Concurrent Updates to Pages with Fixed-Size Rows Using Lock-Free Algorithms

Raghavendra Thallam Kodandaramaih
Microsoft
One Microsoft Way
Redmond, WA-98052
raghavt@microsoft.com

Hanuma Kodavalla
Microsoft
One Microsoft Way
Redmond, WA-98052
hanumak@microsoft.com

Girish Mittur Venkataramanappa
Microsoft
One Microsoft Way
Redmond, WA-98052
girishmv@microsoft.com

ABSTRACT

Database systems based on ARIES [11] protocol rely on Write Ahead Logging (WAL) to recover the database in the event of a crash. WAL protocol requires changes to the database are recorded to the transaction log before updating the underlying database page. WAL also mandates that the log record corresponding to the change is persisted to disk before the updated page. While WAL allows updates to the databases using in-place updates or using shadow paging, database systems that perform in-place updates typically latch the page exclusively for the entire duration of log generation and the change on the page. The exclusive latch on the page prevents other threads from modifying the page at the same time, reducing the concurrency, and negatively impacting the throughput of the system. While approaches like Segment-Based recovery [16] attempt to solve the contention by pushing the burden of synchronization to the application along with a proposal for recovering parts of pages, this paper takes a different approach by providing a mechanism to support concurrent updates to certain kinds of pages under a shared latch using lock-free algorithms. The pages are recovered using existing ARIES protocol with a few modifications. This approach significantly boosts the throughput of an ARIES based database system, without any application changes. The paper describes in detail the challenges of implementing the mechanism and how the ARIES concepts like page LSN, logging and checkpoint are handled to support concurrent updates on space maintenance pages in Microsoft SQL Server. The paper also presents the experimental results showcasing the impact of the work.

PVLDB Reference Format:
Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Girish Mittur Venkataramanappa. Concurrent Updates to Pages with Fixed-Size Rows Using Lock-Free Algorithms. *PVLDB*, 13(12) : 3195-3203, 2020.
DOI: https://doi.org/10.14778/3415478.3415544

1. INTRODUCTION

ARIES based database systems like System R, DB2, Sybase ASE, PostgreSQL and Microsoft SQL Server contain dedicated set of pages for space management. Some of the notable examples of the pages are - the free space inventory pages (FSIPs) in System R and Space Map Pages (SMPs) in DB2 [10] that track the space management in the data pages, Free Space Map (FSM) [14] in PostgreSQL that tracks free space, PageFreeSpace (PFS) [3][8] pages in Sybase ASE and SQL Server that track the free space in system. These pages track the space utilization of the individual data pages and are essential for optimizations like avoiding reading of empty pages [10], identifying pages that can be used for new inserts without scanning the entire object (table or index). The space management pages increase the level of concurrency in the system; however, frequent updates to those pages becomes the source of latch contention in the system. There are optimizations for updating the free space only when the free space on the page moves across thresholds like 50%. Even with such optimizations in place, the space management pages run into contention. IBM seems to have solved the contention problem on FSIPs/SMPs by avoiding latching/locking to provide higher concurrency as mentioned in footnotes in [10], but the details of the implementation are not mentioned in the paper.

Customers of SQL Server have faced similar contention on space management pages like PFS for decades. There is a PFS page for every 64 MB chunk of data in SQL Server. Customers that run into the PFS contention have workloads that perform lots of page allocations, which update the state of the pages in PFS. Spreading the allocations across different 64MB chunks or spreading the allocation across different files can help alleviate the contention, but customers do not have the ability to force allocations to a specific 64MB chunk of a file or specify a file among set of available files. They sometimes work around the contention problem by adding additional files to the database in their installations. The presence of multiple files acts as a hint to the allocation algorithm in SQL Server to select a different file for every set of allocations. While this doesn't necessarily eliminate the PFS contention entirely, it does help alleviate the problem to some extent. However, as more and more databases are moving to the cloud, customers may not have the configuration knobs to add more files due to the restrictions imposed by the cloud provider. Such customers would end up settling for lower transaction throughput. The mechanism described in this paper helps eliminate the page latch contention on pages like the PFS pages. With this solution, customers no longer have to provision multiple database files as workarounds or settle for lower throughput.

This paper describes the overall design of "Concurrent updates to a page with fixed size rows using lock-free mechanisms" concept which is applicable to a wide range of pages, whose updates have

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415544

a higher level lock to guarantee that no two threads in the system are updating the same location concurrently. Even though the idea is applicable to a broader set of pages, the paper uses PFS page as an example to help explain what it takes to update a given ARIES based recovery system to support concurrent updates to a page. The team implemented concurrent updates to PFS pages and enabled by default in the box release of SQL Server - SQL Server 2019, and enabled the feature in the cloud counterpart – Azure SQL Database for over a year. Section 2 begins with the background of the database pages in SQL Server and the mechanism used to manage PFS pages in memory to adhere to WAL. Section 3 outlines the design and the architecture of supporting concurrent updates on PFS pages using lock-free mechanisms under shared page latch, while still adhering to most WAL semantics. Section 4 presents the experimental results showing the performance impact on certain workloads. Section 5 describes the future work and the other areas of databases to which this idea can be extended to. Section 6 explains the applicability of the work in other systems. Section 7 discusses the related work. Finally, Section 8 presents the conclusions.

