Life Cycle of Transactional Data in In-memory Databases

Amit Pathak#1, Aditya Gurajada#2, Pushkar Khadilkar#3

SAP Labs

Magarpatta City, Pune - India

1 amit.pathak@sap.com, 2 aditya.gurajada@sap.com, 3 pushkar.khadilkar@sap.com

Abstract— **The last decade has witnessed an explosive growth in database engines optimized for main memory based execution. However, the requirement to store all the data in-memory makes such processing commercially costly and unviable for very large databases. In this paper, we present novel techniques optimized for transactional workloads where only a small portion of the entire database is "hot" and needs to be in-memory. The SAP ASE hybrid database engine delivers high-performance transaction processing transparently on tiered storage consisting of traditional page-oriented disk based storage, for cold data, and in-memory row storage for fast processing.**

Our techniques make storage choice for the rows based on access patterns of the OLTP workload. This is done by monitoring and analysing the workload running in the system with minimal impact to the transaction performance. The techniques also adapt to the workload to alter storage choices for the data, which is completely transparent to the application and provides continuous data access. The storage choices are not made at gross table or partition level but made at the level of individual row and type of DML operation.

Our result show that these techniques help reduce the memory footprint by a large margin in OLTP workload even while providing performance parity with a setup where all data is stored in-memory.

I. Introduction

Traditionally database system store data on disk, usually in a page-oriented layout, and use various caching techniques to keep important pages in the buffer cache for efficient processing. Many in-memory database engines promise improved performance and often require that the entire database be in-memory, under the premise of availability of cheaper memory [15]. However, we argue that, for very large databases, keeping all the data in-memory may be commercially unviable. Moreover, in transactional systems, we typically see only some small portion of the database to be active, or "hot" during some period of activity. Thus, retaining large unused portions of the database in-memory may potentially be a wasteful use of premium memory resource.

The Business Transactions In-Memory (BTrim) architecture of SAP ASE [3] is a hybrid storage model across traditional page-oriented disk-storage and row-oriented in-memory storage. This allows storing some part of the data (hot data) from one or more tables in the In-Memory Row Store (*aka* IMRS) and the remaining bulk of the data (warm / cold data) is stored in a traditional disk based store (*aka* page-store). Just as the buffer cache is used to hold the working set of data pages in the cache, the IMRS is used to store in-memory "hot" rows, in a row-oriented layout. The IMRS acts as a performance accelerator on top of the buffer cache, and is both a store (for newly inserted or updated rows) and a cache (for frequently accessed hot rows). The BTrim engine supports full ACID-compliant transaction processing to the data independent of its storage without requiring any application changes. For example, a single statement may process (select or update) hot data in the IMRS, and may also process cold data on the page-store. Alternately, a statement may migrate data from the page-store to the IMRS if it finds the data as hot. Subsequent access to such hot data continues in the IMRS.

The hot data in the IMRS may become warm / cold over time and is moved back to the page-store to make IMRS memory available for newer hot data. This data-flow is referred to as Information Life Cycle Management (ILM). Typically, this data-ageing is performed across different data storage tiers. With our ILM-techniques, we achieve row-level data ageing which is tightly integrated in the core database engine.

We design for the data and access patterns seen in OLTP workloads under the assumption that not all data is important enough to be kept in-memory all the time. The goal of our ILM-techniques is to optimally use available memory for hot rows, thereby reducing the memory requirement for the database. Our aim is to deliver performance gains comparable to what can be achieved with a fully memory-resident database.

The rest of the paper is organized as follows. Section II describes at a high-level the hybrid storage architecture. This lays the background for how the temperature-aware data management is integrated inside a commercial DBMS engine. In section III, we present our design objectives and motivation behind this work. Different run-time parameters affecting our design choices are discussed. In section IV we discuss the techniques used for efficiently storing hot data in-memory. There are two significant contributions. Section V presents one core contribution of our work, which is the support for auto IMRS partition tuning. In section VI, we present the other significant contribution of our work, which are the mechanisms to efficiently identify cold data in the IMRS and re-locate (i.e. *Pack*) such data back to the page-store, without adversely affecting run-time performance. In section VII, we discuss how these Pack-ILM techniques are organically integrated with concurrent DMLs. Section VIII discusses some experimental evaluation of this work using OLTP benchmarks based on the TPC-C benchmark. Related work is discussed in section IX followed by our conclusions.

II. BTrim Architecture

The BTrim architecture is a deeply integrated extension to the existing store-and-access layers of the SAP ASE DBMS engine [1][2], offering high-performance access to in-memory "hot" data rows. It extends the capabilities of the existing engine to leverage large memory systems running on multi-core machines, while maintaining full compatibility for existing databases and T-SQL constructs. The new components of the architecture are designed to co-exist with and augment existing server sub-systems (like the buffer cache and logging sub-system) and are designed to use algorithms and techniques that offer superior multi-core scalability and high concurrency.

Figure 1 shows the salient components of the new architecture and how data is organized and accessed.

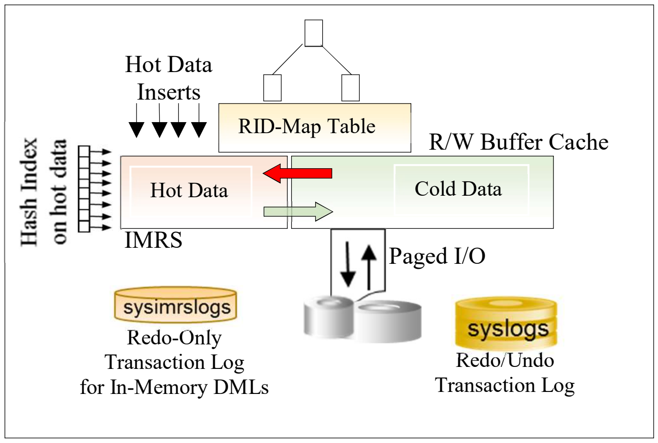

Fig. 1. BTrim architecture for In-Memory Transaction Processing

Traditional disk-based page-oriented storage is provided and data is read into the buffer cache (green shaded box). This data is referred to as page-store data. Periodically, hot data may be migrated from the page-store (buffer-cache) to the IMRS (red shaded box). Once migrated to the IMRS, the buffers holding the page-store image of the data can be recycled for other uses. There is no double-buffering of rows beyond the initial copy created in the IMRS due to migration.

New inserts go directly to the IMRS without any footprint in the page-store. Over-time, cold data is harvested from the IMRS and moved back to the buffer-cache, through an operation referred to as Pack. At that time, any data that was first inserted to the IMRS finds a location in the page-store.

Some or all the tables in a database can be altered to use the IMRS for improved performance, and such tables are referred to as IMRS-enabled tables. All updates to in-memory rows are performed using in-memory versioning, a scheme that is also used to support time-stamp based snapshot isolation for IMRS-enabled tables. Over time, only some part of a table may be in-memory. It is not necessary that a table marked for in-memory storage needs to have all its data in-memory. In Figure 1, the arrows indicating movement of hot data to the IMRS and of cold data to the buffer cache imply that IMRS-enabled tables can straddle the buffer cache and the IMRS. Access methods transparently locate the row from one of the two stores using internal scan methods.

Page-based BTree indexes are enhanced to transparently scan rows either in the page-store or in the IMRS. Index access goes through an in-memory lookup table, the RID-Map table (yellow shaded box), to locate the row either in the IMRS or in the buffer cache. Table-specific non-logged, in-memory hash-indexes are built on top of lock-free hash tables. Hash indexes span only in-memory rows and provide a fast-path performance accelerator under unique BTree indexes.

