Unified Memory Management in Spark 1.6

Andrew Or and Josh Rosen

This document proposes a new memory management model in Spark. In the new design, the existing boundary separating the execution and storage memory regions is made crossable. Execution may borrow unused storage memory and vice versa. Borrowed storage memory may be evicted when memory pressure arises. Borrowed execution memory, however, will not be evicted in the first design due to complexities in implementation.

For a detailed description of the full design, see *Proposal*.

Overview

Memory usage in Spark largely falls into one of two categories: execution and storage. Execution memory refers to that used for computation in shuffles, joins, sorts and aggregations, while storage memory refers to that used for caching and propagating internal data across the cluster.

In Spark 1.5 and before, there is a static, configurable boundary between the two such that neither usage can borrow from the space reserved for the other. There are several limitations with this rigid separation:

- There are no sensible defaults that apply to all workloads
- Tuning memory fractions requires user expertise of internals
- Applications that do not cache use only a small fraction of available memory

The goal is to overcome these limitations by unifying the existing memory regions.

The end result should provide better out-of-the-box performance and eliminate the need for user tuning to achieve optimum memory usage. Further, because memory allocation is no longer statically defined through per-application configurations, a single application should now be able to support multiple types of workload without inducing excessive spilling.

Existing memory management

The existing memory management in Spark is structured through static memory fractions. Conceptually, the memory space is partitioned into three disjoint regions. The size of each region is specified as a fixed fraction of the total JVM heap size through Spark configurations:

Execution. This region is used for buffering intermediate data when performing shuffles, joins, sorts and aggregations. The size of this region is configured through `spark.shuffle.memoryFraction (default 0.2)`.

Storage. This region is mainly for caching data blocks to optimize for future accesses, but is also used for torrent broadcasts and sending large task results. The size of this region is configured through `spark.storage.memoryFraction (default 0.6)`.

Other. The rest of the memory consists mainly of data structures allocated by user code and internal Spark metadata. This category will not be discussed further because it is largely unmanaged `(default 0.2)`.

Within each managed region, in-memory data is spilled to disk when memory usage approaches the limit for the region. In the storage case, cached data may even be dropped altogether. In both cases, dropping data from memory worsens performance either through increased I/O or forced recomputation.

Memory fraction breakdown

To avoid potential OOM's, Spark memory management errs on being overly cautious in case there is, for instance, an unexpectedly large item in the data, which lowers the accuracy of Spark's size estimation. For this reason, safety fractions are used within each region to provide additional buffer for data skew. The breakdown of all memory and related fractions are as follows:

```
spark.shuffle.memoryFraction (default 0.2)
   spark.shuffle.safetyFraction (default 0.8)
spark.storage.memoryFraction (default 0.6)
   spark.storage.safetyFraction (default 0.9)
   spark.storage.unrollFraction (default 0.2)
```

This means the default spill threshold for execution is actually only `0.2 * 0.8 = 16%` of the total heap size! For applications that do not cache any data, the majority of the heap space is

wasted while tasks unnecessarily spill intermediate data to disk just to read them back into memory immediately afterwards. The unroll fraction will be discussed in future sections.

Execution memory management

Execution memory is further divided among the set of active tasks running in the JVM. Unlike allocation across regions, allocation across tasks within the execution region is dynamic. Instead of assigning a fixed chunk to each slot, Spark allows one task to use all available execution memory if there are no other active tasks running in the same JVM.

There are several memory managers responsible for enforcing this behavior:

ShuffleMemoryManager is responsible for global accounting and policy enforcement. It is the central arbiter that decides how much memory to give to which task given the task's existing allocation. There is one of these per JVM.

TaskMemoryManager is responsible for per-task allocation and bookkeeping. It implements a page table to track on-heap blocks and detects memory leaks by throwing an exception if not all pages are freed by the time the task exits. Internally it uses an `ExecutorMemoryManager` to perform the actual allocation and freeing. There is one of these per task.

ExecutorMemoryManager is responsible for the mechanism. It handles both on-heap and off-heap allocations, and implements a weak reference pool to allow freed pages to be re-used across tasks. There is one of these per JVM.

These memory managers interact with each other as follows. Whenever a task wants to allocate some large in-memory data structure, it first requests X bytes from the `ShuffleMemoryManager`. If the request is fully granted, then the task will ask the `TaskMemoryManager` to allocate the X bytes. After the `TaskMemoryManager` updates its internal page table, it will ask the `ExecutorMemoryManager` to actually allocate the X bytes in the system on the task's behalf.

Memory allocation policy

Each task can acquire up to 1/N of the execution memory from `ShuffleMemoryManager`, where N is the number of active tasks. If a request is not fully granted, however, the task spills its in-memory data and releases its share. Depending on the specific operation, the task may then try again or attempt to acquire a smaller chunk.

