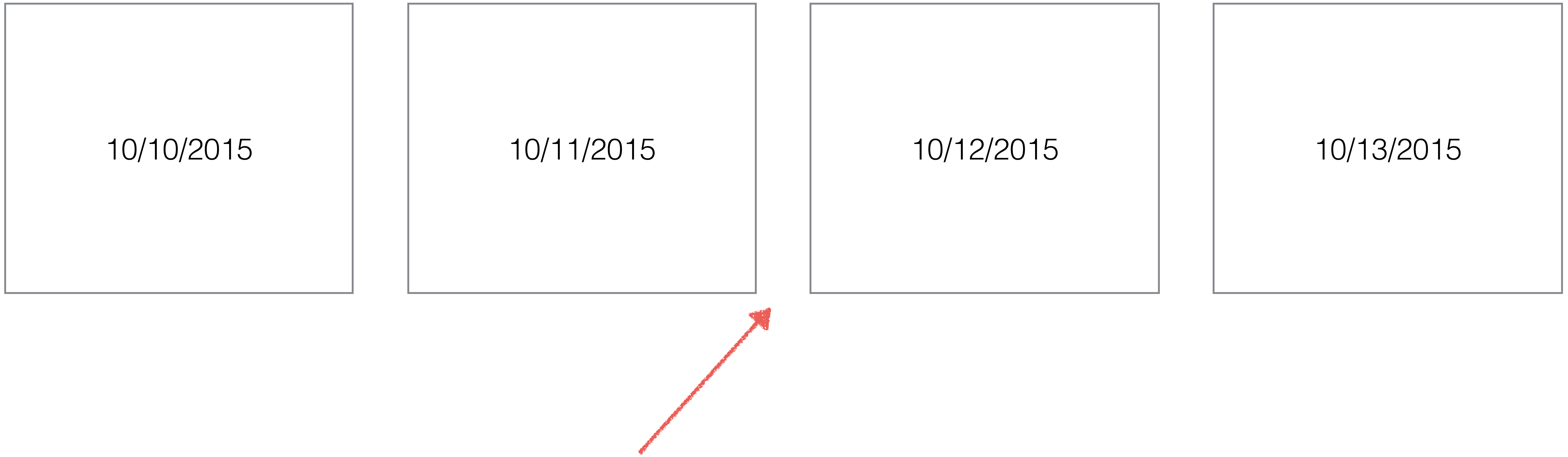
The new InfluxDB storage engine and some query language ideas

Paul Dix
CEO at InfluxDB
@pauldix
paul@influxdb.com

preliminary intro materials…

Everything is indexed by time and series

Shards



Data organized into Shards of time, each is an underlying DB
efficient to drop old data

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

Measurement

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

Measurement

Tags

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

Measurement

Tags

Fields

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

Measurement | Tags | Fields | Timestamp

InfluxDB data

temperature,device=dev1,building=b1 internal=80,external=18 1443782126

Measurement

Tags

Fields

Timestamp

We actually store up to ns scale timestamps
but I couldn't fit on the slide

Each series and field to a unique ID

temperature,device=dev1,building=b1#internal → 1

temperature,device=dev1,building=b1#external → 2

Data per ID is tuples ordered by time

temperature,device=dev1,building=b1#internal → **1**

1 → **(1443782126,80)**

temperature,device=dev1,building=b1#external → **2**

2 → **(1443782126,18)**

Storage Requirements

High write throughput

to hundreds of thousands of series

Awesome read performance

Better Compression

Writes can’t block reads

Reads can’t block writes

Write multiple ranges simultaneously

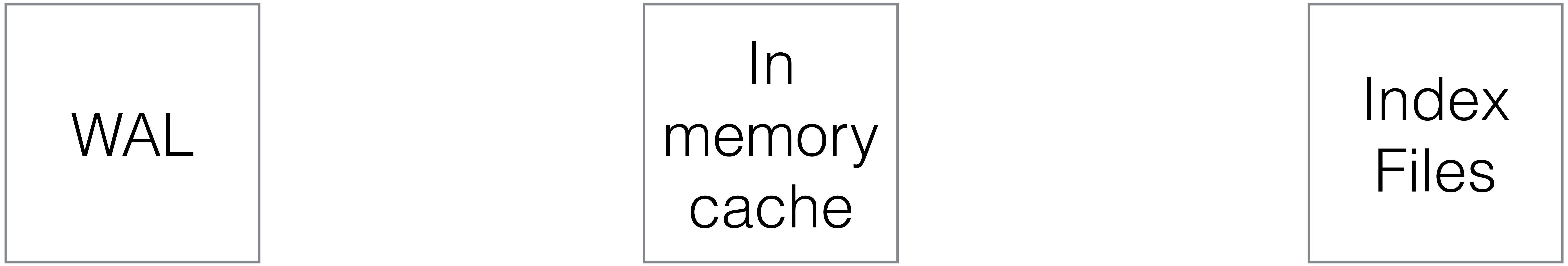
Hot backups

Many databases open in a single process

InfluxDB's
Time Structured Merge Tree
(TSM Tree)