SQL statements and transactions may access tables or data that is in the page-store or in the IMRS, without any restrictions. Changes to page-store rows are logged in the (existing) transaction log labelled as syslogs. Changes to in-memory rows are made durable by logging in a (new) counterpart log, labelled as sysimrslogs above. Both logs are disk-based, and can be placed on SSDs for faster logging performance.

For the page-store, traditional data checkpoint based redo-undo recovery is performed. However, for the data in the IMRS, checkpoint does not flush any data to disk. All the IMRS data is recovered by doing a redo-only recovery of sysimrslogs. The system recovers both transaction logs independently with some lock-step ordering of recovery phases, to ensure a consistent database post-recovery.

Several new sub-systems are added to the product to efficiently use IMRS memory for hot data, without causing application outages. ILM strategies are woven through the access methods to choose whether data is stored in the IMRS or in the page store. Multi-threaded, non-blocking Garbage collection (IMRS-GC) is deployed 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, efficiently relocates cold (transactionally inactive) data out of the IMRS to the page-store (buffer cache). Pack and ILM work together to guarantee stable memory utilization and enhanced performance for OLTP activity

A key sub-system supporting the IMRS is a high-performance fragment-memory manager which is highly optimized for best-fit low-latency memory allocation and reclamation on multiple cores.

III. Design Objectives and Motivation

To motivate our design choices, we use the set of tables in the TPCC schema [12] as a reference workload in our examples. The set of tables in this schema have access patterns commonly-seen in typical OLTP applications such as update-heavy table, insert-only table, read-mostly / read-only lookup table, tables of different sizes and so on. Table names from the TPCC schema are identified by use of **bold** typeface.

ILM-techniques in the BTrim architecture address two important requirements:

1. Identify data as hot which can be stored in the IMRS
2. Identify and remove cold data from the IMRS

Our schemes to address these requirements largely rely on the following aspects of access patterns to decide which data to store and retain in-memory.

- Frequency of data access: It is important to only keep the often-used data in memory and either to not store in the IMRS or move infrequently accessed data from the IMRS to the page-store.
- Contention on the page-store: Traditional page-store are often seen to be non-performant [13][14] due to factors such as page contention, latch contention, etc. IMRS uses a row-

oriented architecture rather than page-oriented, and completely avoids buffer cache access, so it does not have any page-level contention issues.

Our solutions identify potential contention conditions on the page-store and decide to perform such operations in-memory instead of in the page-store.

- Type of operation: Our techniques distinguish between the type of operation – INSERT, SELECT, UPDATE, DELETE (ISUD) – to decide whether to store the data accessed by those operations in-memory. Some operations access and create hot data while other perform just an ad-hoc access to the data. Our techniques distinguish between various types of such operation using runtime statistics gathered from the workload and make choices on a per-row basis to store data in-memory.
- Granularity of storage decision: This is an important aspect as all the data in a database, in a table or partition need not be always hot or cold. Making a choice, say, to store all the data in-memory for an entire partition, or for the whole table, can lead to excessive memory requirements especially for very large tables. Our techniques perform storage decision during every operation on the row to decide if it needs to be stored in memory for faster access. The storage choices are optimized to avoid performance impact in different ways.

In transactional workloads, oftentimes there are partitions which can be considered either completely hot or completely cold, so we also make storage choice decisions at the partition (and operation) level. For example, the **warehouse** table is a small heavily-updated table and can always be in-memory whereas the **history** table is an insert-only but never accessed table so need not be in-memory. Such decisions are made internally in our system.

The main objective of ILM is to keep cache utilization stable while retaining mostly hot rows in memory to provide benefits of fast performance. Other objectives are:

- **Application compatibility**: A major requirement for ILM was to provide application compatibility. Use of ILM / IMRS should not require major application re-writes or lead to application / data outages.
- **Minimal tuning for user input**: One common method used for ILM by in-memory systems such as Hekaton [4] is to accept input from users about placement of rows. Our experience working with enterprise grade systems has revealed that it is hard for users to know which tables are suited for in-memory processing amongst potentially thousands of tables used by the application. ILM should make decisions on row storage requiring as little input from users as possible.
- **Respond to changing workloads**: Applications often cause changing workloads on some tables. If the system can auto-tune to these changing patterns, then user intervention is not required.
- **Low transaction impact**: ILM processing should have low transaction impact on user transactions. Any additional processing should be performed in the background with little or no blocking for user transactions.
- **OLTP characteristic tuning**: ILM processing takes into account table profiles typically seen in traditional OLTP workloads. We anticipate that for most OLTP workloads tables can be characterized into following broad categories.

- Small and frequently updated e.g., **warehouse**.
- Medium table which is frequently inserted or updated.
- Large table which is insert only or update heavy, but usually only a small portion of such large tables are active.

These characteristics serve as guiding design principles for ILM. Small and frequently updated tables are to be retained in the IMRS. For medium-sized tables, data is attempted to be retained while it is active and typically we expect only some portion of the table to be residing in the IMRS. For large tables, we expect that some small slice of data is typically active, so a small percentage of such large tables' data may be in-memory.

IV. STORING HOT DATA IN-MEMORY

We decide whether to store a row in-memory at runtime when a statement makes access to the row. Under the assumption that a newly inserted row will most likely be accessed again, for the most part, inserts will be directed to the IMRS initially, thereby also avoiding any contention at the page-level.

At the first access to a row in the page-store, it is not easy to predict if the row would be accessed frequently in near future. Simple heuristics based on scan type, hotness of buffer etc. are used to determine row hotness. We take into account access patterns specific to a workload to determine which rows are considered "hot". For example, a row from the page-store is brought into the IMRS if it is accessed through a unique index (point query or updates), in anticipation that such rows may be re-accessed by the workload. Most OLTP tables tend to have a primary key, and access driven by unique index key access is a commonly-seen usage. For example, in TPCC, the **warehouse** table is accessed and updated by most of the transactions, driven by primary key access. By our technique, in this case, all rows in the **warehouse** table will be considered "hot" and will remain in-memory.

V. AUTO IMRS PARTITION TUNING

SAP ASE supports semantic partitioning of tables (and indexes). For a partitioned table, individual partitions may be affected differently by the workload. As an example, in a range-partitioned **orders** table, partitioned on the *order_date* column, the rows from partition holding most recent orders that are processed will tend to be "hot". An unpartitioned table is treated as single-partitioned table. All ILM techniques, such as monitoring, metrics collection, analysis etc., are applied at a partition-level. For an unpartitioned table, these will be applied at the table-level. In the following sections, the term *partition* must be understood as an individual data partition for a partitioned table, or the entire table for an unpartitioned table.

This technique disables or re-enables use of in-memory storage for certain ISUD operations on certain partitions by monitoring the workload on rows in the IMRS and in the page-store. Auto IMRS partition tuning results in the following choices:

- Disallow storing rows in-memory for some partition if it finds that the rows brought in the IMRS for that partition are not significantly reused by the workload.
- Enable IMRS use for a partition if performing in-memory operations may provide large performance gain over page-store due to issues like contention (in the buffer cache) or change in workload and possibility of increase in reuse pattern for the rows in a partition.

Using the IMRS for all the tables may require large memory but users may not want to (or may not be able to) handpick some hot tables as IMRS-enabled. Auto-partition tuning technique allows the user to enable IMRS for all the tables and ensures that the server will intelligently use IMRS-storage only if it benefits for a specific partition / table, thereby keeping memory usage optimal.

