BTrim – Hybrid In-Memory Database Architecture for Extreme Transaction Processing in VLDBs

Aditya Gurajada
Hyderabad, India
adityagurajada@yahoo.com

Dheren Gala
SAP Labs
Pune, India
dheren.gala@sap.com

Fei Zhou
Chicago, USA
feizhou111@gmail.com

Amit Pathak
SAP Labs
Pune, India
amit.pathak@sap.com

Zhan-Feng, Ma
SAP China Co. Ltd.
Shanghai, China
zhan-feng.ma@sap.com

ABSTRACT

To address the need for extreme OLTP performance on commodity multi-core hardware supporting large amounts of memory, SAP ASE is re-architected to tightly integrate an In-Memory Row Store (IMRS) within the existing database engine. The IMRS is both a store and a caching layer to host "hot" rows in-memory, in a row-oriented format. The IMRS is an extension to the traditional buffer-cache which deals with data in a page-oriented storage format (referred to as the page-store). Data in individual tables marked as IMRS-enabled can be fully memory-resident or can straddle the page store and the IMRS. Cold data in the IMRS is organically identified, harvested, and "packed" back to the page store. SQL statements and transactions can access data transparently from both stores for the same table. All Transact-SQL capabilities and language constructs are supported with no application or stored procedure code changes required. Full durability for in-memory data is provided, including support for backup and restore of database archives and periodic transaction dumps.

The high-level system design supporting this architecture, along with experimental results and performance benefits is presented.

PVLDB Reference Format:
Aditya P. Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, Zhan-Feng Ma. BTrim – Hybrid In-memory Database Architecture for Extreme Transaction Processing in VLDBs. PVLDB, 11(12) : 1889-1901, 2018.
DOI: https://doi.org/10.14778/3229863.3229875

1. INTRODUCTION

OLTP performance at a low TCO has been a key differentiating offering for the SAP ASE database server. Recent product enhancements [21][22][23] have focused on delivering Extreme OLTP (xOLTP) performance and extreme scalability on commodity multi-core hardware.

Many commercial offerings [2][24][26] in this area require a database or a table to be fully in-memory. With ever-increasing transactional data being generated continuously, customer database sizes tend to grow faster than memory capacities, however OLTP transactions tend to actively use only about 20% of the data [20].

To better understand the run-time contention issues experienced by transactional workloads, we deeply analyzed the execution metrics of two OLTP workloads, by a technique we call transaction log mining. First, we studied profiles from transaction logs of a large Sales Distribution (SD) benchmark, which has become one of the standard for SAP's platform partners. Second, we also analyzed the transaction logs and footprint from a TPCC benchmark. Both benchmarks were executed on an SAP ASE server running with traditional page-storage and the buffer cache. For this investigation, the benchmark configuration, and machine details are not relevant here as we only wish to show how transaction log mining helps us understand concurrency issues and profile of OLTP transactions.

Results from SD Benchmark Analysis: In our internal testing, we analyzed the logging traces of a 35+ minute 8000+ SD users run run on a 120GB page-store database, with about 50GB of data. The transaction log examined was 20GB, containing 1.3+ million transactions generating 90+ million log records. There were less than 0.002% transaction aborts. The average elapsed time of the transactions was 36 milliseconds. The distribution of log records per transaction seen in this profile is given below.

Table 1. # of log records across all transactions in SD Benchmark

Minimum	Average	Std. dev	Maximum
1	63	107	46,496

This shows that although there are a few big transactions generating a large number of log records, for the most part in this workload, the average size and the time-span of the transaction is small.

For this workload, the following Figure 1 shows the distribution of number of transactions by number of log records generated (not including the begin / end transaction log records).

We note from Figure 1 that about 41% of all transactions generate 10 or fewer log records, 82% of all transactions generate 80 or fewer log records. A very small percentage of transactions generate very large numbers of log records (say, over 1000 or more).

We then examined the log activity to identify active tables. We considered 0.5% of the log activity as a low-watermark threshold to qualify a table as being an "active" table in the workload. This translates to about 500,000 log records. A count of tables with about >= 0.5% log activity identifies 22 tables. The remaining activity

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229875

was spread over the remaining ~180 tables in the database where each table had activity less than 0.5% of the total work load. We

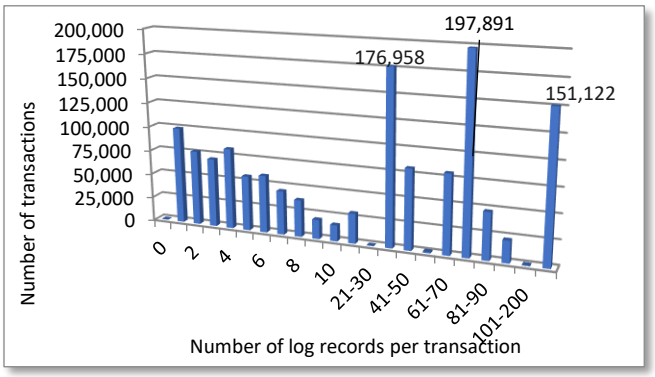

Fig. 1. Number of log records / transaction in SD Benchmark

interpret this to mean that the bulk of the SD benchmark activity is directed towards a small set of tables, which are very active, and a larger set of tables which are comparatively less active.

Identifying conflicts through log records: For the same workload, we analyzed the old and new database time stamps (DBTS) in the log records, matching them with the affected page ID to infer the degree of conflict encountered due to transactions concurrently active on pages. Every change to any database page generates a monotonically increasing DBTS value.

Figure 2 shows a pictorial representation of "events", i.e., the log records, generated by two concurrently executing transactions, affecting some set of pages.

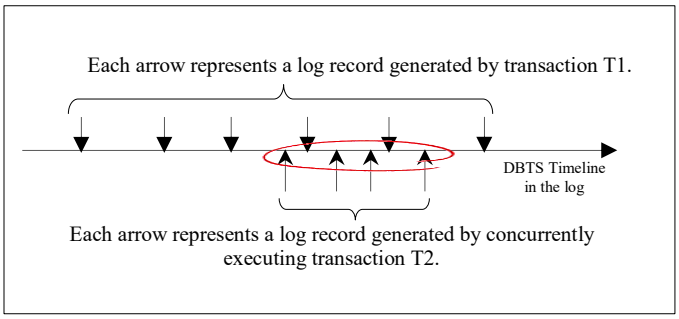

Fig. 2. Timeline representation of changes done by concurrently executing transaction

We mined the affected page IDs, and the old / new DBTS caused by the "event" from the log record to know the time-span when the change was done affecting a said page. If a transaction T1 has updated a page, and before it has committed, if another transaction T2 updates some other row on the same page, this results in a page-conflict to T1.

Logging in the page store [23]: To understand the impact of page conflicts, we diverge slightly to describe at a high-level how buffer management and transaction processing is performed in SAP ASE. For every change affecting pages, in anticipation of the transaction committing, log records are generated, in-line with the change, in every task's private cache of log records, *aka*, Private Log Cache (PLC), and then the PLC contents are flushed to the log page at commit-time. (We refer to this scheme of generating log records through the life of the transaction as drip logging.) PLC is provided to reduce the frequency of writing log records and instead to flush a batch of log records for one transaction to the log. This results in reducing the contention of access to the last-log page, which typically is the main bottleneck in a database system. In order to facilitate Write-Ahead logging (WAL), the modified buffers are pinned to the PLC. As described in [23], page conflicts cause buffer unpinning, resulting in prematurely flushing of the tasks' PLC to the database log. These arise due to updates by concurrent transactions to different rows that may simply be residing on the same data buffer. Space allocation in syslogs, which is done as part of transaction commit, causes spikes of performance degradation. Finally, the shared last-log page insertion point for new log records appended to syslogs by concurrently committing transactions is an omnipresent performance and scalability bottleneck [23].

In the work load analyzed, we found the following metrics:

Table 2. Distribution of page-conflicts in SD Benchmark

	Item	Value	Comment
(a)	Total # of transactions processed	1,374,232	
(b)	# of transactions of type "DML%"	804,697	58% of (a)
(c)	# of distinct pages *not* involved in conflicts	495,892	Split nearly equally, 54-46%, across the workload
(d)	# of distinct pages involved in any conflict	430,921	
(e)	# of "events" (log records) generating conflicts	5,524,349	4X (a) >6X (b)

Using the overlapping DBTS analysis, we observed the folowing distribution of conflicts per affected conflicted-page.

Table 3. Distribution of conflicts per modified page

Min	Avg	Stddev	Max
1	12	111	45,943

This implies that a very large number of pages ran into a pretty steep number of conflicts. On an average, across all pages accessed in this benchmark [sum of (c) and (d) in Table 2], each page can be expected to run into an average of 6 conflicts. Each conflict can result in concurency issues.

