Ignite Persistent Store - under the hood

Following notation is used: words written in italics and wihout spacing mean class name without package name or method name, for example, *GridCacheMapEntry.*

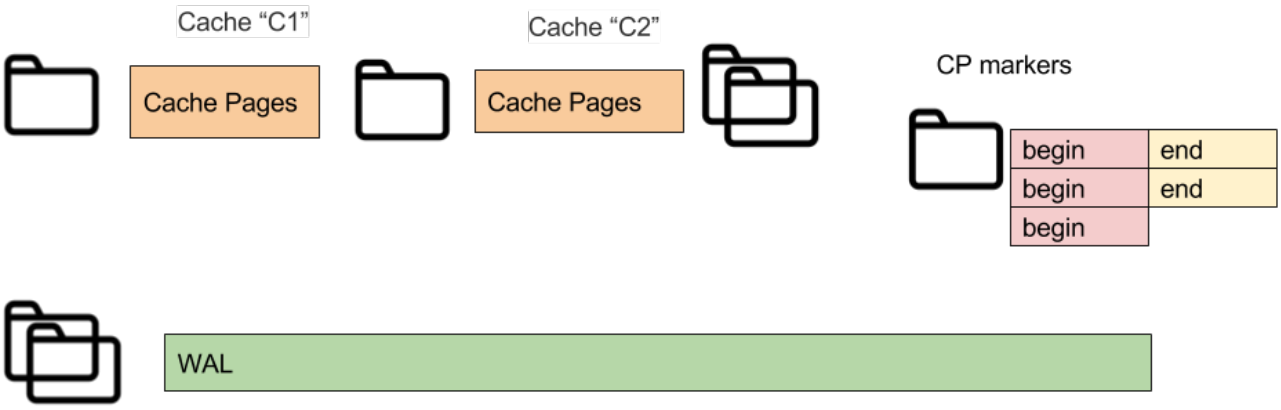
For the sake of simplicity in this article, every cache group is named as a cache. Logical caches (as part of cache group) are not mentioned in this article at all, and a cache always implies a cache group. Each persistence store file naming will contain only cache group name (which is equal to cache name if there is only one logical cache in the group).

Table of Contents:

Ignite Persitent Store

File types

There are following file types used for persisting data: Cache pages or page store, Checkpoint markers, and WAL segments.

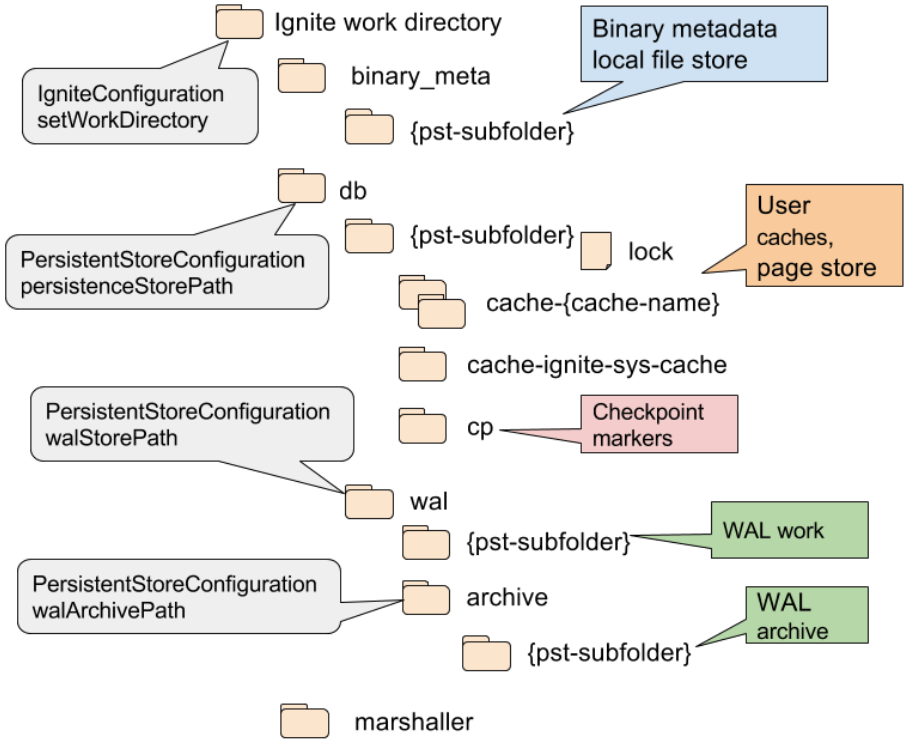


- Write Ahead Log (WAL) segments - constant size files (WAL work directory 0...9.wal; WAL archive 0.wal…).
- CP markers - small files for events of starting and finishing checkpoints (UUID-Begin.bin, UUID-End.bin).
- Page store - contains cache parameters, page store for partitions and SQL indexes data (uses a file per partition: cache-(cache_name)\part1,2,3.bin, and index.bin).

Folders Structure

Ignite with enabled persistence uses following folder structure:

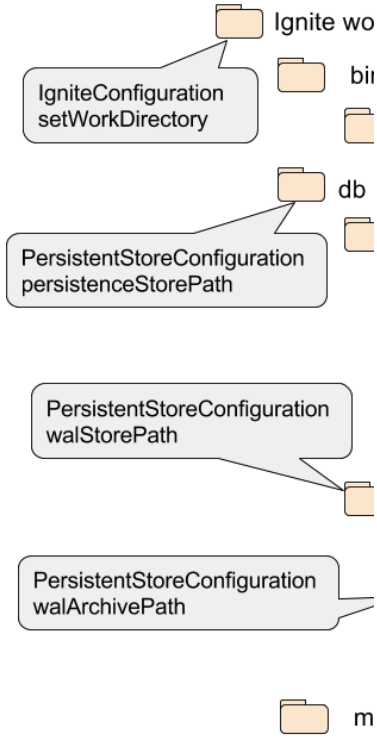
2.3+



The Pst-subfolder name is same for all storage folders.

A name is selected on start, may be based on node consistentId.

Older versions (2.1 & 2.2)