Auto-partition tuning involves a set of strategies that are described below.

A. Monitoring the workload

To effectively perform workload analysis, some counters need to be maintained by the execution engine. However, maintaining counters slows down the transaction performance especially in multi-core system due to cache-invalidations resulting from updates to the counters.

To avoid the performance degradation due to monitoring, an efficient mechanism to monitor the workload is provided. This is implemented using per-CPU core-friendly counters to capture various operations happening in the IMRS and aggregating them across all the counters to get the current value of the counter. This ensures that there is no cache invalidation to modify this counter, as the memory for a counter is updated on only one core and the counter always exists in the cores' L1/L2 cache. Many of the ILM-techniques explained later use these simple counters. Some of the important counters used are: Partition-specific IMRS-memory used, number of rows stored in-memory for a partition, total number of operations which accessed row stored in-memory for the partition (re-use count), number of operations performed on pages in the partition, number of operations on page-store which observed contention etc. In addition, per-row access timestamps are maintained to loosely track row hotness. These timestamps are updated occasionally when rows are accessed, and are not seen to cause any performance overheads.

B. Self-tuning

Auto-partition tuning to make disablement and re-enablement decisions is performed by a background Pack thread. This is important as the user does not need to execute some commands periodically for this to happen and the system responds to the workload continuously. Self-tuning is done by the Pack thread which wakes up after some large number of transactions complete and then examines the above counters to observe various patterns and makes the decision accordingly. The time window between such large number of transaction is referred to as the tuning window, usually in the order of a few minutes. Self-tuning decisions take into consideration counters observed in the previous and present tuning window to respond to changing workload patterns. Since self-tuning uses counter difference to identify new IMRS usage for a partition it results in access-pattern based ageing. For example, if a partition had high re-use initially and then was not used much later, then later tuning-cycle would determine partition as cold as it does not rely on historical counters alone.

The choice of either enabling or disabling IMRS usage for a partition is only applied if the same choice is made successively for few tuning windows. This avoids a situation of hysteresis where dynamically changing workloads repeatedly result in a change in the IMRS-enablement for a partition.

C. Disable in-memory operations on a partition

Partition tuning disables IMRS usage for partitions after careful workload analysis of operations performed in the IMRS. Following heuristics come into play for partition disablement:

- Average reuse of rows: Re-use of rows is the number of select / update / delete operations on rows while they are in the IMRS. Partitions with low re-use rate for rows may not benefit much by storing rows in the IMRS, rather they will unnecessarily consume the IMRS memory. If a partition has a low re-use rate, then its IMRS usage is likely to be disabled.

- Partition IMRS utilization: If the memory footprint of a partition in the IMRS is small (say, < 1% of the IMRS cache), it is not considered for disablement. Such small partitions do not consume much memory, therefore, disabling them may not gain much IMRS cache capacity. This heuristic also guards against a premature disablement decision when a table is newly created, or data is loaded into an empty table.

- IMRS cache utilization: If in-memory storage has lots of free memory then none of the partitions are considered for disablement. Reason behind this heuristic is that it is not necessary to turn off IMRS usage if there is enough memory in the cache. This heuristic guards against a premature disablement decision after a server boot (when applications are initializing new partition accesses) or for a new database creation.

- New IMRS usage by a partition: Slow growing partitions do not cause a huge load on the IMRS cache. Therefore, if there are not enough new rows brought into the IMRS for a partition, then it may remain as IMRS enabled. This heuristic also avoids making a disablement choice for partitions which are active during only some intervals of the day, week, etc. For example, continent specific partitions.

D. Enable in-memory operations on a partition

Partition tuning may turn off IMRS usage for a partition due to low re-use operation on the rows in the IMRS for that partition. Such disablement could result in performance drop in some cases. This technique internally identifies such cases and re-enables use of the IMRS for such partitions using the following heuristics:

- Contention on the page-store: If a partition is disabled for IMRS use and operations on the page-store experience contention then such partitions may be re-enabled for IMRS use.

- Increase in reuse operation: If the number of reuse operations on a partition during the tuning window increases considerably compared to the reuse in the tuning window in which the partition was disabled for IMRS-use then the partition is again re-enabled for IMRS-use.

VI. PACKING COLD DATA FROM THE IMRS

Identifying cold data in the IMRS and relocating such data to the page-store (sometimes also referred to as anti-caching [17]) is a key component of our architecture. We call this operation as Pack. In our BTrim architecture [3], Pack operation is offloaded from user transactions and is performed automatically by one or more background Pack threads. The pack sub-system must balance the volume of data packed versus the load of newer data coming to the IMRS. This section describes the techniques used to achieve the following:

- Determine if a row is cold: It is important to identity if a data-row is cold or not before packing it. If a hot row is packed, it may be accessed by subsequent transaction(s) which will again bring it back to the IMRS. This not only wastes processing performed by the pack operation but also slows down transactions as they have to access hot data from the page-store and migrate it to the IMRS.
- Locate cold rows efficiently: The in-memory store may have a lot of rows and many of them will be hot rows so locating colder rows quickly is important. If the pack sub-system spends a lot of time in finding such cold rows, then it will be inefficient and may not be able to keep with the new load coming to the IMRS.

The rest of this section discuss strategies implemented to efficiently locate and pack cold rows, and to maintain stable IMRS capacity.

A. Steady Cache Utilization

The design goal of ILM and Pack is to keep utilization of IMRS cache stable and at a reasonably high value (e.g. 70%). Keeping it stable is even more important as it ensures predictive performance. Unpredictable variations in system performance due to varying resource utilizations is not something that customers would like to see.

To ensure a steady cache utilization, we provide a user-configurable threshold called **steady cache utilization percentage**. As the workload increases, so does the cache utilization whereas the pack sub-system tries to decrease the utilization. The ILM schemes for transaction processing and pack sub-system try to keep the cache utilization hovering around this threshold. In Sec. VIII (c), we provide experimental evaluation of steady cache utilization in an OLTP setup. This threshold is used as follows to keep the cache utilization stable.

The background pack threads wake up to pack the data only when the cache utilization exceeds this threshold. Pack threads run in one of the following levels based on current IMRS cache utilization:

- Steady-State Pack: This is the default mode for pack where rows are packed only if they are cold as defined by ILM rules.

- Aggressive Pack: If the cache utilization exceeds steady cache utilization and is more than half the difference between that configured value and the cache size, then the pack sub-system start packing more aggressively without applying row-hotness heuristics. In such a case, even hot rows could be packed to free up memory.

If cache utilization increases while aggressive pack was happening, server decides to stop storing new rows in the IMRS until cache utilization drops (as a consequence of pack). Meanwhile, all operations will be performed on the page-store, temporarily resulting in perhaps sub-optimal performance, however, without causing any application outage. This ensures that the pack sub-system is not over-loaded by incoming newer data and needs to pack only the existing cold data in the IMRS.

B. Partition-level Relaxed LRU Queues

To quickly locate the cold rows to perform a pack operation, our system uses a variant of relaxed LRU strategy used in traditional buffer cache-replacement schemes. As individual rows are being identified as being cold, the design attempts to keep the book-keeping overheads of tracking access to rows low.

Relaxed LRU queues are maintained to track cold rows. Cold rows are expected to be found at the head of a queue, and hot rows toward the tail of the queue. Important aspects of such queues that help to efficiently execute various pack heuristics are described below.