We also do note that identifying page-conflicts through log record mining does not capture read activity on tables; only modifications. Therefore any inference from this conflict-metric is only meant to be a lower-bound on potential contention issues. The scalability issues due to shared access to pages is further examined in the section on Performance Evaluation (Sec. 5.1.1).

We performed the same log mining on TPC-C transaction log and concluded that most transactions have a small foot-print of average 18 log records, completing in less than 300 milliseconds.

1.1 Our Contributions

Our motivation behind this work is to build a transaction processing engine which can deliver scalable and high performance using in-memory processing, and overcomes the contention issues in a page-oriented storage model, described earlier. Historically, some of these scalability limitations have been improved in recent versions of SAP ASE, as noted in [23], but there is still scope to deliver enhanced OLTP performance using in-memory computing.

In the **BTrim** architecture, which stands for "**B**usiness **Tr**ansactions **I**n-**M**emory", we provide a hybrid storage model where the existing buffer-cache based page-store layer is **tightly-integrated** with an **In-Memory Row Store (IMRS).** The IMRS is used to host "hot" data for business transactions. Full indexing support is provided for data that can reside either in the page-store or in the IMRS. Concurrent accessors to rows that simply share the data page no longer run into contention [1][9] when the data is decoupled from the page and stored as individual rows in-memory.

For VLDBs much larger than available memory, the buffer cache already provides a "working set" of hot pages. In this layout, the IMRS is seen as providing an extended caching layer for "hot rows" on top of the buffer cache, for improved performance. If the main memory is large enough to host the entire VLDB (i.e. fully memory-resident database), we still run into the contention issues described earlier due to the page-oriented storage, and logging overheads. In this layout, providing any extra memory to the buffer cache yields no further gains. However, this extra available memory can be deployed to the IMRS, where "hot" rows are processed without being hindered by page-oriented contention issues.

Nomenclature: Existing databases using traditional page-based (buffer-cache resident) storage are referred to as **page-store databases**. In the BTrim architecture, individual databases configured to use the IMRS are referred to as **IMRS-enabled database**(s). In an IMRS-enabled database, individual tables altered to use the IMRS for storing "hot" data, supporting data row caching, are referred to as **IMRS-enabled tables**. An IMRS-enabled database can *still contain* ordinary tables, i.e., not enabled for data row caching, and such tables are referred to as **page-store tables**. Transactional data typically goes through 4 stages in its life-cycle – Insert, Select, Update, Delete – and this collection of statements is referred to as **ISUDs**. **Pack** refers to an internal operation that removes cold rows from the IMRS and stores them back in the page-store. On a multi-core system, SAP ASE can be configured to run with multiple OS native threads. Designs to overcome multi-core scalability issues in this configuration are referred to as **thread-local** techniques.

Our contributions from this work are as follows. These enhancements are deeply integrated within the core SAP ASE DBMS engine [24] and can be deployed without destabilizing existing installations and without requiring any application rewrite. The IMRS provides the backbone for new performance-oriented capabilities added to the ASE product. A new technique called Data Row Caching (DRC) is added to the product. The storage choices incorporate Information Life Cycle Management (ILM) [19] which is intrinsic to certain kinds of transaction workloads. High-performance is delivered by storing the rows either in-memory or on the disk-based page-store. Hash-indexes spanning data in the IMRS are provided as a fast-path accelerator for point queries using a unique index access method. Full durability is provided to the data modified in the IMRS in a new logging system that overcomes many of the shortcomings of the existing transaction log. Replication is fully supported for IMRS-enabled tables.

The BTrim architecture is also the foundation for supporting industry-standard snapshot isolation using in-memory versioning offering MVCC capability; a functionality that is now available in the ASE capability set.

The rest of the paper is organized as follows. Section 2 presents the overall architecture and key design choices we made along the way. Section 3 discusses internal changes to how index scans work. In section 4, we discuss the durability layer of the IMRS, another key contribution from this work. In section 5, we present performance results from in-house experiments using micro-benchmarking and high-end OLTP workloads. In section 6, we study related work in this area, and finally present our conclusions.

2. BTRIM ARCHITECTURE

The BTrim architecture extends the capabilities of the existing engine to deliver enhanced performance, even while maintaining full compatibility for existing databases *and* T-SQL constructs. It is neither a separate product nor a bolt-on.

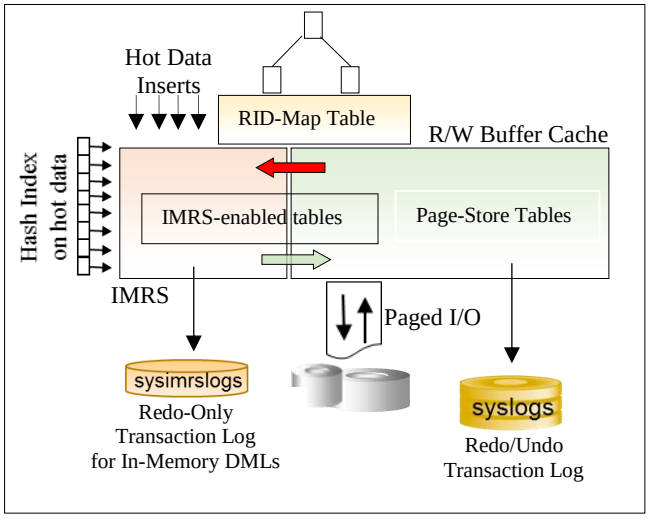

Fig. 3. BTrim architecture for In-Memory Processing

Figure 3 shows the salient components of the new architecture and how data is organized and accessed. Existing storage components such as the disk-based store, page-formatted buffer cache, and the redo/undo transaction logging, referred to as syslogs in the product, remain **unchanged**. The database storage space and buffer cache is "extended" with the In-Memory Row Store, used to *both* **cache existing rows** *and* **store new data**. In Fig. 5, the box labelled IMRS-enabled tables that straddles the buffer cache and the IMRS reflects the behavior that, over time, **only some part of a table may be in-memory**. The arrows represent movement of hot / cold rows to / from the IMRS. It is not necessary for an entire table to be in-memory. Access methods transparently locate the row from one of the two stores using internal scan methods, which are discussed in later sections. A single ISUD statement, and consequently any transaction, may access some rows from the IMRS and some from the page-store, without any limitations. A single transaction can also access / change data from IMRS-enabled and page-store tables without any restrictions.

Page-based BTree indexes continue to be supported in this architecture, however, they are enhanced to transparently scan rows either in the page-store or in the IMRS. BTree indexes are further enhanced to use a latch-free concurrency control [21] mechanism for enhanced performance. In-memory hash-indexes, shown to the left of the IMRS, span only the in-memory rows of a table.

ILM – Row-level Data Ageing: A key differentiating feature of the BTrim architecture is a tight integration with the data life-cycle patterns for transactional workloads to assign efficiently in-memory storage to "hot" rows [19]. ILM is a set of strategies, rules and heuristics designed to efficiently use IMRS resources for frequently accessed "hot" data rows. The decision to use the IMRS for row caching leverages run-time access patterns to optimally store "hot" rows in the IMRS. When hot data becomes less frequently used in the IMRS, it may be replaced back in the page store, by one or more background Pack threads.

Persistence and durability to data updated in the IMRS is provided through a new redo-only logging device, named in the Fig. 3 by its system catalog, **sysimrslogs**. Sysimrslogs is the logging counterpart for the existing redo / undo transaction log, syslogs; the latter used to persist changes to data in the page-store. The vertical arrows from the IMRS and the page-store tables to the two logs, respectively, indicate that changes to in-memory rows are logged in sysimrslogs and those to page-store rows in syslogs.

A key new sub-system supporting the IMRS is a high-performance fragment-memory manager which is highly optimized for best-fit memory allocation and reclamation on multiple cores. The memory manager supports allocation of fixed-length fragments, such as those needed by metadata structures, and variable-length fragments, such as those needed by row versions. Efficiency is achieved by managing memory fragments in multiple like-sized buckets, load-balanced and distributed across multiple cores.

Multi-threaded, non-blocking Garbage collection (IMRS-GC) is provided to efficiently reclaim memory from older versions without affecting transaction performance. Pack is a new sub-system that in cooperation with the memory manager and based on ILM-rules, relocates cold data out of the IMRS to the page-store. Multiple pack threads are provided to guarantee stable memory utilization. The details of the Pack/ILM sub-system are presented in [19].

Row Identity: In-memory rows are identified uniquely similar to existing page-store rows; i.e. using a (page-ID, row#) pair. The fabricated RID for inserted rows is referred to as a Virtual RID (VRID). A RID is easily identified as a physical RID (for page-store rows) or a VRID by examining the row# component, which, for VRIDs, will be beyond the valid value for page-store rows.