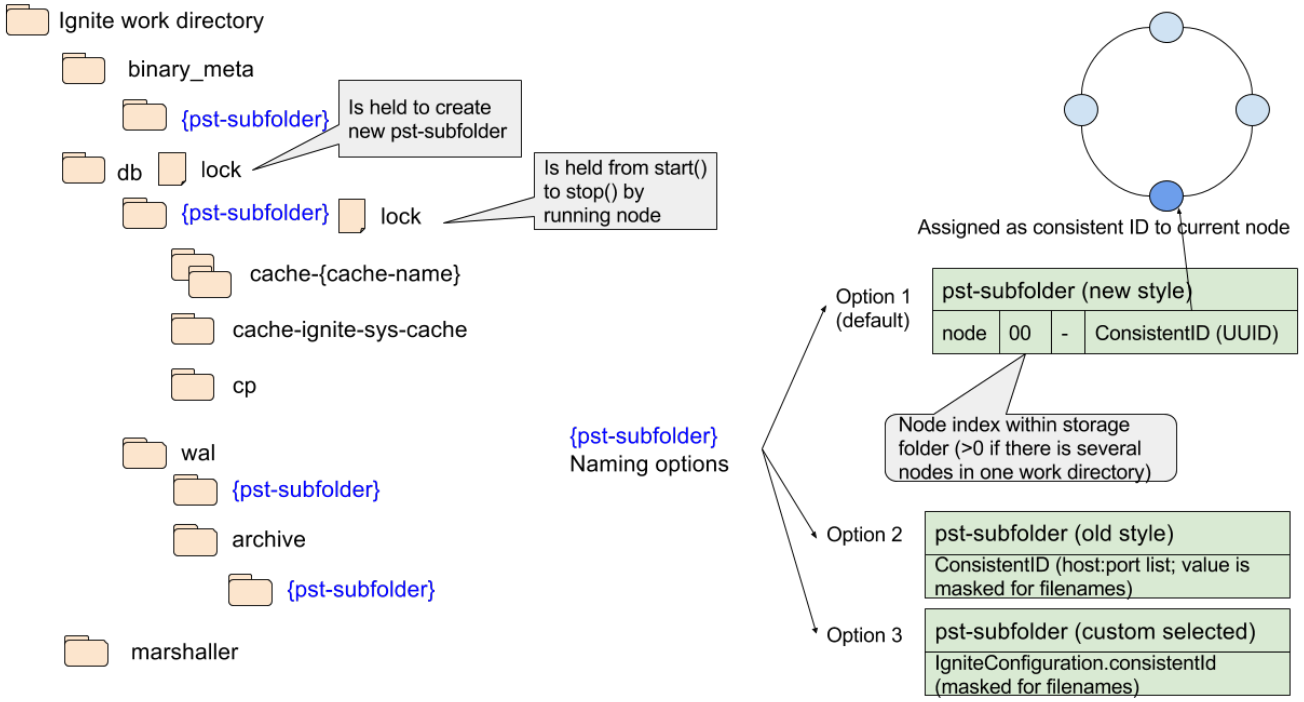


Consistent ID may be configured using Ignit

Subfolders Generation

The subfolder name is generated on start. By default new style naming is used, for example node00-e819f611-3fb9-4dbe-a3aa-1f6de4af5d02

- where 'node' is a constant prefix,
- '00' is node index, it is an incrementing counter of local nodes under same PST root folder,
- remaining is string representation UUID, and this UUID became node's consistent ID.



PST subfolder naming options explained:

Option 1. For case ignite is started for clear persistence storage root folder, this (new style) naming is used.

Option 2. This option is used in case there is existing pst-subfolder with exact the same name as for compatible consistent ID (local host IPs and ports list). If there is such folder, Ignite is started using this one, and consistent ID is not changed.

Option 3. This option is applied in case there is a preconfigured value from *IgniteConfiguration.*

In case there is an old style folder, but its name doesn't match with compatible consistent ID, following warning is generated:

```
There is other non-empty storage folder under storage base directory [work\db\127_0_0_1_49999, 299718 bytes,
modified 10/04/2017 04:33 PM ]
```

There are two file locks used in folders selection.

The first one is used to check if there is no up and running node which is using the same directory.
This lock is placed in work/db/{pst-subfolder}/lock (work/db may be still customized by storage folder property)

The second lock is placed in the storage root folder: work/db/lock. This lock is held for a short time when new pst-subfolder is being created. This protects from concurrent folder initialization by nodes which are starting simultaneously.

Exact generation algorithm and code references:

1) A starting node binds to a port and generates old-style compatible consistent ID (e.g. 127.0.0.1:47500) using *DiscoverySpi.consistentId().* This method still returns ip:port-based identifier.

2) The node scans the work directory and checks if there is a folder matching the consistent ID. (e.g. work\db\127_0_0_1_49999). If such a folder exists, we start up with this ID (compatibility mode), and we get file lock to this folder. See *PdsConsistentIdProcessor.prepareNewSettings.*

3) If there are no matching folders, but the directory is not empty, scan it for old-style consistent IDs. If there are old-style db folders, print out a warning (see warning text above), then switch to new style folder generation (step 4).

4) If there are existing new style folders, pick up the one with the smallest sequence number and try to lock the directory. Repeat until we succeed or until the list of new-style consistent IDs is empty. (e.g. work\db\node00-uuid, node01-uuid, etc).

5) If there are no more available new-style folders, generate a new one with next sequence number and random UUID as consistent ID. (e.g. work\db\node00-uuid, uuid overrides uuid in *GridDiscoveryManager).*

6) Use this consistent ID for the node startup (using value from *GridKernalContext.pdsFolderResolver*() and from *PdsFolderSettings.consistentId()*).

There is a system property to disable new-style generation and using old-style consistent ID (*IgniteSystemProperties.IGNITE_DATA_STORAGE_FOLDER_BY_CONSISTENT_ID*).

Page store

Ignite Durable Memory is the basis for all data structures. There is no cache state saved on heap now.

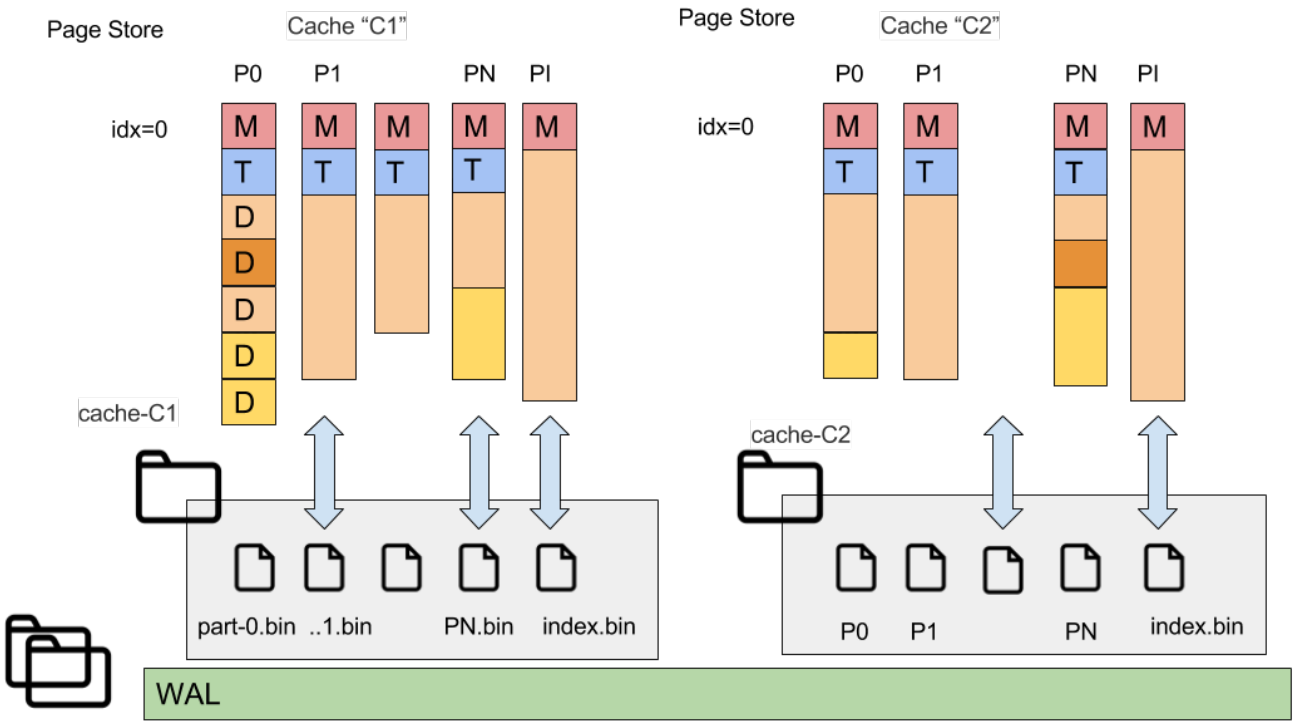
To save the cache state to hard disk we can dump all cache's pages to a file. First prototypes used this simple approach: stop all updates and save all pages.

