Self-managed collections: Off-heap memory management for scalable query-dominated collections

Self-managed collections: Off-heap memory management for scalable query-dominated collections

Fabian Nagel
University of Edinburgh, UK
F.O.Nagel@sms.ed.ac.uk

Gavin Bierman
Oracle Labs, Cambridge, UK
Gavin.Bierman@oracle.com

Aleksandar Dragojevic
Microsoft Research,
Cambridge, UK
alekd@microsoft.com

Stratis D. Viglas
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

ABSTRACT

Explosive growth in DRAM capacities and the emergence of language-integrated query enable a new class of managed applications that perform complex query processing on huge volumes of data stored as collections of objects in the memory space of the application. While more flexible in terms of schema design and application development, this approach typically experiences sub-par query execution performance when compared to specialized systems like DBMS. To address this issue, we propose self-managed collections, which utilize off-heap memory management and dynamic query compilation to improve the performance of querying managed data through language-integrated query. We evaluate self-managed collections using both microbenchmarks and enumeration-heavy queries from the TPC-H business intelligence benchmark. Our results show that self-managed collections outperform ordinary managed collections in both query processing and memory management by up to an order of magnitude and even outperform an optimized in-memory columnar database system for the vast majority of queries.

1. INTRODUCTION

This work follows two recent trends in data management and query processing: language-integrated query and ever-increasing memory capacities.

Language-integrated query is the smooth integration of programming and database languages. The impedance mismatch between these two classes of languages is well-known, but recent developments, notably Microsoft's LINQ and, to a lesser extent, parallel streams and lambdas in Java, enrich the host programming language with relational-like query operators that can be composed to construct complex queries. Of particular interest to this work is that these queries can be targeted at both in-memory and external database data sources.

Over the last two decades, DRAM prices have been dropping at an annual rate of 33%. As of September 2016, servers with a DRAM capacity of more than 1TB are available for under US$50k. These servers allow the entire working set of many applications to fit into main memory, which greatly facilitates query processing as data no longer has to be continuously fetched from disk (*e.g.,* via a disk-based external data management system); instead, it can be loaded into main memory and processed there, thus improving query processing performance.

Granted, the use-case of a persistent (database) and a volatile (application) representation of data, coupled with a thin layer to translate between the two is how programmers have been implementing applications for decades and will certainly not go away for all existing legacy applications that are in production. Combining, however, the trends of large memories and language-integrated query is forward-looking and promises a novel class of *new* applications that store huge volumes of data in the memory space of the application and use language-integrated query to process the data, *without* having to deal with the duality of data representations. This promises to facilitate application design and development because there is no longer a need to setup an external system and to deal with the interoperability between the object-oriented application and the relational database system. Consider, for example, a business intelligence application that, on startup, loads a company's most recent business data into collections of managed objects and then analyses the data using language-integrated query. Such applications process queries that usually scan most of the application data and condense it into a few summarizing values that are then returned to the user; typically presented as interactive GUI elements such as graphs, diagrams or tables. These queries are inherently very expensive as they perform complex aggregation, join and sort operations, and thus dominate most other application costs. Therefore, fast query processing for language-integrated query is imperative.

Unfortunately, previous work [12, 13] has already shown that the underlying query evaluation model used in many language-integrated query implementations, *e.g.,* C♯'s LINQ-to-objects, suffers from various significant inefficiencies that hamper performance. The most significant of these is the cost of calling virtual functions to propagate intermediate

result objects between query operators and to evaluate predicate and selector functions in each operator. Query compilation has been shown to address these issues by dynamically generating highly optimized query code that is compiled and executed to evaluate the query. Previous work [13] also observed that the cost of performing garbage collections and the memory layout of the collection data which is imposed by garbage collection further restricts query performance. This issue needs to be addressed to make this new class of applications feasible for application developers.

Our solution to address these inefficiencies is to use *self-managed collections* (SMCs), a new collection type that manages the memory space of its objects in private memory that is excluded from garbage collection. SMCs exhibit different collection semantics than regular managed collections. This semantics is derived from the table type in databases and allows SMCs to automatically manage the memory layout of contained objects using the underlying type-safe manual memory management system. SMCs are optimized to provide fast query processing performance for enumeration-heavy queries. As the collection manages the memory layout of all contained objects and is aware of the order in which they are accessed by queries, it can place them accordingly to better exploit spatial locality. Doing so improves the performance of enumeration-heavy queries as CPU and compiler prefetching is better utilized. This is not possible when using automatic garbage collection as the garbage collector is not aware of collections and their content. Objects may be scattered all over the managed heap and the order they are accessed may not reflect the order in which they are stored in memory. SMCs are designed with query compilation in mind and allow the generated code low-level access to contained objects, thus enabling the generation of more efficient query code. On top of this, SMCs reduce the total garbage collection overhead by excluding all contained objects from garbage collection. With applications storing huge volumes of data in SMCs, this further improves application performance and scalability.

The remainder of this paper is organized as follows. In §2, we provide an overview of SMCs and their semantics before presenting a type-safe manual memory management system in §3. In §4, we introduce SMCs and show how they utilize our manual memory manager to improve query processing performance compared to regular collections that contain managed objects. Finally, we evaluate SMCs in §7 using microbenchmarks as well as some queries from the TPC-H benchmark. We conclude this work in §9.

2. OVERVIEW

SMCs are a specialized collection type designed to provide improved query processing performance compared to regular managed collections for application data accessed predominantly by language-integrated queries. This performance improvement may come at the expense of the performance of other access patterns (*e.g.,* random access). SMCs are only meant to to be used with data that is dominantly accessed in queries.

SMCs have a new semantics: they own their contained objects and hence the collection itself determines the lifetime of the objects. In other words, objects are created when they are inserted into the collection and their lifetime ends with their removal from the collection. This accurately models many use cases, as objects often are not relevant to the application once they are removed from their host collection. Consider, for example, a collection that stores products sold by a company. Removing a product from the collection usually means that the product is no longer relevant to any other part of the application. Managed applications, on the other hand, keep objects alive so long as they are still referenced. This means that a rogue reference to an object that will never be touched again prevents the runtime from reclaiming the object's memory. Object containment is inspired by database tables, where removing a record from a table entirely removes the record from the database.