2. BACKGROUND ON SQL SERVER

This section provides a summary of the system pages in SQL Server that keep track of space-related information, the mechanisms used to track the dirty page lists and the source of the contention on these pages.

2.1 System Pages in SQL Server

2.1.1 Page Free Space (PFS) Page

Space in SQL Server is managed in units called extents consisting of eight logically contiguous 8KB pages. Each PFS page contains one byte for every page in an 8088-page range of a file.

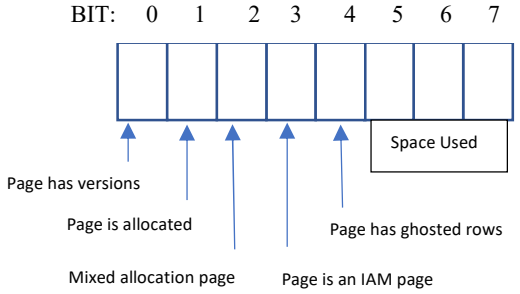

Figure 1. Structure of a byte on a PFS Page

Figure 1 shows the structure of a byte on a PFS page:

- Bit 0 indicates if the page has potentially stale versions that need cleanup when the Constant Time Recovery (CTR) [13] feature is enabled.
- Bit 1 indicates whether the page is allocated or not.
- Bit 2 indicates whether the page is from a mixed extent (pages belong to different objects) or not.
- Bit 3 indicates if the page is an Index Allocation Map (IAM) page.
- Bit 4 indicates if the page contains ghost records.
- Bits 5 through 7 are taken as a three-bit value indicating the page fullness in pages belonging to heaps (non-index objects):
    - 0: empty
    - 1: less than 50% full
    - 2: 50% to 80% full
    - 3: 80 to 95% full
    - 4: 100% full.

While some of the information in the PFS Pages can be derived from the content on the page, the PFS pages are quite useful in the system. SQL Server has a couple of background threads - ghost cleanup and version cleanup that leverage the bits in the PFS pages to identify the pages containing the older versions of the data that are no longer necessary in the system. Without the tracking bits, the cleanup threads would have to perform a scan of the entire database.

2.1.2 Index Allocation Map (IAM) Page

IAM pages keep track of the extents in a 4-GB section of the database file that are allocated to an allocation unit. An allocation unit is a set of pages belonging to a single partition in a table or an index.

2.1.3 Global Allocation Map (GAM) Page

The GAM pages record the allocated extents using one bit per extent. There is one GAM page for every 64000 extents or 4GB of the file.

2.2 Log Sequence Number (LSN)

SQL Server transaction log records track the changes made to the database pages. It stores enough information to allow SQL Server to recover the database in the event of crash. The log is maintained in one or more log files associated with the database. Each log record is labeled with a unique Log Sequence Number (LSN). Just like other ARIES [11] databases, SQL Server adheres to write-ahead logging (WAL) – the changes to the pages are written to the transaction log before the changes are written to the database pages. Each page contains the LSN of the last log record that contains the change to the page. Maintaining the LSN on the page is an essential aspect of ARIES; all three phases of ARIES recovery – analysis, redo and undo rely on it.

2.3 Buffer Pool

The buffer pool is the main memory component of SQL Server. It manages the I/O functions for bringing the pages into memory and flushing them to disk. Every buffer in the buffer pool has a header that contains status and other information that is used to manage the lifetime of the buffer from the time a page is read from disk to the time the page is evicted. SQL Server employs LRU-k [4] as the page replacement policy.

2.4 Checkpoint

SQL Server checkpoint process scans the buffer pool periodically to write out any dirty pages to disk thus ensuring that the number

of such pages is low in the system. This reduces the database recovery time in the event of crash.

2.5 Dirty Page Manager

The dirty page manager tracks the dirty buffers for a database in two lists. One list has the buffers for pages that have been prepared to dirty but do not have any changes yet and another has the buffers for pages that have been dirtied.

2.6 Dirty Page Context (DPC)

The Dirty Page Context is a structure that is maintained in every buffer. It tracks the checkpoint ID that the buffer corresponds to. A checkpoint is deemed complete only if all the buffers corresponding to the checkpoint ID are successfully flushed to disk. The DPC also tracks the first LSN that dirtied the buffer in this checkpoint, which is used to determine the oldest page LSN for the checkpoint.

2.7 Latch Modes in SQL Server

SQL Server supports the following latch modes used to perform read/write operations on database pages.

2.7.1 Shared (SH) Latch

Multiple threads can acquire SH latch on the page at the same time. Original intention was to support concurrent threads reading the same page.

2.7.2 Update (UP) Latch

Only one thread can acquire the UP latch on a page and is used to perform updates on a page. UP latch is compatible with SH latch but not with UP or EX latch. The UP latch is mostly used for the system pages like the PFS/GAM/IAM pages where the updates modify the fixed-size rows in-place. Such updates do not cause the page content to shift. Concurrent reads using SH latch on the page may read a stale value in the presence of concurrent updates but not a bogus value.

2.7.3 Exclusive (EX) Latch

Only one thread in the system can acquire the EX latch on a page at a time. Most pages, other than the PFS/GAM/IAM pages are updated using EX latch. EX latch is not compatible with UP or SH latches.