InfluxDB's
Time Structured Merge Tree
(TSM Tree)
like LSM, but different

Components



Components

Similar to LSM Trees

WAL

In memory cache

Index Files

Components

Similar to LSM Trees

WAL

In memory cache

Index Files

Same

Components

Similar to LSM Trees

WAL	In memory cache	Index Files
Same	**like MemTables**	

Components

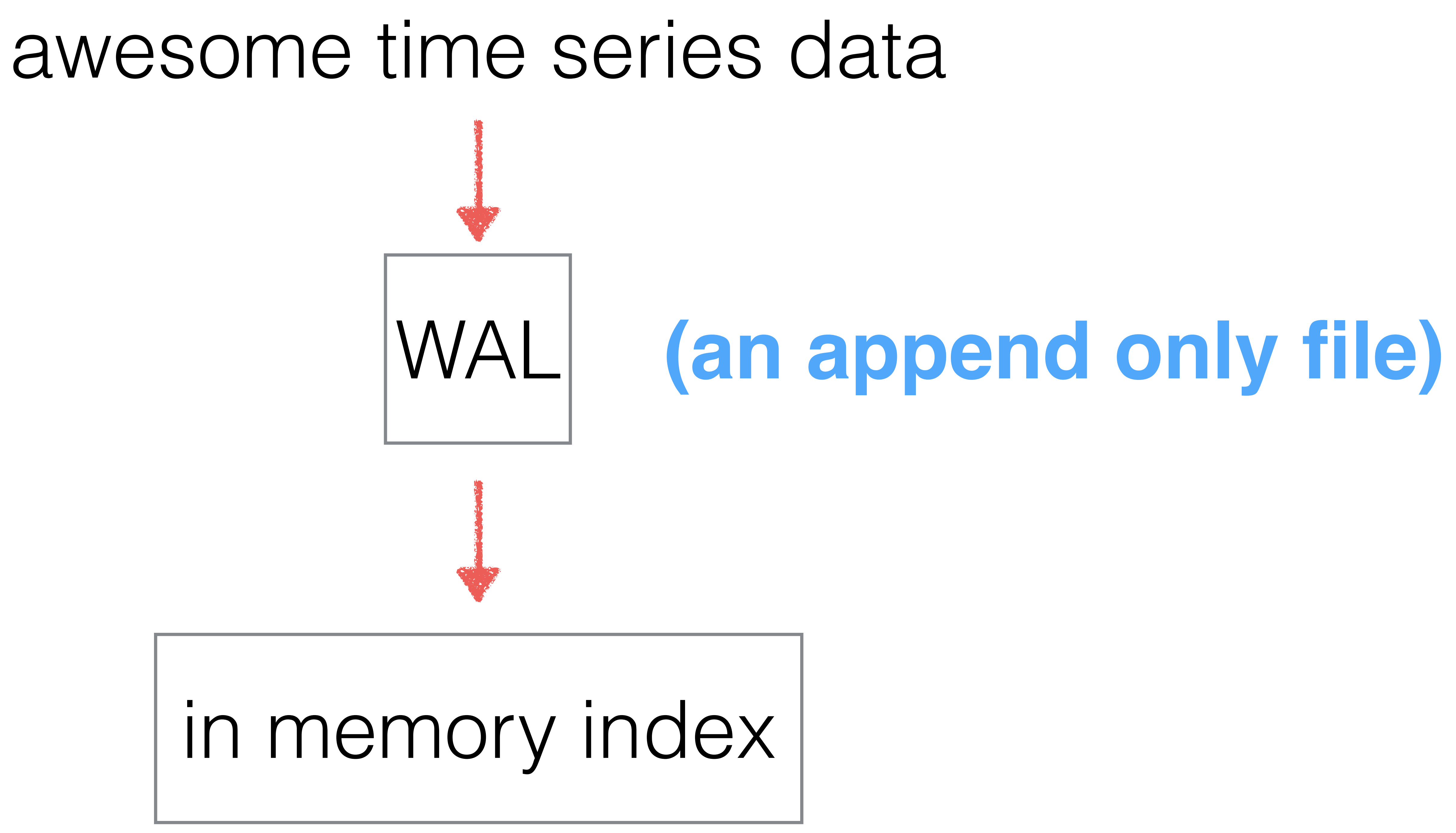
Similar to LSM Trees

WAL	In memory cache	Index Files
Same	**like MemTables**	**like SSTables**

awesome time series data

WAL

(an append only file)


awesome time series data
WAL
(an append only file)
in memory index

In Memory Cache

```
// cache and flush variables
cacheLock                      sync.RWMutex
cache                          map[string]Values
flushCache                     map[string]Values
```

temperature,device=dev1,building=b1#internal

In Memory Cache

```
// cache and flush variables
cacheLock           sync.RWMutex
cache               map[string]Values
flushCache          map[string]Values
```