To avoid excessive spilling, a task does not spill unless it has acquired up to 1/(2N) of the total memory. If there is not enough free memory to acquire even up to 1/(2N), the request will block until other tasks spill and free their shares. Otherwise, new incoming tasks may spill constantly while existing jumbo tasks continue to occupy much of the memory without spilling.

Example: An executor starts off with a single task A, which quickly acquires all the available execution memory before a second task B comes along. The new N will be 2, and task B will block until it can acquire 1/(2N) = 1/4 of the total memory, when task A spills. Once task B has acquired up to 1/4, both tasks are subject to spilling.

Note: Also task A doesn't actually spill until it fails to acquire memory from the manager. In the meantime, all new tasks may be starved because the existing tasks have consumed more than their fair shares of memory. This can be fixed by some kind of force spilling mechanism, but this is outside the scope of this document.

Storage memory management

The storage region is managed by the more general `BlockManager`. Although the primary use case is caching RDD partitions, `BlockManager` is also used for torrent broadcasts and sending large task results to the driver.

Storage level

Each block is associated with a storage level, which specifies whether a block should be stored in memory, on disk, or off-heap. A block may also be stored both in memory *and* on disk, in which case it will be dropped to disk when it is evicted from memory.

The storage level also specifies whether a block should be stored in serialized form. One particular storage level to highlight is `MEMORY_AND_DISK_SER`, where the block is already stored in a byte array in memory and so dropping this block to disk does not require serializing the data again. Blocks with this storage level are the cheapest to evict.

Eviction policy

The existing eviction policy is largely LRU and applies only to in-memory blocks. There are two notable exceptions to this. The first is that we *never* evict existing RDD blocks in order to cache additional blocks of that RDD. The second is that if unrolling fails, the new block that we attempted and failed to unroll is evicted directly.

Unrolling

When the `BlockManager` receives data in the form of an iterator, it unrolls the iterator into an array if the block should ultimately be stored in memory. However, because the entire iterator may not actually fit in memory, `BlockManager` must unroll the iterator gradually to avoid OOM's, periodically checking whether there is still enough unroll space before expanding the array.

The memory used for unrolling is borrowed from the storage space. If there are no existing blocks, unrolling can use all of the storage space. Otherwise, unrolling can drop up to M bytes worth of blocks from memory, where M is a fraction of the storage space configurable through `spark.storage.unrollFraction (default 0.2)`. Note that this sub-region is not statically reserved, but dynamically allocated by dropping existing blocks.

If unrolling a block fails even after dropping existing blocks, the `BlockManager` treats the new block as if it were evicted from memory, dropping it to disk if applicable.

Design

This section proposes a design, discusses the considerations behind this proposal and compares it with several alternative schemes. This section is primarily concerned only with high level memory management policies.

Proposal

The boundary between execution and storage is now crossable. When execution memory exceeds its own region, it can borrow as much of the storage space as is free and vice versa. Borrowed storage memory can be evicted at any given time. Borrowed execution memory, however, will *not* be evicted in the first design due to complexities in implementation. The following new configurations will be introduced:

- **`spark.memory.fraction (default 0.75):`** fraction of the heap space used for execution and storage. The lower this is, the more frequently spills and cached data eviction occur. The purpose of this config is to set aside memory for internal metadata, user data structures, and imprecise size estimation in the case of sparse, unusually large records.

- **`spark.memory.storageFraction (default 0.5):`** The size of the storage region within the space set aside by `spark.memory.fraction`. Cached data may only be evicted if total storage exceeds this region.

- **`spark.memory.useLegacyMode (default false):`** Whether to enable the legacy memory management mode used in Spark 1.5 and before. The legacy mode rigidly partitions the heap space into fixed-size regions, potentially leading to excessive spilling if the application was not tuned. All deprecated memory fraction configurations take effect only if this is enabled:
  - `spark.storage.memoryFraction`
  - `spark.storage.safetyFraction`
  - `spark.storage.unrollFraction`
  - `spark.shuffle.memoryFraction`
  - `spark.shuffle.safetyFraction`

The rest of this section will discuss the tradeoffs involved in this and alternative designs.

Response to memory pressure

Because the same unified space is now used for both execution and storage, we must decide what to evict when memory pressure arises. The following alternatives will be compared on several dimensions:

- Prefer to evict execution memory
- Prefer to evict storage memory
- No eviction preference

Cost of eviction

The cost of evicting storage memory depends significantly on the storage level. Blocks stored with `MEMORY_ONLY` are potentially the most expensive to evict because future accesses will require recomputation. On the other hand, blocks stored with `MEMORY_AND_DISK_SER` are the cheapest to evict because doing so does not involve serializing the content again, so the only cost in this case comes from I/O.