2.8 Existing Algorithms for Updating a PFS Page

The latch types mentioned above are available in other ARIES based database systems too. We have already established that the PFS page updates are derivative of the update to the underlying data page. While we will continue to mention PFS as the example in the following descriptions, many of the aspects are also applicable to other space usage pages in other databases. Updates to the PFS pages are always done under the scope of higher order lock while doing physical data page updates like updates to rows in those pages or logical data page updates like allocation/deallocation. The order in which the data operation and the PFS page operation are done within a transaction can vary. For example, during the forward processing of update to a page that results in updating the space utilization bits in PFS, the update to the data page is done first and then the update to PFS page. However, during rollback processing of the transaction that is undoing the effects of the transaction, the PFS state is updated first and then the data page modification. This is necessary to ensure that recovery works correctly, even in the presence of optimizations like PFS update log records being redo only log records in some cases as explained in [10]. Irrespective of the order of the data page update or PFS update, the key is that the PFS update is done within the scope of the exclusive latch of the data page.

```
Void UpdatePage()
{
  Latch the data page exclusively.
  {
    Compute the necessary PFS update;
    If the transaction is in rollback, update the PFS page;
    Generate log record for the data page;
    Update the data page;
    If the transaction is not in rollback, update PFS page;
  }
  Release the latch on the data page;
}
```

Figure 2. Method to update a data page

Figure 2 shows the sequence of steps in updating a data page in SQL Server, which is very similar to other ARIES based database systems. The data page is latched exclusively; under this scope, the log record for the page update is generated, page content is updated and PFS page is updated if necessary. In some cases like transaction rollback, the order of PFS update (again if necessary) and data page updates might be reversed but they are all done under the scope of the exclusive data page latch.

```
Void UpdatePFSPage()
{
  Obtain PFS page corresponding to the data page;
  Update latch the PFS page;
  {
    Prepare the PFS page to be dirtied;
    Generate log record for the PFS update;
    Update the byte on PFS page;
    Set LSN on PFS page;
    Mark PFS buffer as dirty;
  }
  Release the latch on the PFS page;
}
```

Figure 3. Original method to update a PFS page

Figure 3 provides the details of the original method to update the PFS page. Given the data page id whose PFS byte needs to be updated, the PFS page buffer hosting the information is identified

and latched for update. Then the PFS buffer is prepared to track the page for the current checkpoint using the dirty page manager. As part of this tracking, if an element in the dirty page manager called the dirty page context (DPC) exists, it is used, if not a new entry is created. The DPC tracks the LSN of the first update that dirtied the buffer in the current checkpoint. For a checkpoint, the minimum of all the dirty page LSNs is used to compute the oldest dirty page LSN in the system, which is used to hold up the log truncation for the database. After the PFS page is prepared to dirty, the log record for the PFS page is generated. Next the PFS page is updated and the LSN of the log record is stored in the PFS page header. Then the PFS page buffer is marked as dirty in the DPC by setting the *DIRTY_BIT* in the buffer. The LSN of the log record generated is contributed to the DPC as well. At this point, all the actions on the PFS page are completed, and the update latch is released.

To conclude, there are several buffer pool and log manager operations that are performed under the scope of the update latch on the PFS page. The update latch implies only one PFS update can be performed at a time, while other updates on the same PFS page get blocked. This serialized mode of updating PFS pages is the source of page latch contention. While we have described the problem in detail for the PFS page, this pattern repeats in other pages like GAM pages and IAM pages in SQL Server as well as the space maintenance pages in other database systems. This decades-old contention problem motivated us to find a solution.

3. CONCURRENT UPDATES WITH ONLY SHARED LATCH

This section covers the design and implementation details of the concurrent updates to a page with fixed format under shared latch. Concurrent updates to pages like PFS pages can be applied if they modify different bytes. In the case of PFS page update, there is either an exclusive page latch (for version/ghost/space PFS bit updates) or an exclusive page lock (in the case of allocation PFS bit updates) on the data page whose PFS byte is being updated. This provides the necessary guarantee that no two threads would be updating the same PFS byte concurrently. If a thread has an exclusive latch on the data page D and is in the process of updating its space usage in the corresponding PFS page P, then no other thread in the system can attempt to update other properties like version or ghost or allocation bit of page D as they would get blocked on the exclusive latch on D. This effectively blocks the update on same PFS byte for page D.

```
Void UpdatePFSPageUsingSHLatch()
{
   Obtain PFS page corresponding to the data page;
   Share latch the PFS Page;
   {
      Prepare the PFS buffer to be dirtied;
      Generate log record for the PFS update;
      Update byte on PFS page using interlocked operations;
      Set LSNon PFS page header using interlocked operations;
      Mark PFS buffer as dirty;
   }
   Release the latch on the PFS page;
}
```

Figure 4: Update a PFS page using shared latch

Updating a PFS page involves several buffer pool maintenance operations described earlier. To support concurrent updates, each of the operations must be made thread safe to ensure the overall intent of each and continue to recover the page per ARIES recovery. This is achieved by using interlocked operations for in-memory data structures and other lock-free algorithms for persisted structures. Figure 4 describes the pseudo-code to update the PFS under shared latch with the **highlighted** parts indicating the main differences from the old protocol in Figure 3. Note that there aren't any changes around the way the data page is latched or locked for which the PFS update is being done. When the update on the data page triggers a corresponding PFS update, the PFS page is latched using SH latch instead of the traditional UP latch, which effectively allows other threads to read and write the PFS page.

