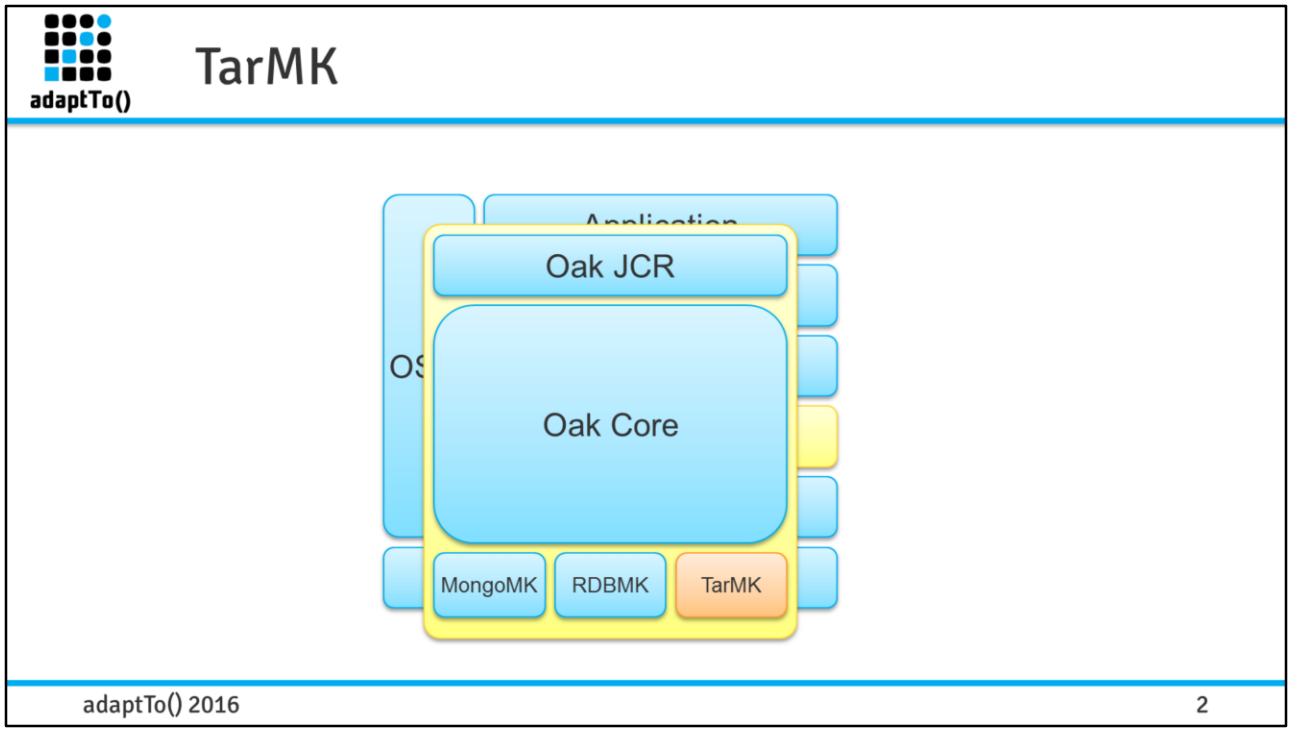
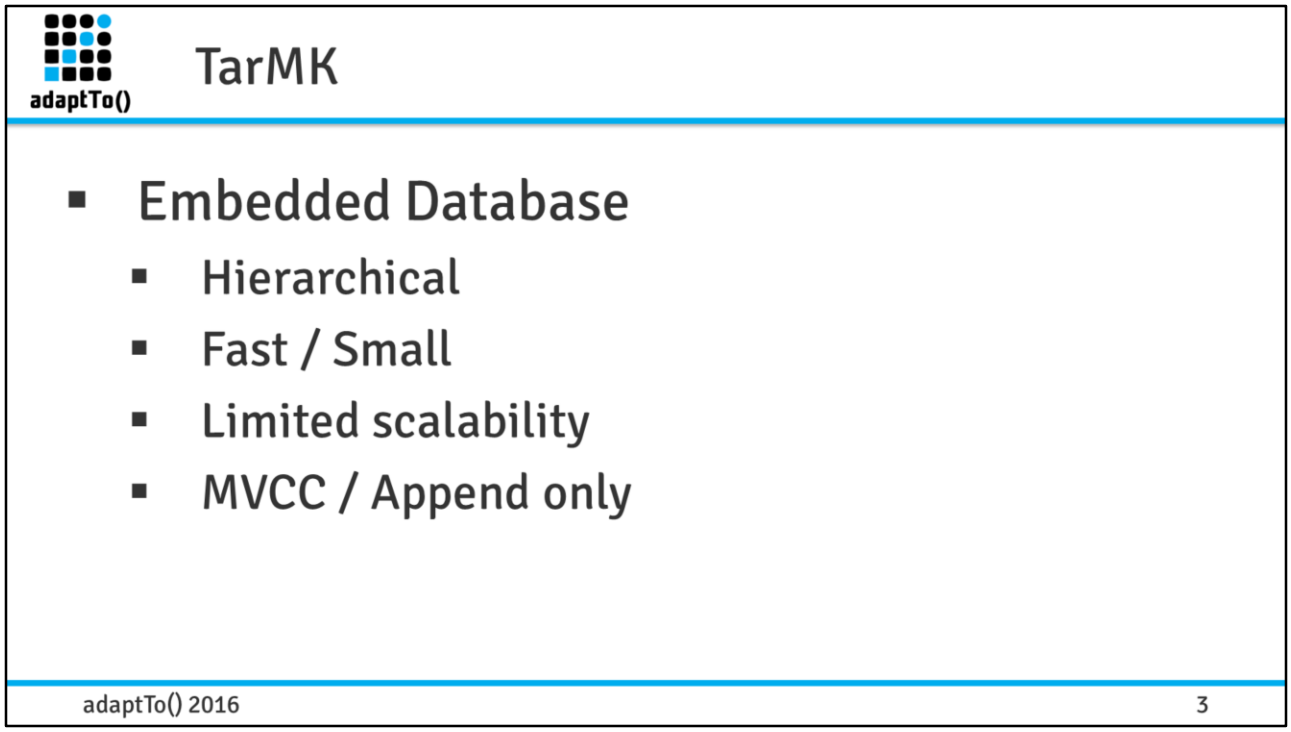
Ever wondered what is inside the TarMK's tar files? What is a segment and what is a record? How garbage collection works and why (or why not)?
This session will answer these questions and many more. It will shed light on the inner working of the TarMK, its system requirements and performance characteristics. It will help participants to better understand and diagnose the cause of common problems and present tools and techniques for diagnosing and debugging.
Finally there will be a preview of what new

features and enhancements we are currently working on.



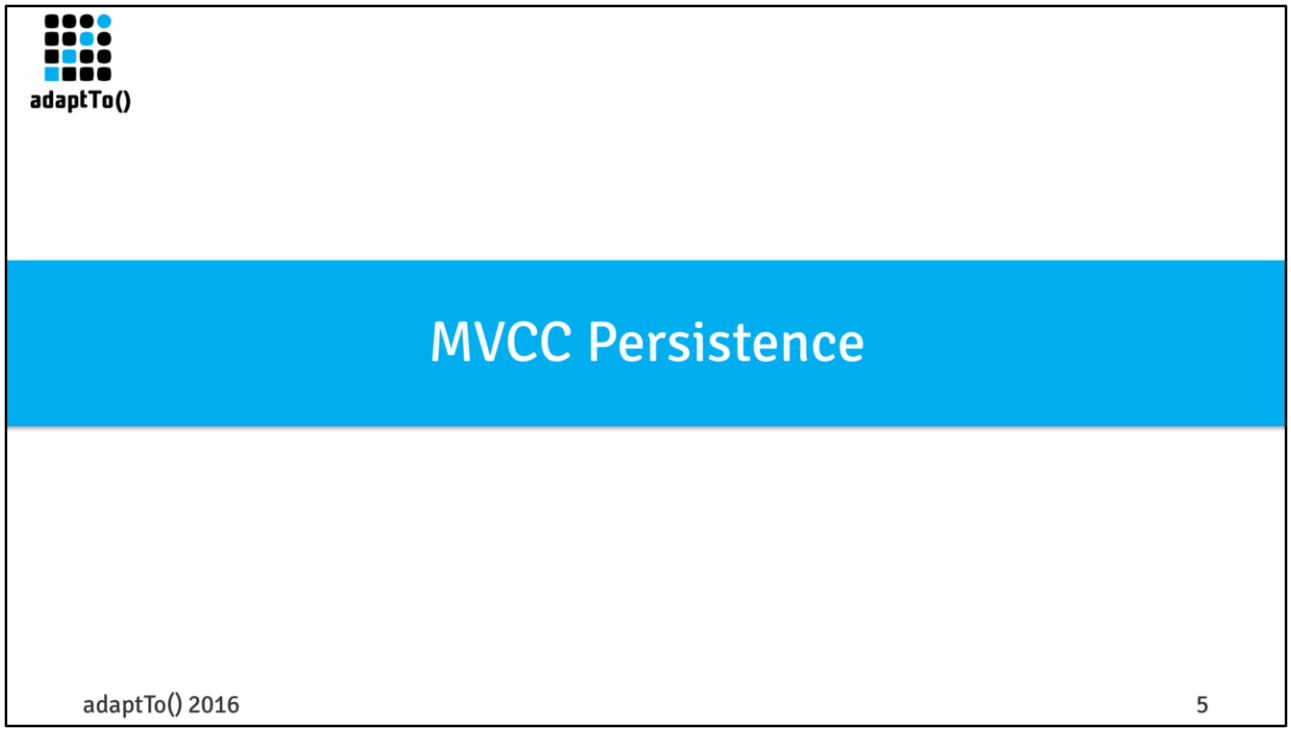
The TarMK is a tiny part of the whole AEM stack. It is one of multiple persistence options of the Java Content Repository implementation Jackrabbit Oak.