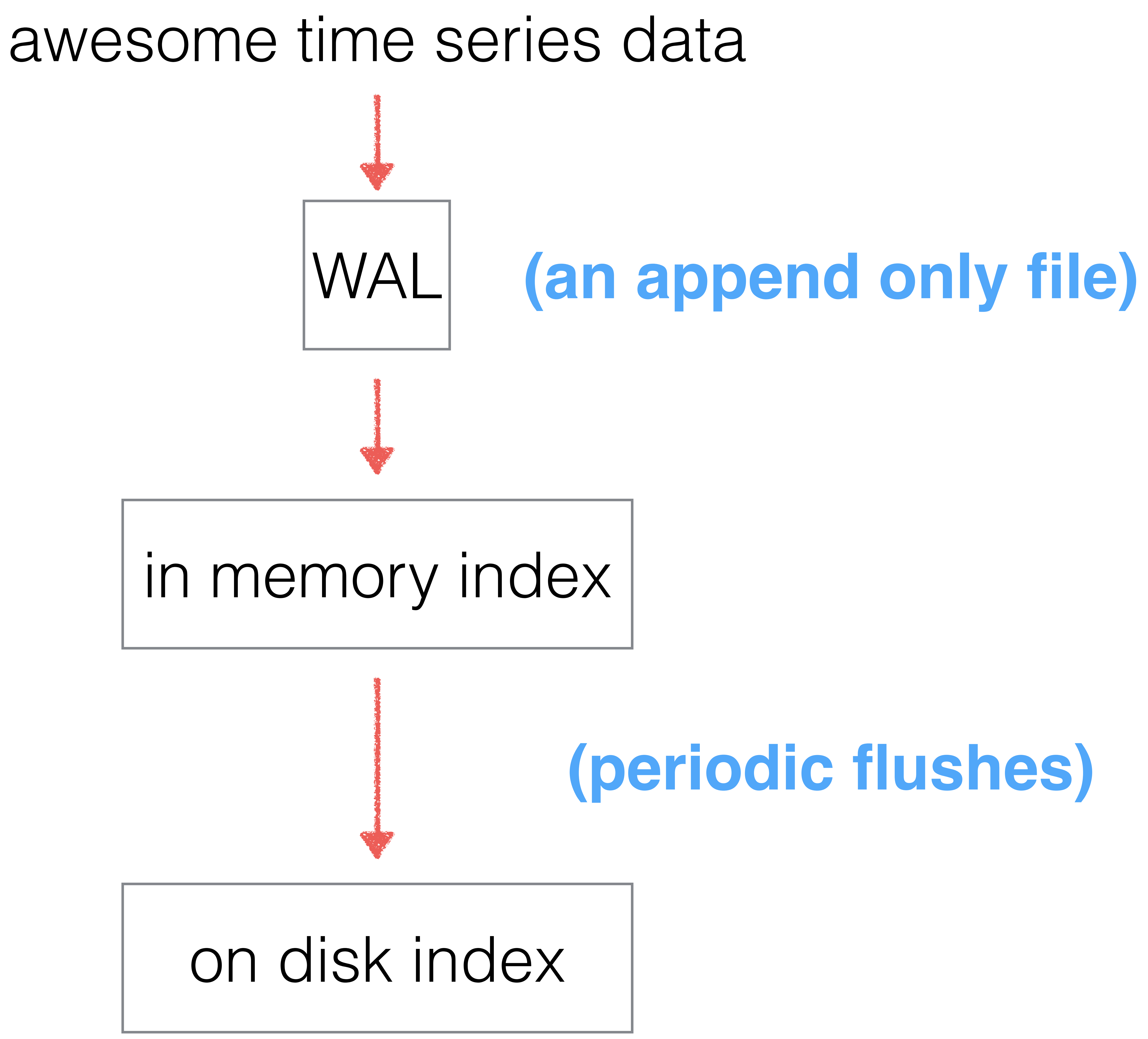
writes can come in while WAL flushes

```go
// cache and flush variables
cacheLock                      sync.RWMutex
cache                          map[string]Values
flushCache                     map[string]Values
dirtySort                      map[string]bool
```

values can come in out of order.
mark if so, sort at query time

Values in Memory

```
type Value interface {
    Time() time.Time
    UnixNano() int64
    Value() interface{}
    Size() int
}
```


awesome time series data
WAL
(an append only file)
in memory index
(periodic flushes)
on disk index

