

```
procedure REPLAY
    repeat
        last_flushed_lsn ← WAITING-FOR-LOG-FLUSH()
        l ← last_checkpoint_lsn
        while l ≤ last_flushed_lsn do
            record ← PARSE(l)
            ADD-TO-HASH-TABLE(table, record)
            stat ← UPDATE-HOTNESS()
            if THROTTLE-SHOULD-FIRE(stat) then
                dirty_pages ← APPLY(table)
                FLUSH(dirty_pages)
            end if
            l ← NEXT-RECORD-LSN()
        end while
    until connection is broken
end procedure
```

InnoDB of current implementation doesn't flush recovered pages to disk immediately. Page flush follows basic algorithm of its *page cleaner* later. However, log grinder should make checkpoint after applying records. Thus, we parallelize page flush with multiple flusher threads.

Chapter 5

Evaluation

We compare the recovery time and the overhead of our log grinder with that of original InnoDB storage engine. We first run MySQL server with Sysbench OLTP workloads and it executes log grinder process. To force database to crash, we kill MySQL server process. Then, connection between them is closed, log grinder tries to finish current job and terminates execution. We run our tests on Intel Xeon E7-8837 server, which has eight Quad-Core processors at 2.67GHz and is equipped with 132GiB of memory and Samsung SSD 850 Pro connected via SATA-II 3Gb/s. But, we utilized small portion of the total memory because SSD capacity is small. Ratio of table file size to memory size is set as 4:3 (8GiB:6GiB). And log file is created with the capacity of 5GiB.

5.1 Methodology

We set up the following metric for evaluation.

Recovery time For baseline, we start original MySQL alone and kill it

when uncheckpointed log size reaches the level we want. Then, we run MySQL again with log grinder doing replay this time and kill the server process when the same log level is reached. After crash, MySQL server restarts with the shadow tablespce. These two recovery time are compared.

Write amplification InnoDB flushes dirty pages during runtime whenever it decides to flush some amount of pages as fas as not disturbing currently running transactions. It also writes log records to log files. Therefore, total size of data the original InnoDB writes is the sum of runtime flush size and log write size. We compare this sum with the sum of runtime flush data log grinder generates and the size of log it receives.

Transaction throughput As mentioned in section 2.3, our system is expected to outperform the original database system in terms of transaction throughput (TPM) because it eliminate the need of page flush by storage engine. We compare the throughtput of storate engine working with log grinder with the original one.

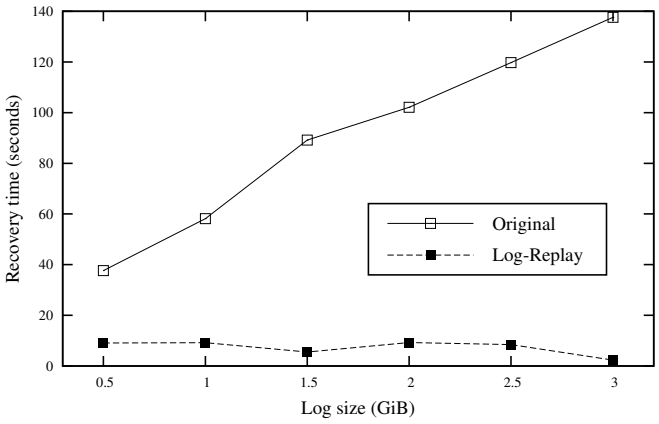

Figure 5.1 **Recovery time**: Recovery time is controlled within a range.

5.2 Result on recovery time

Figure 5.1 shows recovery time of original InnoDB and log grinder. As mentioned in previous chapter, InnoDB's recovery procedure doesn't include page flush, So, recovery time is dominated by the amount of log records and increases near-linearly with it. On the other hand, log grinder controls recovery within specific range; less than 10 seconds in our experiment. It didn't lag behind the reception of flush event and could maintains some level of remained records by means of throttling. If crash occurs with little records remaining, recovery time gets low.

5.3 Result on I/O amplification

Figure 5.2 shows write amplification of log replay. The fact that log grinder maintains some level of replay distance implies periodic disk write. It also implies that write amount keeps increasing as time goes by. But, our system model provides "no-flush" functionality. Definitely, write amount is bigger than original system but, the level of write amplification is maintained at some level. This result comes because write amplification is compensated for by the elimination of page flush that original storage engine must have done and that in turn causes runtime overhead.

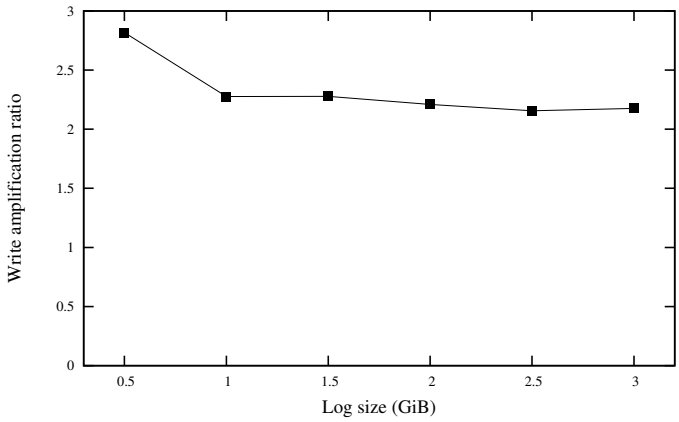

Figure 5.2 **Write amplification**: Write amplification keeps increasing as log size grows

5.4 Result on transaction throughput

Transaction throughput is measured with Sysbench OLTP benchmark. The R:W ratio is around 5:3. The line of baseline (Figure 5.3) fluctuates whenever page flush occurs. Whereas, throughput of our system shows stable line without page flush and it outperforms original system by up to 60% in terms of average throughput (180000 TPM vs. 110000 TPM). This enhancement comes from "no flush" feature.