- Partition level queues: Separate queues for each partition are maintained as opposed to one queue for all rows in the database (or in the IMRS, across all tables). We chose per-partition queues rather than a single LRU queue across all tables in the IMRS for the following reasons. Individual per-partition queues better reflect the activity which may vary across partitions, and over time. Per-partition queues also help to quickly locate packable-cold data from colder partitions. Moreover, our overall pack system is driven with the help of workload analysis on different partition accesses. A single-queue of rows across all tables runs the risk that a certain row may appear "cold" relative to all the rows in the IMRS, but is likely to be a more active row for the small set of rows in the specific partition the row belongs to. A key distinction between classical cache-replacement strategy and Pack is that the latter is intimately tied to operating on a tables' rows. Cache replacement may simply evict "cold" rows from the cache, but Pack, on the other hand, has to *collect* a bunch of cold rows from one partition (or table), **remove** them (logged-delete) from the IMRS and **move** them (logged-insert) to the page store. Partition level queues for cold rows help in consolidating these operations while packing candidate rows from one partition. For example, accessing the metadata of a table to pack a set of rows can be done once for a batch of packable rows. This allows pack threads to move data to the page store more efficiently for a single partition than by using a single queue at the database level, wherein cold rows from different tables could be inter-mingled.
- Multiple queues for each partition: Each partition has multiple queues based on the operation which brought the rows into the IMRS. There are separate queues, one each for *inserted* rows, *migrated* rows (rows updated from page-store to IMRS) and *cached* rows (rows selected from page-store and cached in the IMRS). Having separate queues help because hotness

characteristics for each of the row types or partitions may be different. For example, the **new_orders** table being a heavily-inserted table ends up with more rows in the inserted rows queue. This table, being a queue-like table, is more likely to have newly-inserted rows updated / processed. Older rows that reside in the page-store are less likely to be updated or scanned, so the migrated or cached queues, respectively, for such rows tend to be less useful.

• Queue Maintenance offloaded from transactions: IMRS-GC threads have to process every IMRS row created by a transaction to reclaim memory from obsolete versions. We piggy-back on this activity as follows. GC threads insert a newly created IMRS row(s) at the tail of the ILM-queue. If a pack thread finds a hot row at the head of the queue, it will not pack the row but move it to the tail of the queue. This way, hot rows will be gradually relocated to the tail of the queue, bubbling up colder rows to the head for further packing. This gives a behaviour similar to that of LRU and also avoids performance overheads of constant row shuffling.

This design helps in two ways: (a) Since it is not performed in a transaction's execution path, transaction response time is not affected. (b) As queues are maintained by background threads, which are far fewer in number as compared to number of active transactions, any contention to maintain such queues is very low.

C. Partition-Aware Pack Selection

We expect that in an OLTP-workload the data coldness depends a lot on table partitions, their sizes, and type of operations on the partitions. Analysing these patterns provides information regarding the cold data in the IMRS. Some examples are mentioned below.

- *Number of reuse operation*: We call SELECT, UPDATE, DELETE as operations which could re-use rows which are bought in the IMRS by a previous operation. A partition having a lower rate of re-use operation (w.r.t. number of its rows in the IMRS) has more number of cold rows compared to a partition having higher reuse rate. For example, **history** being an insert-only table has very low reuse rate compared to **orders** table so **history** table would have more cold data to pack.

- *Growing vs stable partition*: Our design anticipates an access pattern that in a partition which is constantly growing some part of data is hot for some time and may not be hot afterwards. Usually in growing tables, newly inserted data is more hot, and then cools off after the business activity is completed. Whereas in small and stable tables, (number of rows remains mostly static) most of the rows will be equally hot and may not have a lot of cold rows to pack.

The following techniques ae used to efficiently identify cold rows based on access patterns to partitions.

• Pack cycle and pack transactions:

The Pack sub-system packs data in time epochs referred to as a pack cycle. In each pack cycle, the pack sub-system tries to pack some small percentage of current IMRS cache utilization, referred to as NumBytesToPack. The idea being that, due to the transactional workload, if cache utilization is growing sufficiently to trigger a pack activity, then the pack sub-system's goal is to bring down the utilization gradually by small percentages, and not dramatically. This small percentage of current cache utilization translates to number of bytes that need to be packed.

A naïve approach could be to distribute the NumBytesToPack bytes uniformly across all active partitions. This has the downside that all or most of the rows from some small partition (e.g. **warehouse** table) are unnecessarily packed, even though they are hot. Our approach is an improved design over this naïve solution. At the beginning of a pack cycle, this number of bytes to pack are distributed among active partitions of IMRS enabled tables based on the current footprint (memory-usage) of the partitions and their (re)usability in the IMRS. This process is referred to as apportioning bytes to pack for each partition for the pack cycle. Cold rows from one partition are packed by one thread, in smaller pack transactions, till the target number of bytes apportioned to each partition are released after packing. Once all target bytes are processed the current pack cycle finishes and the next pack cycle starts with latest metrics for memory footprint and re-usability across all partitions.

• Pack cycle-byte distribution:

Based on the metrics collected, various indexes, as described below, are computed, which lead to the target number of bytes to pack value for each partition.

- Usefulness Index (UI): UI is an indicator of how useful it is (or has been) for storing rows in-memory based on the re-use of those rows. Usefulness of rows for each partition UI_ρ is determined by considering SELECT, UPDATE, DELETE operations that happened on the rows stored in-memory for the partition ρ. More SUD operation means more usefulness. UI is computed by averaging the usage metrics across all IMRS-enabled partitions P.

$$UI_\rho = \frac{(Sel_\rho + Upd_\rho + Del_\rho)}{\sum_{\rho\in\text{P}}(Sel_\rho + Upd_\rho + Del_\rho)}$$

Number of inserts to a partition does not figure in this index as the usefulness is determined by number of reuses that have occurred to rows already in an IMRS. For example, in an insert-only partition, the number of new inserts may be high, but usefulness index is low if subsequently the inserted rows are not selected or updated.

- Cache Utilization Index (CUI): CUI is a relative metric across partitions comparing memory footprint in the IMRS for different partitions. This is determined by comparing memory consumption $Bytes_\rho$ of partition ρ to cache utilization by other partitions. The larger partitions being prime candidates for packing are taxed heavily (i.e. more of their colder rows may be packed) so as to make more memory available.

$$CUI_\rho = \frac{Bytes_\rho}{\sum_{\rho\in\text{P}} Bytes_\rho}$$

Note, pack sub-system only comes into play when cache utilization is beyond a configurable threshold. So, in above expression if the total cache usage (denominator) is still low (say, < 50%), pack sub-system is not activated. In other words, the algorithm is sensitive to CUI only when IMRS memory is used sufficiently.

- Packability Index, PI: Based on these two indexes, a packability index (PI) of a partition ρ is computed as below. This index gives a relative score for what proportion of a partitions' rows in the IMRS could be packed. If a partition has high cache utilization, then its usefulness index has to also be higher otherwise its rows are candidates to be packed (i.e. due to low usefulness index).

$$PI_{\rho} = \frac{\left(CUI_{\rho}/UI_{\rho}\right)}{\sum_{\rho \in \mathrm{P}}\left(CUI_{\rho}/UI_{\rho}\right)}$$

- Bytes to Pack: Finally, the bytes to pack (PACK_BYTES) from each partition ρ during a pack cycle are determined by distributing the total number of bytes to pack $NumBytesToPack$ in a pack cycle across all the partitions P in the proportion of their packability index.

$$PACK\_BYTES_{\rho} = PI_{ptn} \times NumBytesToPack$$

D. Determining Row Hotness

The Pack sub-system tries to keep the most recently accessed rows in memory. It uses a timestamp based filtering mechanism to retain rows that are accessed recently as well as frequently in the IMRS. Timestamp filtering mechanism tries to filter rows based on most recent access to rows in the IMRS. Both SELECT and UPDATE statements are counted as accesses. (Deletes are not interesting for this technique as this operation removes the row from the IMRS.) Server internally maintains and learns the timestamp filter based on the load created on the IMRS cache by current workload.