The following code excerpt illustrates how the `Add` and `Remove` methods of SMCs are used:

```
Collection<Person> persons = new Collection<Person>();
Person adam = persons.Add("Adam", 27);
/* ... */
persons.Remove(adam);
```

The collection's `Add` method allocates memory for the object, calls the object's constructor, adds the object to the collection and returns a reference to the object. As the lifetime of each object in the collection is defined by its containment in the collection, mapping the collection's `Add` and `Remove` methods to the `alloc` and `free` methods of the underlying memory manager is straightforward. When the `adam` object is removed from the collection, it is gone; but it may still be referenced by other objects. Our semantics requires that all references to a self-managed object implicitly become `null` after removing the object from its host collection; dereferencing them will throw a `NullReferenceException`.[1]

SMCs are intended for high-performance query processing of objects stored in main memory. To achieve this, they leverage query compilation [12, 13] and support bag semantics which allows the generated queries to enumerate a collection's objects in memory order. In order to exclude SMCs from garbage collection we have to disallow collection objects to reference managed objects. We enforce this by introducing the `tabular` class modifier to indicate classes backed by SMCs and statically ensure that tabular classes only reference other tabular classes. Strings referenced by tabular classes are considered part of the object; their lifetime matches that of the object, thereby allowing the collection to reclaim the memory for the string when reclaiming the object's memory. We further restrict SMCs not to be defined on base classes or interfaces, to ensure that all objects in a collection have the same size and memory layout.

In contrast to regular managed collection types like `List<T>` our collection types require a deeper integration with the managed runtime. As collections allocate and free memory for the objects they host, we introduce an off-heap memory system to the runtime that provides type, memory and thread safety. The `alloc` and `free` methods of the memory system are part of the runtime API and are called by the collection implementation as needed. The type safety guarantees for tabular types are not the same as for automatically managed ones. We guarantee that a reference always refers to an instance of the same type and that this instance is either the one that was assigned to the reference or, if the instance has been removed from the collection, `null`. This differs from automatically managed types that

[1]This suggests that an ownership type system could be useful to statically guarantee such exceptions are not raised; but we leave this to future work.

guarantee that a reference points to the object it was assigned to for as long as the reference exists and refers to that object. To ensure type-safe reference accesses, we store additional information with each reference and perform extra checks when accessing an object. For managed types, references are translated into pointer-to-memory addresses by the just-in-time (JIT) compiler. As the logic for tabular types is more complex, we modify the JIT compiler to make it aware of tabular type references and the code that must be produced when dereferencing them.

We use query compilation to transform LINQ queries on SMCs into query functions that process the query. To improve query performance, the generated code directly operates on the collection's memory blocks (using `unsafe`, C-style pointers). All objects in the collections are stored in memory blocks that are private to the collections. Note that these blocks are not accessible outside the collection and the code generator. We assume that the structure of most LINQ queries is statically defined in the application's source code with only query parameters (*e.g.,* a constant in a selection predicate) dynamically assigned. We modify the C♯ compiler to automatically expand all LINQ queries on SMCs to calls to automatically generated imperative functions that contain the same parameters as arguments. Queries that are dynamically constructed at run-time, can be dealt with using a LINQ query provider as in [13]. The generated imperative query code processes the query as in [13], but on top of SMCs that enable direct pointer access to the underlying data.

3. TYPE-SAFE MANUAL MEMORY MANAGEMENT

Our manual memory management system is purpose-built for SMCs. It leverages various techniques to allow SMCs to manually manage contained objects and to provide fast query processing.

3.1 Type stability and incarnations

The memory manager allocates objects from unmanaged memory blocks, where each block only serves objects of a certain type. By only storing objects of a certain type in each block and disallowing variable-sized objects to be stored inplace we ensure that all object headers in a block remain at constant positions within that block, even after freeing objects and reusing their memory for new ones. We align the base address of all blocks to the block size to allow extracting the address of the block's header from the object pointer. This allows us to store type-specific information like `vtable` pointers only once per block rather than with every object. We refer to the memory space in a block that is occupied by an object as the object's *memory slot*. Object headers contain a 32-bit *incarnation number*. We use incarnations to ensure that objects are not accessed after having been freed. For each slot, the incarnation number is initialized to zero and incremented whenever an object is freed. References to objects store the incarnation of the object together with its pointer. Before accessing the object's data, the system verifies that the incarnation number of the reference matches that in the object's header and only then allows access to the object [1]. If the application tries to access an object that has been freed (*i.e.,* non matching incarnation numbers), then the system raises a null reference exception. The JIT compiler injects these checks when dereferencing a manually managed object. We do not expect incarnation numbers to overflow in the lifetime of a typical application, but if overflows should occur, we stop reusing these memory slots until a background thread has scanned all manually managed objects and has set all invalid references to `null`. Single-type memory blocks combined with incarnation numbers ensure type-safe manual memory management as defined in §2.

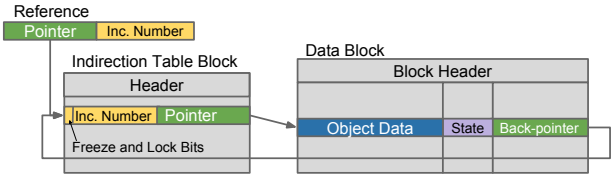

Figure 1: Accessing object data through indirection

3.2 Memory layout

We illustrate the memory layout of our approach in Figure 1. We do not store a pointer to an object's memory slot in its reference, but instead use a level of indirection. We will require this for the compaction schemes of §5. The pointer stored in object references points to an entry in the global indirection table which, in turn, contains a pointer to the object's memory slot. We store the incarnation number associated with an object in its indirection table entry rather than its memory slot. This allows us to reuse empty indirection table entries and memory blocks for different types without breaking our type guarantees.

As shown in Figure 1, each data block is divided into four consecutive memory segments: block header, object store, slot directory, and back-pointers. The object store contains all object data. Each object's data is accessible through a pointer from the corresponding indirection table entry or through the identifier of the object's slot in the block. The slot directory stores the state of each slot and further state-related information (for a total of 32 bits). Each slot can be in one of three states: *free i.e.,* the slot has never been used before, *valid*, *i.e.,* it contains object data, or *limbo i.e.,* the object has been removed, but its slot has not been reclaimed yet. Back-pointers are required for query processing and for compaction; they store a pointer to the object's indirection table entry. The slot directory entry and the back-pointer are accessible using the object's slot identifier.

3.3 Memory contexts

We have so far grouped objects of the same type in blocks private to that type. In many use cases, certain object types exhibit spatial locality: objects of the same collection are more likely to be accessed in close proximity. *Memory contexts* allow the programmer to instruct the allocation function to allocate objects in the blocks of a certain context (*e.g.,* a collection). The memory blocks of a context only contain objects of a single type and only the ones that have been allocated in that specific memory context.

