Working with filesystem in time series database

FOSDEM 2026

Aliaksandr Valialkin, CTO @ VictoriaMetrics

Let’s meet

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)
    - VictoriaMetrics - database for metrics

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)
    - VictoriaMetrics - database for metrics
    - VictoriaLogs - database for logs

Let’s meet

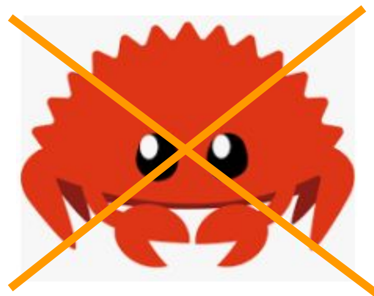
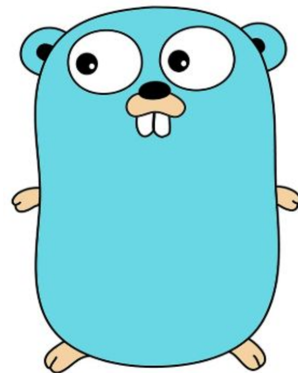
- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)
  - VictoriaMetrics - database for metrics
  - VictoriaLogs - database for logs
- These databases are fast and cost-efficient

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)
    - VictoriaMetrics - database for metrics
    - VictoriaLogs - database for logs
- These databases are fast and cost-efficient
    - They aren’t written in Rust

Let’s meet

- I’m Aliaksandr Valialkin (nickname valyala - google it to know more about me)
- I’m fond of writing code optimized for performance and low resource usage
- I created specialized time series databases (open-source)
  - VictoriaMetrics - database for metrics
  - VictoriaLogs - database for logs
- These databases are fast and cost-efficient
  - They aren’t written in Rust
  - The are written in Go

Specifics of time series databases

Specifics of time series databases

- A time series contains (timestamp, value) samples

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels
- The number of time series in typical cases is from 10K up to billions

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels
- The number of time series in typical cases is from 10K up to a billions
- Typical data ingestion rate: 100K … 10M samples/sec (needs high performance during data ingestion)

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels
- The number of time series in typical cases is from 10K up to a billions
- Typical data ingestion rate: 100K … 10M samples/sec (needs high performance during data ingestion)
- Typical query needs to scan and process 1M - 100M samples (needs high performance during querying)

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels
- The number of time series in typical cases is from 10K up to a billions
- Typical data ingestion rate: 100K … 10M samples/sec (needs high performance during data ingestion)
- Typical query needs to scan and process 1M - 100M samples (needs high performance during querying)
- The number of samples in typical cases is trillions

Specifics of time series databases

- A time series contains (timestamp, value) samples
- Values can be numeric (aka metrics), plaintext of structured blob (aka logs or events)
- Every time series may have arbitrary number of {key=”value”} labels
- The number of time series in typical cases is from 10K up to a billions
- Typical data ingestion rate: 100K … 10M samples/sec (needs high performance during data ingestion)
- Typical query needs to scan and process 1M - 100M samples (needs high performance during querying)
- The number of samples in typical cases is trillions
- Typical data size - from terabytes to petabytes (doesn’t fit RAM)

How to achieve high performance at databases?

How to achieve high performance at databases?

- Database performance is limited by disk IO

How to achieve high performance at databases?

- Database performance is limited by disk IO
    - Especially when the database size doesn’t fit RAM

How to achieve high performance at databases?