The cost of evicting execution memory is less variable. There are no risks of recomputation, as all evicted execution data will be spilled to disk. With recent unsafe additions, execution data is primarily stored in compact format already and thus inexpensive to serialize.

One thing to note is that spilled execution data will always be read back, while evicted storage data may never be referenced again, so the expected cost of evicting execution is presumably higher when no recomputation is involved.

Implementation complexity

Storage eviction is simple; we simply drop blocks using the existing eviction mechanism. Execution eviction, however, is more complex. There are several alternatives:

- Register a spill callback with all execution memory requests
- Cooperatively poll and spill

In both cases, we need to worry about whether a particular page holds the data that we're about to cache. Further, many operators ensure they have at least a page of execution memory to work with. If we force execution memory to spill, then we need to be careful to not evict the reserved pages and thus starve the operators.

Additionally, we need to worry about what to do about the blocks waiting to be cached while execution memory is evicted. The simplest solution here is to simply wait until sufficient execution memory is freed, but this may be subject to deadlocks especially if the contents of the new blocks rely on outstanding in-memory execution data. An alternative is to drop the new blocks to disk in the meantime and load them back into memory as soon as space frees up. To avoid always dropping all waiting blocks, we can set aside a small in-memory buffer (e.g. 5% of heap space) to keep at least a fraction of the blocks in memory.

Given that both of these approaches bring additional complexity to the design, they will be left out in the first implementation. If it turns out that the case where storage should evict execution is common and important, then we should consider adding them.

Preference

This document favors the preference to evict cached data. One of the main motivations is to provide execution with more space to work with, and forcing execution memory to spill under pressure does not guarantee this. This preference is also significantly simpler to implement.

However, if we simply allow all cached data to be evicted we may observe noticeable regressions in applications that actually rely on caching. To remedy this we may need some notion of minimum reserved memory for cached blocks.

Minimum reservations

Whether we choose to evict cached blocks or spill execution memory, we may want to allow the user to provision a minimum reserved region so neither side is starved. There are two ways of provisioning this: statically and dynamically. The existing memory management in Spark 1.5 provide a static reservation for both execution and storage. The downside to such minimum reservations is that they essentially impose a cap on the other side.

The new design provides a *dynamic minimum reservations for both storage and execution*. This means storage can borrow from execution but may be evicted as soon as execution attempts to claim the space back. The same is largely true for execution with one notable exception. If an application attempts to cache a block when execution memory already uses up all the storage space, we simply evict the new block directly instead of attempting to evict execution memory, which is complicated to implement as discussed above.

Additionally, we need to set aside some space for non-execution and non-storage purposes, e.g. internal metadata and user data structures. In Spark 1.5 and before, the default size for this space is 20% of the heap and fixed for the duration of the application. This requirement does

not change in the new design, so the existing *static minimum system reservation* stays. However, it will now be explicitly configurable through `spark.memory.fraction (default 0.75)`, which represents the inverse of this minimum reservation.

Alternative designs

Having outlined the relevant design considerations, we will now compare a few proposals. Note that only designs that prefer to evict cached blocks under memory pressure are suggested due to aforementioned reasons.

In all of the new designs (A - C), the space shared between execution and storage will be limited to a fraction of the heap space, configurable through `spark.memory.fraction (default 0.75)`. Additionally, to provide backward compatibility, the user may enable `spark.memory.useLegacyMode (default false)` to use the old memory management model, i.e. design (X).

> **(X) Existing behavior.** Restrict execution and storage to their own respective memory fractions. No cross-region eviction or spilling is possible. Sizes of memory regions are statically configured and fixed for the duration of the application. *This is the approach taken in Spark 1.5 and before.*
>
> **(A) Evict cached blocks, full fluidity.** Execution and storage share one unified region. When memory pressure arises, cached blocks are evicted. Execution memory spills only if there is still not enough space after evicting storage memory.
>
> **(B) Evict cached blocks, static storage reservation.** This is like design (A) but with a reserved storage region that execution memory cannot borrow from. Cached blocks are evicted only if actual storage exceeds this region. The size of the reserved region is configured through `spark.memory.storageFraction (default 0.0)` and fixed for the duration of the application.
>
> **(C) Evict cached blocks, dynamic storage reservation.** This is like design (B), except the storage space is not statically reserved, but dynamically allocated. This difference is that execution can borrow as much of the storage space as is available. *This is the chosen design.*

Design (A) is rejected because it does not provide a solution for multi-tenancy and applications that rely heavily on caching. Thus, we propose design (B), which adds a minimum reserved region for storage.