3.4 Concurrency

Incarnation numbers protect references from accessing objects that have been freed. However, they do not protect objects from being freed and reused while being accessed. Consider Figure 2: Thread 2 frees and reuses the memory

Thread 1	Thread 2
`if (CHECK_INC(adam))`	
	`persons.Remove(adam);`
	`Person tom = persons.Add("Tom", 25);`
`PRINT(adam.name);`	

Figure 2: Concurrency conflict

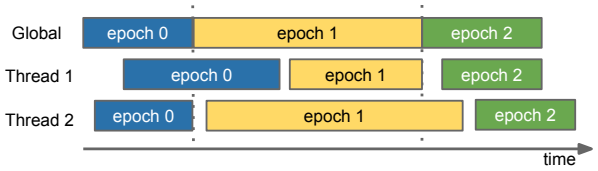

Figure 3: Epoch-based memory reclamation

slot referenced by the `adam` reference just after Thread 1 successfully checked the incarnation numbers for the same object. As Thread 1's incarnation number check was successful, the thread accesses the object, which is now no longer `Adam`, but `Tom`. This behavior violates the type-safety requirement of always returning the object assigned to a reference, or `null` if the referenced object has been freed. We refine the requirement for the concurrent case by specifying the check of the incarnation numbers to be the point in time where the requirement must hold. Thus, all accesses to objects are valid as long as the incarnation numbers matched at the time they were checked. To enforce the type-safety requirement, the memory manager ensures that if an object is freed, its memory slot cannot be reused for a new object until all concurrent threads have finished accessing that object.

We use a variation of epoch-based reclamation [7] to ensure thread safety. In epoch-based reclamation, threads access shared objects in grace periods (critical sections). The memory space of shared objects can only be reclaimed once all threads that may have accessed the object in a grace period have completed this grace period. Thus, grace periods are the time interval during which a thread can access objects without re-checking their incarnation numbers to ensure type safety. Epochs are time intervals during which all threads pass at least one grace period. The system maintains a global epoch; each thread maintains its thread-local epoch. In Figure 3, we show how we track epochs. Upon entering a critical section (grace period), each thread sets its thread-local epoch to the current global epoch. To leave a critical section, a thread can increment the global epoch if all other threads that currently are in critical sections have reached the current global epoch. Hence, threads can either be in the global epoch e or in $e-1$. Memory freed in some global epoch e can safely be reclaimed in epoch $e+2$ because by that time, no concurrent thread can still be in epoch e.

To implement epoch-based reclamation, the JIT compiler automatically injects code to start and end critical sections when dereferencing manually managed objects. Critical sections are not limited to a single reference access; several accesses can be combined into a single critical section to amortize the overhead of starting and ending critical sections. The following illustrates the code to start and end a critical section:

```
void enter_critical_section() {
  global->sectionCtx[threadId].epoch = global->epoch;
  global->sectionCtx[threadId].inCritical = 1;
  memory_fence(); }

void exit_critical_section() {
  memory_fence();
  global->sectionCtx[threadId].inCritical = 0; }
```

Upon entering a critical section, each thread sets its local epoch to the current global epoch and sets a flag to indicate that the thread is currently in a critical section; on exit the thread clears this flag. We have to enforce compiler and CPU instruction ordering around these instructions to ensure that the session context is set before we access the object and not unset until we have finished, hence, the memory fences. In contrast to [7], we do not increment global epochs *modulo* three, but as a continuous counter. We also do not increment the global epoch and reclaim memory when exiting critical sections, but in the memory manager's allocation function. This allows us to lazily reclaim memory on demand when allocating new objects.

3.5 Memory operations

When freeing an object, we increment its incarnation number to prevent subsequent accesses to it. We refer to memory slots that are freed, but not yet available for reuse as *limbo* slots. We set the memory slot's state to limbo and set its removal timestamp to the current global epoch in the slot directory. This bookkeeping ensures that the slot cannot be reclaimed until at least two epochs have passed. Memory blocks become candidates for reclamation when they surpass a threshold fraction of limbo slots, the reclamation threshold. If this is the case, we add the block to a queue of same-type memory blocks that may be reclaimed, along with the earliest timestamp when the block can be reclaimed (global epoch plus two).

All allocations are performed from thread-local blocks so that only one thread allocates slots in a block at a time (though there can be concurrent removals from the same block). Thread-local blocks are taken from the reclamation queue of the appropriate type if there are blocks ready for reclamation; if the queue is empty they are allocated from the unmanaged heap. To find a memory slot for a new object the allocation function scans all entries in the slot directory from the slot of the last allocation until either a free slot or a reclaimable limbo slot is found. The maximum number of slots scanned before finding a limbo slot that can be reclaimed depends on the reclamation threshold. For instance, if blocks can host one hundred objects and are added to the queue once they contain more than 5% limbo slots, then each allocation scans at most twenty slots to find a reclaimable limbo slot. The actual number is likely to be smaller as removals might have happened in the meantime. The allocation function attempts to increment the global epoch counter once there are blocks in the reclamation queue that cannot be reclaimed yet because two epochs have not passed.

4. SELF-MANAGED COLLECTIONS

SMCs use the type-safe memory management described in §3 and support the semantics of §2. The objects contained in an SMC are managed by the collection itself and not by the garbage collector. This, along with bag semantics, enables SMCs to place objects in memory based on the order the objects are touched when enumerating the collection's content in a query. This improves the locality of memory ac-

cesses when enumerating the SMC, leading to improved performance compared to iterating over the collection's content through references that may point anywhere in the managed heap (as is the case for all conventional .NET collections). A convenient side-effect of disallowing SMCs to contain standard objects is that it significantly reduces the size of the managed heap and the volume of memory that has to be scanned during garbage collection and, in consequence, the duration of garbage collection, which improves the overall performance of the application.

SMCs use the type-safe memory manager of §3 to manage contained objects. The semantics of SMCs mean that the `Add` and `Remove` methods can directly be mapped to the memory manager's `alloc` and `free` methods. In addition to allocating memory for the object, the `Add` method calls the object's constructor and returns a reference to the object.

Each SMC has a private memory context to allocate all objects added to the collection. This ensures that all objects in an SMC end up in the same set of private memory blocks. The SMC can access all of these blocks through the memory context. Recall from §2 that we automatically transform LINQ queries over SMCs into calls to specialized query functions that use query compilation to improve the performance of query processing. By giving the SMC access to these memory blocks, we also allow the query compiler to access them to enumerate over the SMC's objects. The following illustrates a simple compiled query that enumerates over all objects in the SMC by iterating over all valid slots in all blocks in the SMC's memory context, checking a predicate on the `age` field, and returning references to all qualifying objects:

```
enter_critical_section();
foreach (Block* blk in collection.GetMemoryContext())
  foreach (Slot i in blk)
    if (blk->slots[i] == VALID)
      if (blk->data[i].age > 17)
        yield new ObjRef { ptr = blk->backptr[i],
                           inc = blk->backptr[i]->inc };
exit_critical_section();
```

The query uses the memory block's slot directory `blk->slots` to check if the corresponding memory slot contains a valid object (in contrast to a free or limbo slot). As each entry in the slot directory is only four bytes wide and stored in a consecutive memory area, it is fairly cheap to iterate over the slot directory to check for valid slots. The query touches the object's data only if the slot is valid. If the slot also satisfies the selection predicate, the query returns a reference (`ObjRef`) to the object. To do so, it uses the back-pointer field `blk->backptr` to obtain a pointer to the corresponding indirection table entry. The reference contains this pointer and the current incarnation number of the object to ensure that the memory slot can safely be reclaimed once the object is removed from the SMC. To generate code for more complex queries we follow a similar strategy as in previous work [10, 12, 13, 14].