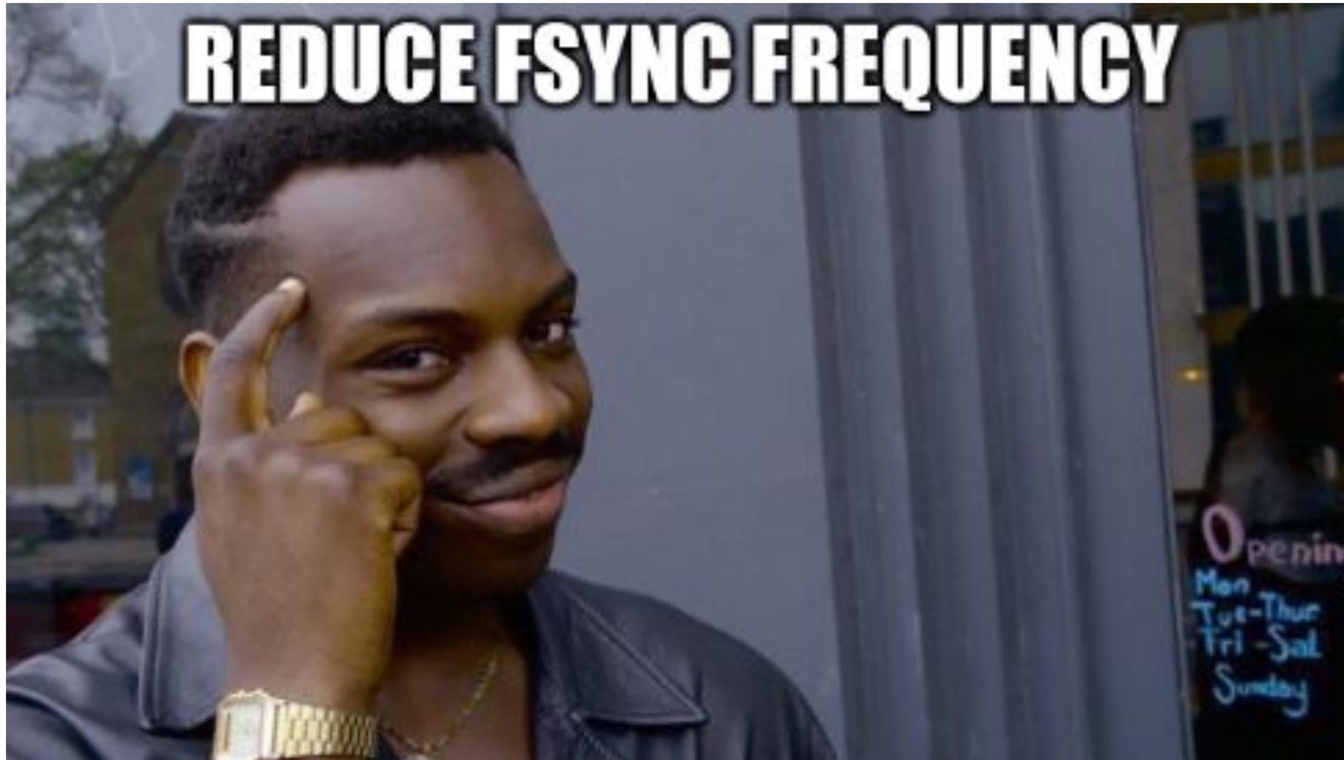
- Database performance is limited by disk IO
  - Especially when the database size doesn’t fit RAM
- Reduce the number of disk reads and writes

How to achieve high performance at databases?

- Database performance is limited by disk IO
  - Especially when the database size doesn’t fit RAM
- Reduce the number of disk reads and writes
- Reduce the amounts of data written and read from disk

How to achieve high data ingestion rate?

How to achieve high data ingestion rate?



Why fsync?

Why fsync?

- When you write data to files, it isn't stored to disk

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined

Why fsync?

- When you write data to files, it isn't stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined
- The page cache data can be partially stored to disk on power off

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined
- The page cache data can be partially stored to disk on power off
    - Data loss

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined
- The page cache data can be partially stored to disk on power off
    - Data loss
    - Data corruption

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined
- The page cache data can be partially stored to disk on power off
  - Data loss
  - Data corruption
- Fsync is a system call which forces storing the data from the page cache to disk

Why fsync?