An ISUD accessing data rows may find the row either in the page-store or in the IMRS, however logical row protection is provided using row-locks on the RID. We did not change the existing pessimistic row-level locking when accessing data rows from an IMRS-enabled table. ANSI Isolation levels, which are currently implemented based on the lock manager, are, thus, fully supported on IMRS-enabled tables. These design choices thereby ensure no changes to application behavior.

Minor DDL extensions are provided to configure new or existing databases with IMRS resources, such as the IMRS-cache and sysimrslogs and to create new IMRS-enabled tables or to alter existing tables to make them IMRS-enabled. **All table schemas and properties**, such as identity columns, unique and non-unique indexes, triggers and rules, defaults, check constraints etc., are **fully supported** for IMRS-enabled tables.

2.1 ISUDs and Data Row Caching

Row caching is a table-level property that can be enabled for datarows locked tables in an IMRS-enabled database.

Each in-memory row has an immutable header structure, IMRHDR, off which hang multiple in-memory versions, IMRVERS, created by updates. Figure 4 shows a pictorial layout of an in-memory row, with multiple versions created by updates to a row. Each row version may be of differing sizes when updates cause the row size to change. Once minted, the row version memory is immutable and only used for reads (or memory reclamation). The memory for older versions is periodically reclaimed by the GC threads.

The older versions of all rows are read-only and are used to support snapshot isolation. This shared design of row layouts with versioning is chosen so that for the same IMRS-enabled table we can support ANSI isolation levels (by acquiring a lock on the latest version of the row), and to also support timestamp based snapshot isolation scanners to locate their "seen" version.

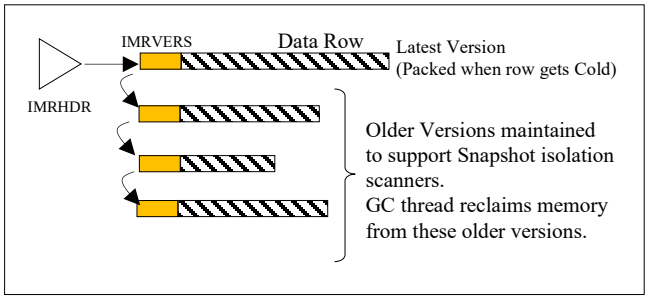

Figure 4. Layout of in-memory structures for one data row

Three types of rows can be stored in the IMRS for each data partition in a table – inserted, cached and migrated rows, as discussed below. All strategies to maintain such rows, including memory usage, row count and other metrics are all tracked finely per the row type, using thread-local, contention-free counters.

Inserts are performed directly in the IMRS and such rows are referred to as ***inserted rows***. Newly inserted rows have no foot-print in the page-store. By directly inserting new rows to the IMRS, all the overheads associated with page-oriented storage are removed. Costs of page allocation is exchanged with the cost of memory allocation, which is highly-optimized and cheaper in contrast.

Selects of inserted rows are returned directly from the IMRS, without needing to access the page store as the row is found only in the IMRS. BTree indexes have the intelligence to locate the latest version of rows from the IMRS using a mapping table called RID-map lookup, a technique which is described further on.

Selects driven by fully-qualified unique index scan returning a single row (point-queries) from the page-store portion of an IMRS-enabled table [usually] cache the row to the IMRS. Such rows are referred to as ***cached rows***. As point-query access by unique index is commonly seen in OLTP workloads, the design principle is that by caching such rows in the IMRS, subsequent scans to rows cached in-memory, done without any page-latching concurrency protocols, will be more efficient.

Cached rows are initially read-only rows but can subsequently be updated in-memory if required. Only the in-memory version of the cached row is updated, whereas the page-store image may remain stale. Updating cached rows only in-memory again avoids page-concurrency issues [1] that will arise when multiple rows residing on the same data page are updated frequently and concurrently.

Updates of existing rows residing on the page-store are usually performed in the IMRS by first re-locating the rows to the IMRS. This process is referred to as *row migration* and such (updated) rows are referred to as ***migrated rows***. For example, point updates qualifying a single-row with a fully-qualified unique index will always cause row migration. Such updates are very common in transactional systems, and, thus, are optimized to be performed in-memory. The trade-off made here is to optimally use IMRS memory for "hot" rows, thereby single-row update statements cause migration. To avoid flooding the IMRS with rows occasionally updated by a single batch-update statement, not all but only some number of the affected rows are migrated.

Updates producing new row-versions do not cause the rows' identity (RID or virtual RID) to change. Only the in-memory row-image is updated with a new row version.

Updates (and deletes) to in-memory rows produce a new version efficiently slotted-in to this chain off the header using CAS. This scheme has several advantages. The expectation is that concurrent transactions are busily chasing after non-intersecting rows, each of which goes through this update sequence on different cores, without causing scalability issues. Another key aspect of our design is that scanners of these row versions do not need any form of latching or reference count to provide stable access to the row versions. Hence, multiple readers of "hot" data on different cores also do not run into cache-line invalidation issues.

Deletes of inserted rows is performed entirely in-memory, producing a stub delete-version as the post-image of the delete. **Deletes of migrated rows** is performed similarly, with the additional work of deleting the data row from its page-store location deferred to background Pack processing.

2.1.1 In-Memory Row Formats and IMRS Garbage Collection

For compressed tables, rows are stored in a compressed form on the page. This results in additional costs of decompression while reading, and compression, following an update. To avoid these overheads, we chose a design of storing in-memory rows in an uncompressed format. This may result in an increased memory consumption for row versions, in exchange for improved run-time performance. Finally, when the cold rows are packed to the page-store, they will be compressed. This scheme retains the storage cost gains of data compression in large tables while improving the performance of accessing uncompressed rows from the IMRS.

IMRS-GC and transaction management collaborate to reclaim memory from older versions that are no longer "seen" or required in the system. In our experiments, we have seen that for TPCC-like OLTP workloads generated by 240 users, about 2 IMRS-GC threads were able to keep up with this work of reclaiming older version memory. Any overheads due to GC activity is greatly amortized over the work done by multiple user connections.

2.1.2 Information Life Cycle Management and Pack

Various Information Lifecycle Management (ILM) schemes to retain only the hot data in memory and colder data in traditional page-store are described in detail in [19]. The hotness of data is measured using frequency of access, and the contention caused while accessing it through page store. While the high-level characteristic of partitions is considered, the decision on whether to have data in IMRS or page store is made during every ISUD operation on a row. When memory utilization crosses a threshold, the cold data is moved back to page store using the Pack operation. To facilitate harvesting cold rows easily, rows are maintained using relaxed LRU queue based on their hotness. A novel technique referred to as Timestamp Filter (TSF) is also used for determining coldness. The ILM checks, queue management and operations such as TSF are optimized to have minimal performance impact to user transactions. Our experiments show that, with the help of ILM techniques, the performance gains of in-memory processing are realized without requiring that all data be in-memory [19].

3. INDEX MANAGEMENT

In-memory data is fully indexed and accessible using existing page-based BTree indexes. All index properties, such as unique and non-unique indexes, primary key constraints, identity columns in indexes, Latch-Optimized BTree indexes [21], compressed indexes, local and global indexes on partitioned tables are fully supported for indexes on IMRS-enabled tables. All index operations such as inserts, deletes, splits, shrinks etc. continue to be logged in the page store; i.e. syslogs.

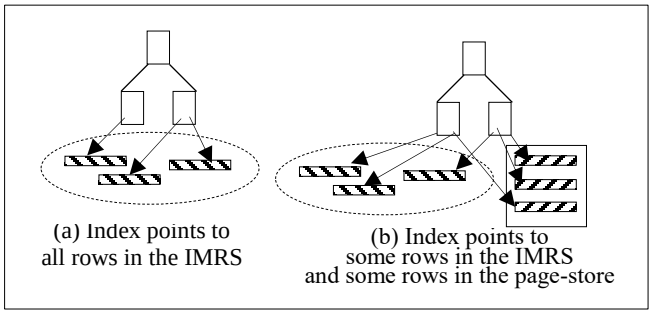

Figure 5. Index spanning rows in IMRS or in page-store

We evaluated the design of changing the leaf-row formats to contain the in-memory row address, but this would lead to migration issues for existing indexes. The leaf level of the index continues to hold {key-values, RID}. The RID in the index leaf row is a Virtual RID (VRID) for inserted rows, while it is a physical RID for page-store rows. This allows existing databases to easily migrate to the new storage architecture without having to unload – reload the data or perform an index rebuild before altering page-store tables to become IMRS-enabled tables. Figure 5 shows two forms of a BTree index spanning rows in the two stores.

For access to in-memory rows, index scans use a RID-Map lookup table to map the RID to a memory address. Every row in the IMRS is guaranteed to have an entry in the RID-map table. Figure 6 shows the multi-level segmented-array layout of the RID-map table.