To ensure that the accessed objects are not removed and their memory slot is not reclaimed while directly accessing objects in a query, we have to be in a critical section. This applies to objects in the primary SMC that we enumerate as well as to objects in other SMCs that we access through references from the primary SMC. Instead of entering and exiting a critical section around each object access, we process huge chunks of data in the same critical section. This amortizes the cost of critical sections (in particular, memory fences) and, hence, is a cornerstone of providing good query performance. The query remains in the same critical section either for its entire duration, or for the duration of processing a single memory block. The query compiler chooses the desired granularity for each query based on the requirements of the query. Staying in the same critical section for the duration of the query allows to generate code that stores direct pointers to the memory locations of SMC objects in intermediate results and data structures (otherwise the query may only use object references). However, it also increases the time until the memory manager can increment the global epoch to reclaim limbo slots. As LINQ queries are lazily evaluated, we enforce that critical sections are exited before a result object is returned and, hence, control is returned to the application. Since queries often contain several blocking operations (*e.g.,* aggregation or sorting), most query processing is performed in a single critical section. Objects that are concurrently removed from an SMC while a query enumerates the SMC's content are included in the query's result if: (*a*) the query reads the object's slot directory entry before the slot is set to limbo, or (*b*) the query follows a reference to the object before its incarnation number is incremented. Objects added to an SMC behave accordingly. SMCs use a lower isolation level than database systems, in line with other managed collections.

4.1 Columnar storage

While SMCs manage the memory space of contained objects themselves, they keep the memory layout of the object's data unchanged. Previous work in database systems, *e.g.,* [2], has shown that some workloads, however, greatly benefit from a columnar layout, instead of the row-wise layout of SMCs. Since SMCs store all object data in blocks that only contain objects from the same collection and, hence, the same type, they can be easily extended to leverage a columnar layout. The only requirements are that: (*a*) the JIT compiler injects the code required to access columnarly stored data when following references to such objects, and (*b*) the query compiler is aware of the data layout and also generates code that accesses the data in a columnar fashion. For columnar layouts, we store the object's block and slot identifiers in the object's indirection table entry instead of a pointer to the object's memory location. To access the data of an object, we look up its memory block using an array of memory blocks indexed by their block identifier, and then use the slot identifier to find the position of the value in its column.

5. COMPACTION

Common uses of SMCs do not cause them to shrink significantly; they stay at a stable size or grow steadily. However, when facing heavy shrinkage of an SMC, we perform compaction to reduce the SMC's memory footprint and improve query performance. When relocating objects as part of a compaction, we have to ensure that concurrent accesses to them do not exhibit inconsistencies. Inconsistencies may arise from accesses through references or from queries directly operating on the SMC's memory blocks.

5.1 Reference access

The indirection table allows us to move data objects within and across memory blocks without having to update all ref-

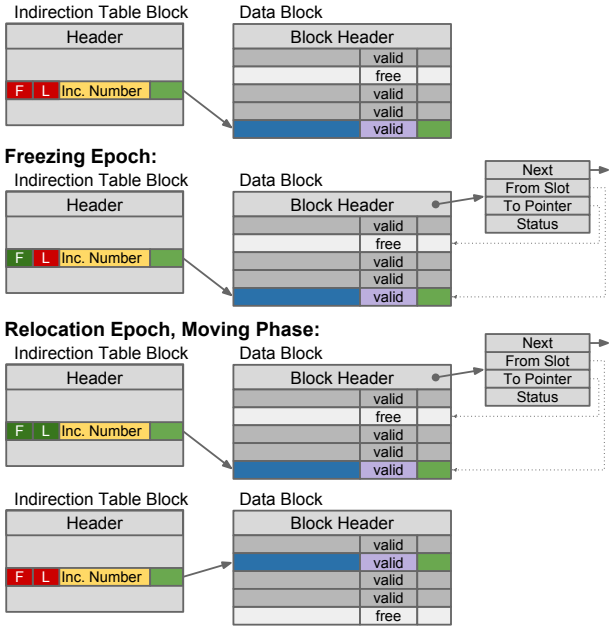

Figure 4: Relocating an object

erences held by the application. Atomically updating the pointer in the indirection table suffices to ensure that all threads can correctly reach the object. However, threads that already are in critical sections and point to the old location might cause inconsistencies by performing updates on outdated memory locations. To compact data blocks without stopping the application we extend the epoch scheme for object relocation. We reserve the two most significant bits of the incarnation number in the indirection table for a *frozen* flag [3] and a *lock* flag. After a thread successfully increments the global epoch, it checks if a compaction is necessary. The global epoch cannot be increased in the meantime because the thread is still in a critical section using the previous epoch. If compaction is necessary we set the global `nextRelocationEpoch` to $e+2$ (e is the thread-local epoch and $e+1$ is the global epoch we just incremented) and then awake the compaction thread. Once a relocation epoch is set, no other but the compaction thread can increment the global epoch until the compaction is finished (epoch $e+3$). To guarantee this, we run the compaction thread in a critical section that uses the thread-local epoch e, which prevents all other threads from incrementing the global epoch.

The compaction thread is active through two epochs: the freezing epoch $e+1$ and the relocation epoch $e+2$. In the freezing epoch it iterates over all blocks that need compaction (marked by previous allocations/removals). For each block, it constructs a list of all slots that have to be moved and the memory address the slots have to be moved to. This list is accessible through the block's header. The thread then sets the frozen bit in the indirection table entry of each slot that is scheduled to be copied.[2] Once all blocks are prepared for compaction, the thread waits until all other threads are in the freezing epoch ($e+1$) and then increments the global epoch to $e+2$ to start the relocation epoch. The relocation epoch consists of two phases: the waiting phase, which lasts until the compaction thread observes that all other threads are in the relocation epoch, and the moving phase that starts thereafter. While waiting, the compaction threat continuously tries to increment the global epoch to proceed to the moving phase. Once in the moving phase the compaction thread makes this phase globally visible by setting a global variable to indicate that frozen objects may now be moved. It then iterates over all blocks scheduled for compaction. For every slot to be moved, it atomically locks the incarnation number by setting the lock bit and copies the object to the new location, updates the pointer in the indirection table, unsets the lock and freeze bits, and marks the relocation as successful in the block's relocation list. Once all scheduled relocations are done, the compaction thread increments the global epoch to $e+3$ (all threads are guaranteed to be at $e+2$ by this point), exits its critical section to allow other threads to increment the global epoch, and goes back to sleep. Figure 4 illustrates the steps to move an object inside a memory block.