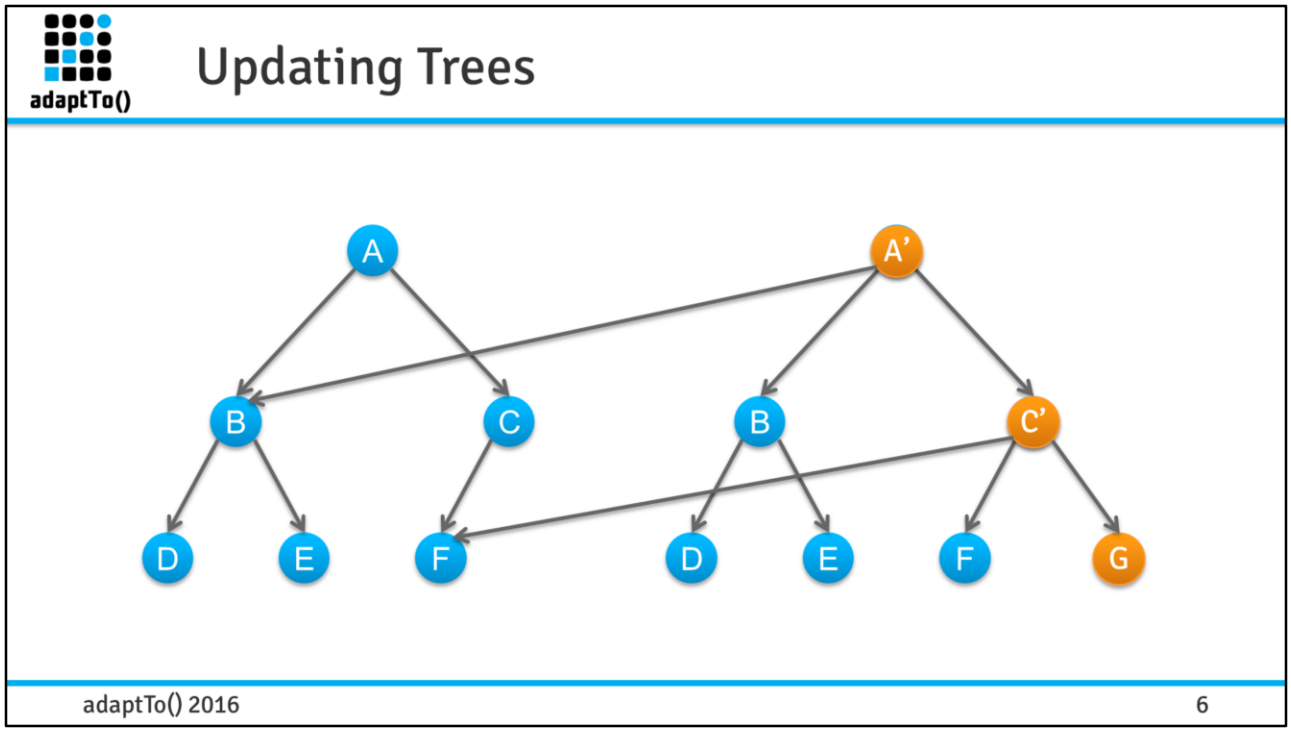
The TarMK is a fast, small and simple embedded hierarchical database engine serving as a persistence backend for the Jackrabbit Oak Java Content Repository. It implements multi-version concurrency control and stores all data in tar files in an append only way.

Agenda

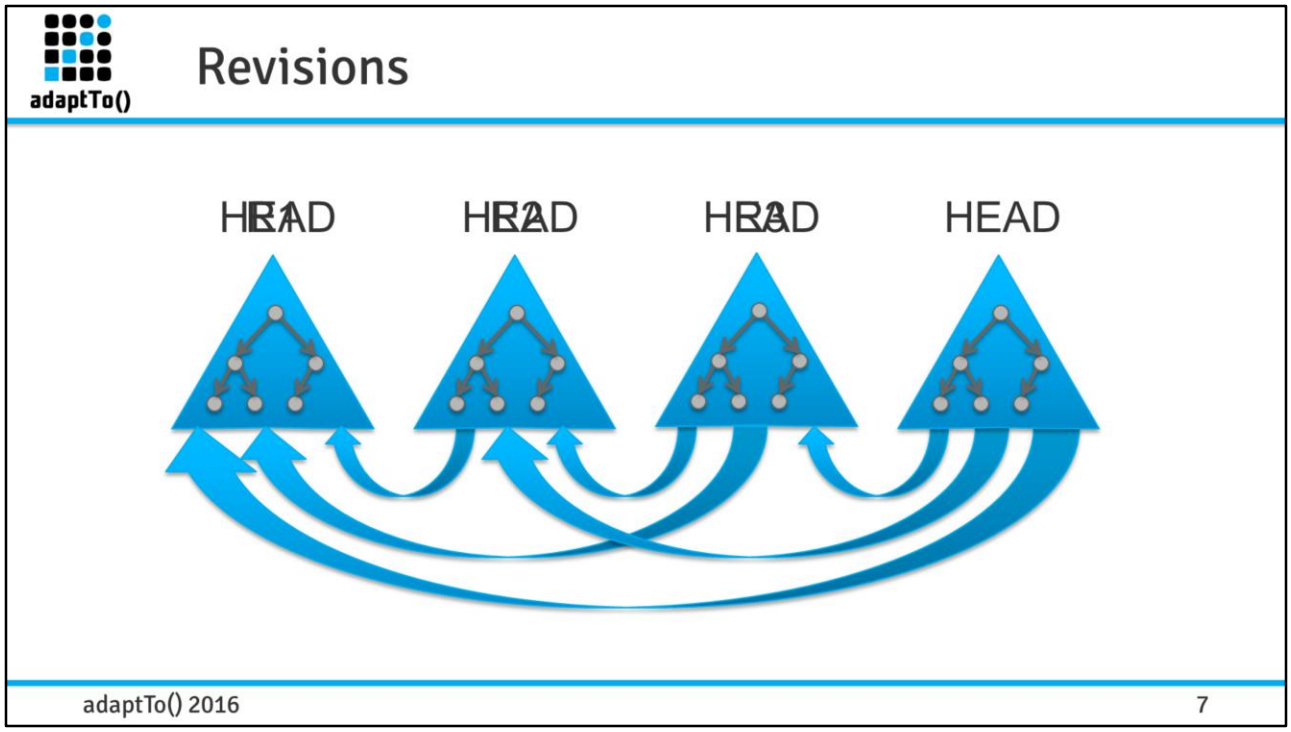
- MVCC Persistence
- Revisions, Recovery, Rollback
- Garbage Collection
- Upcoming Improvements



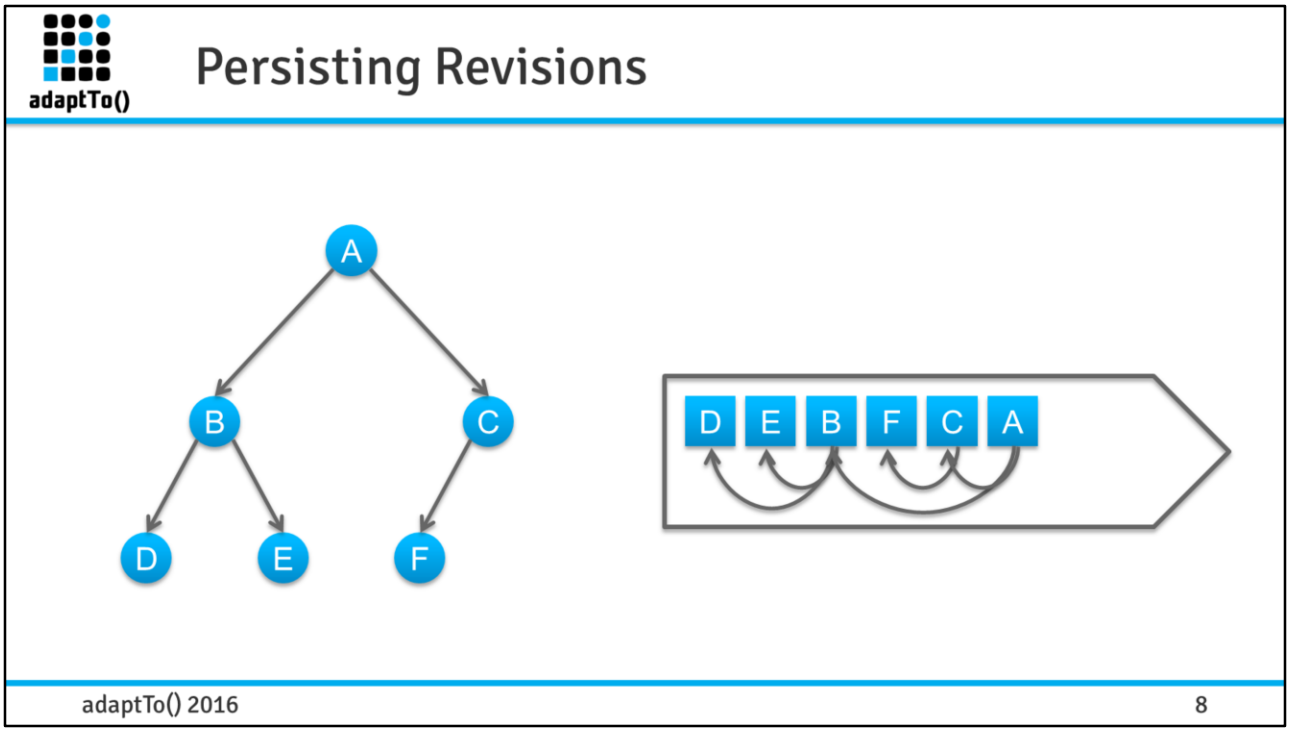
Multi version concurrency control coordinates concurrent access by giving users (the illusion of having) exclusive access to the repository.