Each level of this table maps to contiguous chunks of pages in the database's space map. As the IMRS is designed to hold "hot" active rows, only some small percentage of rows are expected to be in the RID-Map table. This is a sparse table blossoming out to allocate and fill-out sub-arrays at each level holding pointers to sub-arrays at the next level. Over time, as rows are removed from the IMRS,

the structures at each level are de-allocated, and this table shrinks to maintain sparseness. For rows that are not in-memory, the RID-Map probe typically returns early without having to traverse non-existent sub-levels. This table is managed using very efficient lockless lookups and CAS-based updates to grow and shrink the memory at different levels. RID-map probes are not expected to be a performance bottleneck, which has also been validated in our internal benchmarking. Two RID-map lookup tables are provided – one holding addresses of inserted rows, and the other for migrated and cached rows. The RID itself is self-identifying as a VRID or a RID, simplifying which RID-map table to lookup.

Inserted rows only exist in the IMRS. Therefore, the RID-Map lookup will always return the row's in-memory address. The handling of migrated or cached rows is slightly different. For every access to rows in the page-store, an additional probe in the RID-

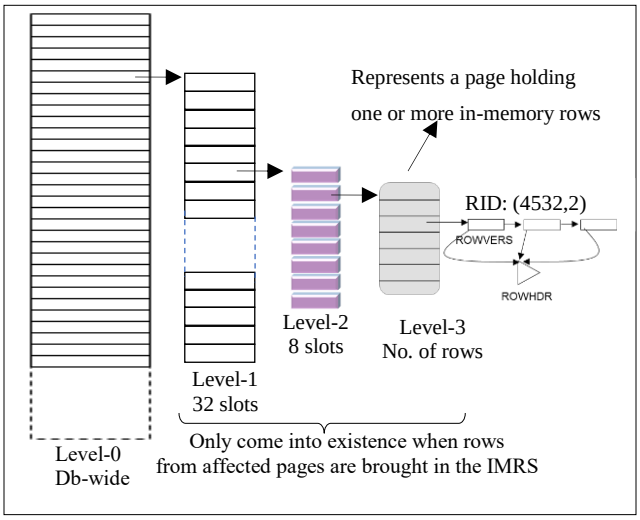

Figure 6. RID-Map Table: Multi-level segmented array access

Map lookup table is needed to ascertain if the (latest version of the migrated or cached) data row may be in the IMRS. If the probe fails to find the RID, then the latest copy of the row is guaranteed to be on the page-store. Scan reverts to normal page-based scan methods.

3.1 Hash-Cached Indexes

In highly-concurrent OLTP workloads driven by multi-level BTree, index scans are known to encounter scalability issues due to access to index pages from different cores. BTree index scan takes significant part of the overall data access time. Latch-Free BTree [21] indexes improve this situation to some extent. In our architecture, "hot" rows are in the IMRS, thus data row access can be much faster than before.

It is efficient to support a full hash index if all data rows reside in memory. However, for cases that only some small percentage of data rows reside in-memory, building a hash index on the full table may result in extra memory consumption without reaping any benefits from the hash-index on data that is never referenced. Thus, keeping in line with the "hot" rows concept of this architecture, we chose to not build a hash-index on the full table; i.e. on the page-store data as well.

Many OLTP tables are seen to have unique indexes, and selects / updates using such indexes to return one row are common. We designed in-memory hash indexes to address this spectrum of OLTP workloads. In the BTrim architecture, we support a feature called "index hash caching" where, for a unique BTree index, an in-memory non-logged hash index on in-memory rows of the table is created. Such an index is called Hash Cached BTree (HCB) index. BTree index continues spanning all data rows that are in the IMRS and the page-store, and is still the access method for range queries. Hash index scans are designed as fast-path accelerator under **fully-qualified unique BTree index scans**.

The hash index is initially empty when created. New data inserted to the IMRS will not be loaded into the hash index, till the time such rows are accessed. For a point query, before traversing the possibly multiple levels of a BTree index, a light-weight hash-index probe is performed to locate the (one) row in-memory. If this probe succeeds, return the row from the IMRS, thereby the potentially expensive BTree search is completely avoided. If the probe fails, the query continues with the BTree search. When "hot" data rows are accessed from the IMRS by the BTree search, new rows will be loaded as entries in the hash index. We expect that over-time, as data rows that are stored in the IMRS are scanned, the corresponding hash index will be populated. Thus, for frequently scanned in-memory rows, the hash-index lookup will return rows faster than via an index scan. Finally, when cold data rows are evicted from the IMRS, or they are deleted, the corresponding hash nodes will be deleted.

As we show in our experimental results (Sec. 5.1.1), re-directing BTree scans to the hash index greatly reduces the code-path of index scans, thereby improving performance. This hash index design has several multi-core scalability and performance benefits:

1. The hash index is built on top of lock-free hash table, and all changes to it are done using compare-and-swap (CAS).
2. No extra logging is done for hash table maintenance.
3. The payload of a hash node (e.g., information about the corresponding IMRS data row, hash value, timestamp used for memory reclamation, row identity, linkage pointers) is designed to be cache-line friendly, and fits within 64-bytes.
4. All hash nodes mapped to one bucket are ordered. This permits short-circuiting scans if no matching hash node is found.

4. IMRS DURABILITY

Transactions affecting in-memory rows are fully durable and recoverable. Crash recovery and recovery from archives (dump / load of database and / or series of transaction logs) is fully integrated into the product. This section discusses changes made to the transaction management, and to the logging and recovery layers to make transactions affecting in-memory rows fully durable.

4.1 IMRS Log – Sysimrslogs

Sysimrslogs is the [database-specific] disk-based transaction log holding full row-images of committed rows residing in the IMRS. It is used both for durability of committed changes at run-time and eventually as the data source to re-instantiate the contents of the IMRS during crash recovery, or recovery from database / transaction log archives.

4.2 Logging and Transaction Management

Transactions and ISUDs are allowed to span ordinary tables and IMRS-enabled tables. A single DML statement on an IMRS-enabled table may affect both in-memory rows and page-store rows. As an example, a simple insert to the IMRS for a table with a single index produces an in-memory data row and a page-store

resident index row. Another example is an update statement updating some rows in the page-store and others in the IMRS.

Each portion of the statement's execution, or more generally the transactions', affecting in-memory rows or page-store rows is made durable in the corresponding transaction log. For the prior examples, a single row insert will log the insert of the data row in sysimrslogs and log the insert of the index row in syslogs. Thus, a transaction affecting IMRS-enabled tables may have a set of changes committing to syslogs and another set of changes committing to sysimrslogs.

A transaction is deemed to be committed only when both sets of changes are made durable. The reason for this design choice of two-legged logging is to overcome bottlenecks traditionally seen in syslogs-based logging. One example is the way run-time rollback is implemented. Rollback of changes to rows in the page-store is performed by scanning log records in syslogs. However, transaction rollback and rollback to savepoint for changes to in-memory rows is performed by scanning the in-memory versions created as part of a transaction, thereby improving performance.

4.3 Commit-Time Logging – No Undo

New designs are introduced in sysimrslogs to overcome existing scalability bottlenecks seen with syslogs in the page store.

Logging in sysimrslogs: Firstly, as in-memory rows are not resident in data buffers, the overheads of dirtying buffers, pinning dirty buffers to the PLC, or to syslogs, unpinning buffers etc. are **automatically non-existent**. Even for updates to migrated rows, the home-buffer holding the migrated row is no longer involved in the update. That holding data buffer could well have been decoupled from the in-memory row and recycled out of the buffer cache, and does not participate in the update transaction. Secondly, a new technique called aggregate logging is employed for in-memory updates. If multiple updates are done to an in-memory row in a transaction, only the final after-image is logged to sysimrslogs. (Contrastingly, such multiple updates in a transaction to the same page-store row generate a log record for each update.) Next, only **commit-time logging** is performed. At commit-time, the in-memory versions, which are located easily off the transaction descriptor, are scanned, and log records are generated directly into sysirmslogs for the final images of modified rows. This way, at transaction run-time, no intermediate copies of log records are generated (in PLC) and the double-copying, like in syslogs, is avoided. Commit-only logging has significant advantages to crash recovery where only redo-processing is needed and the undo phase is completely avoided. Also, we claim that commit-time logging strategy is eminently suited for the IMRS. Implementing this for the page-store rows is practically unviable as we would have to hunt down potentially several buffers which were changed as part of a single transaction. This would introduce another degree of contention on buffer / page management.

4.4 Log Space Management

All the bottlenecks discussed earlier for syslogs are overcome by basic changes in the way space management and logging is done to sysimrslogs.