Let's define page store as the storage for all pages related to a particular cache. And more precisely, cache's partitions and SQL indexes.

Let's introduce a page identifier. Our requirement it should be possible to map from a page ID to a file, and to position in a particular file. Page ID can be defined as follows:

```
pageId = ... || partition ID || page index (idx)
//pageId can be easily converted to file + offset in this file
offset = idx * pageSize
```

Partitions of each cache have a corresponding file in the page store directory (the particular node may own not all partitions).



Each cache has a corresponding folder in the page store (named as 'cache-(cache-name)'). And each owning (or backup) partition of this cache has its related file.

Cache page storage contains the following files:

- part-0.bin, part-1.bin, ..., part-1023.bin (shown as P1,P2,PN at picture) - Cache partition pages.
- index.bin - index partition data, special partition with number 65535 is used for SQL indexes and saved to index.bin.
- cache_data.dat - special file with stored cache data (configuration). A *StoredCacheData* includes more data than the corresponding *CacheConfiguration*, e.g., query entities are saved in addition.

Persistence

Checkpointing

We can define checkpointing as a process of storing dirty pages from RAM on a disk, with results of consistent memory state is saved to disk. At the point of process end, page state is saved as it was for the time the process begins.

There are two approaches to implementation of checkpointing:

- Sharp Checkpointing - if a checkpoint completes: all data structures on disk are consistent, data is also consistent in terms of references and transactions.
- Fuzzy Checkpointing - means state on disk may require recovery itself.

The approach implemented in Ignite is Sharp Checkpoint; Fuzzy Checkpointing- may be done in future releases.

To achieve consistency checkpoint read-write lock is used (see *GridCacheDatabaseSharedManager#checkpointLock*)

- Cache Updates - holds read lock.
- Checkpointer, begin checkpoint - holds write lock for a short time. Holding write lock means all state is consistent, updates are not possible. Usage of checkpoint lock allows doing sharp checkpoint

    Under checkpoint write lock held we do the following:

    i. Add WAL marker: checkpoint (begin) record is added - *CheckpointRecord* - marks consistent state in WAL.
    ii. Collect pages, which were changed since the last checkpoint.

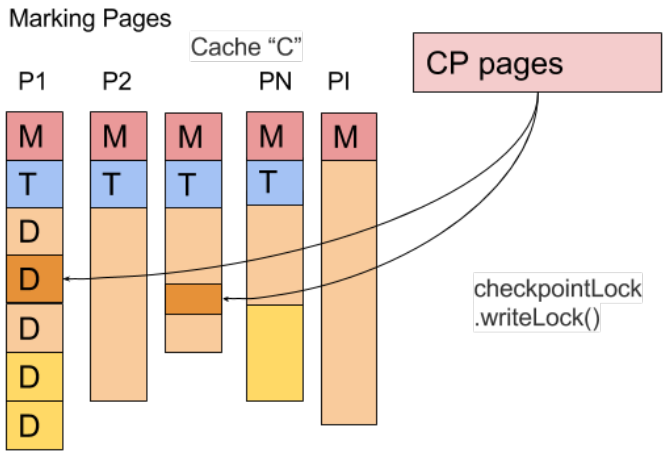
    And then checkpoint write lock is released, updates and transactions can run.

Usage of *checkpointLock* provides the following warranties

- 1 begin checkpoint, and 0 updates (no transactions commit()s) running or
- 0 begin checkpoint and N updates (transactions commit()s) running

Checkpoint begin does not wait transactions to finish. That means a transaction may start before a checkpoint but the transaction will be committed after the checkpoint ends or during its run.

The durable memory maintains dirty pages set. When a page from non-dirty becomes dirty, this page is added to this set.



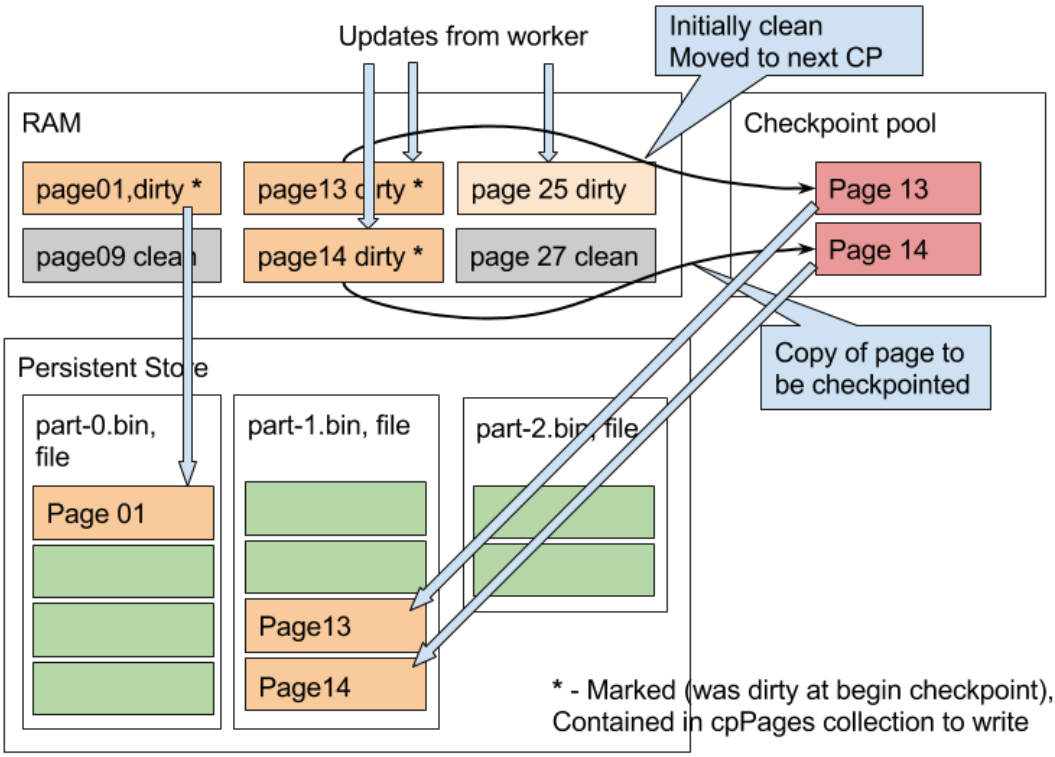
Collection of pages (*GridCacheDatabaseSharedManager.Checkpoint#cpPages*) is a snapshot of dirty pages at checkpoint start. This collection allows writing pages which were changed since the last checkpoint.