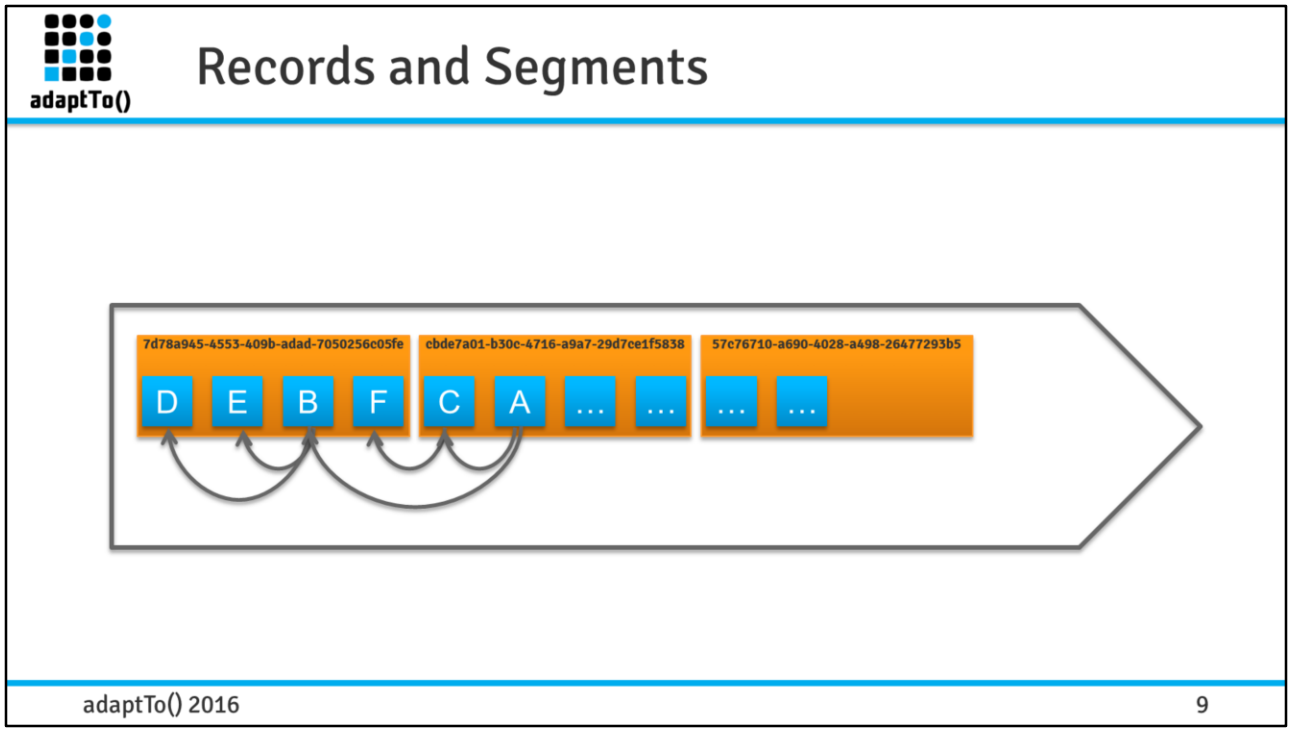
Updating a tree creates a complete new copy of that tree. Unchanged nodes are referenced in the previous tree to avoid duplicating them. Note how changing any node will always cause its whole parent hierarchy to change.



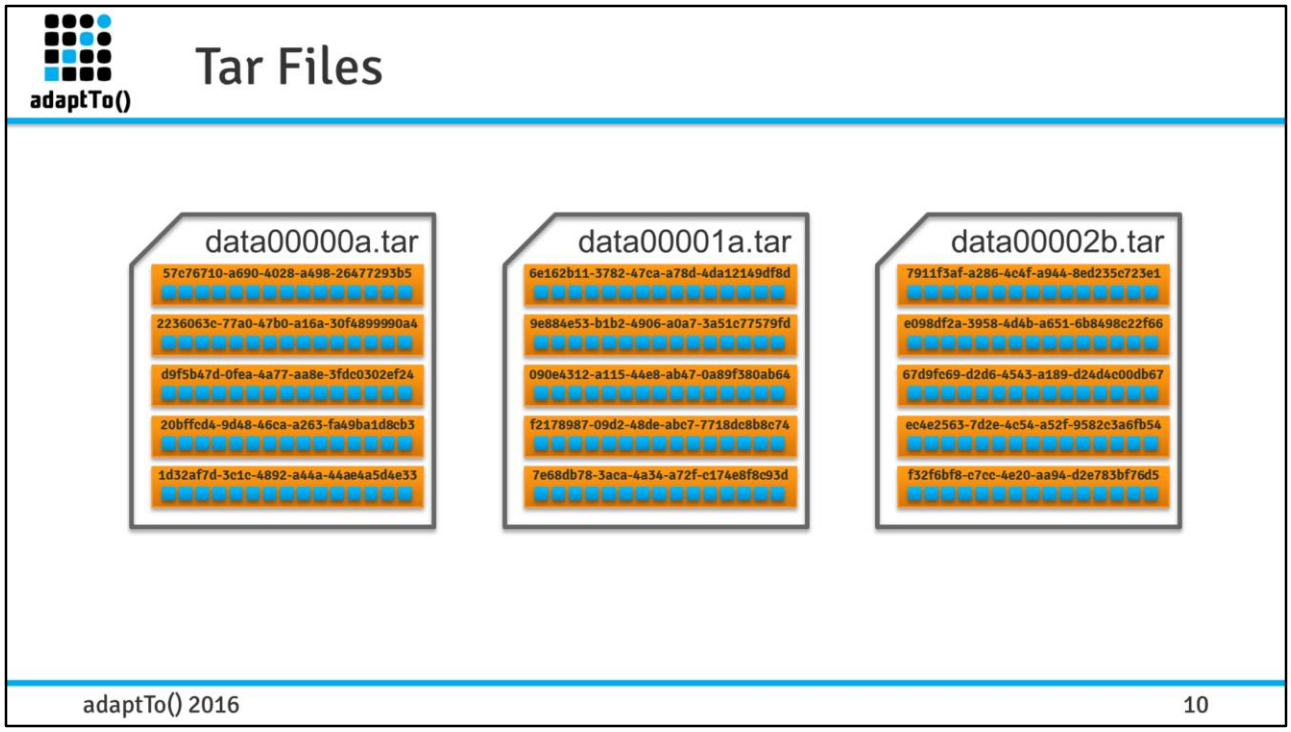
Conceptually each changed node creates a new tree. Unless the same node is edited over and over again each tree references all its predecessors.
Each tree represents a revisions of the repository. The ordered list of the trees form a revision history, which reflects how the repository evolved to its present state. For the TarMK a revision is represented by the identifier of the root node of the respective tree and a revision history is simply a list of such identifiers.



A revision is persisted by serialising the nodes of its respective tree into a stream of records. Serialisation progresses in post order to ensure dependencies are always stored first. This guarantees that a serialised node is always fully readable even if a crash occurs at anytime.



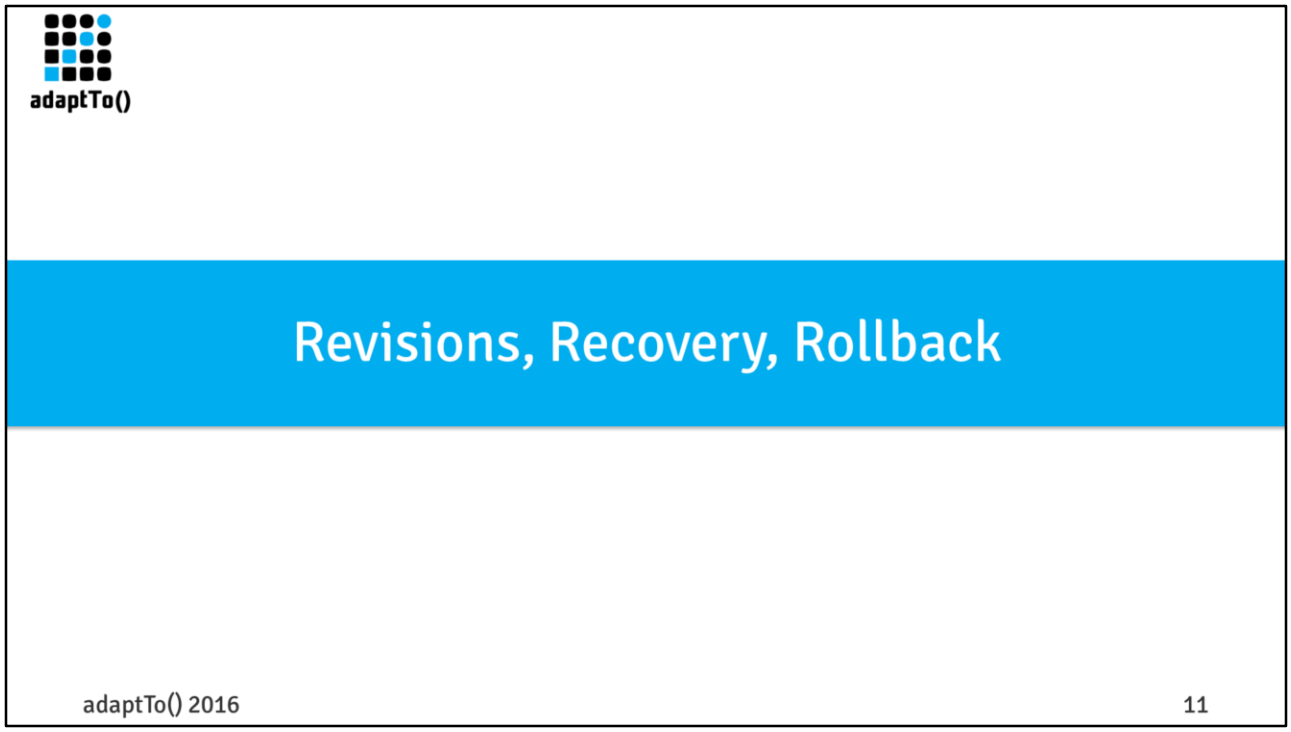
To make records addressable the stream of records is chunked up into segments. A segment is identified by a random UUID (its segment id). Segments contain some header information and a list of records. Records are addressable inside a segment via its offset. A record id is thus a pair consisting of a segment id and an offset. The maximum size of a segment is determined by its address space. The offset of a record id is a 16 bit integer and records are 4 byte aligned in their segment resulting in a maximal segment size of 262'144 bytes.