The space on the imrslog device is dedicated to one sysimrslogs of one database and is not shared by other objects. Space on the imrslog device is serially allocated to sysimrslogs as contiguous pages. This space remains allocated. No page allocation or deallocation ever occurs once the space has been allocated to the imrslog in the database. Transactions do not have to contend with allocation costs at run-time. The dynamically active portion of sysimrslogs is tracked using metadata structures, while the first/last pages of the space map remain fixed. Serial allocation facilitates space pre-reservation at commit time, as explained next.

The last-log page contention point problem is overcome by a scheme termed as **transaction blocking** which provides multiple insertion points to the sysimrslogs page stream to concurrently committing transactions.

Figure 7 shows the log record streams for different transactions, T1, T2, … T6, that have committed on different log pages P1, P2, P3 etc. All the log records from a transaction appear in a contiguous chunk of space, shown in hashed regions, referred to as the transaction block. At run-time, while row versions are created, simple counters are maintained off the transaction descriptor for the total logging space that is going to be required at commit time. When a transaction commits, under briefly held exclusive access protection to the current end of the active portion of sysimrslogs, sufficient space is reserved and assigned to one committing transaction, following which the exclusive access is released. Subsequently other committing transactions serially get access to the new active end of the page chain to reserve the space for their log records. Once this space is made available, multiple transactions can commit simultaneously generating their log records directly into this reserved set of pages, without any further concurrency issues.

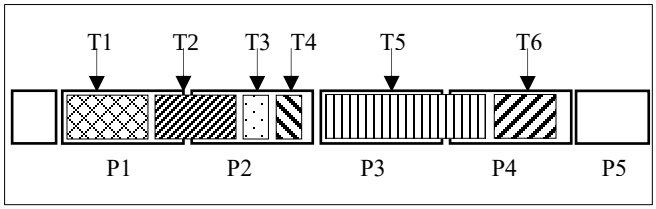

Figure 7. Transaction Blocking in sysimrslogs

In the case where a transaction block contains more than one page (as in T2 and T5 above), only the boundary pages (first and last page) of the transaction block may be shared among transactions. Since the transaction block space of different transactions assigned to the same page is also disjoint, writing log records in these transaction blocks is done concurrently without holding any exclusive access to the buffer (e.g. T2, T3 and T4 can be writing their log records concurrently to the same page P2). For small transactions committing few rows, log pages can be shared between committing threads, in which case traditional group-commit methods are used to ensure I/O efficiencies during logging.

Conventional WAL techniques are needed for logging changes to page-oriented rows, whereas in sysimrslogs, as we do not have to deal with dirty'ing pages due to changes to in-memory rows, WAL is essentially moot.

For large transactions, there *may* appear to be significant "extra work" at commit time to generate all the records and to persist them to the log pages. Drip logging in the page-store which is amortized over the transaction's code path by the use of WAL is, perhaps, more needed for disk systems that were prevalent when such techniques were first introduced to databases. Currently, we believe with availability of devices and SSDs capable of supporting very

high I/O rates, commit-time latency should no longer be an issue. This is confirmed in our experiments where we did not notice any measurable commit-time latency arising from this design.

4.5 Crash Recovery – IMRS Recovery

IMRS recovery using sysimrslogs as the durable store is performed as follows. As only commit-time logging is performed, only a redo-phase is required for processing all committed changes found in the log. No undo is needed, therefore logging of CLRs or other such scheme is skipped (both at run-time and at recovery time).

Unlike for the page store where checkpoint writes out all dirty buffers including log and data to the disk, no data in the IMRS will be written to disk. Only some book-keeping information is persisted to enable recovery processing.

Recovery of the sysimrslogs starts from the transaction that inserted the oldest row in the IMRS cache. Earlier sections in the sysimrslogs will not be used by recovery since the rows created by those transactions no longer exist in the IMRS cache.

Recovery is done by transactions. Since transactions are grouped in disjoint blocks in sysimrslogs, if there were any transaction blocks that were reserved but not yet fully committed at the time of crash, those invalid transaction blocks will be skipped over by recovery.

5. PERFORMANCE EVALUATION

The objective of our performance evaluation was to understand the costs of in-memory processing and whether the benefits overcome the bottlenecks seen in high-volume transactional workloads run against the page-store. In all experiments, we ensured that the page-store data is fully-cached in the buffer cache, thereby, avoiding any I/O costs. Therefore, no further gains can be obtained by providing any more memory to the buffer cache. Instead, we show that any additional memory can be deployed to the IMRS to deliver enhanced performance for large classes of workloads.

Using *the same version of the software*, we conducted a series of micro-benchmarks and end-to-end OLTP benchmarking tests. For each experiment, we uniformly performed a warm-up run followed by 3 runs. The metrics from the warm-up run were discarded.

In all our experiments, we observed that the throughput metric across the 3 runs were within 10%, therefore we used the median value of the metric across these runs for comparison between the page store and the IMRS.

To reiterate, in this section, a page-store table refers to ordinary tables which have data stored in pages, and use the buffer cache. IMRS refers to tables enabled with the row-caching feature and store rows in-memory. HCB refers to the addition of hash-indexes on such IMRS-enabled tables, where the hash-index is used as a performance accelerator under unique BTree indexes. In general, as the goal of our experiments was to study the benefits of the IMRS, built as the new architecture, we did not pursue any further optimizations to the page-store code even though it *may appear* that there are low-hanging short-comings in the current page-store machinery that are easily fixable. (We consider that as a never-ending game of cat-and-mouse performance tuning!)

5.1 Micro-benchmarks

We performed a series of micro-benchmarks to study the execution behavior of inserts, selects and updates. These operations constitute the bulk of OLTP workloads. The benchmarks focused on overall throughput (e.g. number of transactions / minute, or number of selects / minute). As our focus was to measure the gains from in-memory processing on the server-side, all experiments were conducted using stored procedures. This, therefore, eliminates any network traffic of result sets to the client. We did not focus on measuring latency, for example, the time to return the 1st row from a select, as that was not the areas re-designed in this work.

The RID-map lookup and hash-index probes done for in-memory rows *could* appear to be an additional overhead, but we did not find these to be of any noticeable cost. In any of the CPU profiles done in all our experiments (including end-to-end benchmarking), we noted the relevant functions to be consuming less than 2% of the overall CPU cycles. Similarly, we noticed that the overheads of IMRS-GC, which runs as a background thread, is not significant, contributing less than 3% of overall CPU usage even in high-end OLTP workloads. Thus, we did not perform specific micro-benchmarking to further evaluate the impact of these new techniques.

We used the **stock** table of the standard TPCC benchmark for all micro-benchmarks, designed for 240 warehouses. The **stock** table has 17 fixed-length columns – 6 integer columns and 11 char columns, with a fixed row-size of 312 bytes. The choice of a fixed row-size for this table allowed us to accurately measure logging overheads and log space consumption in both the transaction logs. The table has a composite clustered index on two integer columns (warehouse_id and item_id), for an index row-size of 12 bytes. For each warehouse, the **stock** table has 100K item IDs, for a total of 24 million rows. In our implementation on 16K database page sizes, for 24 million rows, the height of the index was three (root and two levels). For the select and update micro-benchmarks, the table was fully loaded, but the activity was focused on about 5 – 100+ pages of the table. This reflects the usage pattern where we expect to see benefits from the IMRS; i.e. highly concurrent activity on data. The thin slice chosen of the (large) **stock** table is meant to represent either a small but very active table, or some "hot" slice of data of large tables. For the insert micro-benchmark the table was initially empty and multiple clients insert rows concurrently.

We designed the micro-benchmarking experiments to evaluate the additional benefits of using HCB in conjunction with the IMRS as follows. For the select micro-benchmark, we measured the select throughput for (a) just the IMRS (no HCB) and (b) IMRS plus HCB. For inserts, the **stock** table did have HCB enabled, but as the HCB is not maintained (or affected) during inserts, it was not necessary to separately measure insert throughputs for the IMRS-only case.

For updates, we anticipate that the scan portion of an update of in-memory rows will be benefitted by the presence of an HCB. We performed the update micro-benchmark with both the IMRS and HCB to study the combined gains in comparison to the costs of performing the updates in the page-store.

For all experiments, the scalability was tested up to 64 CPUs (using 64 ASE threads). Since these are all throughput workloads, the experiments were executed using as many users as needed to drive maximum throughput on the server-side.

The experiments presented in this section were performed on a machine with an Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz processor, having 4 sockets, 112 cores (224 logical CPUs), with 1TB RAM. All concurrent clients and the server were run from the same box, which eliminates any network overhead for client-server communication.

5.1.1 *Highly Concurrent selects of hot rows*

We designed this experiment to measure the performance and scalability with increasing number of concurrent selects. In this experiment, we varied the number of ASE threads from 2 to 64, and had multiple clients repeatedly select a row each from the **stock** table using fully qualified index predicate. As each client's select was driven through stored procedures (i.e. without any network overheads), it was sufficient to have the number of concurrent clients in each experiment be the same as the number of ASE threads. Each client repeatedly selects the same row corresponding to its client-id, so different rows are selected across clients. In the warm-up run of this experiment, selects done by individual clients cause their affected rows to be cached in the buffer cache or in the IMRS. Effectively, this experiment compares the performance between selects of rows fully cached in the buffer-cache versus rows fully cached in the IMRS.