1) Applying Timestamp Filter (TSF)

In a running system, this design attempts to keep utilization of in-memory cache stable and at a higher value (e.g. 70%). We call this percentage as "steady cache utilization" percentage. We use this steady percentage to apply TSF for in-memory storage.

Time Stamp Filter (Ʈ) approximates the number of transactions which would cause memory utilization in the in-memory cache to increase by a small percentage of current cache utilization. From that number, we extrapolate the number of transactions which would cause cache utilization to increase by steady cache utilization percentage P. If memory utilization has already reached this steady level, then pack needs to pack rows. With recent access being the parameter for determining hotness, a row which is being operated by any of the last Ʈ transactions should not be packed as it is a hot row and the IMRS probably has more cold rows to pack.

Databases usually maintain an atomic counter which is incremented when transaction in the database completes; this is called as database commit timestamp. Thus, during pack operation, a row is considered cold if its last access timestamp is greater than commit timestamp by at least Ʈ value.

$$ROW\_IS\_COLD(\gamma) \overset{\text{def}}{=} COMMI\ \_TS(db) - ACCESS\_TS(\gamma)\ > \tau$$

Learning/tuning TSF delta is performed heuristically by monitoring how many transactions in the workload cause memory usage to increase by small percentage (e.g. 1-5%). This learning is performed in background as transactions complete in the database.

- When the tuning-cycle starts, current cache utilization ($Size$) and current commit-ts is recorded (T_1).
- During a tuning-cycle, when memory utilization increases by the required small percentage (δ), current commit-ts is recorded (T_2). TSF (Ʈ) is then computed as

$$\tau = \frac{\left((T_2 - T_1) \times \mathrm{P}\right)}{\delta}$$

To handle the change in workload, system re-learns the TSF again after some time.

2) Partition Awareness for TSF

We consider recency as well as frequency of accesses to data while determining if a row is cold/hot to pack.

- Recency of access:

The above learnt timestamp filter is applied during the pack operation. If a pack operation finds the difference between the current database timestamp and oldest modification timestamp of the row is less than the timestamp filter, (i.e. the row was updated sometime in a window given by the timestamp filter) then such rows are considered hot and are skipped for packing. This application of timestamp filter considers recency of access to the data rows.

- Frequency of access:

However, we don't apply the timestamp filter for all the partitions as we know some of the partitions don't have access pattern for very high row re-use. For such partitions, discarding rows to pack is not desirable as it wastes the processing cost without much gains. In fact, our pack cycle mechanism prioritizes rows from such partitions to be packed first even if they were inserted or updated in the IMRS later than rows in some other high-reuse partitions.

We don't apply timestamp filter to determine row hotness during pack if the reuse rate is very low for a partition. This technique ensures that the frequency of access to the data rows is considered.

$$REUSE\_RATE_{\rho} = \frac{Sel_{\rho} + Upd_{\rho} + Del_{\rho}}{NumRows_{\rho}}$$

Consider for example, due to page contention seen on the page-store, ILM-techniques decide to perform insert on the **history** table in the IMRS. However, this table has very low re-use rate so it is desirable to pack early from this partition and make space available for newer data. In fact, the Pack cycle heuristics make sure that rows from this table are scheduled aggressively for packing. Even though some of these rows could be very recently inserted in the IMRS, due to low reuse rate they would get packed as timestamp filter is not applied on them.

VII. PACK-ILM INTEGRATION WITH CONCURRENT ISUDS

ILM methods are woven through data processing to seamlessly manage data movement between the in-memory and the page store.

A. Granularity of Data Movement

Individual rows can be moved between the page store and the IMRS. This allows fine tuning of hot data in memory. Data movement from the page store to the IMRS is done by ISUDs as part of the statement execution. Data movement from IMRS to the page store is done by pack sub-system in the background.

B. Non-blocking Online Data Movement

Data movement from the IMRS to the page store and vice versa is done in an online manner. DMLs move data to the IMRS while holding row level locks. This does not prevent other DMLs or pack threads from moving other rows from one store to another. Scanners are transparently redirected to the appropriate store. Scanners can be active on a row while there is data movement between stores by DMLs or pack. Scanners which need consistent data (isolation level read committed and above) handle this by looking up the row in the IMRS after acquiring a lock. Since data movement needs locks on the rows, scanners can safely access the row. Scanners which do not take row level locks may access stale copy in IMRS or page store. Physical consistency of IMRS data seen by such scanners is provided by an internal technique called statement registration which blocks garbage collection until the scanner completes its work. Pack threads request a conditional lock on rows. If a row-lock cannot be granted, row is skipped for pack. This prevents active DMLs from blocking pack. Each pack transaction packs only a small number of rows and commits frequently. This prevents DMLs being blocked for a long time by rows which are already locked by pack.

VIII. EXPERIMENTS

We demonstrate the benefits of ILM classification and hot / cold data movement through an OLTP benchmark based on the TPC-C benchmark [17]. Experiments were run on a machine with Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz processor having 4 sockets / 60 cores / 120 logical CPU system and 1 TB RAM, SSD storage for data and log devices. Unless otherwise specified, experiments were done using scale factor of 240 warehouses for TPCC schema, 200 concurrent users, and SAP ASE with 64 threads.

In many of our experiments, we compare two setups for evaluation of ILM strategies.

- **ILM_OFF:** Does not use any of the ILM heuristics mentioned in the paper. In this run, all accessed data is fully memory-resident in the IMRS through the workload. All the ISUD operations store data in the IMRS. There is no background pack activity happening to move data to cold store and cache utilization keeps on increasing. This is akin to an unlimited IMRS size. Practically we configured 150 GB of IMRS cache.
- **ILM_ON:** Uses all the ILM heuristics mentioned in the paper to keep only hot data in the IMRS. Pack sub-system is configured to use 12 pack threads. The design goal of ILM_ON is to maintain stable cache utilization.

Our experiments focus on end to end throughput, transactions per minute (TPM), which is the metric conventionally used for this benchmark. By our design, as online transactions are unaffected by ILM / Pack, we do not anticipate any increase in transaction commit-latency. However, this has not been specifically measured, and is something that can be investigated in future work.

A. TPCC Tables and Workload Pattern

Our experiments were run on tables from the TPCC schema, so for quick reference and understanding of experiment data, this section provide typical workload pattern observed on these tables in TPCC benchmark run.

Table Name	**Workload Pattern**
warehouse, district	Small, medium-sized table respectively with high scan and update rates
stock	Large table with frequent update rates
item	Medium-sized read only table
history	Insert Only table
order_line, orders	Large tables. Heavy inserts, very low scans/update
customer	Medium-sized table. Heavy updates and some selects
new_orders	Both inserts and deletes (e.g. queue table)

TABLE 1: PROFILE OF TABLES SEEN IN THE TPC-C SCHEMA

B. Benefits of ILM Strategies

In this section, we capture the benefits of ILM strategies with help of the following parameters comparing transactions per minute (TPM).

