FLIP-50: Spill-able Heap Keyed State Backend

Authors: Yu Li, Pengfei Li

Status

Current state: *Accepted*

Discussion thread: https://lists.apache.org/thread.html/a10cae910f92936783e59505d7b9fe71c5f66ceea4c1c287c87164ae@%3Cdev.flink.apache.org%3E

JIRA: FLINK-12692 - Support disk spilling in HeapKeyedStateBackend OPEN

Released: <Flink Version>

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

HeapKeyedStateBackend is one of the two KeyedStateBackends in Flink, since state lives as Java objects on the heap in HeapKeyedStateBackend and the de/serialization only happens during state snapshot and restore, it outperforms RocksDBKeyeStateBackend when all data could reside in memory.

However, along with the advantage, HeapKeyedStateBackend also has its shortcomings, and the most painful one is the difficulty to estimate the maximum heap size (Xmx) to set, and we will suffer from GC impact once the heap memory is not enough to hold all state data. There're several (inevitable) causes for such scenario, including (but not limited to):

- Memory overhead of Java object representation (tens of times of the serialized data size).
- Data flood caused by burst traffic.
- Data accumulation caused by source malfunction.

To resolve this problem, we proposed a solution to support spilling state data to disk before heap memory is exhausted. We will monitor the heap usage and choose the coldest data to spill, and reload them when heap memory is regained after data removing or TTL expiration, automatically.

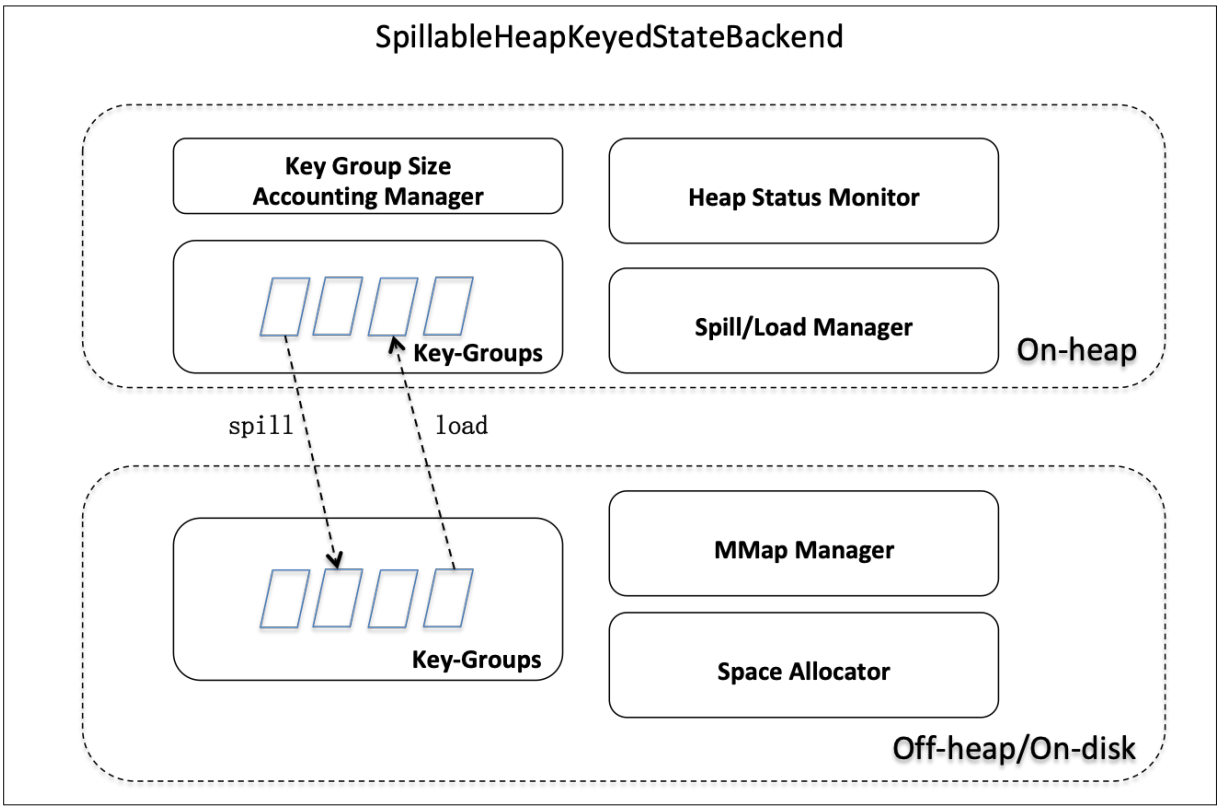
Similar to the idea of Anti-Caching approach [1] proposed for database, the main difference between supporting data spilling in HeapKeyedStateBackend and adding a big cache for RocksDBKeyedStateBackend is now memory is the primary storage device than disk. Data is either in memory or on disk instead of in both places at the same time thus saving the cost to prevent inconsistency, and rather than starting with data on disk and reading hot data into cache, data starts in memory and cold data is evicted to disk.