Checkpoint Pool

In parallel with the process of writing pages to disk, some thread may need to update data in the page being written (or scheduled to being written).

For such case, checkpoint pool (or checkpoint buffer) is used for pages under simultaneous update with write. This pool has limitation.

Copy on write technique is used. If there is modification required in a page which is under checkpoint, Ignite creates a temporary copy of this page in checkpoint pool.



To perform write to a dirty page scheduled to be checkpointed following is done:

- write lock is acquired to the page to protect from inconsistent changes,
- then a full copy of the page is created in checkpoint pool (checkpoint pool is the latest chunk of each durable memory segment),
- actual data is written to regular segment chunk after the copy is created,
- and later copy of the page from the checkpoint pool will be written to disk.

If a page was not involved into checkpoint initially, and it is updated concurrently with the checkpointing process following is done:

- it is updated directly in memory bypassing checkpoint pool,
- and actual data will be stored to disk during next checkpoint.

When a dirty page flushed to disk, the dirty flag is cleared. Every future write to the before-mentioned page (which was initially involved into the checkpoint, but was flushed) does not require the checkpoint pool usage, it is written directly into a segment.

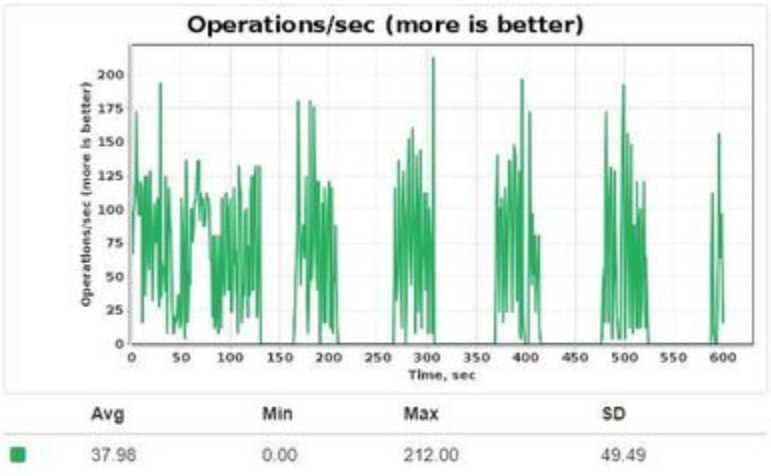
Checkpoint Triggers

Following events triggers checkpointing:

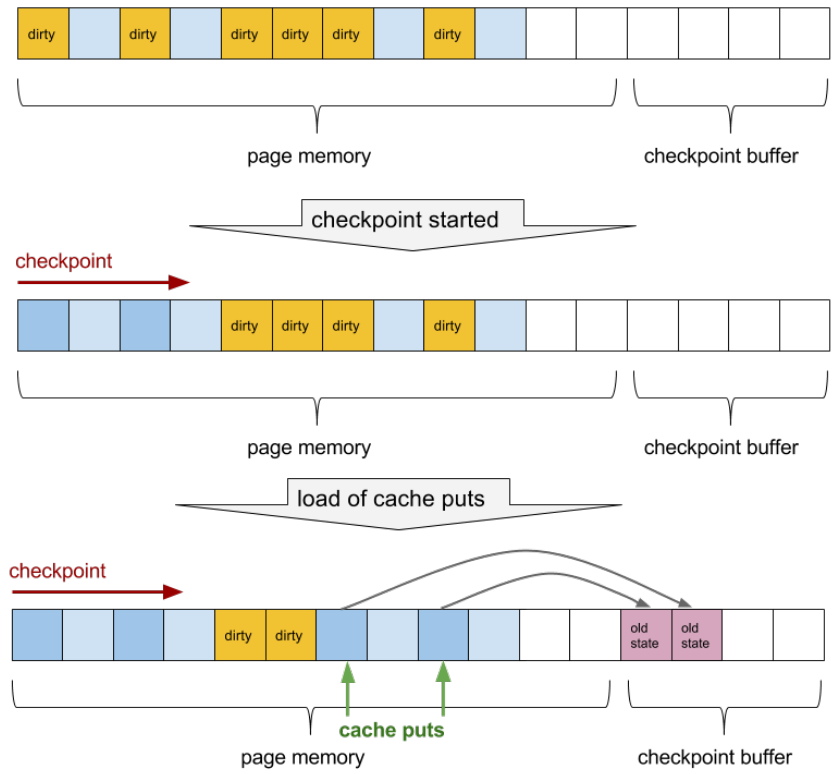
- Percent of dirty pages is a trigger for checkpointing (e.g. 75%).
- Timeout is also a trigger as well. User may specify to do a checkpoint every N seconds.

Pages Write Throttling

Sharp Checkpointing has side-effects when the throughput of data updates is higher than the throughput of a physical storage device. Under heavy load of writes, the rate of operations per second periodically drops to zero:



When offheap memory accumulates too many dirty pages (pages with data not written to disk yet), Ignite node initiates checkpoint — a process of writing a onsistent snapshot of all pages to disk storage. If a dirty page is changed during ongoing checkpoint before being written to disk, its previous state is copied to a special data region — checkpoint buffer:



Slow storage devices cause long-running checkpoints. And if a load is high while a checkpoint is slow, two bad things can happen:

- Checkpoint buffer can overflow
- Dirty pages threshold for next checkpoint can be reached during current checkpoint

Any of two events above causes Ignite node to freeze all updates until the end of current checkpoint. That's why operations/sec graph falls to zero.

Checkpoint buffer overflow protection is always enabled.

- If checkpoint buffer is being filled too fast. Fill ratio more than 66% triggers throttling.

Since Ignite 2.3, data storage configuration has *writeThrottlingEnabled* property. If it's enabled, following possible situation that can trigger throttling:

- If percentage of dirty pages increases too rapidly.

If throttling is triggered, threads that generate dirty pages are slowed with LockSupport.parkNanos(). Throttling stops when none of two conditions above is true (or when checkpoint finishes). As a result, a node provides constant operations/sec rate at the speed of storage device instead of initial burst and following long freeze.

There are two approaches to calculate necessary time to park thread:

1. exponential backoff (start with ultra-short park, every next park will be <factor> times longer)
2. and speed-based (collect history of disk write speed measurements, extrapolate it to calculate "ideal" speed and bound threads that generate dirty pages with that "ideal" speed)

Ignite node chooses one of them adaptively.

How to detect that throttling is applied

There are two ways to find out that Pages Write Throttling affects data update operations.