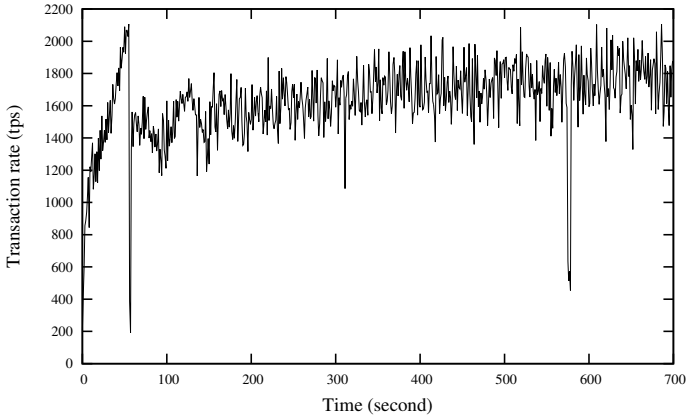

Figure 5.3 **Baseline**: Performance of original system running Sysbench OLTP workloads.

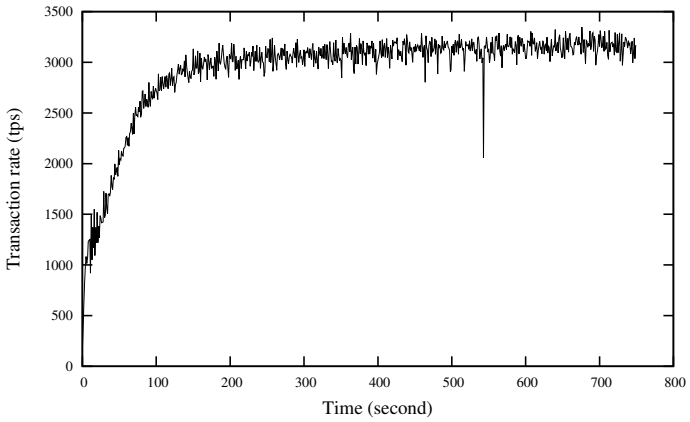

Figure 5.4 **Enhancement**: Transaction rate shows good performance when working with log ginder.

Chapter 6

Conclusion

We have present a new storage model for facilitating instant recovery with proactive background log replay. It supports "log-only write" interface and "no-flush" feature. It assumes storage exists as a special computing resource. We expect this model to enhance runtime performance by eliminating page flush overhead. Based on it, we implement pilot system to demonstrate feasibility of the model. It is shown that crash recovery time is controlled within a range by experiment. And inevitable write amplification during background replay can be compensated with "no-flush" feature, which is expected to enhance transaction performance.

Bibliography

[1] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*, McGraw-Hill, New York, 2010.

[2] C. Mohan, D. Haderle, and B. Lindsay et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, Vol. 17, No. 1, pp. 94-162, March 1992.

[3] G. Goetz, "Instant recovery for data center savings," *SIGMOD Record*, Vol. 44, No. 2, pp. 29-34, June 2015.

[4] W. Kang, S. Lee, and B. Moon et al., "Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases," *Proceedings of the 2014 ACM International Conference on Management of Data (SIGMOD '14)*, Snowbird, USA, June 2014.

[5] I. Petrov, R. Gottstein, and T.Ivanov et al., "Page Size Selection for OLTP Databases on SSD Storage," *Journal of Information and Data Management*, Vol. 2, No. 1, pp. 11–18, February 2011.

[6] MySQL Internals Manual, http://dev.mysql.com/doc/internals/en/

[7] PostgreSQL Manual, http://www.postgresql.org/docs/manuals/

[8] About Log Shipping (SQL Server), https://msdn.microsoft.com/en-us/library/ms187103.aspx

[9] J. DO, Y. Kee, and J. Patel et al., "Query Processing on Smart SSDs: Opportunities and Challenges," *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD '13)*, New York, USA, June 2013.

초록

데이터베이스 시스템의 규모가 커져감에 따라 실행 중의 성능과 신뢰성 뿐만 아니라 시스템 실패시의 성능과 효율이 관심을 모으고 있다. 오늘날 많은 데이터베이스 시스템이 ARIES 계열과 비슷한 복구 방법을 취하고 있다. 전통적인 테크닉인 write-ahead logging (WAL), 전방 로그 스캐닝, 페이지 단위 로그 레코드 체이닝 등이 대표적인데, 몇몇 최적화 기법에도 불구하고 복구 절차 단계들 특히, redo 로그 재생은 상당히 오랜 시간이 걸리는 작업이다. 이 과정은 수분에서 수시간이 걸릴 수 있다. 이에 우리는 능동적 로그 재생 기법을 고안했고, 이것은 실시간 트랜잭션이 계속해서 만들어 내는 로그를 백그라운드에서 지속적으로 재생하고 소모함으로써 시스템 실패 이후 복구 과정을 획기적으로 단축할 수 있는 방법이다. 이 아이디어는 저장소가 또 하나의 계산장치로서 동작하는 환경을 가정하고 있다. 이 저장소는 계속 쌓여가는 로그 레코드들을 백그라운드에서 검색하고 파싱하고 페이지 스냅샷을 만들어내는 일을 한다. 이 논문은 이 스토리지 모델의 타당성에 대한 개념증명으로써의 구현을 보여준다. 그리고, 시험 시스템을 통한 실험으로 데이터베이스 복구 시간을 어느 수준 이하로 제어할 수 있음을 보여주었다. 그리고, 로그 재생과 관련하여 필연적으로 발생하는 디스크 쓰기 증폭 문제에 대하여 설명한다.

주요어: 데이터베이스, 로그 재생, InnoDB

학번: 2014-21793