The Index

Data File

Min Time: **10000**
Max Time: **29999**

Data File

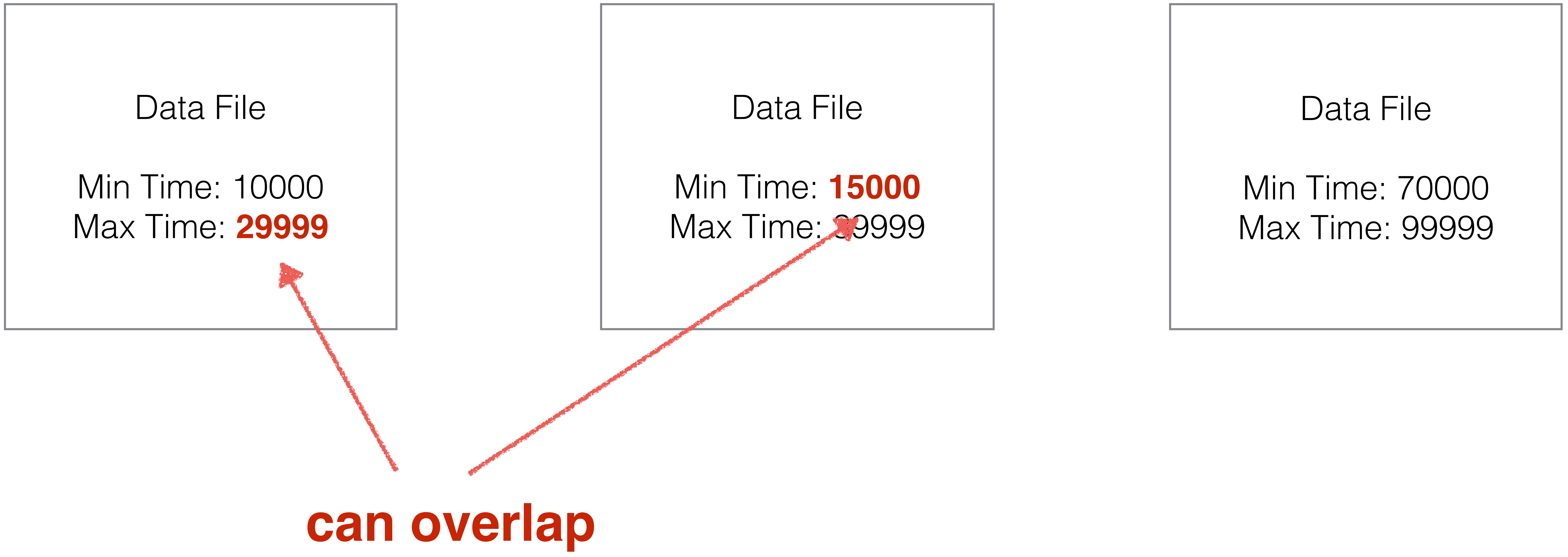
Min Time: 30000
Max Time: 39999

Data File

Min Time: 70000
Max Time: 99999

Contiguous blocks of time

The Index


Data File
Min Time: 10000
Max Time: 29999
Data File
Min Time: 15000
Max Time: 39999
Data File
Min Time: 70000
Max Time: 99999
can overlap

