DalmatinerDB Documentation

Release 0.1.0

Heinz N. Gies

January 02, 2015

Contents

Warning: This documentation and DalmatinerDB are still quite young, please do not use it in mission critical environments without a backup!

CHAPTER 1

Installation

1.1 Binaries

Sorry at this point in time no bianries exist.

1.2 From Sorce

Dependencies:

- Erlang > R16B3
- Make
- GCC

1.2.1 Installing the datastore

```
git clone https://github.com/dalmatinerdb/dalmatinerdb.git
cd dalmatinerdb
make deps all rel
cp rel/dalmatinerdb $TARGET_DIRECTORY
cd $TARGET_DIRECTORY
cp etc/dalmatinerdb.conf.example etc/dalmatinerdb.conf
vi etc/dalmatinerdb.conf # check the settings and adjust if needed
./bin/ddb start
```

Note: DalmatinerDB by default expects to run as the user *dalmatinerdb*, this can be changed or disabled by editing the *bin/ddb* file's entry *RUNNER_USER*

1.2.2 Installing the frontend

```
git clone https://github.com/dalmatinerdb/dalmatiner-frontend.git
cd dalmatiner-frontend
make deps all rel
cp rel/dalmatinerfe $TARGET_DIRECTORY
cd $TARGET_DIRECTORY
cp etc/dalmatinerfe.conf.example etc/dalmatinerfe.conf
```

```
vi etc/dalmatinerfe.conf # check the settings and adjust if needed
./bin/dalmatinerfe start
```

Note: DalmatinerFrontend by default expects to run as the user *dalmatinerfe*, this can be changed or disabled by editing the *bin/dalmatinerfe* file's entry *RUNNER_USER*

Warning: At the moment automatic handling of disconnected upstream servers isn't handled well and might require a restart of the frontend.

CHAPTER 2

Configuration

The configuration files are located in the *etc* subdirectories of the different applications, please consult the inline documentation for details about configuration parameters.

CHAPTER 3

Tradeoffs & Design

An essential part of every database is to make a decision aobut which tradeoff to make, every database makes them in one way or another and I feel it important to be upfront about them. Being open about what is won and what is lost makes it not only honesty but makes it easyer for users to make a informed decision when picking a database for their usecase.

Note: Please keep in mind that all systems have tradeoffs and that just because some are not open about them does not mean they do not make them.

3.1 Foundation

The design decisions for DalmatinerDB are based on a number of observation about metrics. If those observations hold true for you then chances are good that DalmatinerDB is a good fit. If they don't another system with different tradeoffs might be a better choice.

3.1.1 Metrics are imutable

Once a metric is submitted it isn't going to change any more, the CPU usage last mondah at 5:31 will not suddenly spike today. It might however happen that writing the metric is delayed, so writing in the 'past' can happen.

3.1.2 A single value doens't matter

It is more important to have the bulk of data then have a single value. Usually all views are aggregated and missing values can be interpolated.

3.1.3 Everything is a integer

Every metric is either a integer value or can be represented as one (or mutliple). Allowing to scale metrics helps here. As an example `1.5s` can be represented as `1500ms`, a set of percentiles can be represented as multiple metrics (`metric.99`, `metric.95` ...)

3.2 Design

3.2.1 CAP

The perhaps first decision to make is either to pick Consistency or Availability. With metrics and the notion of immutabilit ythere is little harm in picking Availability here, so DalmatinerDB will stay available for read and write options even in the event of a network partition at the cost of giving stall reads on both sides of the pertition until it is healed (given side A can't knoow what was written at side B).

Given the immutability of metrics it can be argued that it is impossible to generate conflicting values on both sides of a split, thus merging is simple and lossless.

3.2.2 Filesystem

DalmatinerDB is designed to run on ZFS and other filesystems are strongly discuraged. While DalmatinerDB will start on any filesystem the experience will be greatly degraded without ZFS or a equally capable filesystem as a base.