1. Take a thread dump - some threads will be waiting at LockSupport#parkNanos with "throttle" classes in a trace. Example stacktrace:

```
"data-streamer-stripe-4-#14%pagemem.PagesWriteThrottleSandboxTest0%@2035" prio=5 tid=0x1e nid=NA waiting
  java.lang.Thread.State: WAITING
          at sun.misc.Unsafe.park(Unsafe.java:-1)
          at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:338)
          at org.apache.ignite.internal.processors.cache.persistence.pagemem.
PagesWriteSpeedBasedThrottle.doPark(PagesWriteSpeedBasedThrottle.java:232)
          at org.apache.ignite.internal.processors.cache.persistence.pagemem.
PagesWriteSpeedBasedThrottle.onMarkDirty(PagesWriteSpeedBasedThrottle.java:220)
          at org.apache.ignite.internal.processors.cache.persistence.pagemem.PageMemoryImpl.allocatePage
(PageMemoryImpl.java:463)
          at org.apache.ignite.internal.processors.cache.persistence.freelist.AbstractFreeList.
allocateDataPage(AbstractFreeList.java:463)
          at org.apache.ignite.internal.processors.cache.persistence.freelist.AbstractFreeList.
insertDataRow(AbstractFreeList.java:501)
          at org.apache.ignite.internal.processors.cache.persistence.RowStore.addRow(RowStore.java:102)
          at org.apache.ignite.internal.processors.cache.
IgniteCacheOffheapManagerImpl$CacheDataStoreImpl.createRow(IgniteCacheOffheapManagerImpl.java:1300)
          at org.apache.ignite.internal.processors.cache.persistence.
GridCacheOffheapManager$GridCacheDataStore.createRow(GridCacheOffheapManager.java:1438)
          at org.apache.ignite.internal.processors.cache.GridCacheMapEntry$UpdateClosure.call
(GridCacheMapEntry.java:4338)
          at org.apache.ignite.internal.processors.cache.GridCacheMapEntry$UpdateClosure.call
(GridCacheMapEntry.java:4296)
          at org.apache.ignite.internal.processors.cache.persistence.tree.BPlusTree$Invoke.invokeClosure
(BPlusTree.java:3051)
          at org.apache.ignite.internal.processors.cache.persistence.tree.BPlusTree$Invoke.access$6200
(BPlusTree.java:2945)
          at org.apache.ignite.internal.processors.cache.persistence.tree.BPlusTree.invokeDown(BPlusTree.
java:1717)
          ...
```

2. If throttling is applied, related statistics is dumped to log from time to time:

```
[2018-03-29 21:36:28,581][INFO ][data-streamer-stripe-0-#10%pagemem.PagesWriteThrottleSandboxTest0%]
[PageMemoryImpl] Throttling is applied to page modifications [percentOfPartTime=0,92, markDirty=9905
pages/sec, checkpointWrite=6983 pages/sec, estIdealMarkDirty=41447 pages/sec, curDirty=0,07, maxDirty=0,
26, avgParkTime=741864 ns, pages: (total=169883, evicted=0, written=112286, synced=0, cpBufUsed=15936,
cpBufTotal=241312)]
```

   The most relevant part of this message is **percentOfPartTime** metric. In the example it's 0.92 - writing threads are stuck in LockSupport.parkNanos() for 92% of the time, which means very heavy throttling.
   Each message appears in the log when **percentOfPartTime** reaches 20% border.

CRC validation

For each page, a control checksum (CRC32) is calculated. The CRC calculation is performed before each page is written to the page store. When a page is read from the page store, CRC is again calculated against the read data and validated against the CRC field also stored in the page.

If CRC validation fails, Ignite logs the failure and attempts to recover the page from WAL. If recovery succeeds, the node keeps running. If recovery fails, then node shuts down itself.

WAL

We can't control a moment when a node crashes. Let's suppose we have saved tree leafs but didn't save tree root (during pages allocation they may be reordered because allocation is multithread). In this case, all updates can be lost.

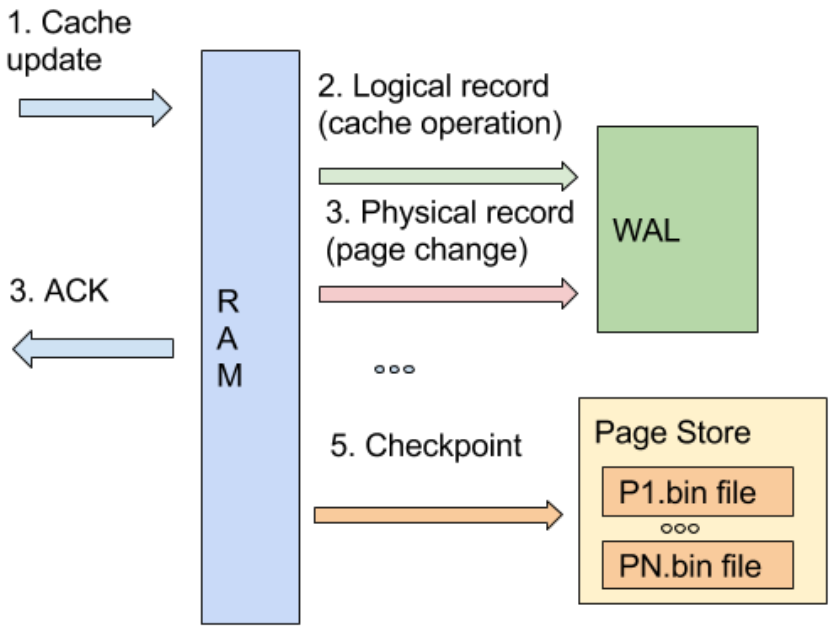
In the same time, we can't translate each memory page update to disk write operation each time - it is too slow.

A technique to solve this named write-ahead logging: Before doing an actual update, we append planned change information into a cyclic file named WAL log. WAL write operation named as WAL append/WAL log.

After the crash, we can read and replay WAL using already saved page set. We can restore to state, which was the last committed state of the crashed process. Restore operation based on both: pages store and WAL.

Practically we can't replay WAL from the beginning of times, Volume(HDD)<Volume(full WAL). And we need a procedure to throw out oldest part of changes in WAL, and this is done during checkpointing.

Consistent state comes only from a pair of WAL records and page store data.



Operation is acknowleged after operation was logged, and page(s) update was logged. Checkpoint will be started later by its triggers.

Crash Recovery

Local Crash Recovery

Crash Recovery can be

- Local (most DB are able to do this)
- and distributed (whole cluster state is restored).