- When you write data to files, it isn’t stored to disk
- It is stored to the OS page cache (RAM)
- The OS stores data from the page cache to disk in background
- The order of storing the data from page cache to disk is undefined
- The page cache data can be partially stored to disk on power off
  - Data loss
  - Data corruption
- Fsync is a system call which forces storing the data from the page cache to disk
- All the production-ready databases **must use fsync**

Why reducing fsync frequency?

Why reducing fsync frequency?

- fsync is slow

Why reducing fsync frequency?

- fsync is slow

Why reducing fsync frequency?

- fsync is slow
  - Including SSDs, since they store data in 512KB-4MB blocks (erase blocks), even if you store a single byte

Why reducing fsync frequency?

- fsync is slow
    - Including SSDs, since they store data in 512KB-4MB blocks (erase blocks), even if you store a single byte
    - Expect a few thousand fsyncs per second

Why reducing fsync frequency?

- fsync is slow
  - Including SSDs, since they store data in 512KB-4MB blocks (erase blocks), even if you store a single byte
  - Expect a few thousand fsyncs per second
  - There are exceptions - enterprise-grade SSDs with capacitors preventing from the loss of recently written (buffered) data on power off

Why reducing fsync frequency?

- fsync is slow
  - Including SSDs, since they store data in 512KB-4MB blocks (erase blocks), even if you store a single byte
  - Expect a few thousand fsyncs per second
  - There are exceptions - enterprise-grade SSDs with capacitors preventing from the loss of recently written (buffered) data on power off
- Bypassing OS page cache and writing the data to disk directly (aka direct IO) doesn’t help too much

Why reducing fsync frequency?

- fsync is slow
    - Including SSDs, since they store data in 512KB-4MB blocks (erase blocks), even if you store a single byte
    - Expect a few thousand fsyncs per second
    - There are exceptions - enterprise-grade SSDs with capacitors preventing from the loss of recently written (buffered) data on power off
- Bypassing OS page cache and writing the data to disk directly (aka direct IO) doesn’t help too much
    - Every direct write to disk is equivalent to an fsync (i.e. it takes the same amounts of resources and time)

How to reduce fsync frequency at databases?

How to reduce fsync frequency at databases?

- Send data to the database in big batches

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client
  - Bigger batches -> less frequent fsyncs

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client
  - Bigger batches -> less frequent fsyncs
- Merge concurrent inserts into a single data block and store it with a single fsync

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client
  - Bigger batches -> less frequent fsyncs
- Merge concurrent inserts into a single data block and store it with a single fsync
  - Run fsync every 10ms (100 fsyncs per second), so every insert is delayed for up to 10ms while building a single data block from concurrent inserts

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client
  - Bigger batches -> less frequent fsyncs
- Merge concurrent inserts into a single data block and store it with a single fsync
  - Run fsync every 10ms (100 fsyncs per second), so every insert is delayed for up to 10ms while building a single data block from concurrent inserts
  - Return “success” to clients when their data is fsync’ed (probably together with the data from concurrent inserts)

How to reduce fsync frequency at databases?

- Send data to the database in big batches
  - Every batch requires at least a single fsync for data persistence before returning “success” to the client
  - Bigger batches -> less frequent fsyncs
- Merge concurrent inserts into a single data block and store it with a single fsync
  - Run fsync every 10ms (100 fsyncs per second), so every insert is delayed for up to 10ms while building a single data block from concurrent inserts
  - Return “success” to clients when their data is fsync’ed (probably together with the data from concurrent inserts)
  - The drawback - a single client is limited to 100 sequential inserts per second

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
    - Store data in WAL, fsync it, and then store the data to the main storage without fsync
        - Replay WAL after power off
    - Many databases use this technique
    - It doesn’t reduce fsync frequency too much
        - Every write to WAL must be persisted in order to prevent from data loss on power off
        - So, every write to WAL must be finished with fsync

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off
    - So, every write to WAL must be finished with fsync
  - Databases use various tricks for reducing fsyncs at WAL

