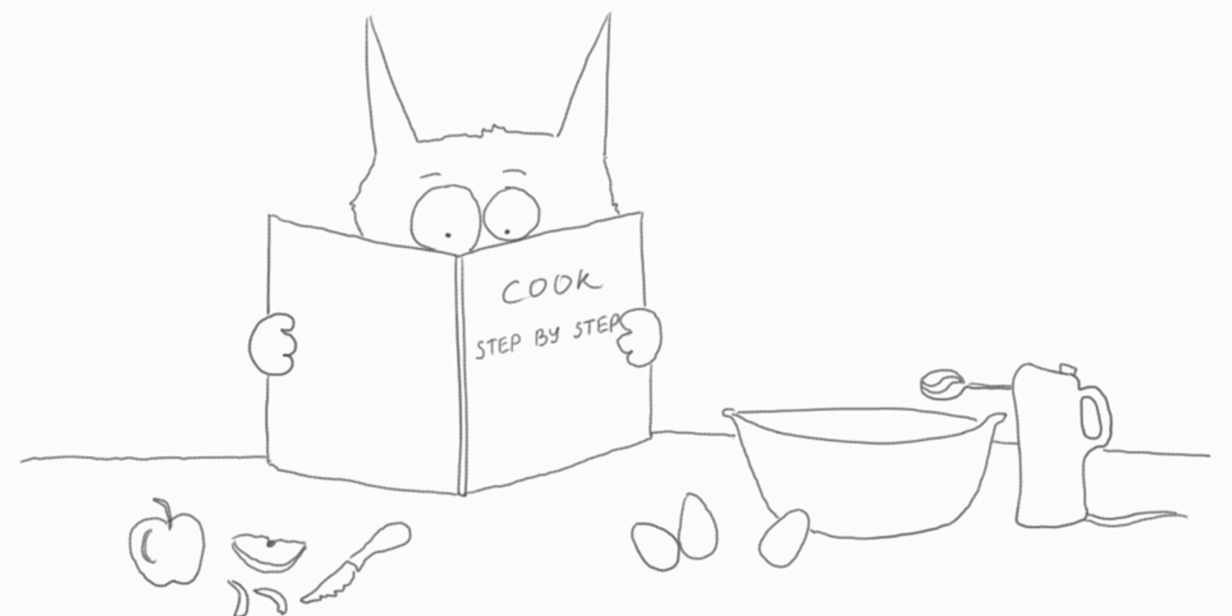
# Step-By-Step Guide to Building an Application Analytics System

Anton Lavrik
WhatsApp



# Motivation: mobile application analytics

- Context: ~450M active apps (mobile clients) in extremely diverse environments
- Goals: best product, best user experience
- Strategy: identify and fix problems, keep improving

  => Need to collect and analyze stats from the app itself

# Wam: WhatsApp Application Metrics

- Tool for collecting and analyzing application performance data
- Primary users: engineers
- Scalable – rough estimate:

500M daily active clients x 1000 data points/day = 500B data points/day

x @10 bytes per data point = 5 TB/day

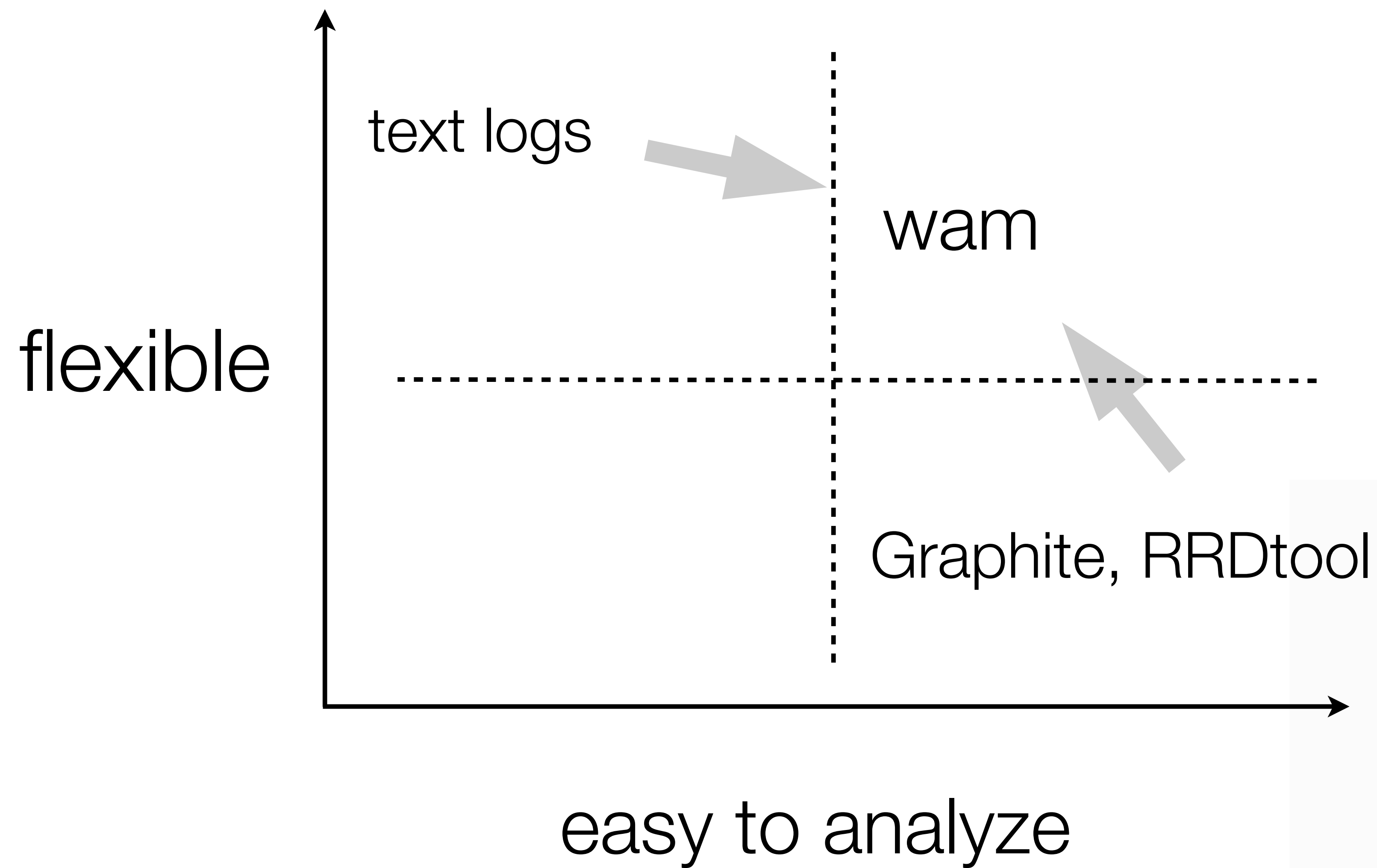
Demo: where we are today

# You can do it too!

HOWTO:

- data model + schema
- query language
- parallel query engine
- distributed storage
- shell
- API
- interactive UI
- collection

# Application analytics?



| Graphite/RRDtool | wam |
|---|---|
| strictly time-series | time factor may or may not be significant |
| data is pre-aggregated | full resolution (log) |
| low-dimensional | any number of dimensions |
| only built-in analysis capabilities | “SQL”, then #pydata on aggregates and samples |

# Wam at its core

fast parallel SQL-like queries

over distributed sparse log

| data warehouses | wam |
| --- | --- |
| full SQL | SQL subset: no heavy joins, subqueries, ... |
| any number of tables | one logical table per application (application log) |
| optimized | best-effort optimizations |
| needs ETL | no ETL! |
| manual schema management | dynamic schema evolution |

# Data model

Input event is set of key-value pairs:

```
[{field_id, field_value}]
```

# Data model: example 1

Media upload event:

| field_id | values |
|---|---|
| event_id | media_upload |
| media_upload_type | photo \| video \| audio |
| media_upload_result | ok \| duplicate \| error |
| media_upload_size | size in bytes |
| media_upload_t | time in milliseconds |
| ... | |

# Data model: example 1

Media upload event:

| field_id | values |
|---|---|
| event_id | media_upload |
| media_upload_type | photo \| video \| audio |
| media_upload_result | ok \| duplicate \| error |
| media_upload_size | size in bytes |
| media_upload_t | time in milliseconds |
| ... | |

dimensions

measures

# Data model: common dimensions

- timestamp: integer (milliseconds)
- user_id: integer
- platform: android | iphone | window_phone | s40 | symbian
- app_version: string, e.g. "2.11.136”
- os_version: string, e.g. "2.3.6"
- device_name: GT-S6102B
- country: us | gb | se | cn | ...
- mcc, mnc (mobile network code, mobile carrier code)