- Relative TPM w.r.t. ILM_OFF: This parameter compares TPM with ILM_ON v/s ILM_OFF strategy.
- % operations in the IMRS (Hit rate): This parameter captures percentage of all operations done in the IMRS with ILM_ON. For ILM_OFF setup this implies hit rate of 100%, as data is fully cached.
- % reduction in cache utilization: This parameter measures how much less cache we could work with when ILM is on v/s ever increasing cache usage when ILM is off.

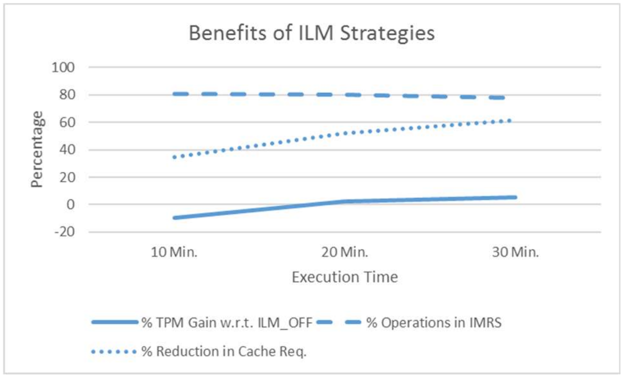

Fig. 1. Benefits of ILM strategies comparing relative throughput metrics

Fig. 1 shows the benefits of ILM by measuring the above parameters between the two schemes. The TPM gain is as compared to a baseline TPCC run on the page-store with the database fully-cached in the buffer cache. The TPM gain with ILM_ON is within +/- 10 % of what is observed in ILM_OFF setting (solid blue line in above figure). Note that ILM_OFF setting keeps all accessed data in memory, whereas ILM_ON setup only keeps hot data in memory. After a 30-minute run, the ILM_ON setup is able to operate with 60% of cache used in the ILM_OFF setup (dotted blue line in figure) even while keeping TPM largely unaffected. Even with this reduced cache usage we observed 80% hit rate with ILM_ON, as shown by the dashed blue line.

This shows that using ILM strategies, we can configure a system with a suitably smaller IMRS size and can run the workload in a stable manner without affecting performance.

C. Cache Utilization

This experiment demonstrates effectiveness of ILM strategies with respect to reducing cache requirement. Fig. 2 shows cache utilization as the benchmark runs progress. For ILM_OFF setup (effectively, *infinite* memory), as expected cache utilization keeps on increasing as the benchmark run progresses. With ILM_ON, cache utilization remains stable at around 44 GB. On-going product enhancements further reduce the memory used for in-memory rows by shrinking size of core structures and improved memory allocation. These enhancements are not seen in these experiments but are available in recent editions of the product.

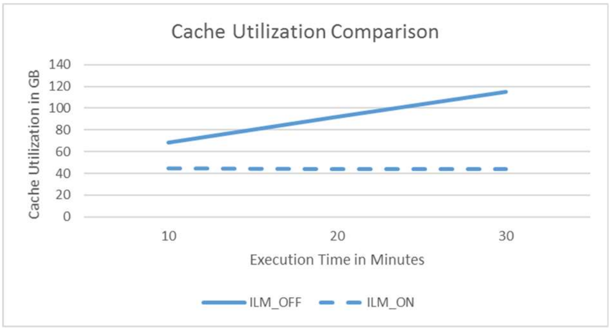

Fig. 2. Cache utilization comparison between ILM ON and OFF schemes

- Partition Level Cache Footprint

In this section, we show the per-table cache footprint between the two strategies. Fig. 3 shows how the IMRS cache footprint increases for ILM_OFF setup for each table as the benchmark run progresses. It can be seen that for most tables cache footprint is growing. This is expected behaviour with ILM_OFF setting as new insert / update / delete commands continue to bring in new data to the IMRS.

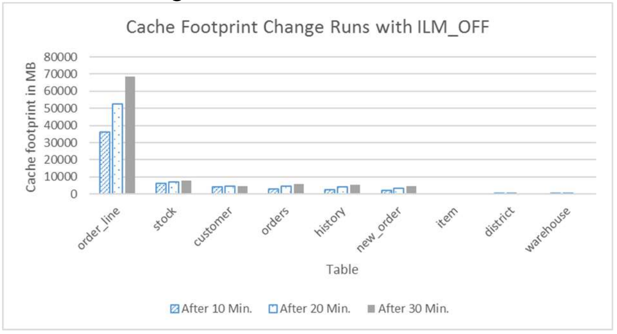

Fig. 3. Memory usage footprint of various tables with ILM_OFF

In comparison, Fig. 4 shows how the IMRS cache footprint changes for ILM_ON setup for each table as the benchmark run progresses Note that cache utilization is mostly stable for all the tables as the benchmark run progresses. This corroborates data in Figure 2, which shows that cache utilization is stable with ILM_ON strategy.

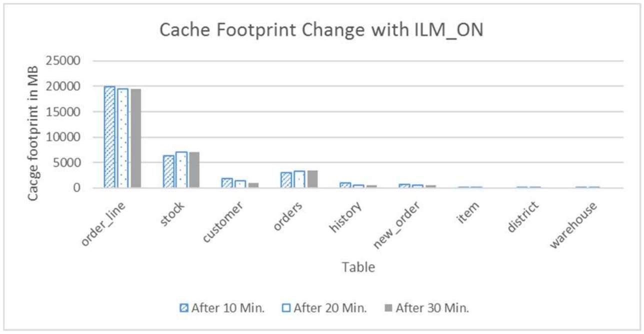

Fig. 4. Memory usage footprint of various tables with ILM_ON

Note that the cache footprint for small, hot tables such as **district** and **warehouse** tables, remains the same with both ILM_OFF and ILM_ON setups. This demonstrates that hot tables continue to remain in the IMRS even with lower cache utilization using ILM heuristics. Lower cache utilization is achieved by packing cold data to keep cache utilization steady. Comparing Fig. 3 and Fig. 4, we can see that most of the reduction in footprint comes from large tables like **order_line** and **orders** (both have high insert rates with low scans), and the **history** (insert only) tables which are cold. IMRS cache utilization for these different table types is in-line with our design expectations.

D. Pack Sub-system

Fig. 5 shows the impact of pack in ILM_ON setup. Pack is a logged data movement background operation affecting both transaction logs and stores. It is not expected to affect TPMs as it is performed by background threads operating on cold data. There is no pack in ILM_OFF setup. In ILM_ON setup, as expected, data packed in MBs increases as the run progresses. However, there is minimal impact on TPM and it remains within 10% of the TPM for ILM_OFF run (used as a reference TPM). This shows that even with the pack processing and logging overhead in both the logs, pack is a low overhead operation, which also helps to keep cache utilization constant.

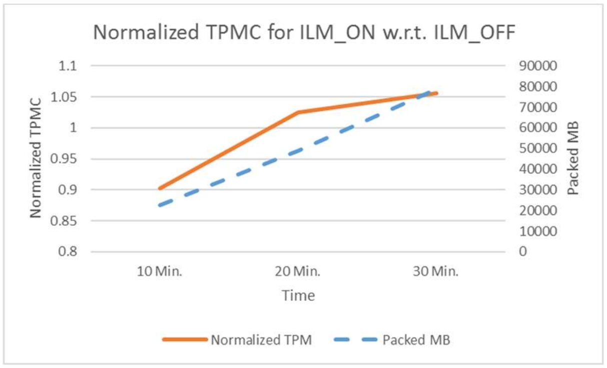

Fig. 5. Normalized TpmC, comparing Pack overheads with both strategies

From Fig. 3, with ILM_OFF the inflow rate of new data to the IMRS is about 2.5 GB / min. Correspondingly the outflow rate of cold data packed from Fig. 5 is also about the same, thereby ensuring steady cache utilization with ILM_OFF, as observed in Fig. 3.