DalmatinerDB's performance relies heaviley on taking advantage of facilities like ARC, ZIL, checksums and volume compression. Expecting those things to be handled on a filesystem level makes it possible to remove most of the code for caching, compression, validation from the application improving code simplicity, stability and performance significantly.

CHAPTER 4

Data Input

DalmatinerDB uses UDP for data ingres, which is one of its trade-offs. UDP is a lot faster than TCP. It prevents the providers from blocking and the consumer from overloading by dropping packages it can not handle. It is possible to have multiple UDP ports for increased concurrency.

4.1 Metric Package

Metrics are sent as size prefixed data. The layout of a metric package looks like this:

```
<<0,                                %% Prefix to denote type of message
  Time:64/integer,                  %% The time (or offset) of the package
  BucketSize:16:integer,            %% The size of the bucket name
  Bucket:BucketSize/binary          %% The bucket to write the metric to
  MetricSize:16:integer,            %% The size of the metric name
  Metric:MetricSize/binary          %% The metric name
  DataSize:16:integer,              %% The size of the metric name
  Data:DataSize/binary              %% The metric name
  >>
```

All sizes are given in bytes. The values are unsigned integers in **network byte order**. Especially with *DataSize* this is to be noted since it does **NOT** reflect the number of datapoints but rather the number of bytes used, thus it has to be a multiple of 9. As a result, a maximum of 7281 datapoints can be sent per metric package, not 65536.

Not only can a metric package have multiple consecutive datapoints, it is also possible to combine multiple metric packages into a single UDP datagram. This allows to set multiple different metrics at once. Combining metric packages allows for some optimizations and is recommended as long as the liveliness of the data doesn't prevent it.

4.2 Data Section

The data section contains raw data for the database, each datapoint consists out of 9 bytes. 1 byte for indicating written data and 8 byte for a 64 bit signed integer in network byte order.

```
<<1, Value:64/signed-integer>>
```

CHAPTER 5

TCP Protocol

5.1 Keepalife

A simple keepalife that can be send, no reply will be send to this message

```
<<0>>.
```

5.2 List Buckets

This command list all buckets, each bucket known to the system. The command is received and a reply send directly.

```
<<3>>.
```

The Reply is prefixed with the total size of the whole reply in bytes (not including the size prefix itself). Then each bucket is prefixed by a size of the bucket name.

```
%% Outer wrapper
<<ReplySize:32/integer, Reply:ReplySize/binary>>.
%% Elements of the reply
<<BucketSize:16/integer, Bucket:BucketSize/binary>>.
```

5.3 List Metrics

Lists all metrics in a bucket. The bucket to look for is prefixed by 1 byte size for the bucket name. .. code-block:: erlang

> **<<1,** BucketSize:8/integer, Bucket:BucketSize/binary>>.

The Reply is prefixed with the total size of the whole reply in bytes (not including the size prefix itself). Then each metric is prefixed by a size of the metric name.

```
%% Outer wrapper
<<ReplySize:32/integer, Reply:ReplySize/binary>>.
%% Elements of the reply
<<MetricSize:16/integer, Metric:MetricSize/binary>>.
```

5.4 Get

Retrieves data for a metric, bucket and metric are size prefixed as strings, Time and count are unsigned integers.

```
<<2,
  BucketSize:8/integer, Bucket:BucketSize/binary,
  MetricSize:16/integer, Metric:MetricSize/binary,
  Time:64/integer, Count:32/integer>>.
```

There will **always** be returned `Count` messages will be returned, if there is no or insufficient data or the bucket/metric doesn't exist the missing data will be filled with blanks.

```
<<Reply:((9*8)*Count)/signed-integer>>.
```

CHAPTER 6

Dalmatiner Query Language

6.1 General

The front end query language is rather remotely related to SQL, which should make it simple to pick up. The focus is on querying one metric at a time, given that those queries are incredible fast. Globs can also be used to query multiple metrics at a time.