If an object's incarnation number is not frozen there is no risk of it being moved in the current epoch, so all threads can access it as before. Note that the incarnation number comparison that we have to do anyway is enough to cover the most common path. If we encounter a frozen incarnation number (*i.e.,* the first incarnation number comparison fails, but a second that excludes frozen and lock bits succeeds), there are three cases: (*a*) We are in the freezing epoch. There will not be any relocation in this epoch, so we can return the data pointer. (*b*) We are in the waiting phase of the relocation epoch and not all threads are in the relocation epoch yet. A relocation might happen while we access the object so we cannot proceed. However, we also cannot relocate the object because not all threads are in the relocation phase so they do not expect relocations yet. Our only option is to bail out from relocating the object. To do so, we find the object's entry in the block's relocation list, atomically set the lock bit in the object's incarnation number, set the status of the relocation to failed (in the object's relocation list entry), and unset the freeze and lock bits. If the lock bit has already been set by another thread, we spin until it is unset and then recheck the object's status. Once the freeze bit is removed, we can return the pointer and proceed. (*c*) We are in the moving phase of the relocation epoch and all other threads are also in the relocation epoch. We again cannot proceed because the object may be moved at any time, but we can help the compaction thread move the object to its new location and then proceed. To do so, we find the object's entry in the block's relocation list, atomically set the lock bit in the object's incarnation number, move the object to its new location, set the status of the relocation to succeeded, and unset the freeze and lock bits. As in the previous case, we spin if the bit is locked, then recheck its status and finally return the pointer after the frozen bit is unset. The following outlines the checks that have to be performed before accessing a manually managed objects through its reference:

[2]By using a CAS operation; this requires `free` to also use CAS to increment incarnation numbers

```
void* dereference_object(ObjRef oref) {
  if(oref.inc == oref.ptr->inc) {
    return oref.ptr->memptr;
  } else if (oref.inc == (oref.ptr->inc & FL_MASK)) {
    // First case:
    if (global->sectionCtx[threadID].epoch
        != global->nextRelocationEpoch) {
      return oref.ptr->memptr;
    // Second case:
    } else if (!global->inMovingPhase) {
      bail_out_relocation(oref);
      return oref.ptr->memptr;
    // Third case:
    } else {
      relocate_object(oref);
      return oref.ptr->memptr; }
  } else {
    throw new NullPointerException(); } }
```

Note that outside freeze and relocation epochs, the first condition is always satisfied if the referenced object has not been freed. If the object access is known to be read-only, we can always use the original location of the object in the waiting phase of the relocation epoch as its memory location cannot be reclaimed while we access it. In this case, the reader does not have to fail the relocation of that object.

When the compaction thread starts iterating over the blocks to be compacted (*i.e.,* the moving phase of the relocation epoch), all failed relocations are visible so the thread can deal with them. If necessary, it extends compaction by one additional epoch to try all unsuccessful relocations again by adding another freezing phase at the end of the relocation epoch and setting the following epoch to be a relocation epoch before exiting the current relocation epoch.

5.2 Block access

Queries directly operating on the memory blocks of an SMC can also cause inconsistencies where the query misses some objects because they are concurrently being relocated or includes them twice. To prevent these inconsistencies, we have to extend the compaction scheme described thus far. We always empty the memory blocks that take part in the compaction by moving their objects to new memory blocks and removing the emptied blocks from the collection. Blocks only participate in a compaction if their occupancy is below a threshold (*e.g.,* 30%). Blocks that participate in a compaction are assigned to *compaction groups* where the objects of all blocks in a compaction group are moved to the same new block. The number of blocks in a compaction group depends on the aforementioned threshold; a 30% threshold results in three blocks per group.

Queries process all blocks of a compaction group in the same thread-local epoch and in consecutive order. This ensures consistent query behavior outside relocation epochs as relocations may not start while processing the compaction group. During relocation epochs, we have to ensure that queries may either only access the pre-relocation state of a compaction group or the post-relocation state. If processing of a compaction group starts in the moving phase of the relocation epoch, the query first helps performing the relocation of the compaction group and then uses the compacted memory block for query processing. If processing of the group starts in the waiting phase, we cannot compact the group's content yet. In this case, we add the group to a list of groups that still have to be processed and continue with the remaining memory blocks. Once all remaining

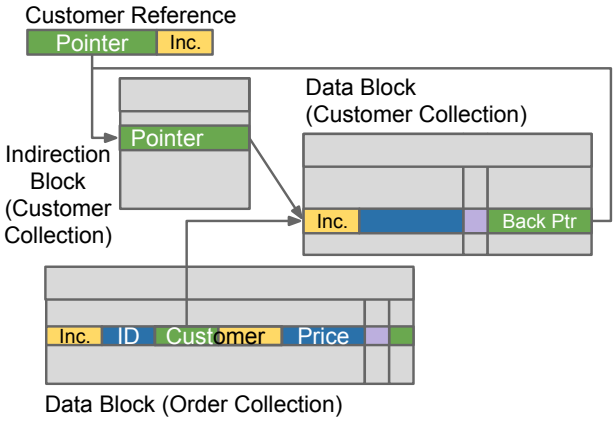

Figure 5: Direct pointer between collection objects

blocks are exhausted, we check if the moving phase has already started and, if this is the case, process all remaining compaction groups by first performing the relocation and then processing the compacted block. If the moving phase has not started yet, we process the compaction group in its pre-relocation state by atomically incrementing a query counter in the compaction group that prevents other threads from compacting the group until the query decremented the counter again. Relocations only occur in the moving phase of the relocation epoch and, hence, once a relocation waits for the query counter of a compaction group to become zero, there are no more queries incrementing it. The compaction thread bails out of compacting a certain group after waiting for a predefined amount of time for the read lock to be released. We do this to deal with queries that return control to the application (*i.e.,* return a result element) while holding the read lock.

6. DIRECT POINTERS

When a query touches an object that contains many references to nested objects, then SMCs may loose ground to automatically managed collections: each dereference not only has to check incarnation numbers, but, more importantly, it has to pay for an additional (random) memory access to the indirection table. We now provide an alternative implementation that solves this problem. We keep indirection for all external references, but, for references between SMCs, we store the direct pointer to the corresponding memory location. To be able to check incarnation numbers in both cases, the incarnation number of a memory slot is moved back into the memory slot (object header) instead of the indirection table. In Figure 5 we show the new layout, which improves query performance for queries that use references to access objects from several SMCs.