The Index

cpu,host=A

Min Time: 10000
Max Time: **20000**

cpu,host=A

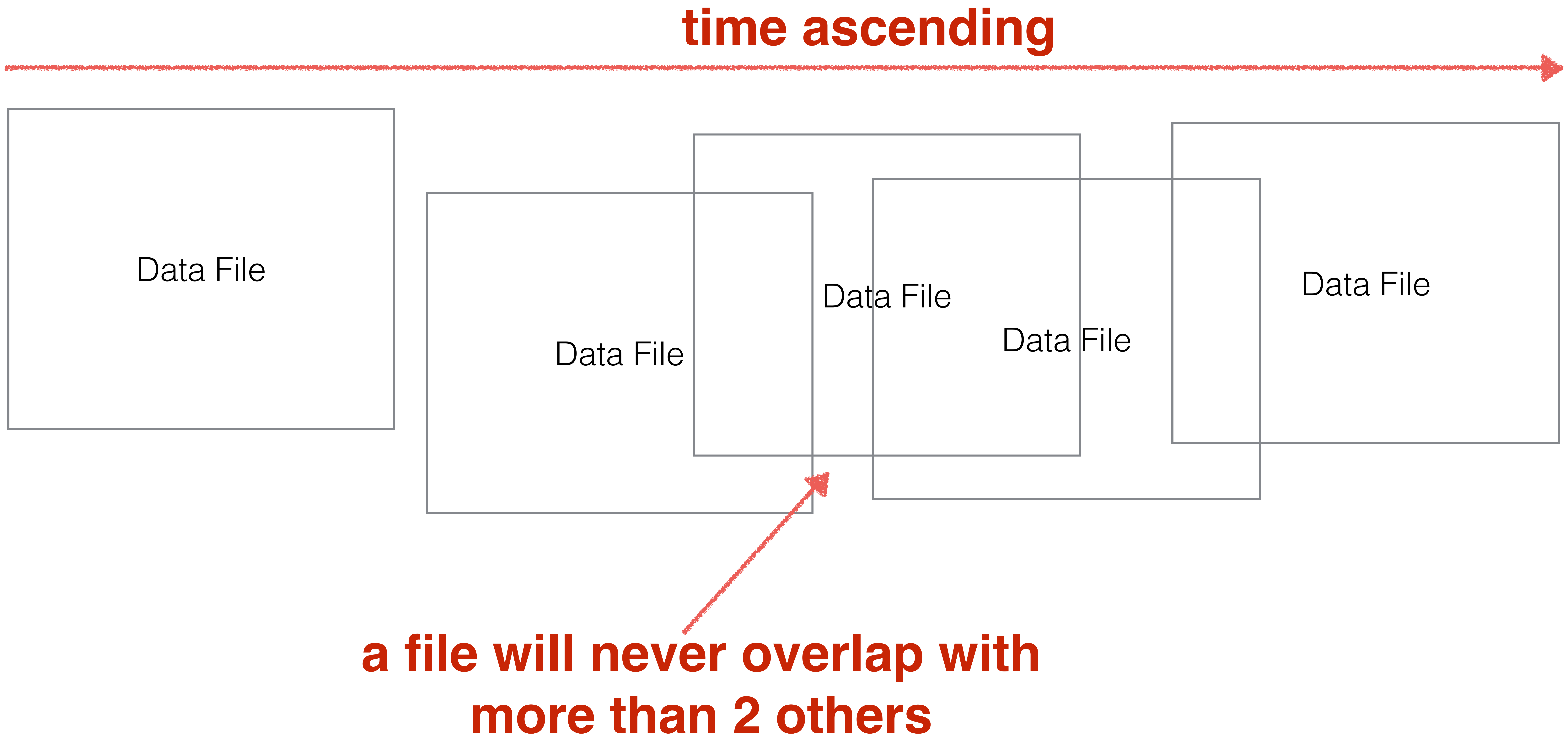
Min Time: **21000**
Max Time: 29999

Data File

Min Time: 70000
Max Time: 99999

but a specific series must not overlap

The Index



time ascending
Data File
Data File
Data File
Data File
Data File
a file will never overlap with
more than 2 others

Data files are read only, like LSM SSTables

The Index

Data File

Min Time: 10000
Max Time: 29999

Data File

Min Time: 30000
Max Time: 39999

Data File

Min Time: 70000
Max Time: 99999

Data File

Min Time: **10000**
Max Time: **99999**

they periodically get compacted (like LSM)

Compacting while appending new data

Compacting while appending new data

```go
func (w *WriteLock) LockRange(min, max int64) {
    // sweet code here
}

func (w *WriteLock) UnlockRange(min, max int64) {
    // sweet code here
}
```

Compacting while appending new data

This should block until we get it

```
func (w *WriteLock) LockRange(min, max int64) {
    // sweet code here
}

func (w *WriteLock) UnlockRange(min, max int64) {
    // sweet code here
}
```