1) Pack Cycle Distribution

As discussed in section VI, the pack sub-system assigns more "tax" to fatter cold partitions. This ILM_ON experiment demonstrates how pack adjusts number of rows to be packed based on re-use counts observed for rows in the IMRS and cache utilization (footprint) for each table.

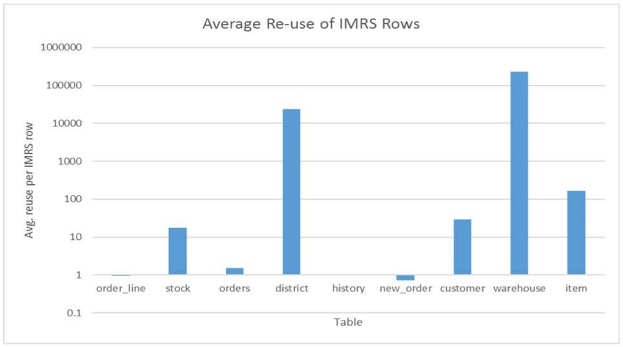

Fig. 6. Average per-row re-use counts across different tables

Fig. 6 shows the average re-use of rows in the IMRS for each table in the experiment over the 30-minute period. Since data access in TPC-C is skewed, logarithmic scale is used. For example, in the 30-minute period, on an average every row in the **warehouse** table was accessed 227K times. Cache footprints for various tables can be seen in Fig. 3. As expected by our design assumptions modelling OLTP workloads, small tables like **warehouse** and **district** show a very high cache re-use rate. Large tables like **order_line** show very little re-use but a larger in-memory footprint of active data. Medium-sized tables like **item** and **customer** show reasonable re-use rates with some limited cache footprint.

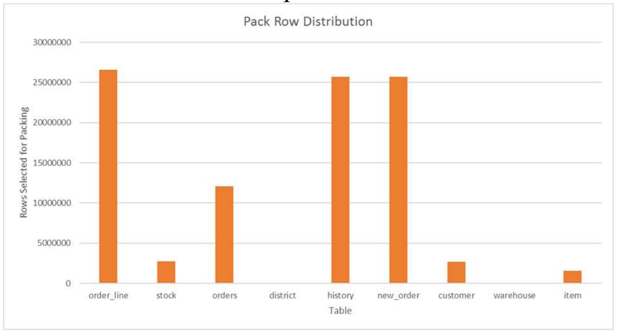

Fig. 7 Packed rows distribution across all tables, over 4 runs

Fig. 7 shows the metrics on number of rows packed across tables, aggregated over 4 runs. The **warehouse** table has high reuse rate (227400), and low cache footprint (71 Kb). Hence only 580 rows were selected for pack. Note that as this data is aggregated over 4 runs, the number of rows selected for pack can be more than number of rows in table. Contrastingly, the **order_line** table has high cache utilization (~19GB) and low re-use rate (0.93), hence large number of rows (26M) are selected for pack. Most of the rows for pack are selected from **order_line**, **orders, history**, and **new_orders** tables, all of which have high cache utilization and low re-use. This also explains their footprint remaining the same despite new DMLs on these tables.

2) Partition Level LRU Queues Distribution

This ILM_ON experiment describes how timestamp filtering is working in the benchmark runs and also explains how the use of partition level queues help to identify cold rows during pack.

Fig. 8 shows the percentage of cold rows in every 10% of rows from the head of the partition level queue for tables in the benchmark. Pack threads remove rows to pack from head of the queue and so having more cold rows at the head of the queue is efficient for the pack sub-system.

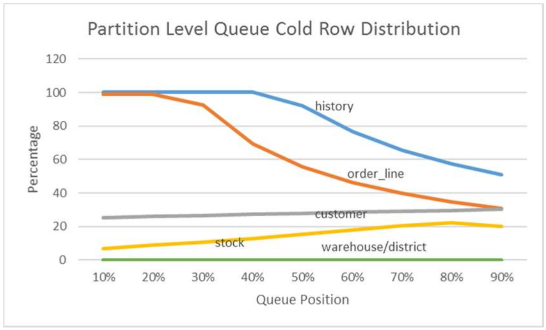

Fig. 8. Percentage of cold rows in every 10% of rows in ILM queues

For tables that are frequently accessed, like the **warehouse**, **district** and **stock** tables, nearly every range has equally hot rows. For other tables like **history** and **order_line**, we see a sheer drop in the percentage of hot rows beyond the initial 2 bands of rows, which is as expected by our designs.

3) TSF Tuning:

From Fig. 8 we can see that TSF is qualifying some rows as hot and some as cold. For example, for the **history** and **order_line** tables all rows up to 20% from the head of the queue are cold. As you near the tail of the queue for these tables, the percentage of rows that are cold drops to about 40-50%. These are large tables with many cold rows. **Warehouse** and **district** are small tables with all rows as hot. This matches with general understanding of the benchmark workload behaviour on the table. This shows that TSF technique is able to correctly model hotness patterns.

4) Partition Level Queues:

The Fig. 8 also partially justifies use of partition level (or, table level) queues, as opposed to a database-wide queue. Different tables and different partitions of the same table in the workload have different characteristics with respect to hot / cold rows. If one database level queue was used, then we anticipate that cold rows could occur anywhere in the queue and locating them during pack would have been difficult.

5) Well-behaved Queues:

Our techniques do not need any special processing to shuffle cold rows from the start of the queue due to transactions operating on such rows. We only move hot rows to the tail of the queue when pack discards such a row upon finding it hot. Due to partition level queues, we observed that the queues are well behaved with majority of cold data at head of the queue and hot data at end of the queue.

6) *Steady Cache Utilization*

To study the impact of the **steady cache utilization** threshold, we re-ran the TPCC experiment with the same setup as described earlier, using ILM_ON. Fig. 9 shows observed highest cache utilization for different values of this threshold. It can be seen that actual value of highest cache utilization follows value of configuration parameter. This shows that pack and ILM mechanisms are able to balance the demands on the IMRS to maintain cache utilization.

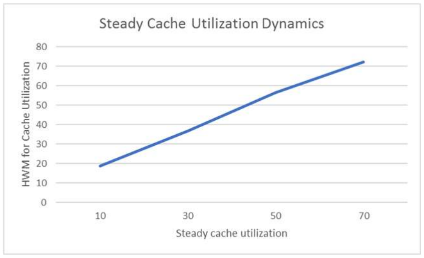

Fig. 9 HWM Cache Utilization for different values of steady cache threshold

For the same experiment, we explored the impact on overall normalized throughput and the work done by Pack sub-system.

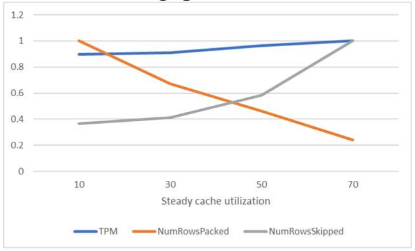

Fig. 10. Normalized ILM / Pack Parameters for steady cache utilization

Fig. 10 shows the normalized values of various parameters, with respect to the maximum value for each metric across different steady cache utilization thresholds studied. For example, the maximum TPM delivered was observed at the steady cache utilization threshold of 70%.