Let us look at how the other components get impacted to support concurrent updates to certain sets of pages while maintaining ARIES assumptions to help recover the page in the event of crash.

3.1 Buffer Management

Buffer manager is responsible for managing the buffers containing the pages. Typically, buffer pool manages all buffers in the same way assuming the updates to page buffers are done in an exclusive fashion. To support the concurrent updates scheme, the buffer manager is modified to be aware of pages that can be updated under shared latches. For every buffer that is prepared to host a page that can take concurrent updates, a bit in buffer header, *BUF_CAN_HAVE_SHARED_UPDATES,* is set to true. This bit is the signal to the buffer pool to use the version of the API that supports concurrent threads.

3.2 Dirty Page Context (DPC) Creation and Setup

Some of the APIs manipulating the DPC were not thread safe and had to be modified in various ways to allow concurrent updates. When the latch on a page is obtained for write operations, the first buffer operation is to prepare the buffer to be dirtied. On buffers that can support concurrent updates, this request was made thread safe. This step, *PrepareToDirty,* involves setting up DPC on the buffer. It is made thread safe using interlocked compare and exchange operator instead of the existing assignment operators. This ensures if two threads attempted to setup the DPC for the same PFS buffer, only one would succeed and the other would bail out and use the one whose setup was successful.

3.3 Log Record and LSN Management

One of the steps in updating the PFS page is generating log record and updating the LSN on the PFS page. The log manager in SQL Server supports generation of log records from concurrent threads irrespective of the page the log record is generated for. The inputs to the log manager are: page id, previous page LSN, previous value of the row and new value of the row. Given this payload, the log manager serializes and returns the LSN at which the payload was serialized to.

In ARIES protocol as part of redo processing of log records, if the LSN on the page does not match the previous page LSN from the log record, it indicates a missed application of the log record and the redo processing fails with a high severity error. However, with

concurrent updates on the PFS page, it is possible that two or more log records on the PFS page could be generated with the same previous page LSN. Since this violates the ARIES protocol [11], redo processing fails with an assert. This assert must be relaxed for PFS pages, which is as good as not utilizing the previous page LSN field in the log records for the redo operations on the PFS pages. This was deemed reasonable as the previous page LSN field on the log record was not used for correctness but used to detect stale pages on the disk (pages for which writes were issued to the I/O subsystem and a positive acknowledgement was received but the writes were missed). Stale pages in the database can be detected by other means like recording the LSN of the page at the time of flush/write in a separate data structure and validating the LSN of the page at read time against the value stored in the data structure.

One of the main challenges was to update the PFS page LSN in an atomic way. The PFS page header typically tracks the largest LSN of the update on the page. In the presence of concurrent updates to PFS pages, the updates to the PFS page header LSN must be synchronized.

```
struct LSN
{
   ULONG m_fSeqNo;
   ULONG m_blockOffset;
   USHORT m_slotId;
};
```

Figure 5. LSN Structure

The LSN on the page header is a 10-byte field that is not 16-byte aligned. Thus existing 16-byte interlocked operations could not be used to set the maximum value of LSNs by various threads. As shown in Figure 5, the 10-byte LSN value is comprised of three parts: 4 bytes for file sequence number, 4 bytes for block offset within the file and 2 bytes for the slot value. The most significant bit on the slot, *SLOT_MASK,* which is previously unused is now used to synchronize writes. Figure 7 provides the details of the *SetPFSPageLsn* API used to achieve this. Among the various threads attempting to update the PFS page LSN, the one to set the *SLOT_MASK* bit using interlocked API InterlockedBitTestAndSet gets to update the LSN in an atomic way while the other threads spin. Once done, the thread resets the *SLOT_MASK* to allow subsequent updates to go through.

```
Void SetPFSPageLsn(LSN targetLSN)
{
   Get the current LSN value on the page;
   If the current LSN > targetLSN
   {
      return;
   }
   Else
   {
      Obtain the lock on m_slotId using interlocked operations;
      {
            Get the current LSN value on the page;
            If Current LSN on page < targetLSN
             {
                Update the LSN value on the page;
             }
      }
      Release the lock on the m_slotId;
   }
}
```

Figure 7. Set PFS Page LSN API

```
LSN GetPFSPageLsn()
{
   do
   {
        read1 = Atomic read of first 8 bytes of LSN;
        read_s1 =Atomic read of last 2 bytes of LSN;
        Memory barrier to prevent compiler reordering;
        read2= Atomic read of first 8 bytes of LSN;
        read_s2 = Atomic read of last 2 bytes of LSN;
   } while (read1 != read2 || read_s1 != read_s2 ||
     SLOT_MASK_set_in_read_s1 ||
     SLOT_MASK_set_in_read_s2);

   Double read succeeded, return the value;
}
```

Figure 6. Get PFS Page LSN API

The counterpart of *SetPFSPageLsn*, *GetPFSPageLSN,* cannot update the fields on the page as the read API might be operating on read-only buffers. As such the read API, *GetPFSPageLSN,* cannot use the lock on the *SLOT_MASK* to synchronize with the writes. At the same time, simple read can result in a torn read of the 10-byte value or an invalid value due to the inflight update that sets the *SLOT_MASK*. To solve this problem, we came up with the lock-free algorithm described in Figure 6. The idea is to atomically read the first 8 bytes of the LSN, then atomically read the last 2 bytes of the LSN field and repeat the two reads once more with a memory barrier in between. If both the attempts provide the same valid value without the *SLOT_MASK* bit set, that value is returned; otherwise, the process is repeated. The double read ensures that only valid and consistent values are returned.