We will describe the overall design in more details in section 2, including the modules required to implement the solution and key technicals used. Then we will describe more details of the design of each module in section 3.

Performance of the newly proposed solution will be depicted in section 4, and limitations observed from online usage and future work will be discussed in section 5 and 6.

Note: to prevent causing unexpected regression to existing usage of HeapKeyedStateBackend, we plan to introduce a new SpillableHeapKeyedStateBackend and change it to default in future if proven to be stable.

Design Overview



Picture 1. Architecture

As illustrated in picture-1, we will monitor the heap status, once we found the heap usage or gc pause time is above some pre-defined threshold, we will choose the proper KG (key group) to spill. On the contrast, if the heap usage is below threshold, we will choose the proper KG to load. Below modules are introduced to support this in heap keyed backend:

- KeyGroupSizeAccountingManager
  - Recording the size of each on-heap and off-heap key group.
- HeapStatusMonitor
  - Monitor the heap status, and trigger spill/load when necessary.
- Spill/LoadManager
  - Choose KG and do spill/load according to pre-defined policy.
- MMapManager
  - We use mmap to optimize performance reading/writing KG from/to disk, and MMapManager is responsible for all mmap operation and management.
- SpaceAllocator
  - Allocate and manage space on off-heap/disk, to store data for different KG.

Design Details

KeyGroupSizeAccountingManager

There’re two kinds of key group data formats, on-heap and off-heap (on-disk), and we use different accountings for them:

- On-heap key group size accounting
  - We estimate the POJO size referring to Lucene’s RamUsageEstimator and count for each KG. Meanwhile we perform ambiguous instead of accurate estimation for performance consideration, and we only need to compare the size of each KG so the estimation doesn’t need to be accurate.
- Off-heap key group size accounting
  - We know the exact size of the serialized key and value, and do simple computation.

HeapStatusMonitor

HeapStatusMonitor is responsible to watch the heap status and decide whether we need to perform a spill or load operation. There’re two main factors to watch:

- Heap usage

        - Since we use approximate rather than accurate estimation for heap size accounting, we don't use the accounting for heap usage. Instead, we use the MXBean for such monitoring.
        - Note that since MemoryMXBean.getUsed won't be updated even after spilling happened if no CMS GC happened, so we need to check CMS GC count from GC beans and only check against heap used when GC count increases.
        - Currently we check against CMS GC rather than G1GC since commonly the heap size for TM won't be too big and CMS GC should be enough.
        - The threshold on heap size is configurable and by default it's 50% of the Xmx setting.
    - GC pause
        - The heap size based threshold might not be enough during data burst, and we will check against the real GC pause as another barrier.
        - By default we will check GC pause every 1 minute and will do spilling when the average GC pause of one type (such as CMS) exceeds 2 seconds. And all these thresholds are configurable.

SpillLoadManager

SpillLoadManager is responsible to choose the proper KG and do spill/load when asked by HeapStatusMonitor

- Two main factors to decide KG to spill/load: KG size and request rate.
- Compute the weighted average on the normalized KG size and request rate, as the weight.
- Spill policy: choose KG with more heap occupancy and lower request rate.
- Load policy: Load policy: choose KG with higher request rate and less heap occupancy.

MMapManager

MMapManager is responsible for all mmap related operations and management, including:

- Support creating and mapping files in rotation on multiple paths.
- Do necessary clean up during initiation and exit (including process crash through shutdown hook).
- Handle exceptional cases of mmap, refer to log4j2 [2] and mapdb [3] experience.