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off
    - So, every write to WAL must be finished with fsync
  - Databases use various tricks for reducing fsyncs at WAL
    - To buffer data in RAM and then periodically write it in blocks to WAL

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off
    - So, every write to WAL must be finished with fsync
  - Databases use various tricks for reducing fsyncs at WAL
    - To buffer data in RAM and then periodically write it in blocks to WAL
    - To write data to WAL without fsync, and to periodically run fsync on WAL

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn’t reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off
    - So, every write to WAL must be finished with fsync
  - Databases use various tricks for reducing fsyncs at WAL
    - To buffer data in RAM and then periodically write it in blocks to WAL
    - To write data to WAL without fsync, and to periodically run fsync on WAL
    - Both tricks may lead to data loss on power off

How to reduce fsync frequency at databases?

- Store data in write-ahead log (WAL)
  - Store data in WAL, fsync it, and then store the data to the main storage without fsync
    - Replay WAL after power off
  - Many databases use this technique
  - It doesn't reduce fsync frequency too much
    - Every write to WAL must be persisted in order to prevent from data loss on power off
    - So, every write to WAL must be finished with fsync
  - Databases use various tricks for reducing fsyncs at WAL
    - To buffer data in RAM and then periodically write it in blocks to WAL
    - To write data to WAL without fsync, and to periodically run fsync on WAL
    - Both tricks may lead to data loss on power off
  - Data corruption is possible when power off happens in the middle of fsync and the WAL has no protection against partial writes

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off
  - There are practical cases where it is OK to lose recently ingested data on power off

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off
    - There are practical cases where it is OK to lose recently ingested data on power off
        - Observability data - metrics, logs and events

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off
    - There are practical cases where it is OK to lose recently ingested data on power off
        - Observability data - metrics, logs and events
    - Buffer data in memory and periodically flush it to disk with an fsync at the end

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off
  - There are practical cases where it is OK to lose recently ingested data on power off
    - Observability data - metrics, logs and events
  - Buffer data in memory and periodically flush it to disk with an fsync at the end
  - After the data is flushed to disk, it is safe from loss and corruption on power off

How to reduce fsync frequency at databases?

- Admit that the recently ingested data can be lost on power off
  - There are practical cases where it is OK to lose recently ingested data on power off
    - Observability data - metrics, logs and events
  - Buffer data in memory and periodically flush it to disk with an fsync at the end
  - After the data is flushed to disk, it is safe from loss and corruption on power off
  - The recently ingested buffered data is lost on power off

The essence of all the databases

The essence of all the databases

- To find quickly the needed data by the given key (prefix)

The essence of all the databases

- To find quickly the needed data by the given key (prefix)
- Databases use indexes for this task

The essence of all the databases

- To find quickly the needed data by the given key (prefix)
- Databases use indexes for this task
- Indexes are data structures, which map keys to the locations of the data (on disk)

The essence of all the databases

- To find quickly the needed data by the given key (prefix)
- Databases use indexes for this task
- Indexes are data structures, which map keys to the locations of the data (on disk)
- Indexes usually have O(log(N)) seek complexity, where N is the number of entries in the index

The most frequently used index types

The most frequently used index types

- B-tree

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
  - Adding a new entry in b-tree may require rewriting multiple nodes on disk

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
  - Adding a new entry in b-tree may require rewriting multiple nodes on disk
  - Entries usually contain pointers to row values, which are stored in other files

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
  - Adding a new entry in b-tree may require rewriting multiple nodes on disk
  - Entries usually contain pointers to row values, which are stored in other files
- LSM (log-structured merge) tree

The most frequently used index types

- B-tree
    - Tree with nodes containing thousands of entries per node
    - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
    - Adding a new entry in b-tree may require rewriting multiple nodes on disk
    - Entries usually contain pointers to row values, which are stored in other files
- LSM (log-structured merge) tree
    - Sorted entries are stored in files