Even though this is a read-only workload, in the page-store we maintain reference counts on the accessed data or index pages. Additionally, shared-latches are required to ensure physical consistency while accessing the page. On multi-core systems, these primitives cause cache-line invalidation, even though multiple clients are accessing different rows, when they happen to be stored on the same data page. This results in increased contention, resulting in degraded scalability.

Figure 8 shows the performance gains with IMRS alone and then, with HCB on top of it, at various number of CPUs.

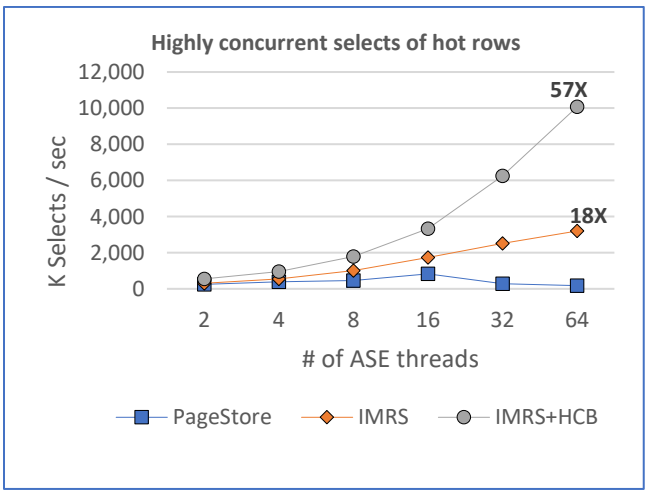

Figure 8: Performance gains for concurrent selects

At lower thread counts (2, 4), there is not much contention in page store runs. The gains observed with IMRS, and especially with HCB (2 – 2.5X) are due to (a) short circuiting the multi-layer B-tree access by HCB and (b) the code-path improvements of faster access of rows in the IMRS. Beyond 16 thread counts, we observed a significant drop in the page store performance, largely due to the contention issues noted above.

As access to in-memory rows is latch-free and completely avoids the page cache, these contention points for data pages are overcome by the IMRS. This results in an improved performance (18X at 64 engines). With HCB in the picture, we then completely eliminate this contention even on index pages. At high core counts, for the highly concurrent select workload, with IMRS + HCB, we were able to deliver significantly improved performance gains as compared to the page-store – 57X at 64 core-counts.

5.1.2 *Highly Concurrent inserts*

We designed this experiment to measure the performance of highly-concurrent inserts on ASE with 64 threads. For a fixed number of concurrently inserting 240 clients, starting from an empty table, each client inserts 100 K rows to its assigned warehouse ID. We varied the transaction size by changing the number of rows inserted per transaction. This allows us to study the impact of transaction size on overall insert throughput and any commit-latency due to logging.

For inserts to the page-store, the presence of a clustered index on (warehouse_id, item_id) attempts to guide the insertion from different clients to different data page. However, at high concurrency, we observed that different clients may end up inserting into a small set of target data pages at a time which causes page-level contention.

Inserts to the IMRS are independent of the clustered index as we do not attempt to maintain any sort of in-memory "clustering" of newly inserted rows. Insert into the IMRS-enabled **stock** table still logs the index insert in the page store. So, we performed two set of experiments – one without the index and one with the index. The results are showin in Figure 9.

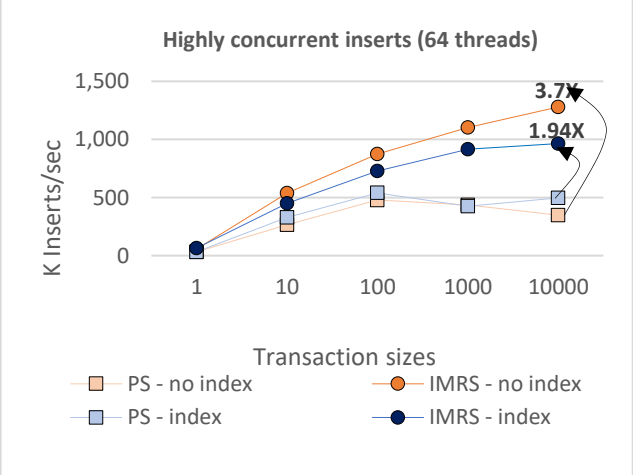

Figure 9: Performance gains for concurrent inserts

Certain aspects of transaction performance, such as the perceived commit-time latency that may occur due to the generation of log records while logging to sysimrslogs, is measured indirectly, and is subsumed in the overall throughput metric.

There are multiple interesting data-points highlighted from these experiments:

1. Larger transaction sizes cause greater degree of conflicts across transaction. As the inserts into the page-store are impacted by page-level contention, the performance is affected negatively by larger transaction sizes. In contrast, IMRS runs keep improving with larger transaction sizes.
2. At lower transaction sizes (especially size of 1), for both page store and IMRS, the performance is bottlenecked by the

commit-time behavior. However, for IMRS, with the design of performing only the space reservation under a semaphore, and due to multiple insertion points to concurrently committing transactions, the performance is greatly improved. Even for the simplest case of 1-row inserts / transaction, insert performance in IMRS is 1.8-2X better than in the page-store. As the transaction size increases, the benefits from multiple insertion points to sysimrslogs are accentuated, resulting in higher relative throughput. For example, at a transaction size of 100 rows, each commit independently writes to, and flushes, slightly more than 2 IMRS log pages. This allows several concurrently committing transactions to drive the imrslog I/O system. Also, as commit-time logging in sysimrslogs generates logging directly to the log page, we avoid the overhead of double-copying seen in the case of logging in syslogs.

3. The increased IMRS performance gains over the page-store for the no-index inserts v/s inserts-with-index case is explained as follows. (For example, for the transaction size of 10K rows, we are comparing 3.7X v/s 1.94X.) Index inserts are logged in syslogs, and the known contention issues with syslogs degrades the performance of insert throughput for the case with index in comparison to without an index.

Note that the amount of log generated for page store versus IMRS for insert workload is in the same ballpark and the difference in performance is not due to the difference in the amount of log generated (unlike in the case of update workload, which we will study next).

In conclusion, we observed that highly-concurrent inserts perform significantly better in the IMRS. We do note that the overheads of inserting to the page-store can also be partly overcome by partitioning the table n-ways, thereby providing multiple insertion points. In practice, however, this would require DBA intervention on a case-by-case basis to partition selected tables, which is detrimental to the overall TCO of the product. Therefore, we did not pursue the experiments to verify the potential gains from partitioning.

5.1.3 Highly Concurrent updates of hot rows

We designed the experiments to measure the difference in throughput due to:

a) Varying degree of contention on updated pages
b) Varying transaction sizes (# of updates per transaction)
c) Varying update width; narrow v/s wide updates. i.e. updates of 1 column v/s several or all columns, respectively.

We used the **stock** table for this experiment, again for the reason that the logging overhead is deterministic due to the fixed row-size. Updates to different rows are performed by 240 clients. Initially we performed narrow updates; i.e., update of a single non-index column. The server was run using 64 threads. Non-intersecting rows were updated by different clients so that we do not run into logical locking issues and focus only on update throughput. In page store, the updated rows may reside on the same page.

For dimension (a), fixed non-overlapping chunks of rows to be updated are "assigned" to a client, and we varied the size of this chunk from 1 row to 5, 25, 125 contiguous rows. Recall that on a 16K data page, about 50 rows fit on a page. Therefore, the average number of clients updating rows on the same page in the page store runs correspondingly varies from 50 (chunk size of 1) to 10, 2 and 0.4 (chunk size of 125). The former extreme would have maximum page contention in page store runs whereas the latter would have least page contention, as each client works on a separate page. With this distribution, 240 clients updating a single row each affects 5 pages, whereas all clients updating 125 rows each affects 600 pages. This simulates the variation in the size of "hot" pages in the workload. This distribution is captured in Table 4.

For dimension (b), we varied the number of updates done by a client within a transaction, from 1, 10 to 100. Note that, in an experiment, the chunk of rows assigned to a client remains fixed. So, all updates within a transaction get done on random rows within this chunk of assigned rows, and, in some cases, a row may get updated more than once within a transaction. For example, with a chunk size of 1 row, and a transaction size of 10 updates, the same row will be updated 10 times. (We realize this may not be a common case, but is included to study the overall distribution.) On the other hand, with a chunk size of 25, and transaction size of 10 updates, on an average, every other row in the chunk assigned to a client will be updated. For completeness of the spectrum of combinations, we still examine the throughput when some rows may be updated multiple times within the same transaction.