The update to the PFS page is done in an atomic way using existing interlocked update API so as not to interfere with the other threads. Apart from the LSN on the page, the LSN on the structures like DPC need to be maintained and they were updated using traditional interlocked operations too.

3.4 Checkpoint problem

The checkpoint process goes over all the DPC entries that have been dirtied in this cycle, reads the LSN from every entry and computes the minimum value as the Dirty Page LSN for the current checkpoint. With concurrent updates to PFS pages using shared latches, it is possible that thread T1 generated a lower LSN

for the PFS page while thread T2 generated a higher LSN for the same PFS page. T2 can race ahead to contribute the LSN to the DPC. Now, if checkpoint process inspects the dirty page LSN for this buffer, before T1 has had a chance to contribute its LSN to the DPC, it can result in the incorrect dirty page LSN computation by checkpoint, leading to recovery failure or corruption.

```
Void PrepareToDirty()
{
   If the page does not already have DPC, create one;
   Increment PENDING_COUNT in DPC in interlocked way;
   Other prepare to dirty aspects;
}

Void Dirty()
{
   Perform other Dirty tracking;
   Set the Dirty bit;
   Decrement PENDING_COUNT on DPC if needed;
}
```

Figure 8. API changes to solve checkpoint problem

Figure 8 describes the changes to buffer pool operations, *PrepareToDirty* and *Dirty,* to find a solution. A bit, *DIRTY_BIT,* is currently maintained in the DPC to indicate if the page corresponding to the DPC has been dirtied in the current checkpoint. This was traditionally set by the first thread that successfully generated the LSN on the page and executed the *Dirty* API on the DPC before releasing the latch on the page. To solve the checkpoint problem, a new counter, *PENDING_COUNT,* is added to the DPC structure. The checkpoint thread uses the *PENDING_COUNT* value on the DPC to determine if the LSN on the DPC is safe to consider or not. If the *PENDING_COUNT* is greater than 0, then it implies that there could be a thread in the system that has not yet contributed its LSN to the DPC and hence the checkpoint thread cannot get an accurate LSN value for the DPC yet. If that happens, the checkpoint thread releases the locks on the DPC structures and rechecks the DPC in a loop till *PENDING_COUNT* becomes 0, basically ensuring all inflight concurrent updates to PFS that started when *DIRTY_BIT* was 0 and have all contributed their LSNs to the DPC. The PFS update that can potentially block the oldest page LSN computation is expected to complete without running into deadlatches and hence not block the checkpoint process indefinitely.

PENDING_COUNT is maintained as follows: when a thread T1 is attempting to generate a log record on the PFS page, if the *DIRTY_BIT* is 0, it implies no other update has happened to the page in the current checkpoint, T1 increments the *PENDING_COUNT* value in an interlocked manner. T1 can decrement the *PENDING_COUNT* value only after contributing the LSN it generated to the DPC. Now, if T2 is attempting to generate a log record on the PFS page and the *DIRTY_BIT* is 1, it implies another thread T1 has already generated a log record, so T2 doesn't need to contribute to the dirty page LSN computation at all, so it no longer needs to increment the *PENDING_COUNT*. This helped resolve the checkpoint problem

3.5 Impact to Other Latch Protocols

The scheme to support concurrent updates using existing SH latches posed one latch compatibility issue. Section 2.7 mentioned that the UP latches are compatible with SH latches. With the scheme, updates on PFS pages are performed under SH latch. This violates the contract for requests in the system that need an UP latch, like the ones that update the PFS page header after scanning the entire contents of the PFS page. Such PFS updates may not necessarily have any other locks or latches held on other pages. In order to continue to support such operations, any UP latch request for PFS page in the system is automatically elevated to EX type to serialize with an update being performed using SH latch. This auto-elevation was made transparent to all UP latch consumers of PFS pages to prevent accidental misuse of UP latch on PFS pages.

3.6 Benefits of the Scheme

To summarize, the scheme involved updating existing algorithms to support concurrent updates under shared latches. This helped improve the throughput of PFS operations in the system and solved the decades-old PFS contention issues customers faced. Another major beneficiary of the scheme was the Constant Time Recovery (CTR) [13] feature in SQL Server. CTR is a novel recovery algorithm that depends on ARIES [11] but leverages row versions generated for MVCC for transaction rollbacks. CTR uses PFS pages for tracking stale versions: any database page modified by an active transaction is marked as having versions in PFS page. Then a background thread uses this information to identify pages that potentially have rows that need to be rolled back. Without the tracking in PFS, the background cleaner would have to perform an entire database scan to clean up the older versions of the rows in the database, which can lead to slow cleanup and inefficient space utilization in the system. This design choice increased the PFS usage and thus contention on PFS pages. Addressing this was necessary for shipping CTR.

4. EXPERIMENTAL RESULTS

Our experimental studies can be classified into four major categories – TPCC benchmarks, features like CTR, simulated allocation-intensive workloads and customer workloads. Let us look at the scheme's impact on each of these.