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
  - Adding a new entry in b-tree may require rewriting multiple nodes on disk
  - Entries usually contain pointers to row values, which are stored in other files
- LSM (log-structured merge) tree
  - Sorted entries are stored in files
  - Smaller files are periodically merged into bigger files in background

The most frequently used index types

- B-tree
  - Tree with nodes containing thousands of entries per node
  - Nodes are usually aligned to some fixed size (4KB - 64KB) and are stored to disk at once
  - Adding a new entry in b-tree may require rewriting multiple nodes on disk
  - Entries usually contain pointers to row values, which are stored in other files
- LSM (log-structured merge) tree
  - Sorted entries are stored in files
  - Smaller files are periodically merged into bigger files in background
  - Entries usually contain row values instead of pointers to rows

B-tree vs LSM tree: practical cases

B-tree vs LSM tree: practical cases

- High data ingestion rate

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO
  - LSM: multiple items are buffered in memory and then stored in a file with a single fsync

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO
  - LSM: multiple items are buffered in memory and then stored in a file with a single fsync
    - No WAL -> no additional disk IO

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO
  - LSM: multiple items are buffered in memory and then stored in a file with a single fsync
    - No WAL -> no additional disk IO
- Reading values in the index order (aka range scans by key prefix)

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO
  - LSM: multiple items are buffered in memory and then stored in a file with a single fsync
    - No WAL -> no additional disk IO
- Reading values in the index order (aka range scans by key prefix)
  - B-tree: the actual row values are scattered across disk, so every row requires a dedicated disk read operation

B-tree vs LSM tree: practical cases

- High data ingestion rate
  - B-tree: every item potentially updates multiple nodes on disk and requires multiple fsyncs in the worst case
    - WAL eats additional disk IO
  - LSM: multiple items are buffered in memory and then stored in a file with a single fsync
    - No WAL -> no additional disk IO
- Reading values in the index order (aka range scans by key prefix)
  - B-tree: the actual row values are scattered across disk, so every row requires a dedicated disk read operation
  - LSM: sorted row values are read linearly from disk in a much smaller number of disk read operations

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM
  - B-tree: requires additional disk read/write operations per each ingested item for index update

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM
    - B-tree: requires additional disk read/write operations per each ingested item for index update
        - Some b-tree nodes may be missing in RAM -> disk read operation

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM
  - B-tree: requires additional disk read/write operations per each ingested item for index update
    - Some b-tree nodes may be missing in RAM -> disk read operation
    - Updated b-tree nodes must be stored to disk before eviction -> disk write operation

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM
    - B-tree: requires additional disk read/write operations per each ingested item for index update
        - Some b-tree nodes may be missing in RAM -> disk read operation
        - Updated b-tree nodes must be stored to disk before eviction -> disk write operation
    - LSM: multiple items are buffered in RAM, sorted and saved to disk in one go (a few sequential disk write operations)

B-tree vs LSM tree: practical cases

- Ingesting big data volumes which do not fit RAM
  - B-tree: requires additional disk read/write operations per each ingested item for index update
    - Some b-tree nodes may be missing in RAM -> disk read operation
    - Updated b-tree nodes must be stored to disk before eviction -> disk write operation
  - LSM: multiple items are buffered in RAM, sorted and saved to disk in one go (a few sequential disk write operations)
    - Data ingestion performance doesn’t depend on the database size

B-tree vs LSM tree: practical cases

- Snapshots

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable
    - Just make hard links for all the LSM files - and the snapshot is done!