When relocating an object, however, the new memory location of the object now has to be updated in the indirection table as well as in all self-managed objects that reference it, which is no longer an atomic operation. We address this by adding a third flag to the incarnation number, the *forwarding* flag. The forwarding flag turns the object's old memory slot into a tombstone. Queries reaching the tombstone through direct pointers use the slot's back-pointer to access the object's indirection table entry which contains a

pointer to its new memory slot. To improve the performance of future accesses to this object, the query also updates the direct pointer to the object's new memory location. The forwarding flag is set by the thread relocating the object after completing the relocation in the same atomic operation that unsets the frozen and lock bits; hence, tombstones cannot be reached through (indirect) references. As was the case for the two other flags, checking the forwarding flag is performed during incarnation number checking and, hence, does not penalize the common case of an unset forwarding flag.

Tombstoned memory slots are not reclaimed until there are no more direct pointers to them. After compacting an SMC, the compaction thread scans all SMCs that have direct pointers to it and updates the pointers to relocated objects. Note that the references between SMCs are statically known and the compiler can produce specialized functions that only scan SMCs that have direct pointers that may have to be updated and only check the corresponding pointer fields. We improve the performance of scanning an SMC to update direct pointers by only following pointers to memory slots that are known to have been relocated. This saves many random memory accesses. We achieve this by building a hash table during compaction that contains the memory addresses of all blocks that are compacted and, instead of following a direct pointer to see if the forwarding flag is set, we first compute the address of the corresponding block, probe it in the hash table and only follow the direct pointer if the block address was in the hash table.

7. EVALUATION

We implemented SMCs as a library using `unsafe` C♯ code. We did not change the JIT-compiler to automatically inject the code for correctly dereferencing references to self-managed objects but added this code by hand to factor out any overhead. We implemented the code generation techniques of [13] and we did not use any query-specific optimizations. Our experimental setup was an Intel Core i7-2700K (4x3.5GHz) system with 16GB of RAM, running Windows 8.1 and .NET 4.5.2. We compare SMCs with the default managed collection types in C♯. Unlike SMCs, most collections in C♯ are not thread-safe (*e.g.,* `List<T>`, C♯'s version of a dynamic array). Thread-safe collection types in C♯ are limited and only `ConcurrentDictionary<TKey, TValue>` and `ConcurrentBag<T>` provide comparable functionality to SMCs; however, `ConcurrentBag<T>` does not allow the removal of specific objects. .NET supports two garbage collection modes: *workstation* and *server*. Both modes support either interactive (concurrent) or batch (non-concurrent) garbage collections. In our tests the server modes outperformed the workstation ones, so we only report results for the server mode and only report both concurrency settings if their results differ.

Our benchmarks are primarily based on an object-oriented adaptation of the TPC-H workload. We have chosen to focus on a database benchmark as we believe it exemplifies the class of large-scale analytics applications that will benefit from SMCs. A relational workload is the most typical example of an application that has traditionally offloaded 'heavy' data-bound computation to an optimized runtime for that data model (a relational DBMS). As such, it is a good indication of both the classes of queries that can be integrated in the programming language, while, at the same time, it

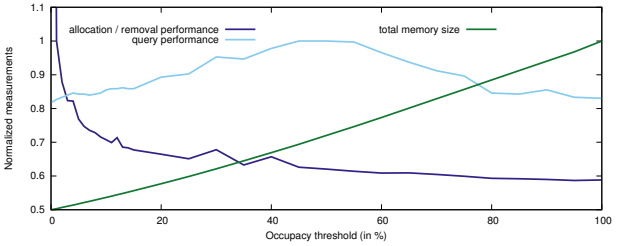

Figure 6: Varying the relocation threshold

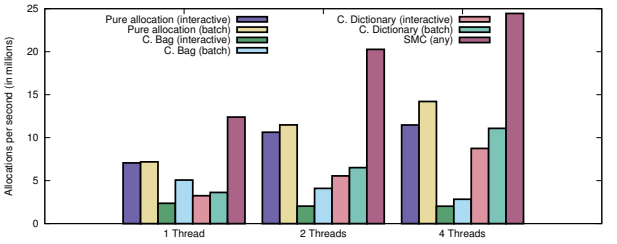

Figure 7: Batch allocation throughput

can provide an immediate performance comparison to the dominant alternative. TPC-H tables map to collections and each record to an object composed of C♯'s primitive types and references to other records (all primary-foreign-key relations). Based on the latter, most joins are performed using references. Unless stated otherwise, we use a scale factor of three for all TPC-H benchmarks. Note that due to a 16-byte-per-object overhead and larger primitive types (*e.g.,* `decimal` is 16 bytes wide) in C♯, a scale factor of three requires significantly more memory than in a database system.

Sensitivity to relocation threshold Recall from §3.4 that the data blocks of SMCs may contain limbo slots that cannot be reclaimed yet and that we use a tolerance threshold of such slots in a block that needs to be surpassed before adding the block to a reclamation queue. Varying this threshold affects the memory size, the cost of memory operations and the query performance of SMCs. In Figure 6 we show how these factors change when varying the threshold (normalized to the maximum value). As the percentage of unused limbo slots grows, so does the memory footprint of the collection. The cost of performing memory operations (*i.e.,* insertions and removals) slowly decreases with an increasing threshold as allocations have to scan less memory slots to find a slot that can be reclaimed. Query performance seems to be less dependent on the additional slot directory entries that have to be processed with an increasing threshold, but more on the branch misprediction penalties when verifying if the slot is occupied. At a 50% threshold, the branch predictor has the most trouble predicting if the slot is occupied. Based on the results of Figure 6, we will use a 5% threshold for the following experiments. For a 5% threshold, the memory requirements of SMCs are comparable to that of storing managed objects in `List<T>`.