Locking happens inside each Shard

Back to the data files…

Data File

Min Time: 10000
Max Time: 29999

Data File

Min Time: 30000
Max Time: 39999

Data File

Min Time: 70000
Max Time: 99999

Data File Layout

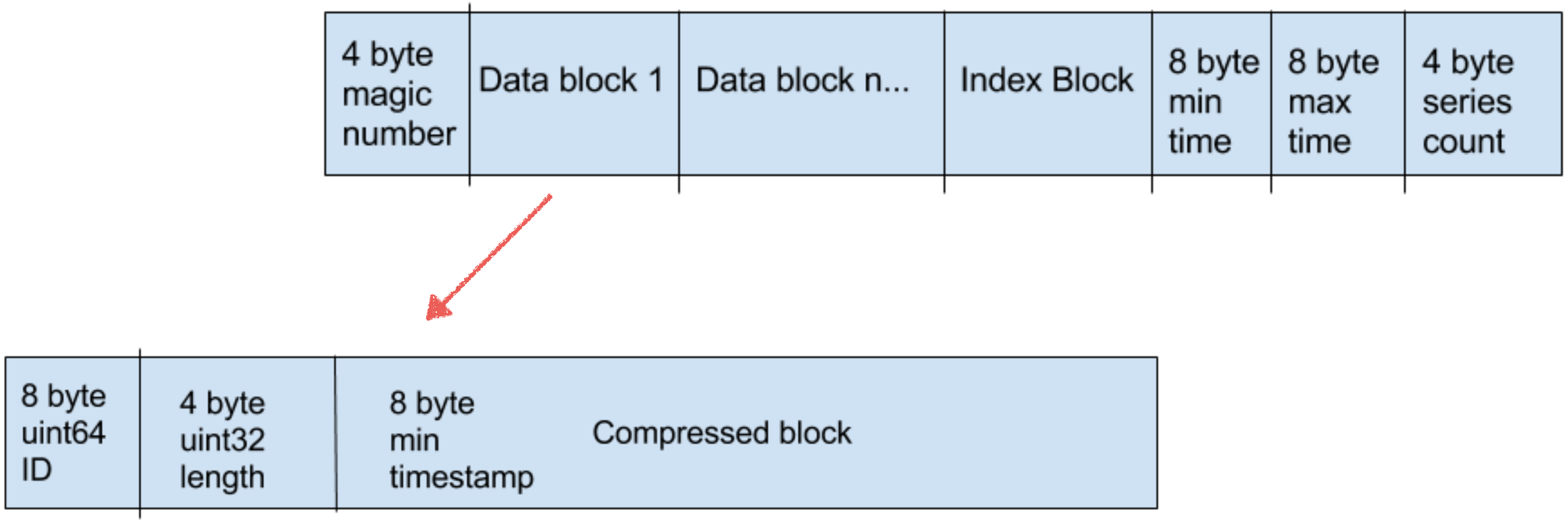
4 byte magic number	Data block 1	Data block n...	Index Block	8 byte min time	8 byte max time	4 byte series count

Data File Layout

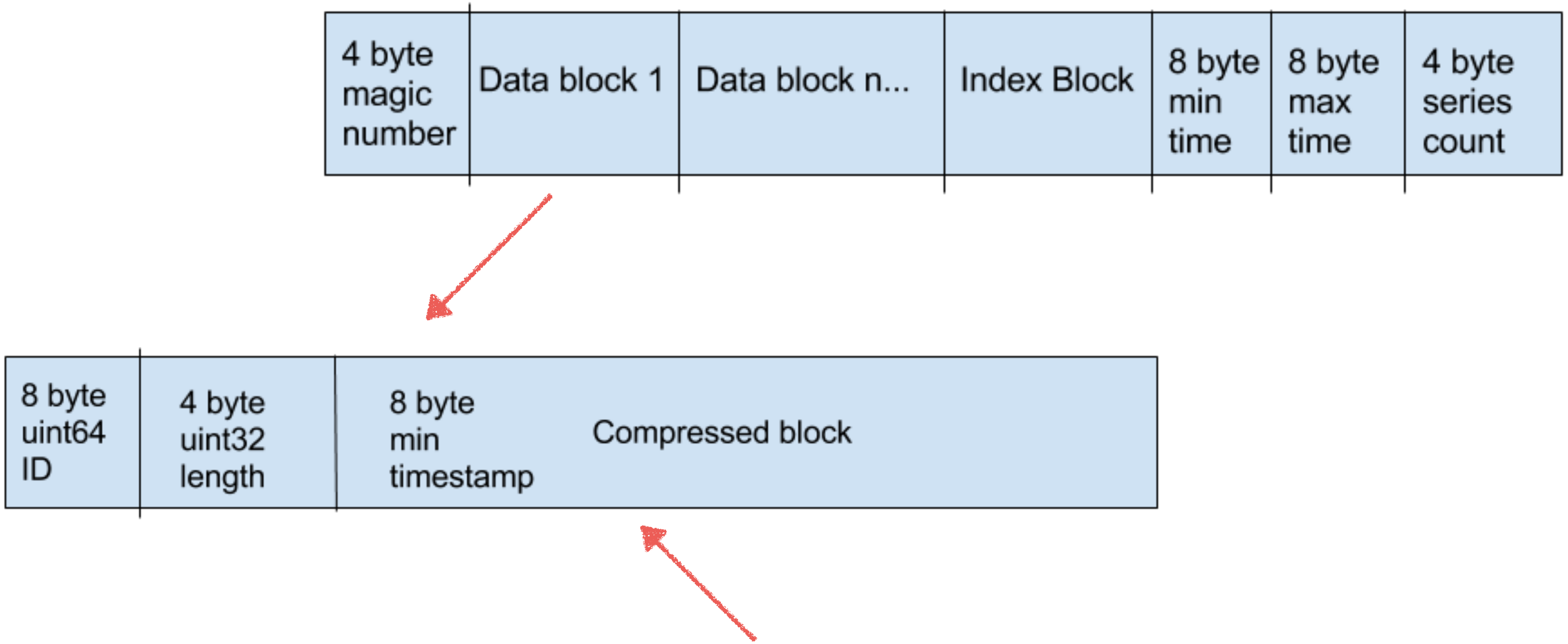
4 byte magic number	Data block 1	Data block n...	Index Block	8 byte min time	8 byte max time	4 byte series count