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable
    - Just make hard links for all the LSM files - and the snapshot is done!
- Network-attached storage (NFS, object storage)

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable
    - Just make hard links for all the LSM files - and the snapshot is done!
- Network-attached storage (NFS, object storage)
  - B-tree: doesn’t work because you need to update files (objects)

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable
    - Just make hard links for all the LSM files - and the snapshot is done!
- Network-attached storage (NFS, object storage)
  - B-tree: doesn't work because you need to update files (objects)
    - This is slow and inefficient

B-tree vs LSM tree: practical cases

- Snapshots
  - B-tree: complicated schemes with MVCC and transaction ids
  - LSM: instant for any database sizes
    - LSM tree files are immutable
    - Just make hard links for all the LSM files - and the snapshot is done!
- Network-attached storage (NFS, object storage)
  - B-tree: doesn’t work because you need to update files (objects)
    - This is slow and inefficient
  - LSM: works perfectly because files are immutable

B-tree vs LSM tree: practical cases

- Disk space usage

B-tree vs LSM tree: practical cases

- Disk space usage
    - B-tree: index isn’t compressible, the row values have bad compression ratio because they are stored in “random” order => bigger disk space usage

B-tree vs LSM tree: practical cases

- Disk space usage
  - B-tree: index isn't compressible, the row values have bad compression ratio because they are stored in "random" order => bigger disk space usage
  - LSM: sorted row values are usually compressed at higher compression ratio => lower disk space usage

B-tree vs LSM tree: practical cases

- Disk space usage
  - B-tree: index isn’t compressible, the row values have bad compression ratio because they are stored in “random” order => bigger disk space usage
  - LSM: sorted row values are usually compressed at higher compression ratio => lower disk space usage
- Disk read bandwidth usage

B-tree vs LSM tree: practical cases

- Disk space usage
  - B-tree: index isn’t compressible, the row values have bad compression ratio because they are stored in “random” order => bigger disk space usage
  - LSM: sorted row values are usually compressed at higher compression ratio => lower disk space usage
- Disk read bandwidth usage
  - B-Tree: more data on disk -> higher disk read bandwidth usage

B-tree vs LSM tree: practical cases

- Disk space usage
  - B-tree: index isn't compressible, the row values have bad compression ratio because they are stored in "random" order => bigger disk space usage
  - LSM: sorted row values are usually compressed at higher compression ratio => lower disk space usage
- Disk read bandwidth usage
  - B-Tree: more data on disk -> higher disk read bandwidth usage
  - LSM: less data on disk -> lower disk read bandwidth usage

B-tree vs LSM tree: practical cases

- Disk space usage
  - B-tree: index isn’t compressible, the row values have bad compression ratio because they are stored in “random” order => bigger disk space usage
  - LSM: sorted row values are usually compressed at higher compression ratio => lower disk space usage
- Disk read bandwidth usage
  - B-Tree: more data on disk -> higher disk read bandwidth usage
  - LSM: less data on disk -> lower disk read bandwidth usage
    - Higher performance for heavy queries which need to scan terabytes of data

Row-oriented vs column-oriented storage

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
    - Great for “SELECT * …” when selecting all the row fields
        - Rarely seen in the practice
    - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
    - Great for “SELECT * …” when selecting all the row fields
        - Rarely seen in the practice
    - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
        - Typical production queries
        - Needs to read data from disk for all the columns per each selected row
    - Low compression rate (data for adjacent rows on disk is “random”)

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
  - Big overhead for “SELECT * …”, since data across all the column files must be read

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
  - Big overhead for “SELECT * …”, since data across all the column files must be read
  - Small overhead for “SELECT a_few_column … WHERE filter_by_a_few_columns”

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
    - Great for “SELECT * …” when selecting all the row fields
        - Rarely seen in the practice
    - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
        - Typical production queries
        - Needs to read data from disk for all the columns per each selected row
    - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
    - Big overhead for “SELECT * …”, since data across all the column files must be read
    - Small overhead for “SELECT a_few_column … WHERE filter_by_a_few_columns”
        - Reads data only for the columns mentioned in SELECT and WHERE clauses

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
  - Big overhead for “SELECT * …”, since data across all the column files must be read
  - Small overhead for “SELECT a_few_column … WHERE filter_by_a_few_columns”
    - Reads data only for the columns mentioned in SELECT and WHERE clauses
  - Per-column data usually compresses very well