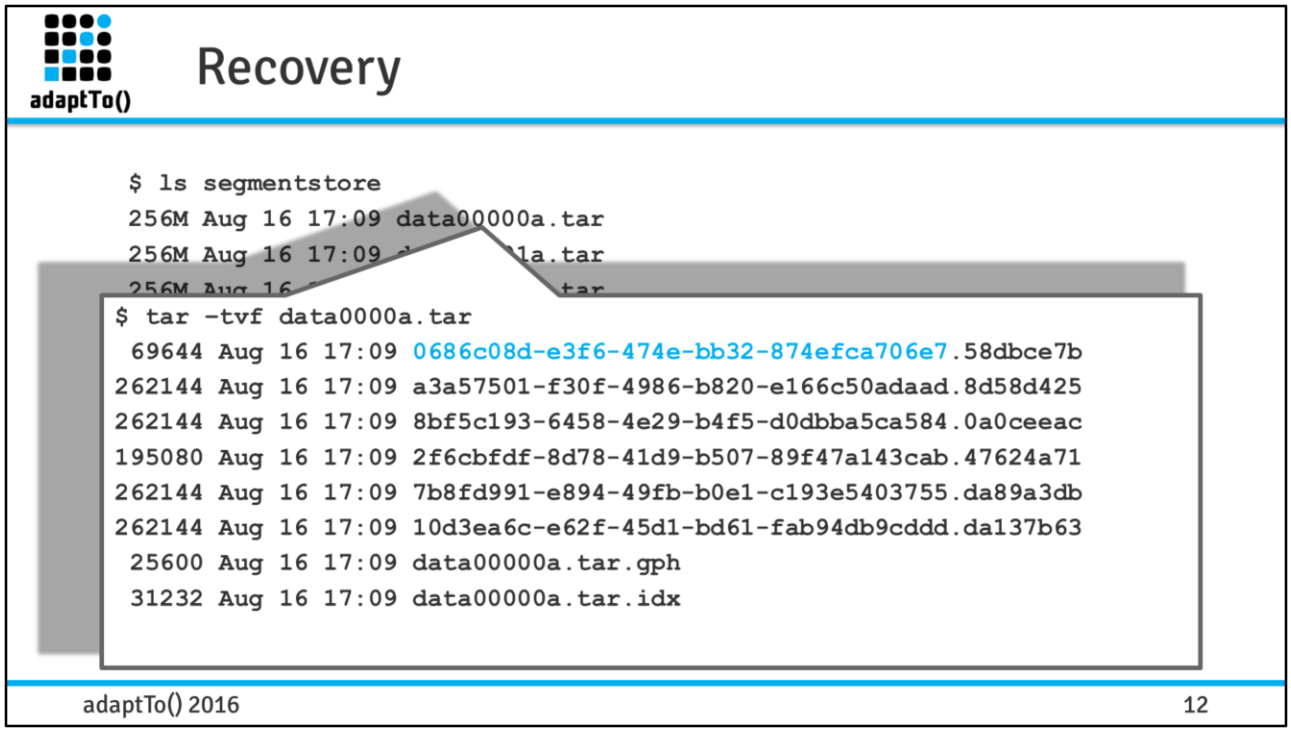
Segments are appended into tar files. Once a tar file becomes full (265MB by default) some auxiliary entries are added and a new tar file is started. Subsequent tar file names include an ever increasing sequence number to maintain a strict order. The auxiliary entries consist of an index of the segments for quicker lookup and a list of segments referenced from this tar files for analysing reachability during garbage collection.

The letter in the tar file's names refer to its generation. When the garbage collector is able to collect enough segments from a tar file such

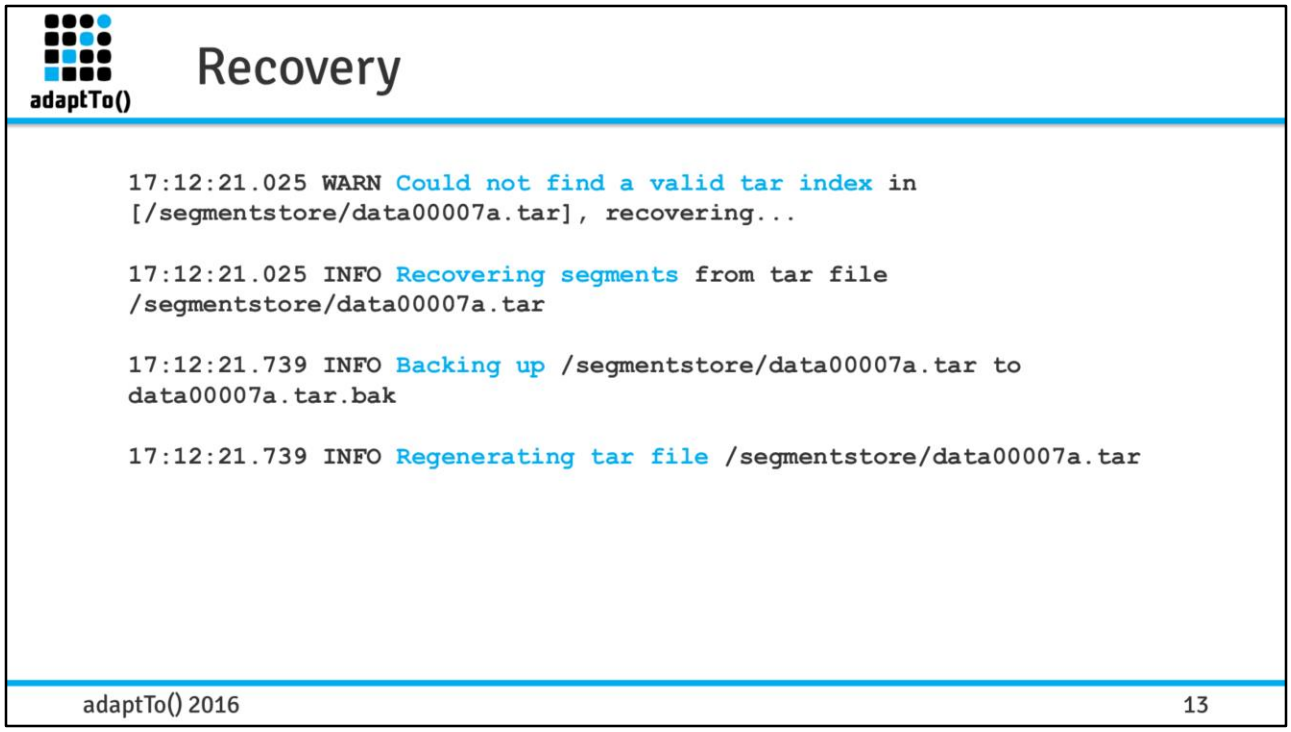
that there is at least 25% space saving for that file, the file is rewritten into a new generation leaving out the garbage collected segments. By default tar files are memory mapped for fast access. So it is important to avoid allocating all available RAM to the JVM (e.g. heap) as otherwise the OS would not have enough space for memory mapping the tar files, which could lead to some form of thrashing.



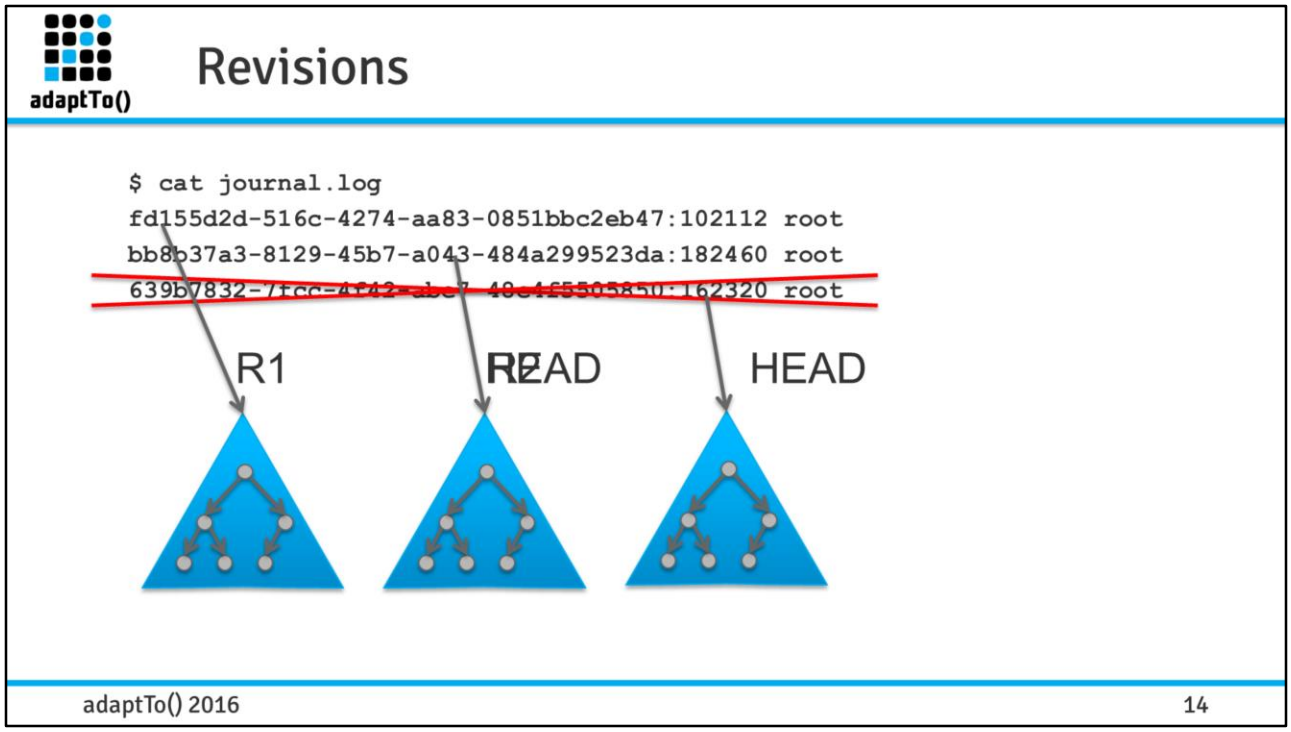
Writing everything in a way such that it only references already written items makes the persistence format resilient against unclean shutdown, crashes, power cuts etc. In these cases recovery is automatic and transparent during the next start-up. More sever corruptions (e.g. bit flip in tar files) need manual intervention to resolve. However, the MVCC nature of the TarMK makes it easy to roll back to the last good state.