One issue with design (B), however, is that the minimum storage fraction still requires user configuration in many cases. The default value of `0` does not work in a shared environment; one user can inadvertently evict the cached blocks of another through a large shuffle.

Additionally, no matter what non-zero value we set the minimum storage fraction to, we will end up statically capping the execution memory. For instance, to avoid potential performance regressions in shared environments, it is natural to set the minimum storage fraction to `0.6` (the default for the old storage fraction). This effectively caps execution memory at `0.4 * 0.75 = 0.3` of the heap space, which is not much better than before especially if the user is not actually caching anything.

Design (C) is an alternative that does not enforce a cap on the execution memory. When the storage space is not used, execution can borrow from it. One issue worth addressing is how to handle the case where the application attempts to cache a block when execution has already used up all the available memory. The initial version of this design simply evicts the new block directly instead of introducing complex execution eviction logic, which can always be added later if necessary.

Design (C) is the only one that satisfies all of the following properties:

- There is no cap on storage memory (not satisfied in design X)
- There is no cap on execution memory (not satisfied in designs X and B)
- A minimum storage space is provisioned (not satisfied in design A)

Therefore, design (C) is the chosen one.

Implementation

A new entity, the `MemoryManager`, will be introduced to manage the global accounting of memory for both execution and storage. This entity will implement the policy of favoring execution data over cached data, as described above. This manager exposes the following methods:

```
/**
 * Acquire N bytes of memory for execution.
 * @return number of bytes successfully granted (<= N).
 */
def acquireExecutionMemory(numBytes: Long): Long

/**
 * Acquire N bytes of memory to cache the given block,
 * evicting existing ones if necessary.
 * @return number of bytes successfully granted (0 or N).
 */
def acquireStorageMemory(blockId: BlockId, numBytes: Long): Long

/**
 * Release N bytes of execution memory.
 */
def releaseExecutionMemory(numBytes: Long): Unit

/**
 * Release N bytes of storage memory.
 */
def releaseStorageMemory(numBytes: Long): Unit
```

All tasks will now go through `acquireExecutionMemory` before allocating large data structures for execution. Internally, this may trigger `BlockManager` to free up to the request number of bytes by evicting blocks from memory. This requires exposing the `ensureFreeSpace` method in `MemoryStore` (the block store that lives in `BlockManager`). A task calls `releaseExecutionMemory` whenever it spills or exits.

All put methods in `MemoryStore` will first attempt to `acquireStorageMemory` before caching the block in memory. If this acquire attempt fails, the block will be dropped to disk directly or evicted altogether. When an existing block is evicted from memory, the `MemoryStore` will call `releaseStorageMemory` with the size of the evicted block.

Allocating execution memory across tasks

The existing logic in `ShuffleMemoryManager` allows a task to consume memory between 1/(2N) and 1/N of the total execution memory, where N is the number of active running tasks. In the new model, however, the total execution memory changes over time. In particular, the total execution memory is the size of the execution region *plus* the amount of free space in the storage region at any given time.

This execution memory policy is complicated, however, because there are now two moving pieces, the number of active tasks and the total execution memory. To simplify this, we will simply assign each task at most 1/S of the total execution memory, where S is fixed at the number of CPU slots available to the application.

Further, to avoid having two separate classes managing execution memory policy, all existing logic in `ShuffleMemoryManager` will be moved to the new `MemoryManager` so more methods can be hidden behind the new interface.

Legacy Mode

The design put forth in this document has far-reaching implications to the performance of most existing applications. In case there are corner cases where performance regresses so significantly that no workarounds are effective, Spark must provide a way for the user to continue using the existing static memory management model.

The user may enable legacy mode through `spark.memory.useLegacyMode`. All old memory fractions are deprecated and read only if legacy mode is enabled, and all new memory configurations are read only if legacy mode is not enabled.

To avoid spawning `if-else`'s everywhere, legacy mode will be implemented through a new `LegacyMemoryManager` that implements the interface defined by the new `MemoryManager`.

Simplifying memory manager structure

This design introduces YAMM (yet another memory manager). This calls for simplifying the existing memory manager structure, which is already densely populated.

Currently, a task must first acquire memory from the `ShuffleMemoryManager`, and then allocate the memory through the `TaskMemoryManager`. This two-step process is duplicated in

many places and requires passing around both memory managers in the constructors of many unsafe data structures.

In the new design, there will be a single `MemoryManager`. Tasks will directly call `acquireExecutionMemory` to acquire *and* allocate execution memory. All existing logic in `ShuffleMemoryManager` and `ExecutorMemoryManager` will be moved to the new `MemoryManager`. The existing `TaskMemoryManager` will be renamed (new name subject to discussion) and used within `MemoryManager` when `acquireExecutionMemory` is called.