4.1 Impact on TPCC Benchmarks

Concurrent PFS updates mechanism eliminates exclusive latches on PFS pages but at the same time it introduces interlocked operations. We wanted to understand the impact of these changes on TPCC workload that doesn't necessarily experience PFS contention. The experiments were carried out on a workstation with 4 sockets, 40 cores (Intel® Xeon® Processor E7-4850, 2.00GHz) and 512GB of RAM. External storage consists of two 1.5TB SSDs, one for data and one for log. The TPCC workload running with concurrent PFS update feature did not degrade compared to running without. The variation between the runs was within the usual TPCC run-to-run variance of 1% or less. This showed that the new mechanism to update the PFS pages does not penalize the workload. Lock-free algorithms can potentially consume more CPU due to retries, interlocked operations and long chains in hash maps. The TPCC results prove that our lock-free scheme doesn't necessarily add any CPU overhead to the system.

In terms of memory overhead in the system, the scheme introduced additional fields in existing data structures. The additional fields (less than 100 bytes) are only created for the BUFs that correspond to PFS pages. For a 100GB database, if all the PFS pages are in buffer pool, then the additional memory overhead would amount to 156KB (approx. 100 bytes for each PFS page). The memory overhead is insignificant compared to the size of the database.

4.2 Impact on Constant Time Recovery (CTR)

On a database that has Constant Time Recovery (also known as Accelerated Database Recovery [7]) enabled, any insert/update/delete in the system needs to track the version bit in the PFS page. This significantly increases the number of PFS trips and thus the PFS contention in the system. We ran TPCC workload on a CTR enabled database with and without concurrent PFS update feature on the 4-socket machine. Table 1 summarizes the results of the experiments.

Table 1. Performance results of TPCC with CTR

Concurrent PFS update configuration	Transaction Throughput in TPMC	CPU Utilization
Enabled	787099.73	98.50%
Disabled	475423.40	45.30%

The throughput in TPCC increased nearly 65% when concurrent PFS scheme was enabled. The huge increase in throughput can be attributed to the increase in CPU usage from 45% to 98.5%. In the runs without Concurrent PFS Update, the PFS contention prevents the TPCC workload to utilize the CPU. From the performance numbers, it became clear that CTR feature benefits tremendously from the concurrent PFS update scheme. The PFS contention problem existed in SQL Server for over a decade. When CTR took a dependency on tracking versioning in the PFS pages, the bottleneck became more prominent, forcing us to think about a practical solution for the problem. While the PFS contention was fixed to address CTR performance issues, it solved the decades-old PFS contention problem that existed even without CTR.

4.3 Impact on Allocation-intensive Workloads

We ran various workloads, some comprising of temp tables, some with table variables and some with logon triggers. Each of them is known to run into PFS contention as reported by several SQL Server customers over decades. While these workloads are not strict benchmarks, they correspond to real workloads that customers have in production. With concurrent PFS update changes, we noticed a 5x to 20x increase in the throughput of some of these workloads. Let us look at one of the experiments in detail. Figure 9 describes the stored procedure that inserts top system messages into a temporary table. This stored procedure creates a temporary table into which the results are sent. Creation of the temporary tables involves allocation and when there are concurrent allocation requests, the workload runs into PFS contention. We leveraged OStress framework [12] that provides capabilities to run SQL workloads such as the above stored procedure.

```
CREATE PROCEDURE usp_WorkLoad
AS
BEGIN
  Select TOP 500 * into #TMP from master.sys.messages
  WAITFOR DELAY '0:0:0.20'
END
```

Figure 9. Stored procedure to repro PFS contention

```
ostress.exe -E -d"tempdb" -Q"exec usp_WorkLoad" -n400 -r300 -b -q
```

Figure 10. Stress workload commands

Figure 10 shows the command line parameters used to generate the workload. The workload was essentially 300 iterations of the above stored procedure being executed by 400 concurrent threads. The workload was run with and without the feature on a 2-socket Intel Xeon 2.4Ghz machine with 256 GB RAM and database files on SSDs. On a run without the concurrent update feature, the wait statistics showed that there were several requests waiting for update latch on the same PFS page.

Table 2. Performance results on 2-socket machine

Concurrent PFS update configuration	Average CPU utilization	Elapsed time (HH:MM:SS)
Disabled	10%	00:10:01
Enabled	32%	00:01:56

Table 2 summarizes the results of the experiments. With the feature, the throughput of the workload increased 5x, as it completed in under 2 minutes compared to 10 minutes without the feature. The throughput increased as the CPU utilization went up from 10% to 32%. Throughout the run, queries were run to identify the contenting latches and results were as expected. Without the feature, at any point in time, there were more than 25 UP latch requests on the PFS page, while with the feature enabled, PFS latches did not show up in the wait statistics at all. The workload was also run on 8-socket Intel Xeon CPU E7-8890 @2.2Ghz with 6TB RAM and database files on SSDs.

Table 3. Performance results on 8-socket machine

Concurrent PFS update configuration	Average CPU utilization	Elapsed time (HH:MM:SS)
Disabled	1%	00:11:28
Enabled	83%	00:04:57

Table 3 summarizes the results of the experiments. With the feature, the throughput of the workload increased 2x, as it completed in under 5 minutes compared to 12 minutes without the feature. The throughput increased as the average CPU utilization went up from 1% to 83%. Throughout the run, queries were run to identify the contenting latches and results were as expected. Without the feature, at any point in time, there were several UP latch requests on the PFS page, while with the feature enabled,

PFS latches did not show up in the wait statistics at all. The results from the 8-socket machine confirm that the algorithms scale up too.