SpaceAllocator

SpaceAllocator is responsible for allocating space on off-heap/disk to store spilled KG data.

Data Structure for off-heap/on-disk Data

We implemented a compacted SkipList which supports copy-on-write. A delta-chain structure is used for copy-on-write support referring to Nitro [4], and more details please refer to this sub page

Implementation

We will introduce a new *flink-statebackend-heap-spillable* sub-module under the flink-state-backends module and add the new backend there, following the way RocksDB does.

Code wise, we will mainly introduce below classes according to the above design details:

- A new SpillableHeapKeyedStateBackend class extending HeapKeyedStateBackend, the main wrapper of the new backend.
- A new HybridStateTable class extending StateTable, where we will do almost all the magic, including:
    - A HeapAccountingManager (newly introduced class) instance to do on-heap key-group memory consumption estimation.
    - A HeapStatusMonitor (newly introduced class) instance to monitor the JVM status.
    - A SpillLoadManager (newly introduced class) instance to decide and execute spill or load action.
    - A MMapManager (newly introduced class) to handle mmap operations.
- A new CopyOnWriteSkipListStateMap class extending StateMap, the major implementation to support storing key-group data on off-heap.

And we will reuse the existing classes for heap backend whenever the code logics remain the same, such as HeapSnapshotStrategy, HeapRestoreOperation, etc.

Performance Evaluation (Preview)

We performed comparison test between the new heap backend and RocksDB backend with word-count job which has 566MB state size (after serialization) in PCIe-SSD environment, and below are the results:

Backend	**QPS (K/s)**	**Note**
Heap (all in memory)	400	-Xmx=10GB
RocksDB (all in memory)	100	Cache=10GB
Heap (memory + disk)	160	-Xmx=3GB (spill ratio=57%)
RocksDB (all in memory)	100	Cache=3GB

Limitations

The solution is already deployed in production in Alibaba and used on Singles' Day in 2018. From our online observation the solution could work well for most cases but also has several limitations, including:

- Lack of control on mmap may exhaust disk IO if too many data spilling
  - The more KeyGroup is spilled, the more dirty page will be generated by random write, thus the higher IO pressure during pdflush
  - The IO pressure will impact the whole system's performance rather than a single task
- Additional CPU cost during checkpoint
  - Same with the original heap backend, data will be serialized when writing to distributed filesystem like HDFS during checkpointing
  - Additionally, the spilled data will be deserialized (during scan) and then re-serialized (during writing to DFS)

We could see the major issues are related to spilled data, so please keep in mind that although we support data spilling with the solution, the premise is that the major part of your state data should be able to reside in memory, and cold data eviction should be regarded as a protection mechanism instead of a regular operation.

Future Work

If only the major proportion of data resides in memory, the anti-caching [1] theory (performance advantage) stands. The more data could be spilled, the bigger state could be supported in HeapKeyedStateBackend. Relative to the limitations mentioned in section 5, we could improve below parts in future work:

1. Fine-grained IO control on data spilling
   - Consider replacing mmap with self-designed approach.
2. Reduce cpu cost during checkpoint
   - Hybrid data format in checkpoint
     Write the on-heap key-group data out to HDFS directly during checkpoint, meantime copy the spilled key-group data to HDFS directly. During restore, firstly we read back the on-heap part and write into heap, then copy the spilled key-group data back to local, and lastly load the spilled data into heap through the SpillLoadManager if there's still enough memory.
3. Support incremental checkpoint
   - Since commonly we only use heap backend when data is small, full checkpoint could also complete quickly, so the priority of supporting incremental checkpoint is low.

References

[1] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-caching: A new approach to database management system architecture," Proc. VLDB Endowment, vol. 6, pp. 1942–1953, 2013.

[2] https://github.com/apache/logging-log4j2, MemoryMappedFileManager

[3] http://www.mapdb.org/blog/mmap_files_alloc_and_jvm_crash

[4] Lakshman S, Melkote S, Liang J, Mayuram R. Nitro: a fast, scalable in-memory storage engine for NoSQL global secondary index. Proceedings of the VLDB Endowment, 2016, 9(13): 1413–1424