Similar to SSTables

Data File Layout

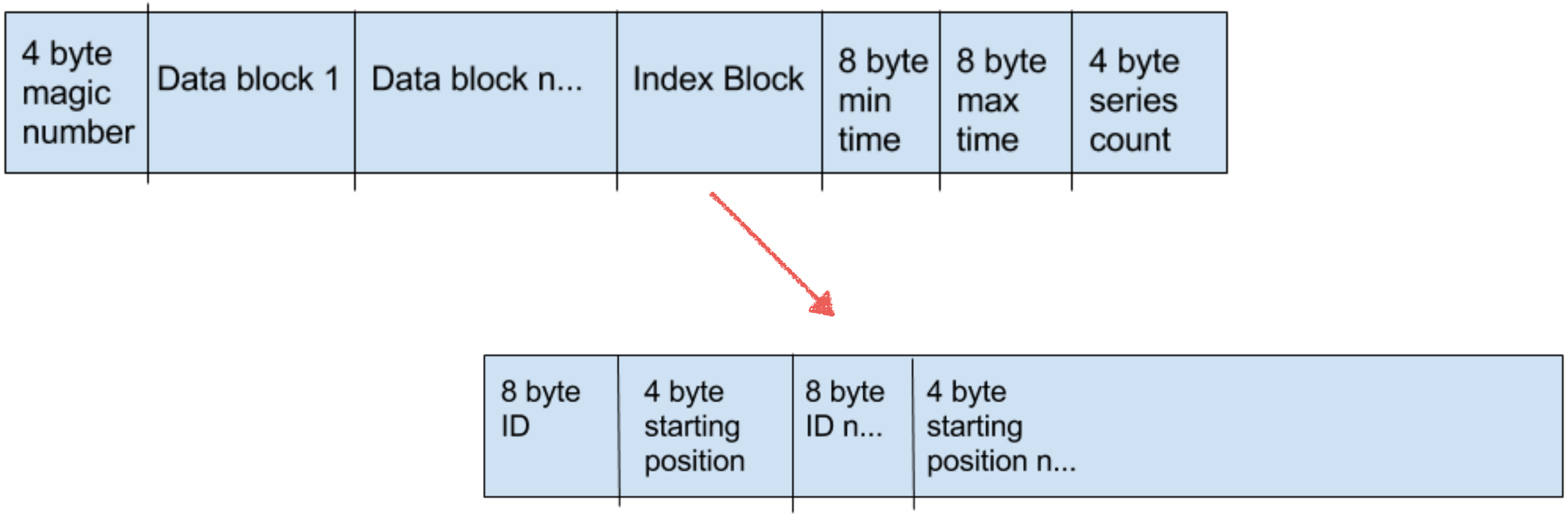


Data File Layout

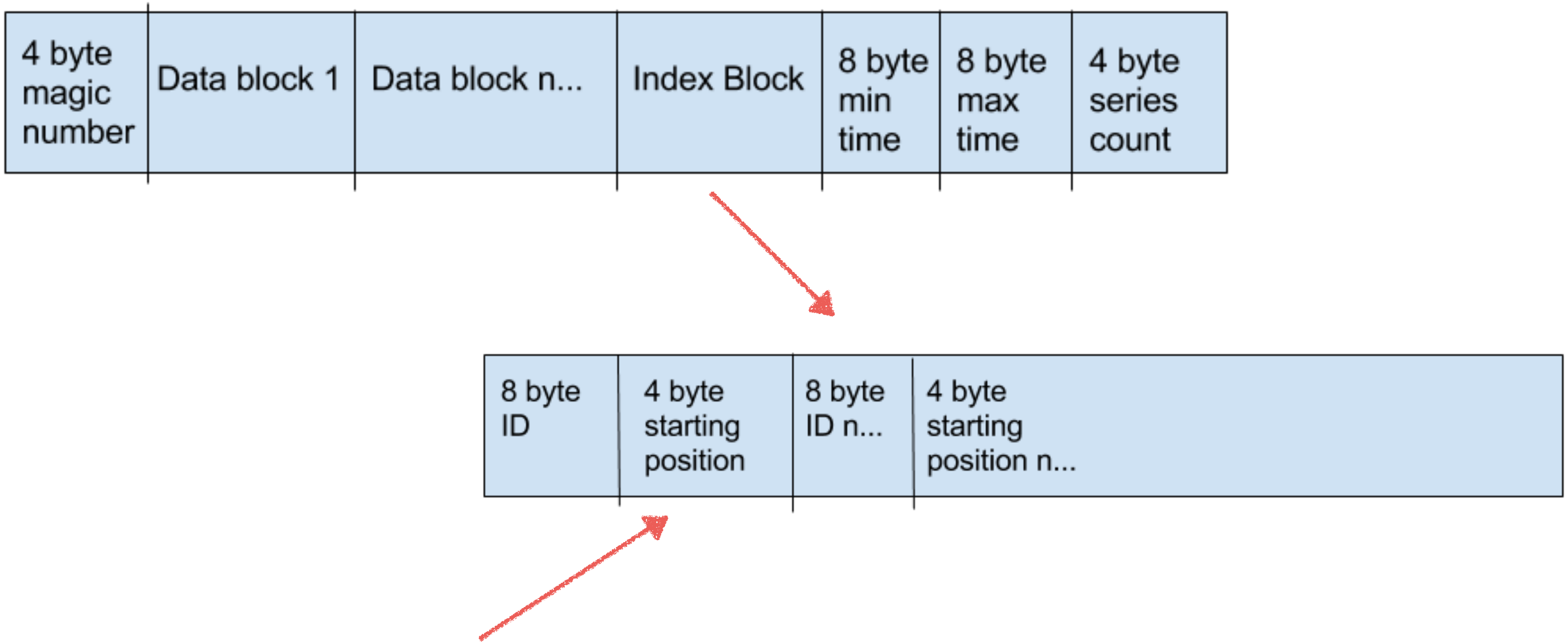


blocks have up to 1,000 points by default

Data File Layout



Data File Layout



4 byte position means data files can be at most 4GB

Data Files

```go
type dataFile struct {
    f         *os.File
    size      uint32
    mmap      []byte
}
```

Memory mapping lets the OS handle caching for you

Compressed Data Blocks

8 byte min timestamp	1 byte header	[1-4] byte Time length	Time bytes	Value bytes

Timestamps: encoding based on precision and deltas

Timestamps (best case):
Run length encoding

Deltas are all the same for a block
(only requires start time, delta, and count)

Timestamps (good case): Simple8B

Ann and Moffat in "Index compression using 64-bit words"

Timestamps (worst case): raw values

nano-second timestamps with large deltas

float64: double delta

Facebook's Gorilla - google: gorilla time series facebook

https://github.com/dgryski/go-tsz

booleans are bits!

int64 uses zig-zag

same as from Protobufs
(adding double delta and RLE)

string uses Snappy

same compression LevelDB uses
(might add dictionary compression)

How does it perform?

Compression depends greatly on the shape of your data

Write throughput depends on batching, CPU, and memory

one test:

100,000 series
100,000 points per series
10,000,000,000 total points
5,000 points per request
c3.8xlarge, writes from 4 other systems
~390,000 points/sec
~3 bytes/point (random floats, could be better)

~400 IOPS

30%-50% CPU

There's room for improvement!