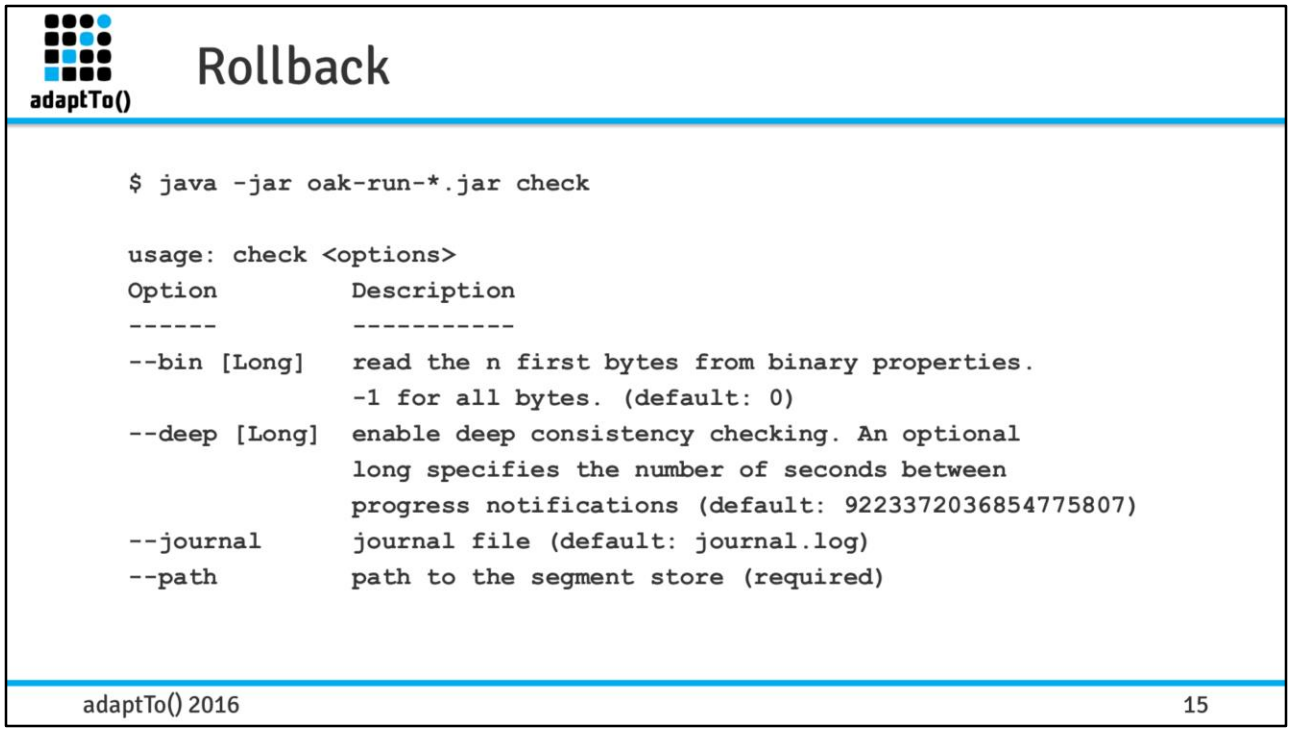
Backup files in a directory listing indicate that a automatic recovery has occurred at start-up. In the case of a crash the tar file that has been last written to might become corrupt. As it hasn't been cleanly closed it will have a missing or corrupt index (the .idx file is always written last and it is check-summed). In the recovery case corrupt tar files are backed up, all recoverable entries are written to a new tar file and a new graph and index entry is added.



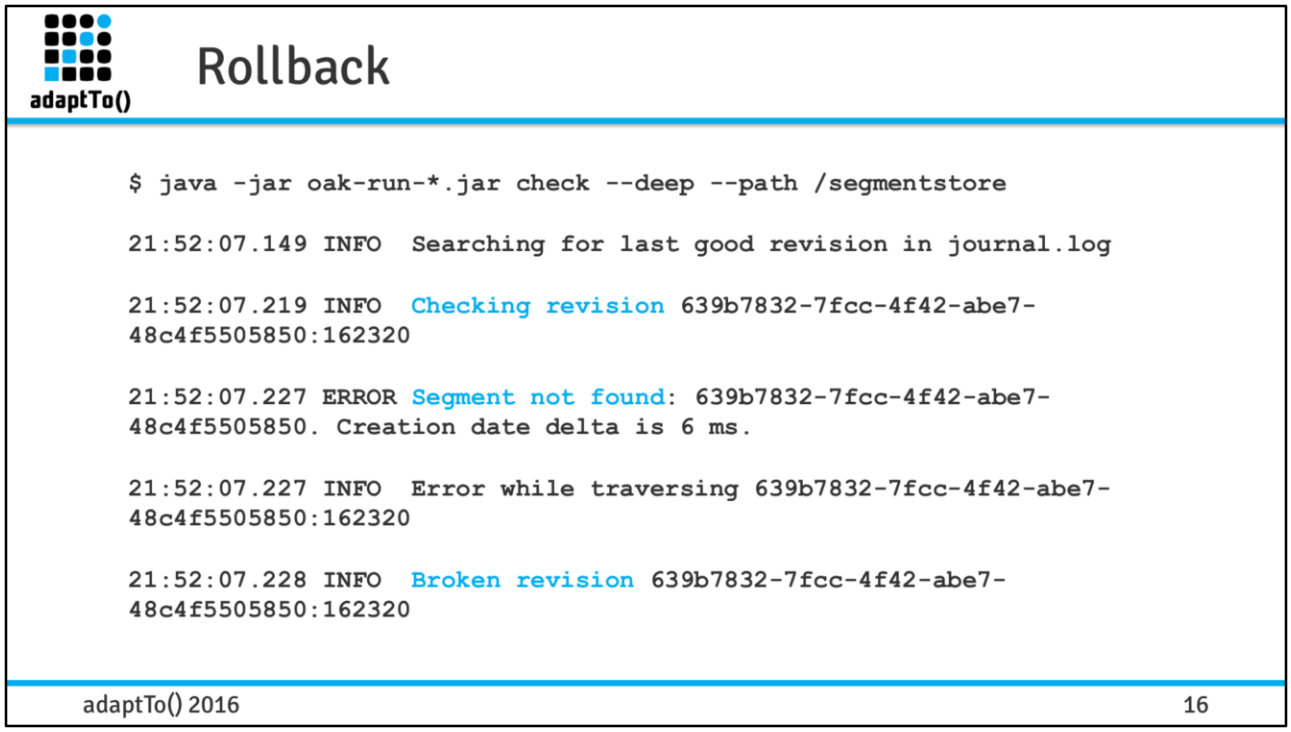
Excerpt of log file entries when an automatic recovery of a tar file occurs at start-up. The process recovers all valid segment entries from the corrupt tar file and regenerates the graph and index entries. The original tar file is backed up before the regenerated one is created.



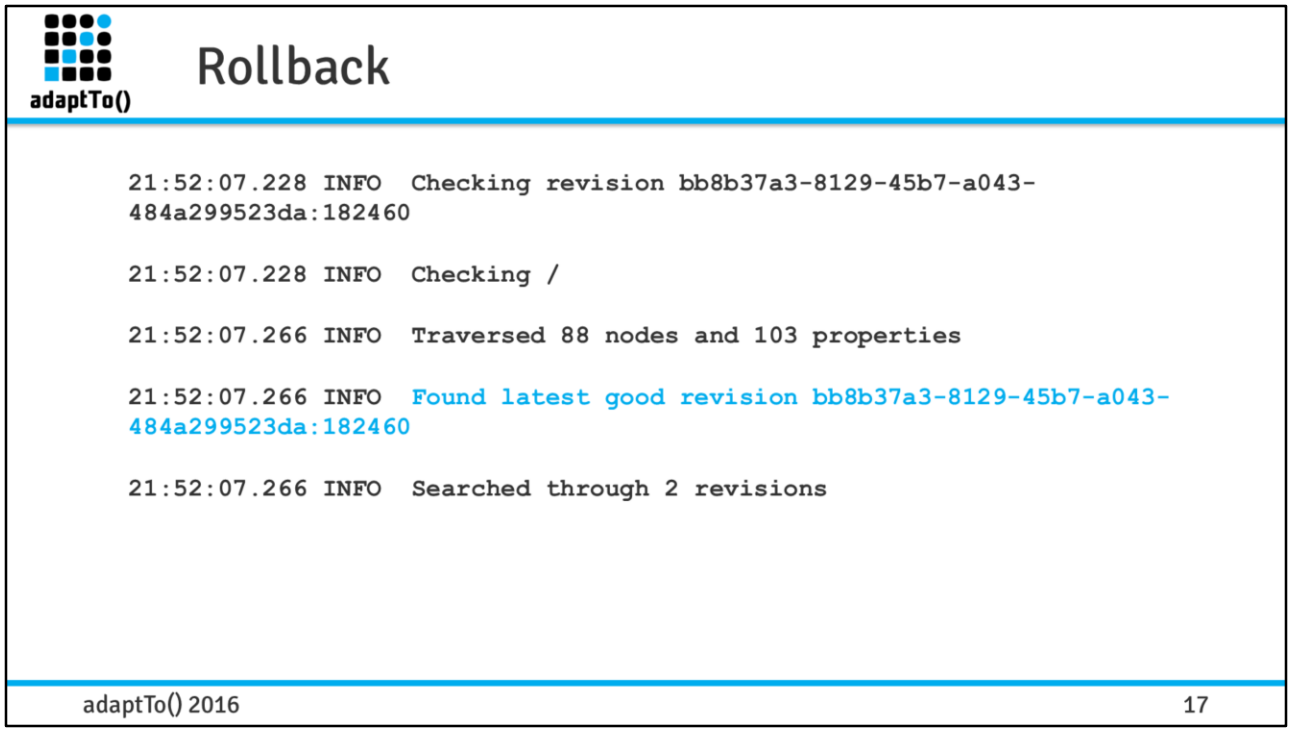
The journal.log file contains an ordered list of revisions (record ids of root nodes) where later entries are appended to the end of the file. Removing entries from the end of the journal causes a roll back of the TarMK to a previous revision.