- NumRowsPacked shows that at lower steady cache utilization values, as expected, more data is packed.
- NumRowsSkipped is a metric for the number of rows skipped by Pack when they were found to be hot. This metric is gradually increasing at a slow rate. This is because at higher steady cache utilization, more rows are considered hot and are, therefore, skipped by pack. This also shows that TSF tuning is adapting well to the changing steady cache utilization.
- The TPM remain mostly unaffected because hot data is still retained at lower values of steady cache utilization. These values also show that background pack is not an expensive operation with reasonably configured cache size.

The results from Figs. 9 and 10 show that with ILM_ON, across different steady cache utilization thresholds, our system is able to maintain IMRS cache-usage reasonably around this threshold, while delivering stable performance gains.

IX. Related Work

Performance of database systems constrained by memory has been studied previously, but with significant differences from our work. In [16], 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.

In [17], 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 design however, the storage model starts initially in-memory and then pushes cold data to disk-storage. 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. Other high-level design choices such as which tables / partitions to consider for row-eviction, the number of cold rows that will be evicted etc., are similar to our work, but the actual logic is different at lower-levels.

Hekaton [4] is a Microsoft SQL Server's new DBMS engine optimized for in-memory processing of OLTP workloads. Hekaton calls tables which are stored in-memory as "Memory optimized" tables which need to be entirely memory resident. This is different from our approach where a table is not required to be fully in-memory, may have its data across page store and the IMRS, and could end up being fully memory-resident if the workload so requires.

Siberia [5] and [9] investigate cold data classification based on data access capture and offline analysis of access log. Siberia is prototyped on Hekaton and [9] is prototyped on VoltDB [11]. [5] and [9] investigate efficient estimation techniques based on log of record accesses. Estimation is done offline using log based on exponential smoothing based algorithms and cold data is moved to cold store (Siberia extensions for cold data movement are described in [10].) As an objective, Siberia estimates K hot rows, where hotness is identified by record access logs. Our work is fully integrated into the SAP ASE generally-available product. We run classification algorithm online using scalable engine level runtime counters. Our work internally adjusts number of hot rows that can be stored in-memory based on resource utilization and data sizes. No additional log collection is required by user. Partition tuning is performed by a set of background threads with configurable periodic cycle. Temperature classification is available in Siberia [5] and [9] at a row level. Our work also takes into account patterns where rows in a partition are accessed in similar manner. For example, in a table partitioned by month, partitions for recent months are more likely to be hot.

In addition, in our work selects can also bring rows to the IMRS, which is not a feature supported in these alternate schemes.

HyPer [7] and HYRISE [8] propose hybrid database systems capable of handling OLTP and OLAP workload. These systems require the whole database to be in memory. Again, the BTrim architecture imposes no such restrictions. Hyper advocates partitioning approach where data is partitioned such that most transactions need to access data only from a single partition. HYRISE auto-partitions tables based on type of access. Both these systems do not handle hot / cold data classification problem because of their full in-memory nature.

Funke et.al. [6] present an access based hot/cold data classification scheme for Hyper. Cold data is stored in a compressed format. Classification is at a page granularity in [7], where page stores a subset of data in a column. BTrim architecture stores data at a row granularity, and individual rows are classified as hot or cold. Classification at a row granularity potentially allows more data if only a few rows on a page are hot. In HyPer, classification is done using hardware assisted component called "Access observer". BTrim uses platform independent, lightweight scalable partition level metrics to determine re-use for classification.

Per our understanding, there is no other commercially available DBMS engine that tightly integrates cold / hot data classification and packing (anti-caching) seamlessly in a single product. We believe the availability of this work is itself a significant differentiator and we look forward to performance details from customer implementations.

X. Conclusions

In this paper, we presented various ILM-schemes to retain only the hot data in memory and store the colder data in traditional page-store. Our experiments show that, with the help of our ILM-techniques, we can get the performance gains of in-memory processing without requiring that all data be in-memory. Backed by ILM-strategies, the tiered in-memory storage was able to achieve performance close to a system where all the data was stored in memory.

The active working dataset often remains stable and our experiments show that we are able to deliver constant performance gains with stable cache utilization in the system. The stable cache utilization is one of the important parameters for using this technology in the field.

To ensure that the in-memory cache is used to store only hot data, it is important to have ILM-strategies for both (a) determining data hotness while storing data into in-memory cache (b) determining data coldness while packing/evicting data from the in-memory cache. The novel ILM-techniques presented in this paper are all driven by workload-characteristics and make use of partition-specific workload patterns to easily and efficiently determine the data hotness and coldness. Our experiments show that our techniques perform well in a representative OLTP workload to successfully identify hot and cold data. We are also looking to provide easy-to-use user configurations drawing on ILM rules to specify, for instance, that a small table be fully memory-resident, overriding ILM rules in specific cases. This will provide features such as fully in-memory tables and "pre-warmed" IMRS caches.

Acknowledgment

We would like to express our sincere appreciation to many other members of the ASE Product development and Server Performance Engineering teams who have contributed to developing and evaluating various techniques mentioned in this paper. We also acknowledge the feedback and internal reviews of this work received from personal mentors of authors.

References

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

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

[3] SAP ASE Product Documentation on In-memory row storage: https://help.sap.com/viewer/a1237e466dba417da6f0e5504cf9fb83/16.0.3.0/en-US/46211551447741638379 84cbe3fe0656.html

[4] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13). ACM, New York, NY, USA, 1243-1254.

[5] Radu Stoica, Justin J. Levandoski, and Per-Ake Larson. 2013. Identifying hot and cold data in main-memory databases. In Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013) (ICDE '13). IEEE Computer Society, Washington, DC, USA, 26-37. DOI: http://dx.doi.org/10.1109/ICDE.2013.6544811

[6] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting transactional data in hybrid OLTP&OLAP databases. Proc. VLDB Endow. 5, 11 (July 2012), 1424-1435. DOI: http://dx.doi.org/10.14778/2350229.2350258

[7] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11). IEEE Computer Society, Washington, DC, USA, 195-206. DOI: http://dx.doi.org/10.1109/ICDE.2011.5767867

[8] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. HYRISE: a main memory hybrid storage engine. Proc. VLDB Endow. 4, 2 (November 2010), 105-116. DOI=http://dx.doi.org/10.14778/1921071.1921077

[9] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13). ACM, New York, NY, USA, , Article 7 , 7 pages. DOI=http://dx.doi.org/10.1145/2485278.2485285

[10] Ahmed Eldawy, Justin Levandoski, and Per-Åke Larson. 2014. Trekking through Siberia: managing cold data in a memory-optimized database. Proc. VLDB Endow. 7, 11 (July 2014), 931-942. DOI=http://dx.doi.org/10.14778/2732967.2732968

[11] M. Stonebraker and A. Weisberg. The VoltDB Main Memory DBMS. IEEE Data Engineering Bulletin, 36(2):21--27, 2013.

[12] Transaction Processing Performance Council TPC-C Standard Specification:. http://www.tpc.org/tpcc/spec/tpcc_current.pdf .

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

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

[15] Plattner H., Zeier A., In-Memory Data Management: An Inflection Point for the Enterprise Applications, Springer publication, 2011

[16] G. Graefe, et al', In-Memory Performance for Big Data, Proceedings of the VLDB Endowment, Vol. 8, No. 1; pp. 37-48, 2014

[17] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, S. Zdonik, Anti-Caching: A new approach to Database Management System Architecture, Proceedings of the VLDB Endowment, Vol 6. No. 14, pp. 1942-1953, 2013.