Memory allocation throughput In Figure 7 we compare the throughput (in objects per second) of allocating `lineitem` objects (using the default constructor) in an SMC to the pure allocation throughput of managed objects in

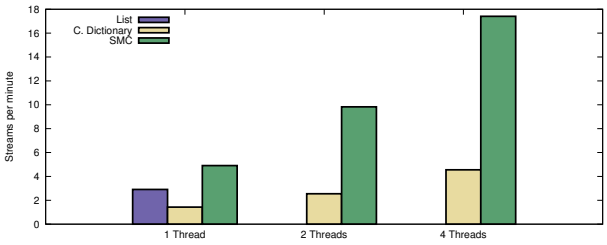

Figure 8: Refresh stream throughput

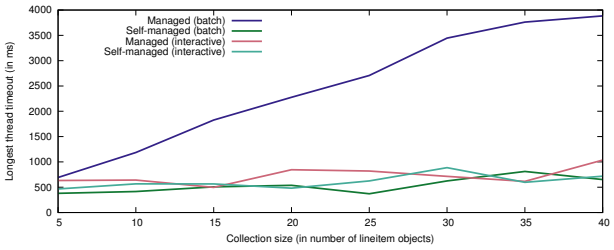

Figure 9: Timeouts caused by garbage collection

.NET[3] and the throughput of allocating managed objects and adding them to a concurrent collection. For managed allocations we report the throughput for interactive and batch garbage collection; the latter consistently provides better performance. SMCs significantly outperform both managed collections and the pure allocation cost of managed objects. All objects remain reachable so the runtime performs numerous garbage collections, with many of them stopping all application threads to copy objects from younger to older generations. SMCs allocate from (previously unused) thread-local blocks, which reduces the synchronization overhead of multiple allocation threads to about one atomic operation per 10k `lineitem` allocations.

Refresh streams To measure the throughput of memory operations we introduce the TPC-H refresh streams. Each thread continuously runs one of two kinds of streams with the same frequency. The first stream type creates and adds `lineitem` objects (0.1% of the initial population) to the `lineitem` collection. The second stream type enumerates all elements in the `lineitem` collection and removes 0.1% of the initial population based on a predicate on the object's `orderkey` value. All 0.1% objects to delete are provided in a hash map and removed in a single enumeration over the collection. This benchmark represents the common use case of refreshing the data stored in SMCs. In Figure 8 we report the stream throughput for SMCs against `ConcurrentDictionary<TKey, TValue>`; `ConcurrentBag<T>` is not included because it does not support the removal of specific elements. SMCs perform better than both types of managed collections in all cases.

Impact of garbage collection Out of the two garbage collection settings reported in Figure 7, the (non-concurrent) batch mode provides the higher throughput. In other garbage collection intensive benchmarks, we found the batch mode to enable a several times higher throughput. However, the

[3]Pre-allocated, thread-local arrays prevent objects from being garbage collected.

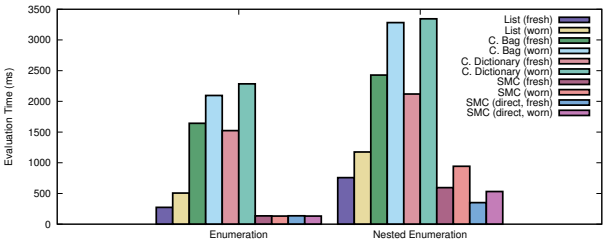

Figure 10: Enumeration performance

higher throughput comes at a price: response time. Where concurrent collectors (interactive) can perform big parts of garbage collection on a background thread without pausing all application threads, non-concurrent collectors have to pause all threads for the duration of the collection. As the size of the managed heap grows, so does the duration of full garbage collections and, hence, the application's maximum response time. To illustrate this, we insert a number of objects into a collection, either managed or self-managed, and then start two threads in parallel. The first thread continuously allocates managed objects with varying lifetimes and the second continuously sleeps for one millisecond and measures the time that passed in the meantime. If it observes that significantly more time has passed than expected, it records the value as it most likely was caused by garbage collection triggered by the other thread. Figure 9 shows the maximum timeout measured for a varying number of objects stored in the collection. For non-concurrent garbage collection, the maximum timeout increases with a growing number of objects stored in a managed collection, but remains fairly stable when these objects are stored in an SMC. Thus, the duration of garbage collections increases with growing data volumes stored in the managed heap. In the batch mode this negatively impacts the responsiveness of the application; in the interactive mode, it negatively impacts the overall application performance as the background collection thread steals processing resources from the application. In both cases, SMCs scale better with increasing data volumes.

Enumeration performance We first report on the pure enumeration performance of SMCs before considering more complex queries. Our queries either: (*a*) enumerate the `lineitem` collection and perform a simple function on each object to ensure that all `lineitem` objects are accessed; or (*b*) enumerate the `lineitem` collection, and for each object follow the `order` reference to a `customer` object and perform a simple function on the latter to ensure that `customer` objects are also accessed. Query performance deteriorates over time as objects are added and removed from the collection. In managed collections, objects may end up scattered all over the managed heap, whereas in SMCs the blocks containing objects may have holes due to limbo slots. In Figure 10 we show the performance of both query types after the collections are freshly loaded (fresh) and after the collections have undergone numerous object removals and insertions (worn). SMCs (indirect) outperform all automatically managed collections. However, when performing nested object accesses, the difference with `List<T>` diminishes because of the additional memory access required by indirection when following self-managed references. By utilizing the direct pointers of §6, we can bypass this look-up and

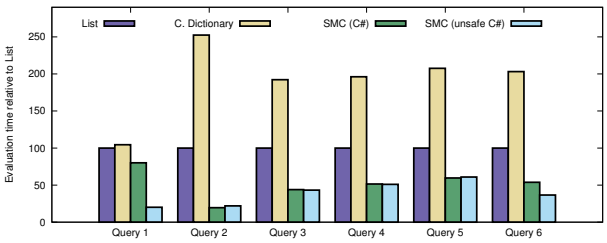

Figure 11: TPC-H Queries 1 to 6

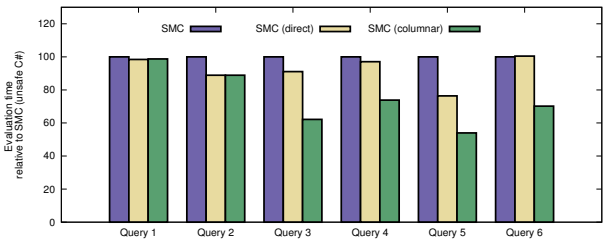

Figure 12: Direct pointer and columnar storage

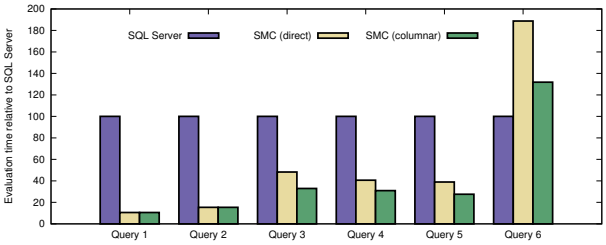

Figure 13: Comparison to SQL Server on a TPC-H-like workload

improve performance. When comparing the fresh and worn states, SMCs only lose performance under nested accesses, whereas managed collections exhibit degraded performance in both cases. As `ConcurrentDictionary<TKey, TValue>` is the best performing thread-safe managed collection, we exclude `ConcurrentBag<T>` in what follows.