Table 4. Distribution of page contention v/s chunk size

Chunk sizes (Rows assigned to a client)	1	5	25	125
Total number of distinct pages in the updated "working set"	5	24	120	600
Concurrent clients updating per page	50	10	2	0.4
Page contention decreases	→			

Figure 10 shows the throughput of single-column updates for page-store and IMRS for all combinations of transaction size and update chunks considered. In the legend of this figure, PS-1 indicates the page-store run with transaction size of 1. Likewise, IMRS-10 indicates the IMRS-run with a transaction size of 10 updates.

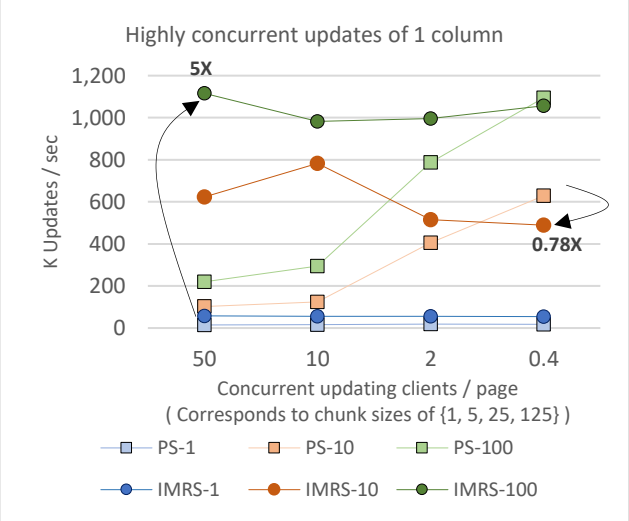

Figure 10: Performance gains for 1-column updates

We observed the following performance characteristics:

1. Across the page store run, as expected, with fewer concurrent clients updating on the same page, the performance keeps improving as there are lesser page contention. In contrast, for the IMRS the performance is relatively constant.
2. With a transaction size of 1 (PS-1, IMRS-1), the throughput is lower in all experiments, because the commit time bottlenecks dominate the performance. However, IMRS performs relatively better (~3X over PS-1) due to the improved sysimrslogs strategies, explained in the previous section.
3. For cases where the same row is updated multiple times in a transaction, IMRS updates benefit due to aggregate logging. (For example, in IMRS-10 with chunk sizes of 1 and 5, and for IMRS-100, with chunk sizes of 1, 5 and 25.) For IMRS, only one log record representing the net change gets logged for a row, thus the amount of log generated is lesser.

Lastly, note that in the absence of page contention i.e. at chunk size of 125 (0.4 clients/page), the IMRS actually performs poorer than page store. This is due to the full-image logging done in IMRS versus the delta-logging done in the page store. Recall that this workload involves updating a single column in the table. IMRS logging would involve writing a new image for the entire row whereas page store logging involves writing only the modify log record for the updated column. Beyond micro-benchmarking, this is not really seen as a drawback for user-workloads.

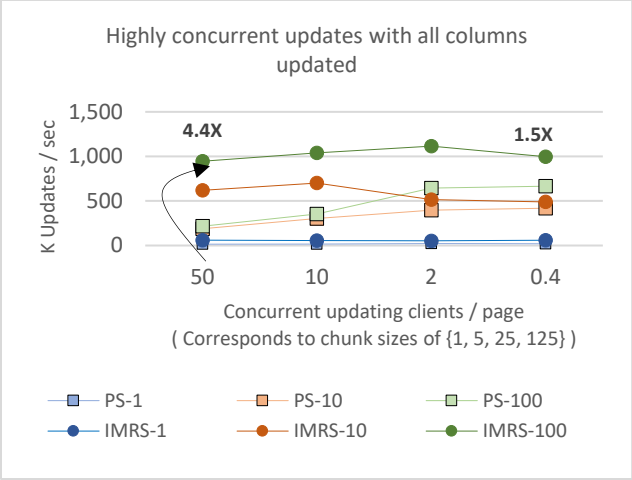

Figure 11: Performance gains for all-columns update

For several cases, such as replicated tables, for tables with triggers, for some classes of multi-row updates driven by a join or, more commonly, when a large number of columns in the row are updated, updates in the page store do generate full before/after image logging. In fact, in such cases, the log record volume is double for page-store changes as compared to changes in the IMRS because we only log the after-image for changes affecting rows in the IMRS. To demonstrate this behaviour we repeated the experiments with all the 15 non-index columns updated. The results are shown in Figure 11.

In conclusion, for highly concurrent updates in small transactions, on small tables and for narrow or wide updates, IMRS out-performs the page store by factor of 3.

5.2 End-to-end benchmarks

While micro-benchmarks are useful to demonstrate particular aspects of the design, what matters finally is the improvements seen in an end-to-end workload. Our end-to-end OLTP benchmark is based on TPCC. The experiments in this section were performed on a machine with an Intel Xeon E7-4880 @ 2.50 GHz processor, having 4 sockets / 60 cores / 120 logical CPUs, with 1TB RAM.

On 64-CPUs, with 250 users performing a mix of ISUD transactions, similar to those in the TPCC benchmark, **we are able to deliver close to 3X gains for throughput metric (total transactions / minute), when comparing the OLTP benchmark running in the page-store v/s in the IMRS.** To conform to low TCO and usability considerations, minimal tunings were used which can be implemented primarily through configuration changes without needing extensive table-, index- or cache-specific tuning. It is important to note that the entire stored-procedures based benchmarking code ran with no need for any code changes when executed against the IMRS.

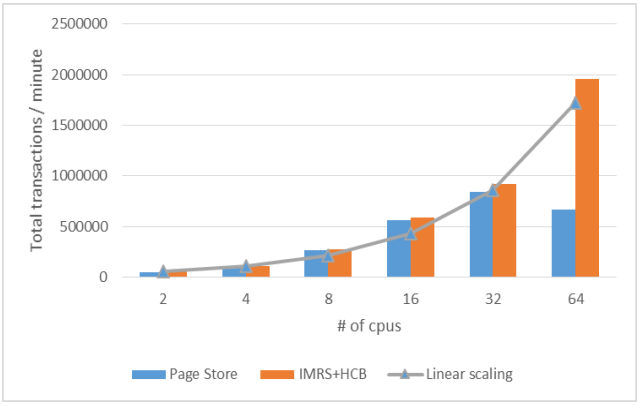

Figure 12: Multi-core scaling for an E2E workload

Figure 12 shows the throughput scalability for this workload across multiple. While both architectures scale similarly at lower number of cores, the page-store is not able to scale beyond 32 cores. IMRS is able to scale almost linearly till 64 cores.

In this workload, the **order_line** and **orders** tables have similar highly concurrent insertion pattern as seen in the insert microbenchmark earlier and thus page store has similar bottlenecks. The **warehouse** and **district** tables which are small tables and are frequently updated, have similar page conflicts in case of page store as observed in the concurrent update microbenchmark. IMRS is able to resolve these scaling issues in the end-to-end workload resulting in significant performance gains. We anticipate other real-life OLTP workloads to also have similar access patterns and, thus, the resulting gains by use of IMRS.

In future, we plan to investigate the performance characteristics of the SAP SD Benchmark when run against the IMRS.

6. RELATED WORK

Several commercial in-memory database engines are available in the market, with different capabilities. SAP Hana [5] [20], [24], Oracle TimesTen [15], [17], Oracle RDB [12], Microsoft Hekaton [2], [3] and VoltDB [26] are a few comparable offerings. There are notable differences between existing offerings and our work.

Many of the existing offerings, such as SAP HANA, Microsoft's Hekaton, and VoltDB require that the entire database or table be fully memory-resident. Our offering does not have this requirement, and allows for smaller portion of "hot" rows from some tables to be in-memory. Oracle TimesTen is more on the lines of an application-tier caching system, which can be deployed with an application like a library or as an extension of the database engine. Although it is similar in capability with our offering, it is still a separate product needing its own installation and administration in addition to that of the core database engine.

Oracle RDB has a feature set of data row caching similar to our offering, but it was not clear from available literature if support is provided for identifying cold rows and moving them from the in-memory cache to the database physical storage. Organic integration of ILM rules within the database engine and IMRS is a key differentiator of our offering. However, Oracle Rdb also offers caching of index rows, which is currently not supported in BTrim.