The basic syntax looks like this:

```
SELECT <FIELDS> [ FROM <ALIASES>] <TIME RANGE> [IN <RESOLUTION>]
```

Please keep in mind that each query can only return a single row of data at the moment.

6.1.1 Fields (*SELECT* section)

A field can either be a metric, an alias for a metric or a function:

- *cloud.zones.cpu.usage.eca485cf-bdbb-4ae5-aba9-dce767 BUCKET tachyon* - a fully qualified metric.
- *vm* - an alias that is defined in the *FROM* section of the query.
- *avg(vm, 1m)* - a aggregation function.

Fields can be aliased for output adding a *AS <alias>* directive after the field.

Multiple fields can be given separating two fields with a *,*. The resolution of fields does not have to be the same, and there is no validation to enforce this!

6.1.2 Aliases (*FROM* section)

When a metric is used multiple times it is more readable to alias this metric in the *FROM* section. Multiple elements can be given separated with a *,*. Each element takes the form: *<metric> BUCKET <bucket> AS <alias>*.

It is possible to match multiple metrics by using a mulitget aggregator and a glob to match a metric. Valid multiget aggregators are *sum* and *avg*. For example: *sum(some.metric.* BUCKET b)*.

6.1.3 Time Range

There are two ways to declare ranges. Although numbers here represent seconds, DalmatinerDB does not care about the time unit at all:

```
BETWEEN <start:reltime> AND <end:reltime>
```

The above statement selects all points between the *start* and the *end*.

The most used query is 'what happened in the past X seconds?', so there is a simplified form for this:

```
LAST <amount:int>|[time:time>]
```

6.1.4 Resolution (*IN* section)

By default queries treat incoming data as a one second resolution, however this can be adjusted by passing a resolution section to the query. The syntax is: *IN <resolution:time>*.

6.2 Data Types

6.2.1 Integer (int)

A simple number literal i.e. *42*.

6.2.2 Time (time)

There are two ways to declare times:

- Relatively, in which case the time is a simple integer and corresponds to a number of metric points used. (i.e. *60*)
- Absolute, in which case the time is an integer followed by a time unit such as *ms*, *s*, *m*, *h*, *d* and *w*. In this case the resolution of the metric is taken into account.

6.2.3 Relative Time (reltime)

Relative times can either be the keyword *NOW*, an absolute timestamp (a simple integer) or a relative time in the past such as *<time> AGO*.

6.2.4 Metric

Metrics are simple strings that are optionally separated by dots and a second string for the bucket. The two strings are separated by the keyword *BUCKET*.

Example:

```
cloud.zones.cpu.usage.eca485cf-bdbb-4ae5-aba9-dce767 BUCKET tachyon
```

6.3 Aggregation Functions

Aggregation functions aggregate a metric over a given range of time and decrease the resolution by doing so. Aggregation functions can be nested, in which case the 'higher' functions work with the decreased resolution of lower functions and not the raw resolution. This means the correct code to get the 1m average over 10s sums from a 1s resolution metric would be *avg(sum(m, 10s), 1m)* not *avg(sum(m, 10s), 6s)* - however this does not apply when using the point and not the time declaration, so it would be: *avg(sum(m, 10s), 6)* not *avg(sum(m, 10s), 60)* (please note the missing *s*).

6.3.1 min/2

The minimal value over a given range of time.

6.3.2 max/2

The maximal value over a given range of time.

6.3.3 sum/2

The sum of all values of a time-range.

6.3.4 avg/2

The average of a time-range (this is the mean not the median).

6.3.5 empty/2

Returns the total of empty data-points in a time-range. This can be used to indicate the precision of the data and the loss occurring before they get stored.

6.3.6 percentile/3

Returns the value of the `n` th percentile, where 0 < `n` < 1. The percentile is given as the second value of the function, the time-range to aggregate over as the third.

6.4 Manipulation Functions

Manipulation functions help to change the values of a value list they do not change the resolution or aggregate multiple values into one.