Detailed writeup

https://influxdb.com/docs/v0.9/concepts/storage_engine.html

Query Language Ideas

Three different kinds of functions

Aggregates

```
select mean(value)
from cpu
where host = 'A'
and time > now() - 4h
group by time(5m)
```

Transformations

```
select derivative(value)
from cpu
where host = 'A'
and time > now() - 4h
group by time(5m)
```

Selectors

```
select min(value)
from cpu
where host = 'A';
and time > now() - 4h
group by time(5m)
```

Then there are fills

```
select mean(value)
from cpu
where host = 'A'
and time > now() - 4h
group by time(5m)
fill(0)
```

How to differentiate between the different types?

How do we chain functions together?

without making breaking changes to InfluxQL

Mix jQuery style with InfluxQL

```
SELECT
   mean(value).fill(previous).derivate(1s).scale(100).as('mvg_avg')
FROM measurement
WHERE time > now() - 4h
GROUP BY time(1m)
```

D3 style

```
SELECT
   mean(value)
      .fill(previous)
      .derivate(1s)
      .scale(100)
      .as('mvg_avg')
FROM measurement
WHERE time > now() - 4h
GROUP BY time(1m)
```

Moving the FROM?

```
SELECT
  from('cpu').mean(value)
  from('memory').mean(value)
WHERE time > now() - 4h
GROUP BY time(1m)
```

Moving the FROM?

```
SELECT
  from('cpu').mean(value)
  from('memory').mean(value)
WHERE time > now() - 4h
GROUP BY time(1m)
```

consistent time and filtering applied to both

JOIN

```
SELECT
  join(
   from('errors')
     .count(value),
   from('requests')
     .count(value)
  ).fill(0)
   .count(value)
WHERE time > now() - 4h
GROUP BY time(1m)
```

Thank you!

Paul Dix
@pauldix
paul@influxdb.com