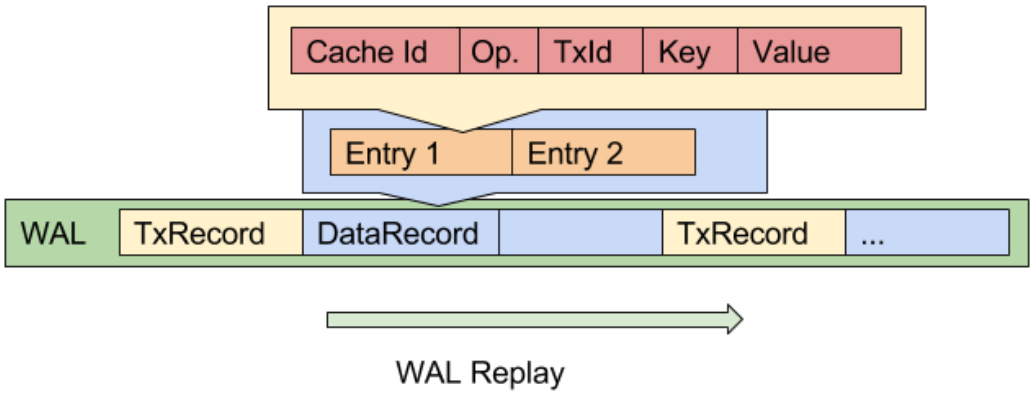
WAL records for recovery

Crash recovery involves following records writtent in WAL, it may be of 2 main types Logical & Physical

Logical records

a. Operation description - which operation we want to do. Contains operation type (create, update, delete) and (Key, Value, Version) - *Data Record*
b. Transactional record - this record is marker of begin prepare, prepared, and committed, or rollback transactions - (*TxRecord*)
c. Checkpoint record - marker of begin checkpointing (*CheckpointRecord*)

Structure of data record:



Data record includes list of entries (entry operations). Each operation has cache ID, operation type, key and value. Operation type can be

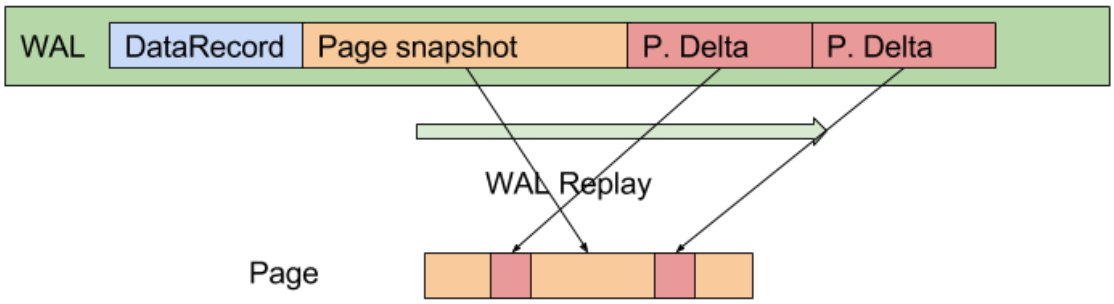
- CREATE - first put in cache, contains key and value.
- UPDATE - put in case for existing key, contains key and value.
- DELETE - remove key, contains key only, value is absent.

Update and create always contain full value value. In the same time several updates of the same key within transaction are merged into one latest update.

Physical records

1. a. Full page snapshot - record is issued for first page update after successfull checkpointing. Record is logged when page state changes from 'clean' to 'dirty' state (*PageSnapshot*)
   b. Delta record - describes memory region change, page change. Subclass of *PageDeltaRecord*. Contains bytes changed in the page. e.g bytes 5-10 were changed to [...,]. Relatively small records for B+tree records

Page snapshots and related deltas are combined during WAL replay.



For particular cache entry update we log records in following order:

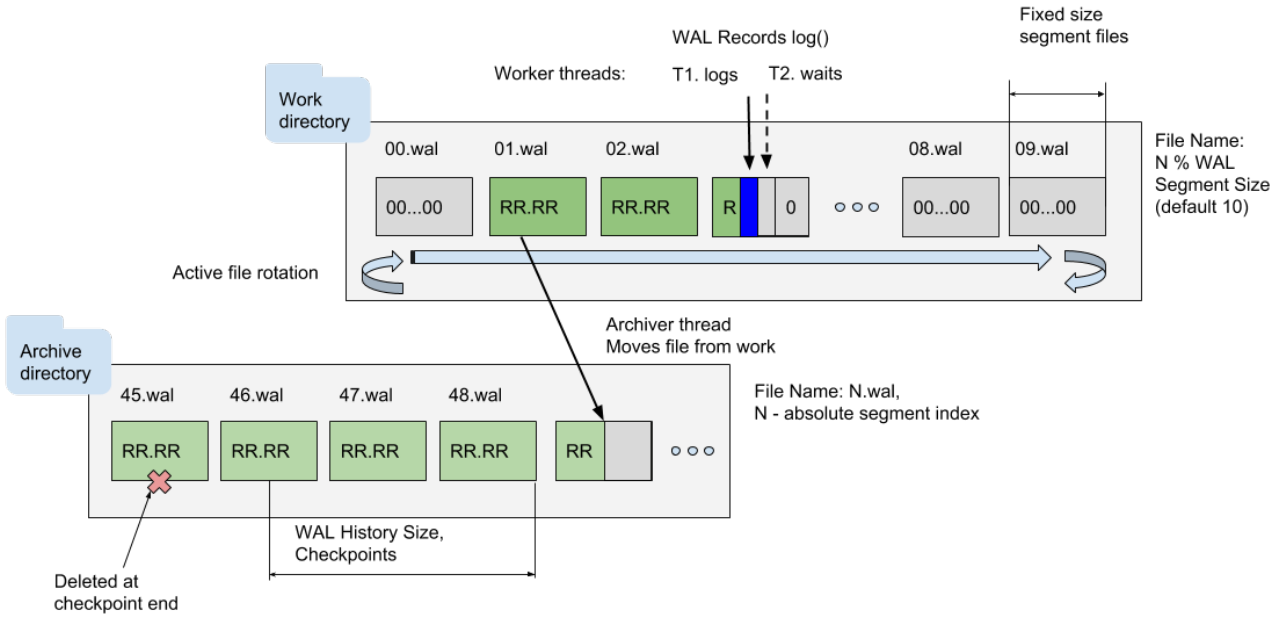
1. logical record with change planned - *DataRecord* with several *DataEntry* (ies)
2. page record:
   a. option: page changed by this update was initially clean, full page is loged - *PageSnapshot*,
   b. option: page was already modified, delta record is issued - *PageDeltaRecord*

Planned future optimisation - refer data modified from *PageDeltaRecord* to logical record. Will allow to not store byte updates twice. There is file WAL pointer, pointer to record from the beginning of time. This refreence may be used.

WAL structure

WAL consist of segments (files). The part of segments creates a work directory and files there are cyclically overwritten. Another part is archive - it is sequentially enumerated files, old files are deleted.

WAL file segments and rotation structure is shown at the picture below:



A number of segments may be not needed anymore (depending on History Size setting). Old fashion WAL History size setting is set in checkpoints number (See also WAL history size section below), the new one is set in bytes. History size setting is mentioned here https://apacheignite.readme.io/docs/write-ahead-log#section-wal-archive

Local Recovery Process

Let's assume node start process is running with existent files.