6.4.1 derivate/1

Calculates the derivate of a metric, meaning N'(X)=N(X) - N(X-1)

Note: Even if the resolution isn't changed this function removes exactly 1 element from the result

6.4.2 multiply/2

Multiplies each element with integer constant.

6.4.3 divide/2

Divides each element with a integer constant.

6.5 Examples

Calculates the min, max and average of a metric over a hour:

```
SELECT min(vm, 10m), avg(vm, 10m), max(vm, 10m) AS max FROM cloud.zones.cpu.usage.eca485cf-bdbb-4ae5-
```

HTTP API

The DalmatinerDB frontend offers a simplistic HTTP API for running queries along with a basic UI. All endpoints respond in regards of the **Content-Type** requested. Three types are currently valid:

- text/html - will return a human readable UI page.
- application/json - returns the result in json format.
- application/x-messagepack - returns the result's MessagePack encoded.

Using MessagePack is recommended as it both conserves computation time and bandwidth during encoding and decoding.

7.1 Bucket Listing

To list all buckets, submit a *GET* request to the endpoint */buckets*. The reponse will return a list of all buckets known to DalmatinerDB.

7.2 Metric Listing

To list all the metrics in a bucket, submit a *GET* request to the endpoint */buckets/<bucket name>*. The response will return a list of all metrics inside the requested bucket.

7.3 DQL Query

To execute a DQL query a *GET* request is performed on the endpint */*, and the query passed with the parameter *q*. The response will return an object of the form:

```
{
 "t": 1.2, // Duration in milliseconds
 "d": [{/*...*/}] // Result objects
}
```

The result object looks like this:

```
{
 "n": "avg", // name of the result
 "r": 1, // resolution of the result
 "v": [42 /*, ...*/] // The array of values
}
```

CHAPTER 8

Benchmarks

Artificial benchmarks only show a portion of reality so all benchmarks were made in a real environment and results shown are based on workloads seen in production. To ensure comparability of results the exact same workload was mirrored to different backends on identical hardware and OS configuration.

Warning: The benchmarks show initial results. The results are by no means a comprehensive comparison but should give a general idea of the state of things.

8.1 Setup

8.1.1 Hardware & OS

Tests run in Zones on SmartOS 20140124T065835Z, each zone has 16 GB of memory, 400% (non-competing) CPU CAP (and shares equivalent to 4 cores on a hyperthreaded dual *E5-2687W v2 @ 3.40GHz*).

Each zone has 15TB of storage on spinning disks with L2ARC and ZIL on mirrored SSDs.

8.1.2 Configuration

DalmatinerDB is configured to use 30 UDP listeners and caches up to 100s in memory before flushing a metric (however most metrics are flushed earlier. A detailed breakdown of this process is given in the DalmatinerDB section) and a R/N/W value of 1.

KairosDB 0.9.3 and Cassandra version 2.0.5 are used with the default configuration (Keyspace configuration based on the KairosDB defaults) with a N value of 1. Since it is not possible to disable compression lz4 is automatically used.

8.1.3 Datastores

- DalmatinerDB - obviously
- Graphite - (based on the SmartOS dataset version)
- KairosDB - (on Cassandra 2.0.5)
- InfluxDB - not included, doesn't compile on SmartOS and running in a KVM would result in an unfair disadvantage

8.2 Workload

The systems are subjected to a workload of a stream of roughly 14,000 metrics per seconds (one datapoint per metric per second). The only batching allowed is on the metric axis not the time axis, which cuts network overhead and does not reduce the liveliness of the database's data.

Each metric consists of:

- A metric identifier (name)
- An epoch timestamp
- A value for the timestamp

8.3 Duration

Measurements presented here are taken after a week of continuous load.