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
  - Big overhead for “SELECT * …”, since data across all the column files must be read
  - Small overhead for “SELECT a_few_column … WHERE filter_by_a_few_columns”
    - Reads data only for the columns mentioned in SELECT and WHERE clauses
  - Per-column data usually compresses very well
    - When the per-column values are constant or contain a small number of unique values

Row-oriented vs column-oriented storage

- Row-oriented storage - store rows on disk one after the other
  - Great for “SELECT * …” when selecting all the row fields
    - Rarely seen in the practice
  - Big overhead for “SELECT a_few_columns … WHERE filter_by_a_few_columns”
    - Typical production queries
    - Needs to read data from disk for all the columns per each selected row
  - Low compression rate (data for adjacent rows on disk is “random”)
- Column-oriented storage - store data per each column in a separate file
  - Big overhead for “SELECT * …”, since data across all the column files must be read
  - Small overhead for “SELECT a_few_column … WHERE filter_by_a_few_columns”
    - Reads data only for the columns mentioned in SELECT and WHERE clauses
  - Per-column data usually compresses very well
    - When the per-column values are constant or contain a small number of unique values
    - Smaller disk space usage and faster query performance, since less data needs to be read from disk

Column-oriented storage properties

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
    - Not so good compression ratio

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
  - Not so good compression ratio
    - Bigger disk space usage

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
    - Not so good compression ratio
        - Bigger disk space usage
        - Slower queries

Column-oriented storage properties

- Doesn't work well with b-trees when rows' data isn't sorted by index
    - Not so good compression ratio
        - Bigger disk space usage
        - Slower queries
- Works great with LSM trees

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
  - Not so good compression ratio
    - Bigger disk space usage
    - Slower queries
- Works great with LSM trees
  - Rows are physically sorted by index key

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
    - Not so good compression ratio
        - Bigger disk space usage
        - Slower queries
- Works great with LSM trees
    - Rows are physically sorted by index key
    - Sorted column values compress much better than the values in “random” order

Column-oriented storage properties

- Doesn't work well with b-trees when rows' data isn't sorted by index
  - Not so good compression ratio
    - Bigger disk space usage
    - Slower queries
- Works great with LSM trees
  - Rows are physically sorted by index key
  - Sorted column values compress much better than the values in “random” order
    - Smaller disk space usage

Column-oriented storage properties

- Doesn’t work well with b-trees when rows’ data isn’t sorted by index
  - Not so good compression ratio
    - Bigger disk space usage
    - Slower queries
- Works great with LSM trees
  - Rows are physically sorted by index key
  - Sorted column values compress much better than the values in “random” order
    - Smaller disk space usage
    - Faster queries

Conculsions

Conculsions

- Fsync is slow

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)
- LSM tree + column-oriented storage => the best performance for big databases which do not fit RAM

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)
- LSM tree + column-oriented storage => the best performance for big databases which do not fit RAM
- Databases which use LSM tree + column-oriented storage:

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)
- LSM tree + column-oriented storage => the best performance for big databases which do not fit RAM
- Databases which use LSM tree + column-oriented storage:
    - ClickHouse

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)
- LSM tree + column-oriented storage => the best performance for big databases which do not fit RAM
- Databases which use LSM tree + column-oriented storage:
  - ClickHouse
  - VictoriaMetrics and VictoriaLogs

Conculsions

- Fsync is slow
- The best way to reduce fsync rate - to admit that the recently ingested data can be lost
- LSM tree uses less disk IO than B-tree
- Column-oriented storage uses less disk space and disk IO than row-oriented storage (including background merges)
- LSM tree + column-oriented storage => the best performance for big databases which do not fit RAM
- Databases which use LSM tree + column-oriented storage:
  - ClickHouse
  - VictoriaMetrics and VictoriaLogs
- Investigate the source code for these databases - they are open source