4.4 Impact on Customer Workload

We had a customer willing to test out a workload that constantly ran into PFS latch contention. Due to the restrictions imposed by the customer, we cannot reveal the details of the workload. However, the workload essentially ran into PFS contention due to amount of allocations in the system. We obtained the workload from the customer and ran it with and without the concurrent PFS update feature. The workload was run on a 2-socket Intel Xeon 2.4Ghz machine with 256 GB RAM and database files on SSDs.

Table 4. Performance results of customer workload

Concurrent PFS update configuration	Throughput (batch req/ sec)	%CPU	Top wait types % (of total wait time)
Disabled	350	35%	PAGELATCH_UP - 38.52% SOS_WORK_DISPATCHER - 59.33%
Enabled	1035	63%	ASYNC_NETWORK_IO - 59.15% SOS_WORK_DISPATCHER - 39.95%

Table 4 summarizes the results of the customer workload. The customer workload benefited from the feature in the same way as the temp table workload. The throughput of the customer workload increased 3x due to the increased CPU utilization. The top wait types were captured during the workload execution. As expected, in the run without the feature, the top wait was the update page latch, *PAGELATCH_UP*, which typically appears for PFS/GAM pages. In the run with the feature enabled, the update page latch disappeared.

We did not perform any micro benchmarks on the lock-free algorithms used in the various buffer pool management and checkpoint management aspects as these are not structured as unit testable components. It was very hard to tease them apart and test them in isolation to see if the lock-free algorithms have an inflection point or not.

5. FUTURE WORK

The work has been currently implemented for PFS pages and enabled by default in both Azure SQL DB and SQL Server 2019. More than five million databases in Azure SQL DB are running with this feature. With this in place, PFS pages are no longer the source of contention in SQL Server. Some of the allocation workloads that ran into PFS contention earlier, now run into GAM (Global Allocation Map) and occasionally IAM (Index Allocation Map) page contention. The next step is to extend the work to GAM and IAM pages. This would eliminate the system page latch contention as part of the allocation and speed up other workloads.

6. APPLICABILITY OF THE WORK

While the paper talks about how the contention on PFS pages in SQL Server was solved, the approach can be applied to other database systems that have metadata pages with fixed size rows, whose updates are derived updates of the actual rows. PostgreSQL [6] uses metadata pages called Free Space Map (FSM) [14] to track free space in the system. If the space usage on the database pages increases with workload, then the free space map needs to be updated. If the space on the database pages is reclaimed due to cleanup of older versions by the process called vacuum [15], then the free space map needs to be updated as well. The information stored in FSM is derived from the operations on the data pages, just like in SQL Server. It is important to store the information in FSM as it reduces the amount of time spent to identify pages that have free space. FSM pages can run into contention when concurrent threads in the system attempt to update the same FSM page. We believe updates to the FSM pages can benefit from the scheme and help improve the throughput of certain workloads in the PostgreSQL database as well.

7. RELATED WORK

We investigated if other databases or storage systems encountered similar problems and how they addressed the issues. [10] describes some of the flexible space management optimizations in database systems like System R [1] and DB2 [9]. A few system pages per file called the free space inventory pages (FSIPs) in System R and space map pages (SMPs) in DB2, track the space management in the data pages.

These system pages are essential to improving the storage utilization and increasing the levels of concurrency in the system. The FSIP/SMP pages are frequently updated due to the updates to the data pages that they are responsible for. While there are optimizations to not update the FSIPs/SMPs for every data operation, the latching of these pages causes contention. [10] mentions that FSIPs/SMPs are normally the hot spots and to alleviate latch contention, they used atomic update instructions to update the relevant FSIP entry, basically supporting concurrent updates to such pages. They updated the page_LSN fields in the FSIPs/SMPs to maintain the highest LSN too. To achieve recoverability of FSIP/SMP pages, the updates were redo only log records without any undo semantics. This was possible with special logic around the order of the log records generated – forward processing of transactions generated the log records for the pages corresponding to FSIPs first and then generated the update to the FSIP, undo processing of transactions generated the FSIP log record if needed prior to generating the undo log record of the data pages. The fact that the data page is latched exclusively when the corresponding FSIP is being updated both during forward processing and undo processing helped avoid latching or locking FSIPs to provide higher concurrency.

DB2 page-level locking led to contention on index leaf pages. Before solving that using record-locking, DB2 implemented mini-page locking [11] by dividing the index leaf page into smaller pages. Each mini-page had its own LSN besides the LSN for the whole leaf page. The page LSN is set to the max LSN of the mini-page LSNs. This approach has the disadvantage of taking up extra space for mini-page LSNs. We could not necessarily adopt a similar scheme for PFS pages as the PFS page format is fixed with no additional space for LSNs for different sections of the page.