1. We need to check if page store is consistent.
2. Or we need to find out if crash was while Checkpoint (CP) was running

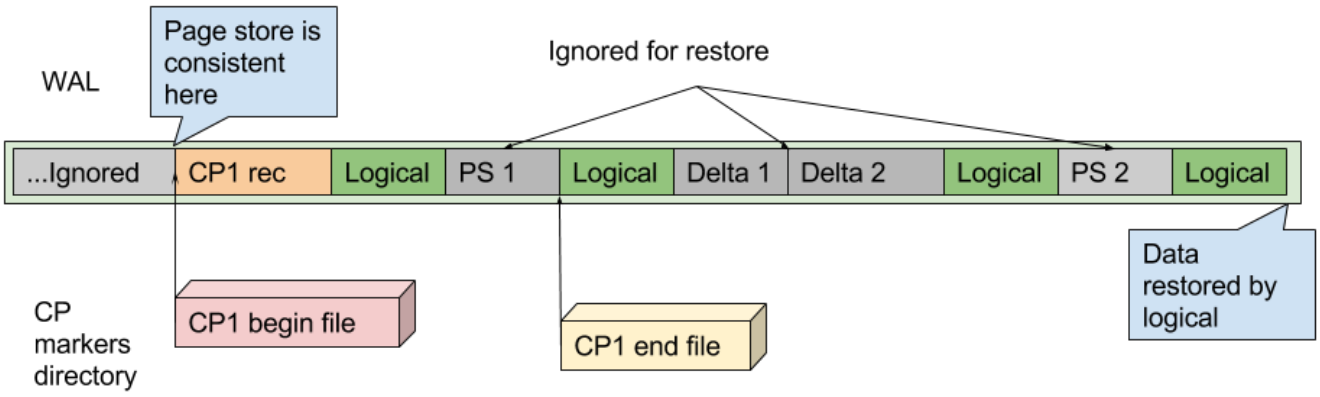
Ignite manages 2 types of CP markers on disk (standalone files, includes timestamp and WAL pointer):

- CP begin
- CP end

If we observe only CP begin and there is no CP end marker that means CP not finished; we have not consistent page store.

No checkpoint process

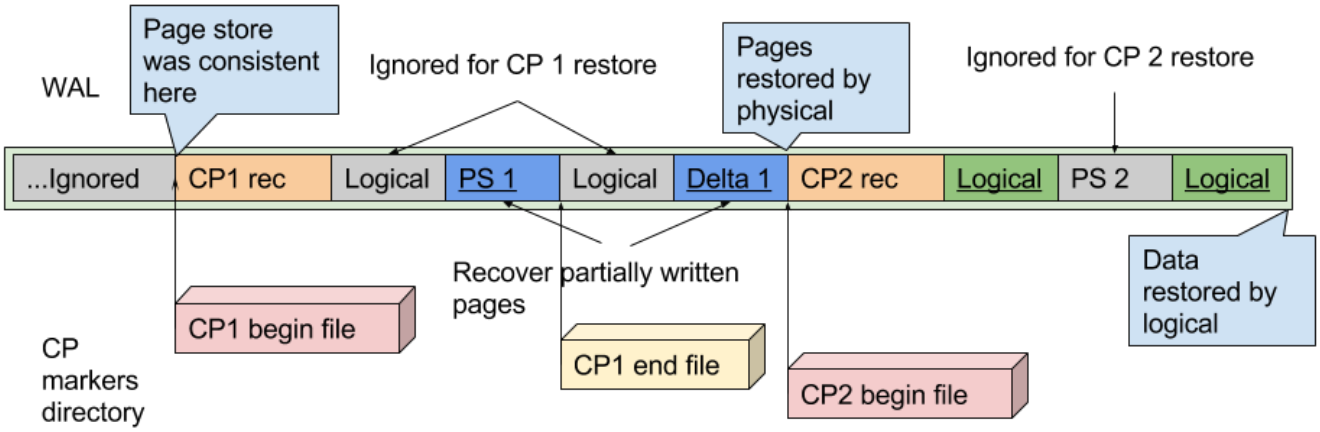
For crash at the moment when there was no checkpoint process running restore is trivial, only logical record are applied.



Physical records (page snapshots and delta records) are ignored because page store is consistent.

Middle of checkpoint

Let's suppose crash occurred at the middle of checkpoint. In that case restore process will discover markers for 1 CP1 and 2 start and CP 1 end.



Restore is split to 2 stages:

1st stage: Starting from previous completed checkpoint record CP1 till record of CP2 start (incompleted) we apply all physical records and ignore logical.

- In example CP2 was started but not finished. At the time of crash CP2 process was writting pages changed before CP2 was started: in this example: page(s) corresponding to PS1 & Delta 1 physical record.
- Write of page content to page store may not incompleted (e.g. broken at the middle of page). To restore potentially corrupted page store we apply page change records. As result of updating same pages at the same place in page store, we got consistent page store.
- Page Snapshot records required to avoid double apply of data from delta records.

2nd stage: Starting from marker of incomplete CP2 we apply only logical records until end of WAL is reached.

When replay is finished CP2 end marker will be added.

If transaction begin record has no corresponding end, tx change is not applied.

Summary, limitations and performance

Limitations

Because CP are consistent we can't start next CP until previous is not completed.

There is possible next situation:

- updates coming fast from worker threads

- CP pool (for copy on writes) may become full with new changes originated

For that case we will block new updates and wait running for CP to finish.

To avoid such scenario:

- increase frequency of checkpoints (to minimize amount of data to be saved in each CP)
- increase CP buffer size

WAL and page store may be saved to different devices to avoid its mutual influence.

Case if same records are updated many times may generate load to WAL and no significant load to page store.

To provide recovery guarantees each write (log()) to WAL should:

- call write() itself.
- but also require fsync (force buffers to be flushed by OS to the real device).

fsync is expensive operation. There is optimisation for case updates coming faster than disk write, fsyncDelayNanos (1ns-1ms, 1ns by default) delay is used. This delay is used to park threads to accumulate more than one fsync requests.

Future optimisation: standalone thread will be responsible to write data to disk. Worker threads will do all preparation and transfer buffer to write.

See also WAL history size section below.

WAL mode

There several levels of guarantees (*WALMode*)

	Implementation	Warranties
FSYNC	fsync() on each commit	Any crashes (OS and process crash)
LOG_ONY	write() on commit Synchronisation is responsibility of OS	Kill process, but no OS fail
BACKGROUND	do nothing on commit (records are accumulated in memory) write() on timeout	kill -9 may cause loss of several latest updates
NONE	WAL is disabled	data is persisted only in case of graceful cluster shutdown (Ignite.cluster().active(false))

But there is several nodes containing same data and there is possible to restore data from other nodes.

Distributed Recovery

Partition update counter. This mechanism was already used in continuous queries.

- Partition update counter is associated with partition
- Each update causes increment of partition update counter.

Each update (counter) is replicated to backup. If counter equal on primary and backup means replication is finished.

Partition update counter is saved with update recods in WAL.

Node Join (with data from persitence)

Consider partition on joining node was is owning state, update counter = 50. Existing nodes has update counter = 150

Node join causes partition map exchange, update counter is sent with other partition data. (Joining node will have new ID and from the point of view of dicsovery this node is a new node.)

Coordinator observes older partition state and forces partition to moving state. Moving force is required to setup uploading newer data.

Rebalance of fresh data to joined node now may be run in 2 modes:

- There is WAL on primary node. WAL includes checkpoint marker with partition update cntr = 45.
    - We can send only WAL logical update records to backup
- If counter in WAL is too big, e.g. 200, we don't have delta (can't sent WAL recods)
    - joined node will have to clear partition data.
    - Partition state is set to renting state
    - When clean up finished partition goes to moving state.
    - We can't use delta updates because there is possible problem with keys deleted early. Can get stale key if we send only delta of changes.