Query processing In Figure 11 we show the performance of the object-oriented adaptation of the first six TPC-H queries. For managed collections, we report the query performance of compiled C♯ code (as in [13] but with reference-based joins). Using LINQ to evaluate the queries instead of compiling them to C♯ code results in a 40% to 400% higher evaluation time, but as this was not the focus of the paper, we do not report it in Figure 11. We report on two versions of compiled code for SMCs: (*a*) Compiled C♯ code that, other than the enumeration code, is equivalent to the code used for managed collections. This illustrates the fraction of the overall improvement contributed by the better enumeration performance of SMCs. (*b*) Compiled `unsafe` C♯ code that contains optimizations only possible on SMCs. One such optimization is to use direct pointers to primitive types in an object (*e.g.,* `decimal` values) as arguments to functions that operate on them (*e.g.,* addition). For managed objects, these functions have to be called by value as the garbage collector may move the object inside the managed heap at any time without notice and, hence, the pointer would become invalid. Another optimization is to use memory regions [16] for all intermediate data during query processing, which improves performance by excluding those intermediates from garbage collection. Figure 11 reports the query processing performance relative to the performance of `List<T>`. SMCs perform significantly better than `ConcurrentDictionary<TKey, TValue>`, the fastest competing thread-safe collection in .NET; and even between 47% and 80% better than `List<T>`. Query 1 is a great example of what can be achieved with direct pointer access to self-managed objects. The query is `decimal` computation heavy and as C♯'s `decimal` type is 16-bytes wide, calling the functions that perform decimal math using pointers and allowing for in-place modifications results in a huge performance gain. The other queries are less `decimal` computation intensive and, hence, show very little improvement from using unsafe code. Generating native C code leads to another 10% to 20% improvement over compiled `unsafe` C♯ code. But as the compiled C code is (mostly) equivalent to the compiled `unsafe` C♯ code, any performance differences can be attributed to more aggressive code-level optimizations by the C compiler.

Direct pointers and columnar storage In Figure 12 we show the impact of the direct pointer optimization introduced in §6 and columnar storage as discussed in §4.1. Direct pointer moderately improve query performance for queries that contain joins, in particular for Query 5. Columnar storage shows further improvements that are enabled by the SMCs decoupling the memory layout of their elements from their definition through managing their own memory.

Comparison to RDBMS To put the SMC results into perspective, we compare the query performance over objects in SMCs to that of a modern commercial database system. We use SQL Server 2014 for this purpose as it is well integrated into .NET and incorporates a compressed in-memory columnar store. We store all tables in the database's column store and, in addition, use clustered indexes on `shipdate` and `orderdate`. We use the `read uncommitted` isolation level and disable parallelized query execution to level the playing field. The results are shown in Figure 13. For most of the queries, SMCs exhibit better query performance. For join-heavy queries, they benefit from using references to perform joins instead of explicit value-based join operations. In other queries the database benefits from the indexes on `shipdate` and `orderdate`.

8. RELATED WORK

Type-safe manual memory management is at the core of SMCs. Region-based memory management [16] groups objects in regions and deallocates entire regions. Deallocating objects at region granularity is too high a storage overhead as objects in the applications we target are long-lived with only incremental insertions and deletions. Memory safety at object granularity is enforced by introducing specialized pointer types, *e.g., smart pointers* in C++11, which use reference counting to ensure that memory is only freed once it is no longer referenced. Reference counting comes at a high cost, especially when objects may be accessed concurrently [11]. *Fat pointers* are frequently used for type and/or memory safety at run-time [1]. Tracking object incarnations [6] is an application of this approach.

We use a variant of epoch-based memory reclamation used in lock-free data structures [5, 7], to ensure thread-safety. *Hazard pointers* and their variants [8, 11] ensure that threads only reclaim memory that is not referenced by other such pointers. This is similar to our epoch-based approach, but it would reduce performance: each query would iterate over objects through a hazard pointer, requiring a memory barrier whenever it is assigned to the next object. Epochs amortize the cost of memory barriers by using the entire query as the granularity of the critical section. Braginsky and Petrank [3] propose a lock-free sorted linked list optimized for spatial locality. Each list element is a sub-list of several data elements stored as a chunk of memory. Hazard pointers ensure safe memory reclamation, while a freeze bit in the elements' next pointer ensures lock-free splitting and merging of chunks. The implementation is limited to a specific format for each list element (`integer` key and value).

To improve query performance, SMCs rely on query compilation [9, 10, 14, 15]. We use popular techniques, *e.g.,* maximizing the processing performed in each loop and merging query operations inside a loop to maximize data reuse [14]. Klonatos et al. [9] propose the use of a high-level programming language for implementation and use query compilation for query processing. In contrast to our approach, the data store and query processor are not integrated with the application and, hence, the database functionality is treated as a black box (*e.g.,* there are no references to data objects). Murray et al. [12] first proposed query compilation for LINQ queries on in-memory objects. Their code generation approach did not go beyond querying C♯ objects in managed collections using compiled C♯ code. Nagel et al. [13] extended that idea by experimenting with different data layouts and identified managed collections as a performance bottleneck; generating native C code that operates on arrays of in-place structs provided the best performance. Our work builds on these findings.

DryadLINQ [17] and Trill [4] both build on LINQ to ease programming and to provide better application integration. DryadLINQ transforms LINQ programs into distributed computations running on a cluster whereas Trill operates on data batches pushed from external sources.

9. CONCLUSION

In this paper we introduced self-managed collections, a new type of collection for managed applications that manage and process large volumes of in-memory data. SMCs have specialized semantics that allow the collection to manually manage the memory space of its contained objects; and the objects of the collection to be referenced from the application and other SMCs. SMCs are optimized for query processing using language-integrated queries compiled to imperative code. We introduced the type-safe manual memory management system of SMCs and then the collection type itself. Our evaluation shows that SMCs outperform managed collections on query performance, batch allocations, and online modifications using predicate-based removal. At the same time, SMCs can improve the response time of the application overall by reducing the stress on the garbage collector and allow it to better scale with growing data volumes. Such scalability is transparent to the developer and eliminates the current required practise of resorting to low-level programming techniques.

10. REFERENCES

[1] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[2] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *VLDB*, 2005.

[3] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *ICDCN*. 2011.

[4] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. In *VLDB*, 2014.

[5] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-level implementations of read-copy update. *IEEE TOPADS*, 23(2):375–382, 2012.

[6] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *NSDI*, 2014.

[7] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, 2004.

[8] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM TOCS*, 23:146–196, 2005.

[9] I. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. In *VLDB*, 2014.

[10] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, 2010.

[11] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *TPDS*, 2004.

[12] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *PLDI*, 2011.

[13] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. In *VLDB*, 2014.

[14] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[15] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. Compiled query execution engine using JVM. In *ICDE*, 2006.

[16] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and computation*, 1997.

[17] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.