Our offering is very similar to Microsoft's Hekaton offering integrated with SQL Server, both using in-memory versioning offering snapshot isolation. However, there are significant differentiating aspects. Hekaton requires that tables marked as in-memory be fully memory-resident, whereas in BTrim this is not a requirement. For very large tables, the design of Hash-Cache BTree indexes under unique indexes in our design provides high-performance for point-query access to "hot" data in-memory without the need for large memory footprint for a hash index spanning all the table's data in the cache (as is the design in Hekaton). As of this writing, we believe Hekaton does not offer schemes to organically harvest cold rows from the cache and migrate the cold-data to disk-based storage. Some work in the area of cold-data management has been published in this area (Siberia [5] and by Levandoski et' al' [13]), but is not, yet, commercially integrated with the core dbms engine. Also, the techniques rely on, howsoever efficient, off-line analysis of log streams which then can lead to identifying cold rows. In contrast, organically identifying and packing cold rows out of the IMRS to the page-store is an integral and differentiating part of our offering. The overheads of this cold-data harvesting is very minimal and is not seen to impact OLTP performance in our experimental results [19].

In [3], techniques referred to as Anti-Caching; i.e. moving cold data from in-memory to disk storage, are presented as an extensible alternative to fully in-memory databases. The anti-caching aspects of this work is close to our Pack design however, their storage model starts initially in-memory and then pushes cold data to disk-storage. This is different from BTrim where we support a hybrid storage model for existing databases which already have on-disk data and augment that with in-memory storage for faster processing. In the Anti-caching work, access to cold data that was evicted (i.e. access from page-store) results in rolling back certain transactions while the system retrieves relevant tuples in the background. This approach seems quite non-user-friendly. Our scheme has no such issues with application outages.

Since the early editions of Hekaton offering with SQL Server, there have been serious limitations on features that could be supported on in-memory tables. Some of these have been bridged with later editions of the product. However, literature [14] suggests that there still are limitations to SQL usage with in-memory tables. Features such as Replication, data compression, non-clustered indexes, partitioning of memory-optimized tables are unsupported on in-memory tables. Our offering is feature-compatible in all these areas with existing versions of the product. With Hekaton, there are documented limitations [15] on the types of accesses that can be done to in-memory tables from natively-compiled stored procedures. As an example: Natively compiled T-SQL modules do not support the FROM clause and do not support subqueries in UPDATE statements (they are supported in SELECT). In our offering, no such limitations exist on what language constructs can be used in any part of the system. Although full support for natively-compiled SQL is not offered, SAP ASE [21] does offer performance improvements by generating compiled code for optimizing execution of tight-loops in search and index qualification code. There is no distinction made when native compiled plan is generated and executed while scanning data for IMRS-enabled tables or page-store tables. Choice of pessimistic locking in our offering allows complete support for ANSI isolation levels on in-memory tables, and full application compatibility. Contrastingly, choice of optimistic locking for Hekaton in-memory tables can result in changes in application logic to retry transactions that may fail due to concurrency conflicts.

In [7], Graefe et al present an architecture that optimizes buffer pool designs to support "big data" workloads which cannot fit in available memory sizes. This work manages buffer pool usage using pointer swizzling, but does not address areas considered by this work around contention issues arising from page-oriented storage and row-level in-memory processing.

7. CONCLUDING REMARKS

In this work, we presented a novel in-memory row-oriented storage extension tightly integrated with the page-based storage, buffer cache and access methods of the traditional SAP ASE database engine, to offer high-performance overcoming contention and scalability issues seen in typical database servers, including those seen in previous editions of this product. Tight and deep integration to the existing architecture delivers **full language and application compatibility** to existing and migrating customers, allowing better use of available memory and multi-core resources for higher-end OLTP workloads, and more importantly, of human-resources needed for any migration effort.

The new architecture **does not require** that all table data, or much worse, the entire database be fully memory-resident. Capturing the patterns of data-ageing, the hybrid architecture allows for **data to flow** through the storage layers (disk, buffer cache or in-memory store), and tightly integrates the in-memory storage techniques to existing access methods.

We believe this technology, with its performance gains, and compatibility levels enabled by the deep integration with the existing product is fundamentally different than any other enterprise-class commercial database engine available today.

8. ACKNOWLEDGEMENTS

We would like to express our sincere appreciation to all members of the ASE Product development and Server Performance Engineering teams who have contributed to making this integrated product offering a reality. Aditya Gurajada and Fei Zhou, among the primary authors of this work, have since retired from SAP. Contributions by Rahul Mittal, Jay Sudrik and Graham Ivey who delivered key pieces of this architecture (but are currently not at SAP) is duly acknowledged. Contributions by Piyush Dungarwal in the area of performance enhancements and benchmarking are duly acknowledged.

9. REFERENCES

[1] Ailamaki A., DeWitt J. David, Hill D.M., Wood A. D., *DBMSs On A Modern Processor: Where Does Time Go?* In VLDB, pages 266-277, 1999.

[2] Delaney K. Online White Paper. SQL Server In-Memory OLTP Internals for SQL Server 2016, June 2016.

[3] DeBrabant J., Pavlo A., Tu S., Stonebraker M. and Zdonik S. *Anti-Caching: A new approach to Database Management System Architecture*. PVLDB, 6(14):1942-1953, 2013.

[4] Diaconu C., Freedman C., Ismert E., Larson P., Mittal P., Stonecipher R., Verma N., Zwilling, M. *Hekaton: SQL Server's Memory-Optimized OLTP Engine*. In SIGMOD, pages 1243-1254, 2013.

[5] Eldawy Ahmed, Lavendoski Justin and Larson Per-Ake. *Trekking through Siberia: Managing Cold Data in an In-memory Database*. PVLDB, 7(11):931-942, 2014.

[6] Farber Franz, May Norma et al. *The SAP HANA Database – An Architecture Overview*. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2012.

[7] G. Graefe, et al. *In-Memory Performance for Big Data*. PVLDB, 8(1):37-48, 2014.

[8] Grund M., Kruger J., Plattner H., Zeier A., Cudre-Mauroux P. and Madden S. *HYRISE – A Main Memory Hybrid Storage Engine*. PVLDB, 4(2):105-116, 2010.

[9] Harizopoulos, S., Abadi, Daniel J., Madden, Samuel, Stonebrake, Michael. *OLTP Through the Looking Glass, and What we Found There*. In SIGMOD, pages 981-992, 2008.

[10] Hobbs, L., Smith I., England K. *Rdb: A Comprehensive Guide*, 3rd edition. Digital Press, Oxford, 1999, 167-169.

[11] Johnson R., Pandis I., Hardavellas N., Ailamaki A., Falsafi B. *Shore-MT: A Scalable Storage Manager for the Multicore Era*. In ACM, 2009.

[12] Lastovica, N., Oracle Rdb Technology Group. *Guide to Database Performance and Tuning: Row Cache Enhancements, A feature of Oracle Rdb*. Rdb Journal, Aug 2003.

[13] Levandoski Justin, Laron Per-Ake, Stoica Radu. *Identifying Hot and Cold Data in Main-Memory Databases*. In ICDE, 2013.

[14] Microsoft SQL Server Online Docs, *Transact SQL Constructs Not Supported by In-Memory OLTP*. https://docs.microsoft.com/en-us/sql/relational-databases/in-memory-oltp/transact-sql-constructs-not-supported-by-in-memory-oltp, Nov 2017.

[15] Microsoft SQL Server online docs, Migration Issues for Natively compiled procedures. www.microsoft.com, 2017.

[16] Oracle TimesTen White paper, Using Oracle TimesTen Application-Tier Database Cache to Accelerate the Oracle Database, Oct 2014.

[17] Oracle TimesTen online documentation, various. www.oracle.com

[18] Oracle Rdb online documentation, *Guide to Database Performance and Tuning: Row Cache Enhancements*, 2003.

[19] Pathak A., Gurajada A. P., Khadilkar P. *Life Cycle of Transactional Data in In-memory Databases*, ICDE, Joint Workshop of HardBD and Active, April 2018.

[20] Plattner H., Zeier A.. *In-Memory Data Management: An Inflection Point for the Enterprise Applications*. Springer publication, 2011, Chap. 5, pages 89-95.

[21] SAP ASE Product Documentation, What's New in SAP ASE 1602, dd. Dec. 2016.

[22] SAP ASE Whitepaper, www.sap.com, What's New in SAP Adaptive Server Enterprise 16.0 SP02, MemScale Option. dd. 2015.

[23] SAP ASE Technical Whitepaper, www.sap.com, *Performance Scalability Enhancements in SAP ASE. Discussion on Scalability Enhancements in ASE 16.0*, dd. 2014.

[24] SAP ASE 16.0.3 online documentation, SAP ASE In-Memory Database User's Guide, June 2017.

[25] Sikka V., Farber F., Lehner W., Cha S. Kyun, Peh T., Bornhovd C., *Efficient Transaction Processing in SAP HANA Database – The End of a Column Store Myth*. In SIGMOD, pages 731-741, 2012.

[26] Stonebraker M., Madden S., Abadi D.J et al. *The End of an Architectural Era (It's time for a complete rewrite)*. In VLDB, pages 1150-1160, 2007.

[27] VoltDB. http://www.voltdb.com