There are other studies in the literature that work around the latency issues of write-ahead-logging protocol of ARIES using a different approach. Segment-Based recovery [16] addresses the latency among components in write-ahead-logging by removing the two core assumptions – pages are the unit of recovery and maintenance of the LSN on each page. They achieve the above by using segments as the units of recovery. The segment can be within a single page or span multiple pages. They delegate the locking of the segments to the application layer and thus avoid

concurrent updates to the same segment. The scheme assumes sector writes to be atomic and handle torn pages by blind writes. Segment-Based recovery eliminates the need to maintain the LSNs on the pages by employing a write-back-caching mechanism – updates to a page only affect the cached copy while the write is performed when the higher level object the page represents is evicted from the cache. This allows the updates to the same page to be re-ordered, enabling higher concurrency in the system. With the need for page latches eliminated, the scheme allows for calls to buffer manager and log manager to be asynchronous and increases the throughput of the system. The paper describes changes to the allocation algorithms to support safe rollback of the transactions by avoiding unrecoverable states. The paper also proves the correctness of the algorithms and demonstrates achieving 20x speed up in some cases compared to page-based approach. While Segment-Based recovery addresses the latency concerns and helps improve the throughput of the application workload on ARIES based system, it requires significant re-write of the underlying system to support the high level of concurrency. It also pushes the responsibility of locking to the application layer, which can potentially lead to bad performance in some cases. Our approach is similar to the concepts used in Segment-Based recovery, in that we attempt to leverage the write back mechanisms to lazily update the LSN on the pages and leverage the higher level locking of the PFS bytes to support concurrent updates to the PFS pages. We achieved this without having to substantially re-write the mature storage engine.

To summarize, we believe that concurrency of pages with fixed-size rows can be significantly improved by adopting lock-free algorithms for such pages.

8. CONCLUSIONS

Based on the experimental results, the concurrent PFS update protocol did not add any performance overhead to the system. It significantly improved the throughput in certain workloads. It's on by default in Azure SQL DB and is also supported in SQL Server 2019 product. Even though the concurrent PFS changes were mainly scoped to buffer pool management aspects and PFS page update API, identifying all the steps that needed to be made lock-free, implementing them and proving correctness was a long journey for the team. Coding lock-free algorithms is hard but rewarding too. The work showed that taking a closer look at a particular subsystem that is based on lock-based algorithms and changing them to be lock-free can help in significantly boosting the performance of the entire system. Contrary to claims in [5], this work proved that it is possible to leverage lock-free algorithms and improve the performance of certain class of operations for some important class of pages in disk-based database systems. While techniques like transactional memory described in [2] could potentially be used in some cases, we haven't explored the option yet and might consider them in the future.

9. ACKNOWLEDGMENTS

We thank Peter Byrne, George Reynya, Wayne Chen, Cristian Diaconu, Chaitanya Ravella, Zhuan Chen and other colleagues for their invaluable advice and help in converting some of the lock-based algorithms in buffer pool page management protocol to lock-free algorithms. We also thank the reviewers for providing critical feedback that helped us improve the paper in many ways. We appreciate C. Mohan meeting with us virtually to inform us of similar work done in System R and DB2 to handle contention on FSIPs. We thank our leadership team for sponsoring such a risky and novel approach.

10. REFERENCES

[1] Astrahan, M., et al. System R: Relational approach to Data Base Management, ACM Transactions on Database Systems, Vol. 1, No. 2, June 1976.

[2] David T, Guerraoui R, Trigonakis V. Everything you always wanted to know about synchronization but were afraid to ask. Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. November 2013. Pages 33–48.

[3] Delaney, K., Randal, P. S., Tripp, K. L., Cunningham, C., Machanic, A. Microsoft SQL Server 2008 Internals. Microsoft Press, Redmond, WA, USA, 2009.

[4] Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum. The LRU–K Page Replacement Algorithm for Database Disk Buffering. Proceedings of the 1993 ACM SIGMOD Conference. Pages 297-306.

[5] Jose, M. F, Daniel, J. A. Latch-free Synchronization in Database Systems: Silver Bullet or Fool's Gold. CIDR 2017.

[6] Internals of Postgre SQL. http://www.interdb.jp/pg/pgsql01.html.

[7] Microsoft, Accelerated Database Recovery. https://docs.microsoft.com/en-us/azure/sql-database/sqldatabase-accelerated-database-recovery.

[8] Microsoft, Under the covers: GAM, SGAM, and PFS pages. https://techcommunity.microsoft.com/t5/SQL-Server/Under-the-covers-GAM-SGAM-and-PFS-pages/ba-p/383125.

[9] Mohan, C. IBM's Relational DBMS Products: Features and Technologies, Proc. SIGMOD International Conference on Management of Data, Washington, May 1993.

[10] Mohan, C., Haderle, D.J. Algorithms for flexible space management in transaction systems supporting fine-granularity locking. In: Jarke M., Bubenko J., Jeffery K. (eds) Advances in Database Technology — EDBT '94. EDBT 1994. Lecture Notes in Computer Science, vol 779. Springer, Berlin, Heidelberg.

[11] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., Schwarz, P. M. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM TODS, 17(1):94–162, 1992.

[12] OStress framework. https://support.microsoft.com/en-us/help/944837/description-of-the-replay-markup-language-rml-utilities-for-sql-server.

[13] P. Antonopoulos, et al. Constant Time Recovery in Azure SQL Database. PVLDB, 12(12): 2143-2154, 2019. DOI: https://doi.org/10.14778/3352063.3352131.

[14] PostgreSQL, Free Space Map. https://www.postgresql.org/docs/9.2/storage-fsm.html.

[15] PostgreSQL, Vacuum. https://www.postgresql.org/docs/9.1/sql-vacuum.html.

[16] R. Sears and E. Brewer. Segment-Based Recovery: Write-ahead logging revisited. PVLDB, 2(1):490–501, Aug. 2009.