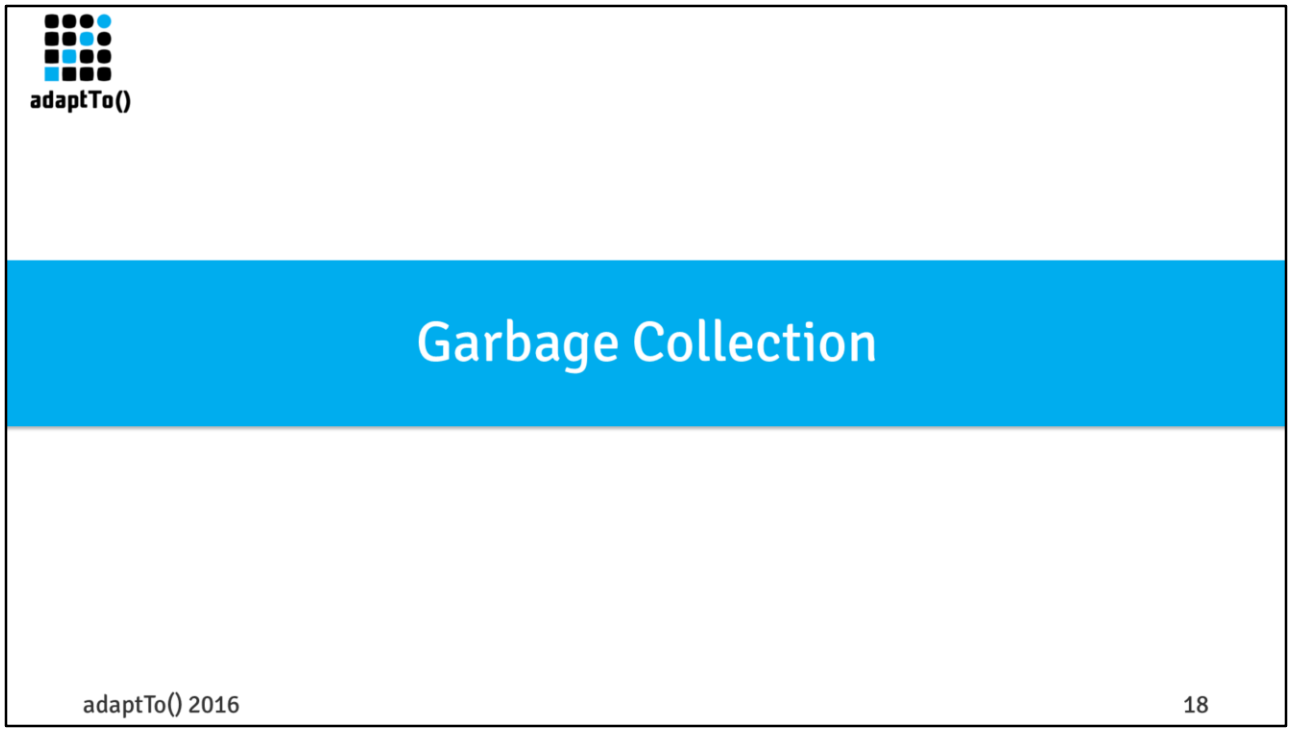
The check run mode of the oak-run utility can be used to find the latest good revision. It traverses all revisions from the journal backward until it finds a good one. Command line arguments specify how thoroughly individual revisions should be checked. In particular the --bin option controls than handling of binaries. Specifying 0 skips reading binaries, which is useful when a blob store is configured. The check process does not modify the repository itself but rather outputs the first good revision it finds (if any). Editing the journal.log file needs to be done manually.



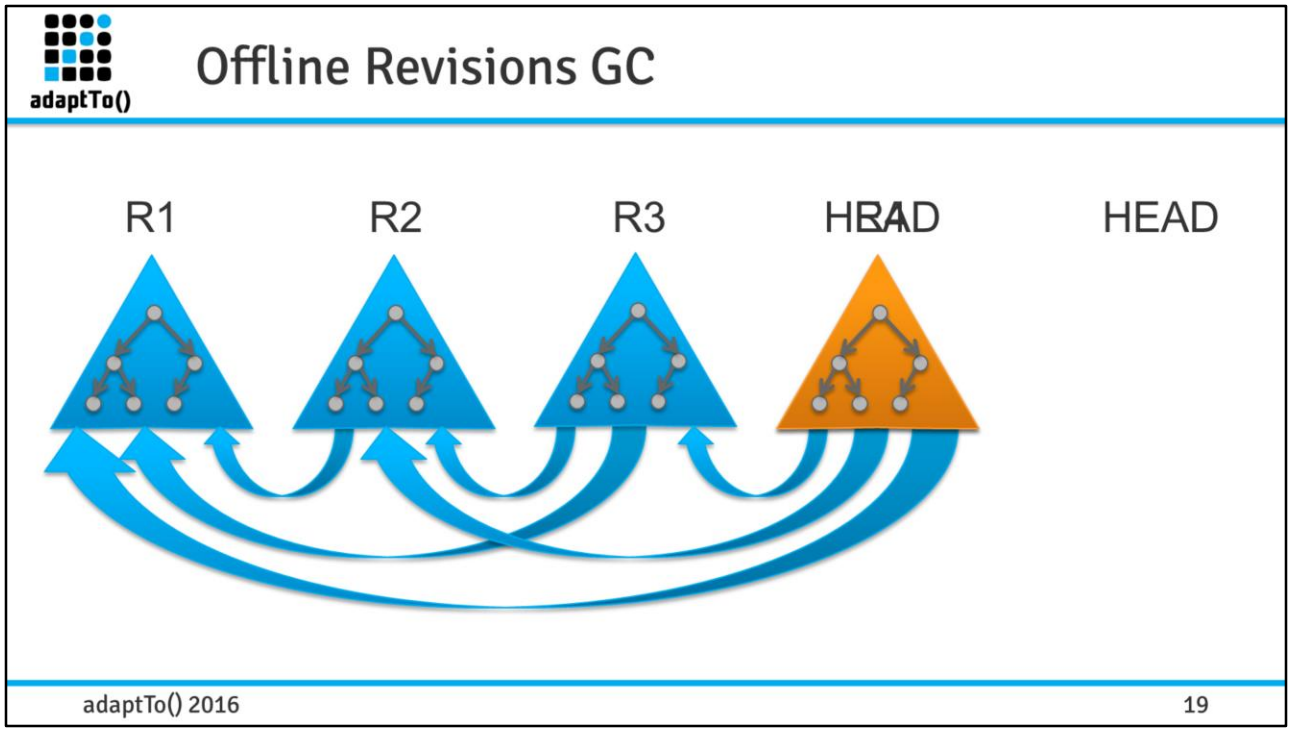
oak-run check outputs the record ids of the revisions it is checking and any errors that occur along the way.



Once oak-run check found a good revision it will output its record id.
Note, that the TarMK also has a basic variant of rolling back the journal.log build into its start up behaviour: if the latest record id in the journal.log cannot be accessed (e.g. because its segment is missing), it will log a warning “Unable to access revision f2178987-09d2-48de-abc7-7718dc8b8c74.63b8, rewinding..” and tries continuing with the previous record id in the journal.

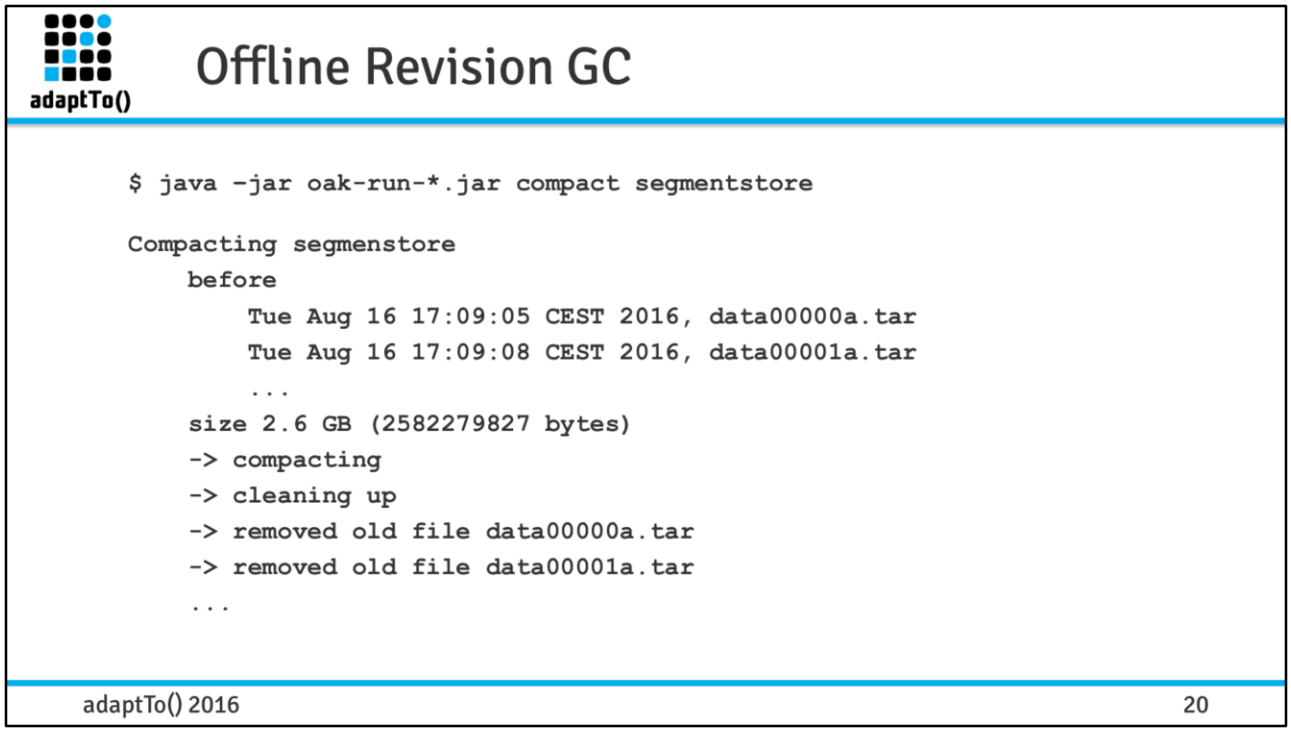


While the append only storage model has many advantages, it leads to a store that only ever grows. No amount of removed nodes will cause the store to shrink. A garbage collection process is required to free space used by unreferenced records. Garbage collection can either run online (concurrent to normal repository operation) or offline (with exclusive access to the store).
Conceptually both modes are almost the same. Their efficacy can greatly vary though.