Possible future optimisation: for full update we may send page store file over network.

Order of nodes join is not relevant, there is possible situation that oldest node has older partition state, but joining node has higher partition counter. In this case rebalancing will be triggered by coordinator. Rebalancing will be performed from the newly joined node to existing one (note this behaviour may be changed under IEP-4 Baseline topology for caches)

Advanced Configuration

WAL History Size

In corner case we need to store WAL only for 1 checkpoint in past for successful recovery (*DataStorageConfiguration#walHistSize*)

We can't delete WAL segments considering only history size in bytes or segments. It is possible to replay WAL only starting from checkpoint marker.

> WAL history size is measured in number of checkpoint.

Assuming that checkpoints are triggered mostly by timeout we can estimate possible downtime after which node may be rebalanced using delta logical WAL records.

By default WAL history size is 20 to increase probability that rebalancing can be done using logical deltas from WAL.

Estimating disk space

WAL Work maximum used size: walSegmentSize * walSegments = 640Mb (default)

in case Default WAL mode - this size is used always,

in case other modes best case is 1 segment * walSegmentSize

WAL Work+WAL Archive max size may be estimated by

1. average load or
2. by maximum size.

1st way is applicable if checkpoints are triggered mostly by timer trigger.
Wal size = 2*Average load(bytes/sec) * trigger interval (sec) * walHistSize (number of checkpoints)
Where 2 multiplier coming from physical & logical WAL Records.

2nd way: Checkpoint is triggered by segments max dirty pages percent. Use persisted data regions max sizes:
sum(Max configured DataRegionConfiguration.maxSize) * 75% - est. maximum data volume to be writen on 1 checkpoint.
Overall WAL size (before archiving) = 2* est. data volume * walHistSize = 1,5 * sum(DataRegionConfiguration.maxSize) * walHistSize

Note applying WAL compressor may significiantly reduce archive size.

WAL compaction

If WAL archive occupies too much space, there's option to enable background WAL compression (*DataStorageConfiguration#walCompactionEnabled*).

If it's enabled, WAL archive segments that are older than 1 checkpoint in past (they are no longer needed for crash recovery) will be filtered from physical records and compressed to ZIP format. In case of demand (e.g. delta-rebalancing in case of topology change), they will be uncompressed back to RAW format. Experiments show that factor between compacted and RAW segments is **10x** on usual data and **3x** in worst case (secure random data in updates). Please note that as long as ZIP segments don't contain any data needed for crash recovery, they can be deleted anytime in case of need for disk space (it will affect rebalancing though).

Setting input-output

I/O abstraction determines how disk features are accessed by native persistence.

Random Access File I/O

This type of I/O implementation opeate with files using standard Java inferface. java.nio.channels.FileChannel is used.

This implementation was used by default before 2.4.

To switch to this implementation it is required to set factory in config (*DataStorageConfiguration#setFileIOFactory*) or change using system property: IgniteSystemProperties.IGNITE_USE_ASYNC_FILE_IO_FACTORY = "false".

This type of IO is always used for WAL files.

Async I/O

This option is default since 2.4.

It was introduced to protect IO module and underlying files from close by interrupt problem.

To set this implementation it is possible to set factory in config (*DataStorageConfiguration#setFileIOFactory*) or change using system property: IgniteSystemProperties.IGNITE_USE_ASYNC_FILE_IO_FACTORY = "true".

Direct I/O

Introduction and Requirements

Since Ignite 2.4 there is plugin for enabling Direct I/O mode.

Direct I/O is a feature of the file system whereby file reads and writes go directly from the applications to the storage device, bypassing the operating system read and write caches (page cache). Direct I/O is useful for applications (such as databases) that manage their own caches, as Ignite does.

Direct I/O mode for file open enforces application to do block read/write operation, that means data having size less than block can't be written or loaded from disk.

Plugin works on Linux with the version of the kernel higher than 2.4.2. This plugin switches the input of the output for durable (page) memory to use the mode Direct IO for files. If incompatible OS or FS is used, plugin has no effect and fallbacks to regular I/O implementation. Durable memory page size should be not less than physical device block and Linux system page size. Durable memory page size should be divisible by underlying OS and FS blocks sizes. Usually both sizes are 4Kbytes, so using default page size is usually sufficient to enable plugin.

Configuration

There is no need to do additional configuration of plugin. It is sufficient to add ignite-direct-io.jar and Java Native Access (JNA) library jar (jna-xxx.jar) to classpath.

Plugin jar is available under optional libs folder:

apache-ignite-fabric-x.x.x-bin\libs\optional\ignite-direct-io\ignite-direct-io-x.x.x.jar

When plugin is applied following messages will be produced

```
Configured plugins:
^-- Ignite Native I/O Plugin [Direct I/O]
^-- Copyright(C) Apache Software Foundation
```

Following message will be produced for successful setup:

```
Page size configuration for storage path [/work/db/node00-3a1415b8-aa54-4a63-a40a-c75ad48dd6b8]: 4096; Linux
memory page size: 4096; Selected FS block size : 4096.
Selected FS block size : 4096
Direct IO is enabled for block IO operations on aligned memory structures. [block size = 4096, durable memory
page size = 4096]
```

However, disabling plugin's function is possible through system Property. To disable Direct IO set IgniteSystemProperties#IGNITE_DIRECT_IO_ENABLED to false.

Enabled direct input-output mode allows Ignite to bypass the system cache pages Linux is fully conveys the management of pages to Ignite.

WAL and Native IO

Write Ahead log does not have blocks(chunks/pages) as Durable Memory Page store has. So Direct IO currently can't be enabled for WAL logging. WAL always goes through the conventional system I/O calls.

However, when the ignite-direct-io plugin was configured successfully, WAL logging still obtains benefit.

ignite-direct-io plugin allows WAL manager to advice Linux system, that file is not needed (Native call posix_fadvise, with flag POSIX_FADV_DONTNEED).

This advice gives only recommendation to Linux system, "do not store the file data in the page cache, as data is not required". This results WAL data is still going to page cache first, but according this advice Linux will flush and remove these data during next page cache scan.

Direct I/O & Performance

Direct I/O can bring possible negative effects to performance of reading pages. In Direct I/O mode all pages are read bypassing Linux cache directly from the disk.

Compared with partially filled page cache direct disk access can slow down

- the scenarios of the initial data load as well as
- loading data pages previously replaced to disk (rotation of pages)

This effect is taking place because using a conventional input-output without flag Direct I/O has a chance to work faster if

- required pages remain in the Linux system cache. RAM access greatly speeds up access to them.
- the Linux kernel makes the so-called pre-fetching when reading a file. Pre-fetching pages allows to read more data, than actually was requested by application. Currently pre-fetching is not implemented in plugin

Direct I/O still can be used for production in case of accurate configuration. But currently the Direct I/O mode is not enabled by default and provided as plugin.

Benefit of using Direct I/O is more predictable time of fdatasync (fsync) operation. As all data is not accumulated in RAM and goes directly to disk, each fsync of page store requires less time, than fsync'ing all data from memory. Direct I/O does not guarantee fsync(), immediately after write, so checkpoint writers still calls fsync at the end of checkpoint.