8.4 Results

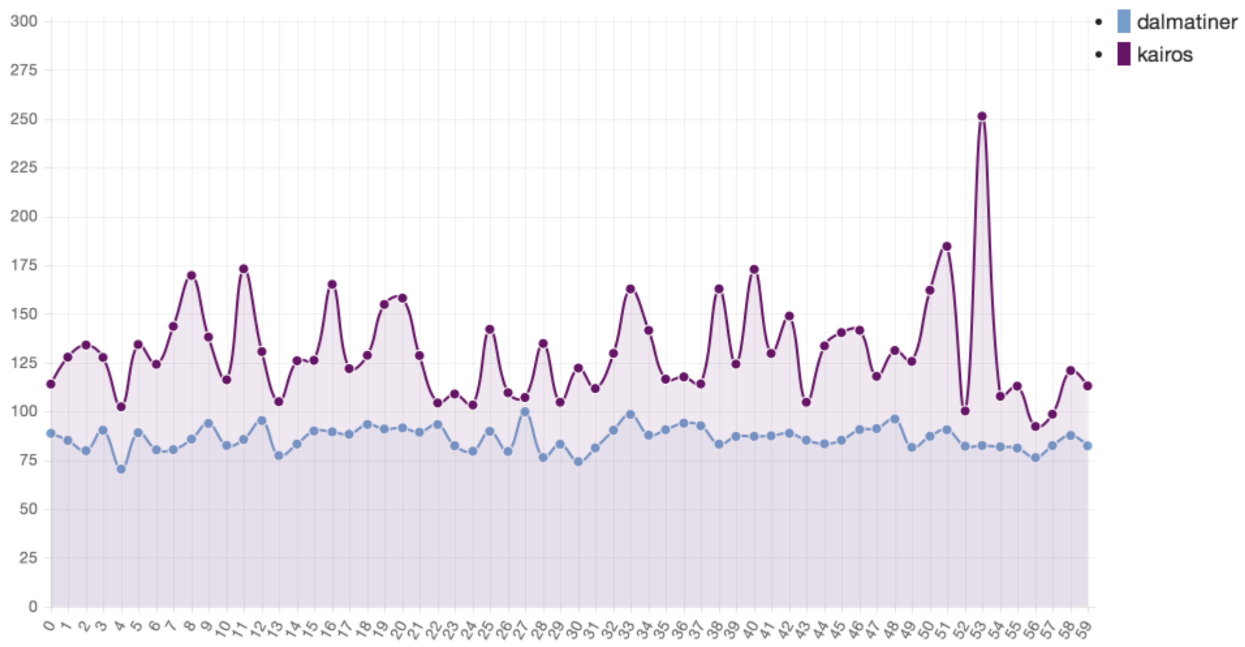
Results for Graphite are not listed since halfway through the test carbon-cache locked up with and consumed 100% memory in its zone.

8.4.1 Usage during operations

The graphs shouw measurements over 1 hour, with the average over a minute taken of a from the described system w/o significant read quaries happening during that timeframe on DalmatinerDB and without any read quaries on KairosDB.