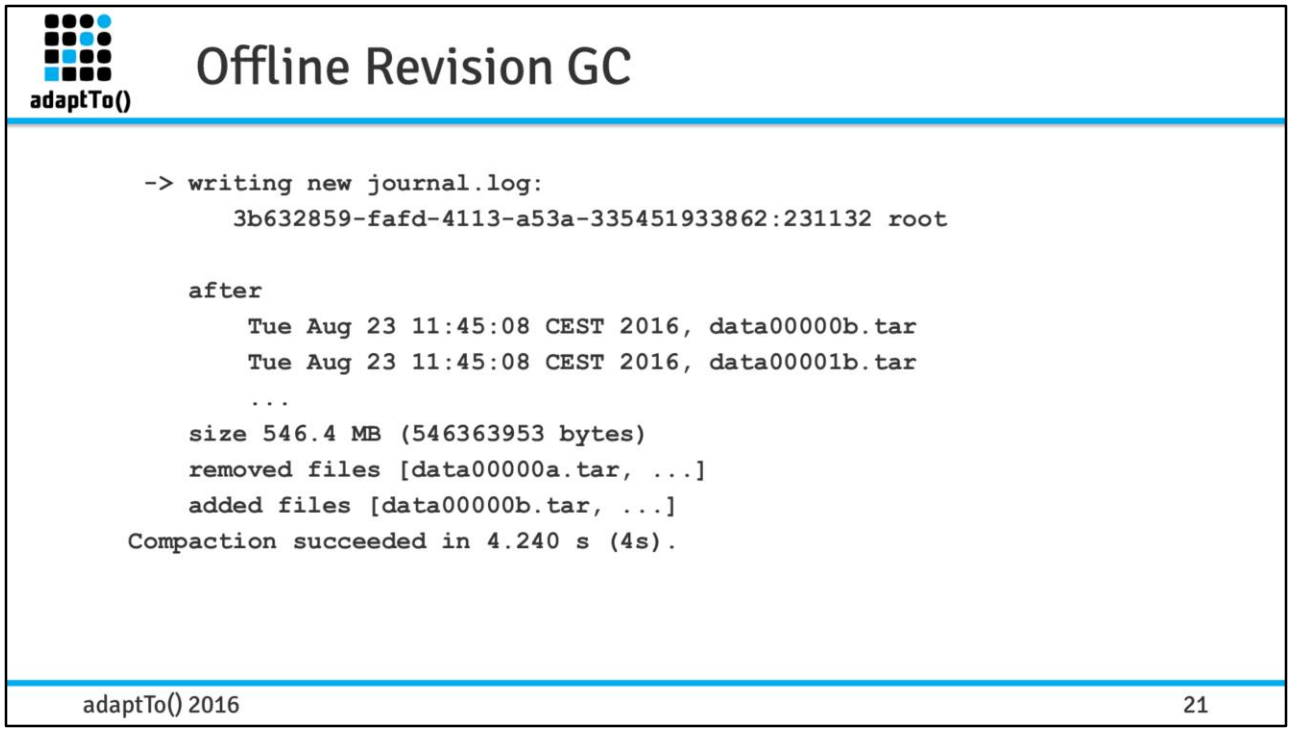


Since tar files, segments and records are immutable, the garbage collector cannot just remove unreferenced items. Instead it will clone the current head state such that it doesn't reference previous states anymore. This is called the compaction phase as it creates a compact representation of the current head state. The subsequent clean-up phase removes segments containing the old, now unreferenced states. Clean-up creates a new generation of any tar file containing at least 25% of garbage (non referenced segments) and removing the old tar file. New

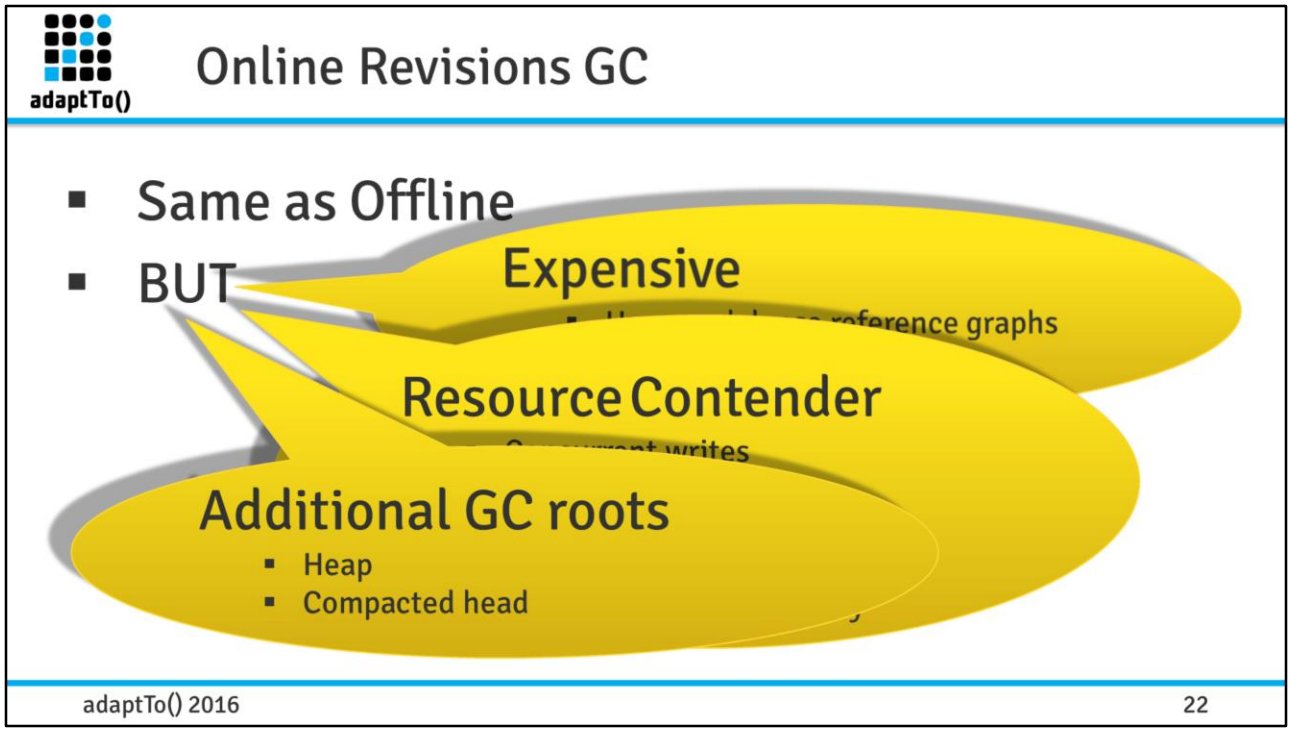
tar files have its generation letter increased. E.g. data00000a.tar will become data00000b.tar.



When running offline revision garbage collection oak-run compact outputs a list of current tar files, the current size of the repository, the steps it is performing (compacting, cleaning up) and a list of the tar files it removed.



It will update the journal to only contain the record id of the new head stated created in the compaction phase and subsequently output a list of tar files after garbage collection concluded as well as the final size, a list of removed files, a list of added files and the time the whole process took.



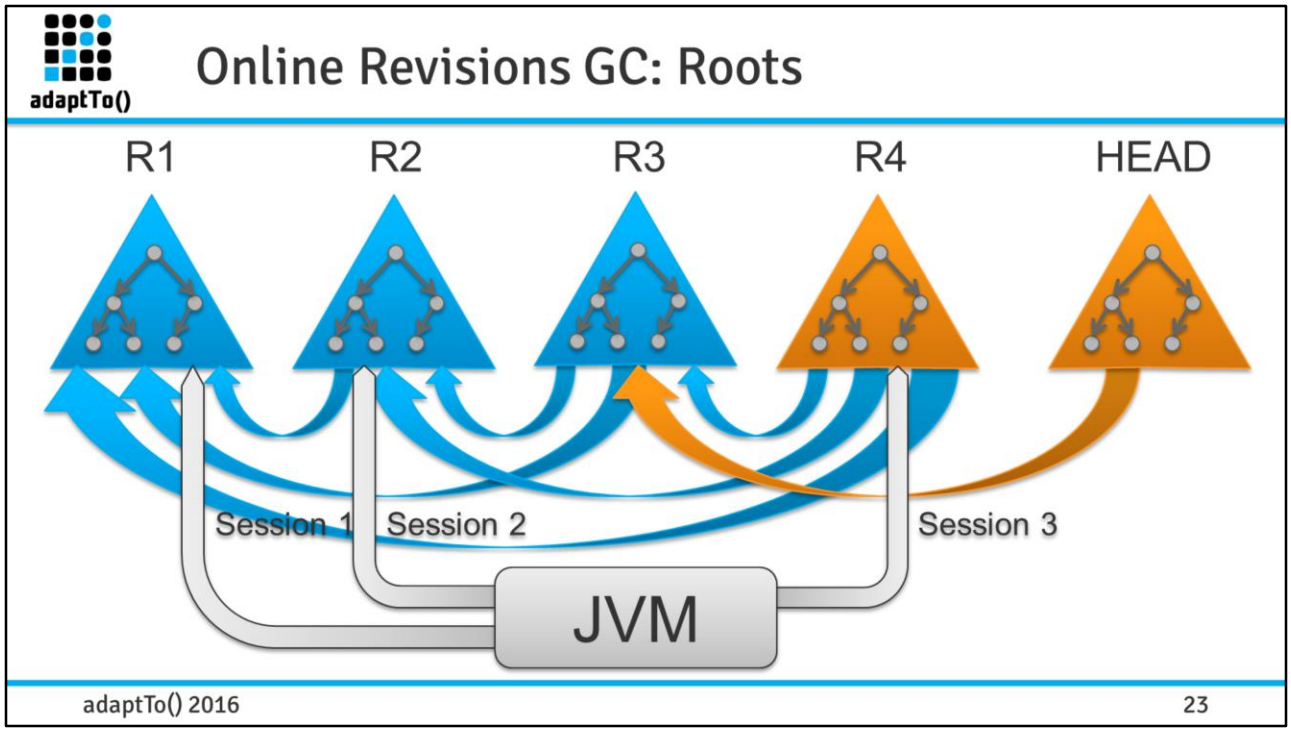
Online revision garbage collection works the same as offline only that it is started from within a running TarMK instance. However, running within an live instance leads to some additional complications:

- Traversing the reachability graph is expensive as the respective graphs are enormously dense. It is a contender for system resources (CPU, disk, lock). As it is a scan operation it also has advert affects on caches that are hot for normal system operation.
- An additional estimation phase should avoid

online revision garbage collection from running if not enough garbage has been accumulated. Unfortunately the estimation phase already has similar effects on normal system operation as the garbage collection process itself.

- The compaction phase races against concurrent writes: when a write was performed concurrently to the compactor creating the clone of the current head state, the new changes need to be compacted first. This process can repeat multiple times as concurrent writes occur. The number of retries can be configured and there is an option to eventually force compaction by acquiring exclusive write access to the store. Both, giving up and forcing compaction is not optimal though. In the first case a lot of work is just thrown away and in the second case concurrent writes start piling up until the compactor eventually finished.
- There are additional gc root from the heap and from later generations blocking segments from being removed when they

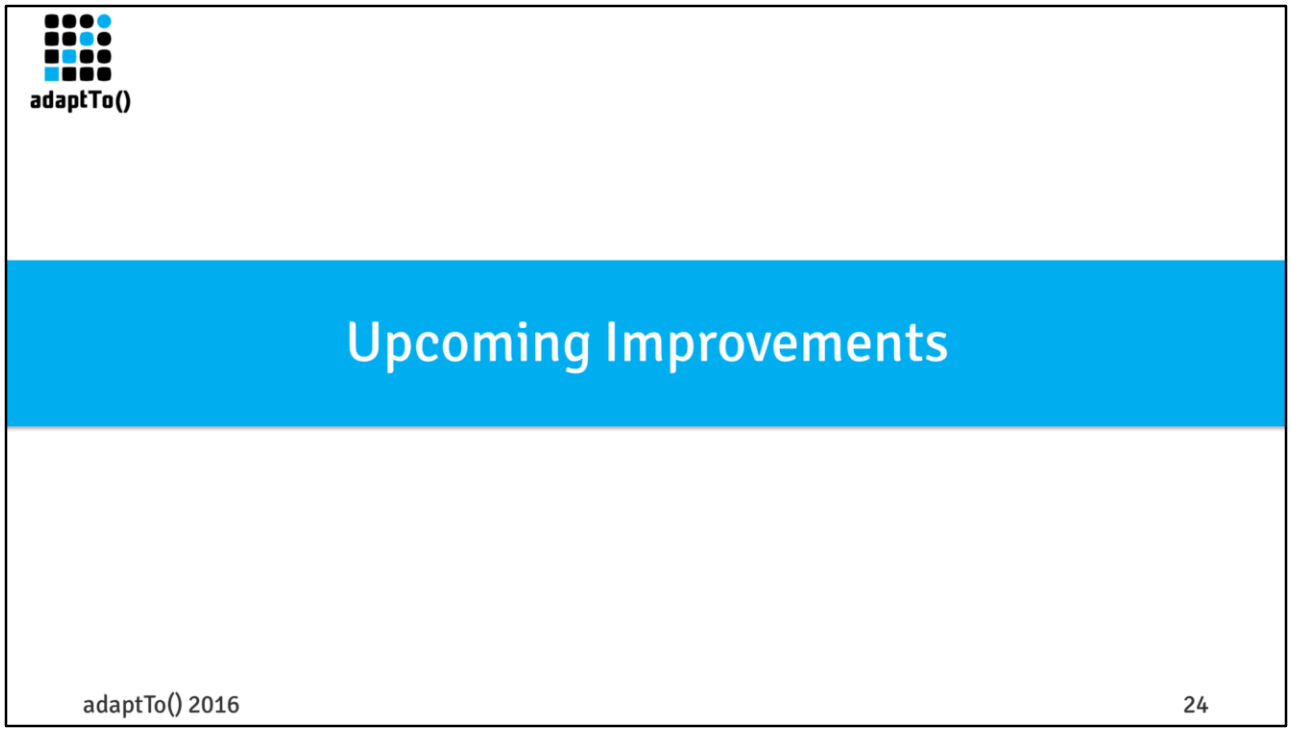
wouldn’t be blocked in the offline revision gc case.



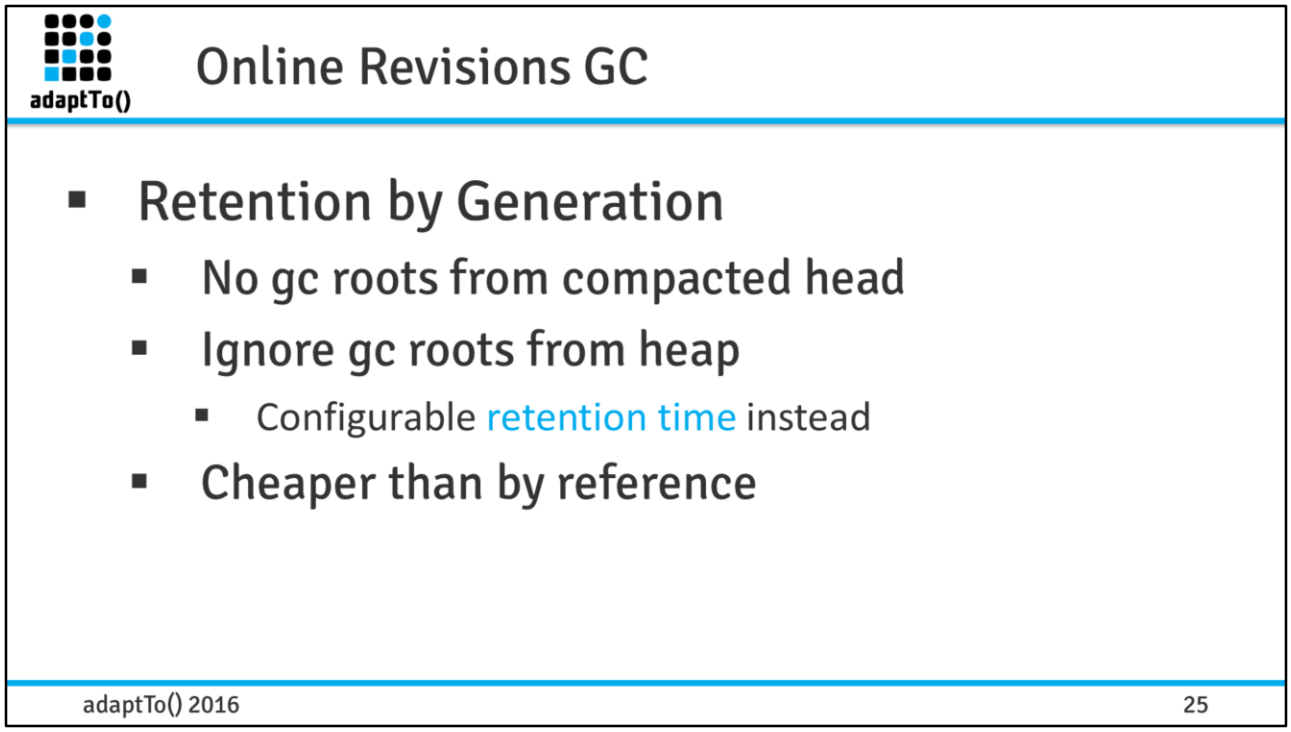
Additional gc roots are introduced by

- A subtle implementation problem during the compaction process. This problem will cause the clone of the head stated created by the compaction phase to reference record in an older revision.
- The application on top of the TarMK referencing older revisions. As a JCR session is based on the head revision from the time it was opened, that revision will ultimately be referenced from the JVM's heap.

Together with the enormous density of the reference graph above two issues often cause the clean-up phase to be less effective than desired.

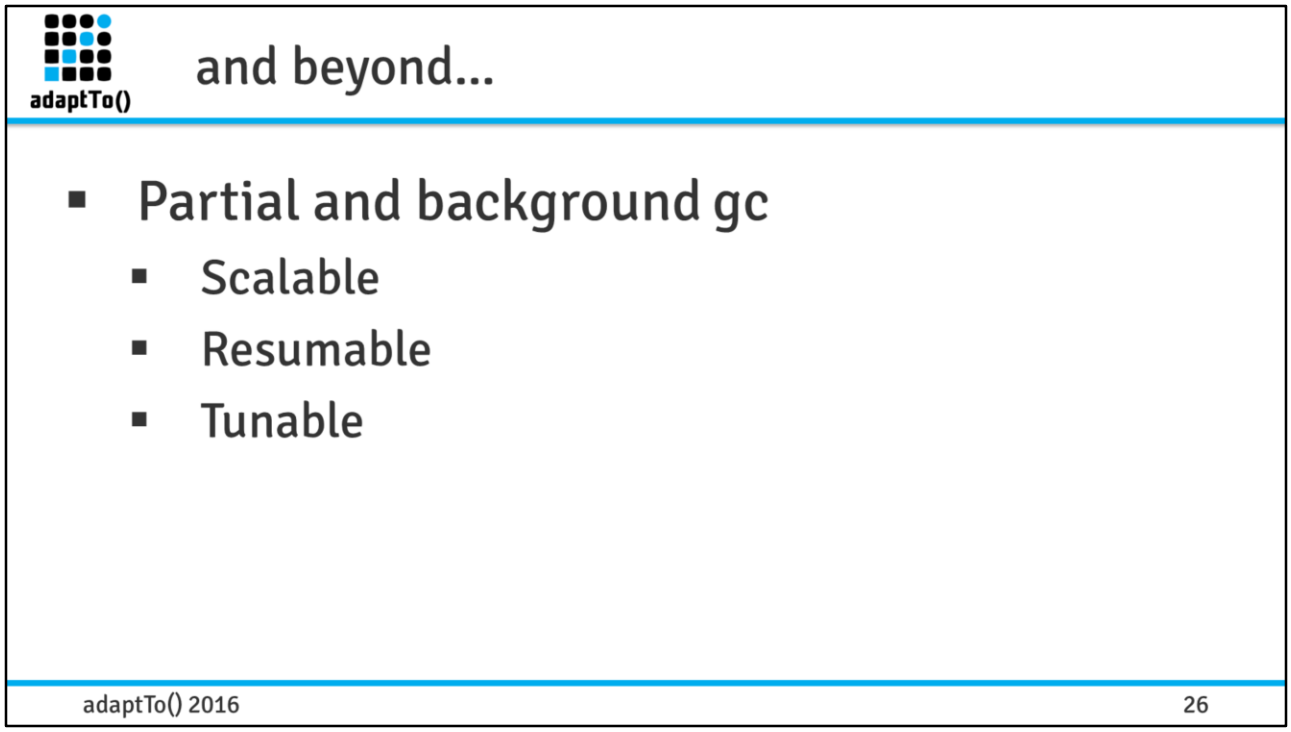


Improving online revision garbage collection requires changes in the segment format. Repositories of older formats are incompatible to the new format and need to be migrated.



The problem with the compacted head referencing older states is fixed in the next version of the TarMK. This leads to a clear separation between gc generations. That is, each time compaction is performed a new generation is written that does not have a reference to any previous generation. To avoid references from heap (“old sessions”) to block clean up from removing old revisions, a retention time base clean-up mechanism is employed: by default anything that is older that one generation is removed and sessions still referring to such old revisions are

automatically refreshed to the current head revision.



In preparation for further improvements (mainly wrt. to performance and scalability) we are working on further changes to the storage format. To improve scalability we mainly aim at making garbage collection a background process that would run during idle times. At the same time there is attempts to partition the compaction step such that it would be possible to run partial garbage collections (i.e. on parts of the tree).

Conclusion

- Many improvements for gc
  - Changed storage format
  - Migration required
- Ground work for future improvements
  - More scalable gc
  - Bigger repositories
  - More write throughput

Questions