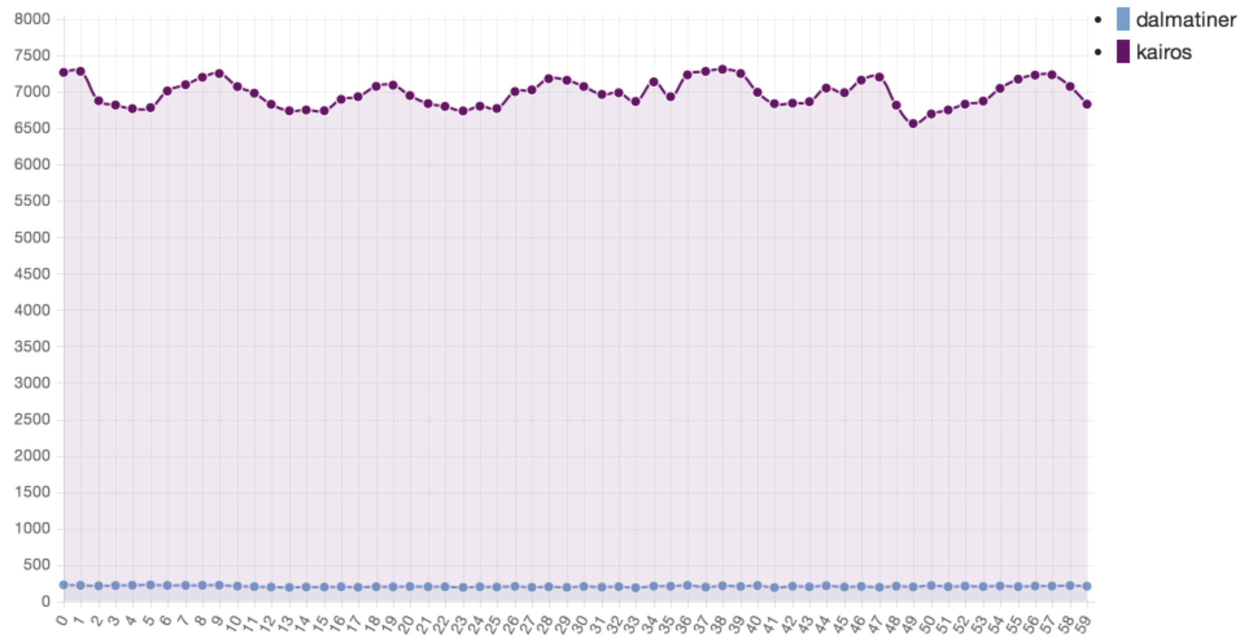
The first graph shows CPU cores used. In it a cpu usage of 3.125 corresponds to 100% of a core used.

Measurement	DalmatinerDB	KairosDB	
		Kairos	Cassandra
CPU Usage %	2.8%	3.5%	1.8%
CPU Cores	0.9	1.1	0.6



The second graph shows MB of used memory.

Measurement	DalmatinerDB	KairosDB	
		Kairos	Cassandra
Memory SIZE	260MB	1323MB	7720MB
Memory RSS	135MB	1303MB	5546MB



8.4.2 Data Size

All systems use compression. Shown is the effective data size on disk:

Note: KairosDB uses compression in Cassandra and there is no option to disable it. This makes it hard to determine the compresison ratio and the effective size per metric.

Measurement	DalmatinerDB	KairosDB
grows 10m	9133B	80514B
compressratio	8.32x	1.02x *
size/point	8.65 bit	???

8.4.3 Query Times

Query performed: The maximum nwait per second over the last hour for a given VM.

```
SELECT max(cloud.zones.cpu.nwait.e2be6f6c-2005-4f2d-aff9-f427b9 BUCKET tachyon, 1m) LAST 1h
```

	DalmatinerDB	KairosDB
First	~2ms	~300ms
consecutive	~1.3ms	~135ms

Query performed: The maximum nwait per hour over the last day for 7 VMs.

```
select
  max(cloud.zones.cpu.usage.f242021c-c5eb-4c53-a609-64bee4 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.b02df988-2abf-4364-8f55-c39eb3 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.7d1a1a3b-f3e9-4388-a938-c3a866 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.986ea915-f274-41c4-9ac5-b3dbd1 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.1333cf62-b8f1-496a-b2e1-5ec9d4 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.c6a34e43-a242-46e5-89af-b25431 BUCKET tachyon, 1h),
  max(cloud.zones.cpu.usage.e86f77ef-27a3-44c2-9348-f2319b BUCKET tachyon, 1h) LAST 1d
```

	DalmatinerDB	KairosDB
First	~120ms	~1600ms
consecutive	~85ms	~1450ms

8.5 Addendum

8.5.1 DalmatierDB write sizes

Actual distribution of write cache as affected by read and out of order flushes:

# Metrics	# Writes
38	3
85	10
49	32
16	69
37	132
83	149
84	417
15	588
62	672
93	672
35	682
63	682
69	682
13	806
14	849
36	1030
12	4030
11	4398
9	11694
1	11719
8	12780
10	13124
3	15206
7	25545
6	29203
101	37089
4	52765
5	85455
2	86841