# Data model: schema

Describe all the fields and types using 3 files:

- dimension.json
- measures.json
- enums.json

```
{
    "id": "media_upload_type",
    "type": "enum",
    "enum": "media_type",
    "code": 55,
    "doc": "size of the media upload"
},
{
    "id": "media_upload_result",
    "type": "enum",
    "enum": "media_upload_result_type",
    "code": 56,
    "doc": "result of the media upload operation"
},
```

supported types:

- string, enum, timestamp (only for dimensions)
- integer, double, bool

# Schema management

- Store dimension.json, measures.json, enums.json in a separate git repo
- Push schema changes to repo
- Tell the system to apply the updates: `./wam-client update-schema`

# Data model: relational interpretation

```
CREATE TABLE <application> (
     -- all dimensions from"dimensions.json"
     <dimension1> <dimension1_type>,
     ...
     <dimensionN> <dimensionN_type>,

     -- all measures from "measures.json"
     <measure1> <measure1_type>,
     ...
     <measureN> <measure1_type>
 );
```

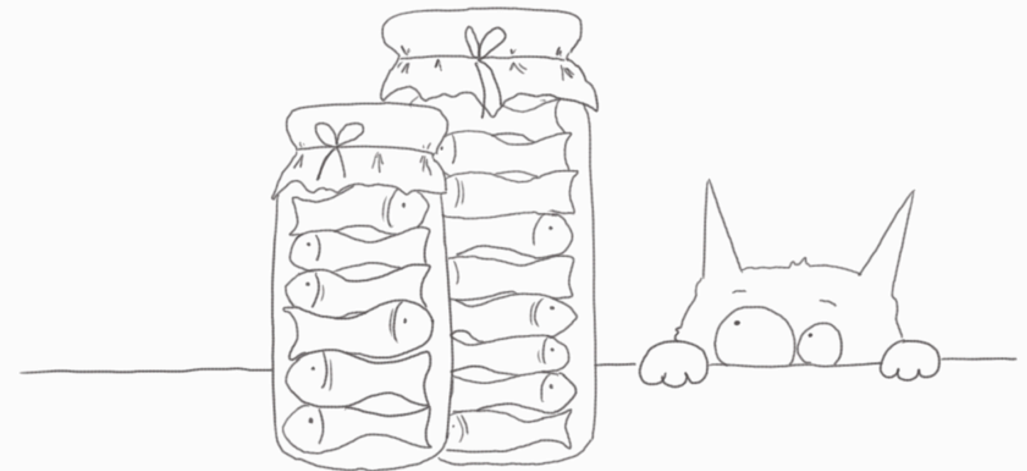
# Storage format: traditional approaches

**row-based** – mostly used in transactional databases (OLTP)

- generally, inefficient for sparse data, because need to store NULLs
- difficult to extend schema

**columnar** – used in analytic databases (OLAP)

- read only necessary columns at query time
- higher compression
- overall, ideal approach
- however, implementation can be non-trivial



# Storage format: wam approach*

so called “long format”: event is represented as a sequence of non-null fields, each encoded as a key-value pair: `[{field_code, field_value}]`

non-string fields:

- `{primitive, FieldCode, PrimitiveValue}`

dictionary-encoded string fields:

- `{string, FieldCode, StringId, StringHash, VariableLengthString}`
- `{string_reference, FieldCode, StringId, StringHash}`

end of record:

- `end_of_record`

** the actual encoding is binary*

# Storage format: wam approach (with RLE)

so called “long format”: event is represented as a sequence of non-null fields, each encoded as a key-value pair: `[{field_code, field_value}]`

non-string fields:

- `{primitive, FieldCode, PrimitiveValue, RunLengh}`

dictionary-encoded string fields:

- `{string, FieldCode, StringId, StringHash, VariableLengthString}`
- `{string_reference, FieldCode, StringId, StringHash, RunLengh}`

end of record:

- `end_of_record`

** the actual encoding is binary*

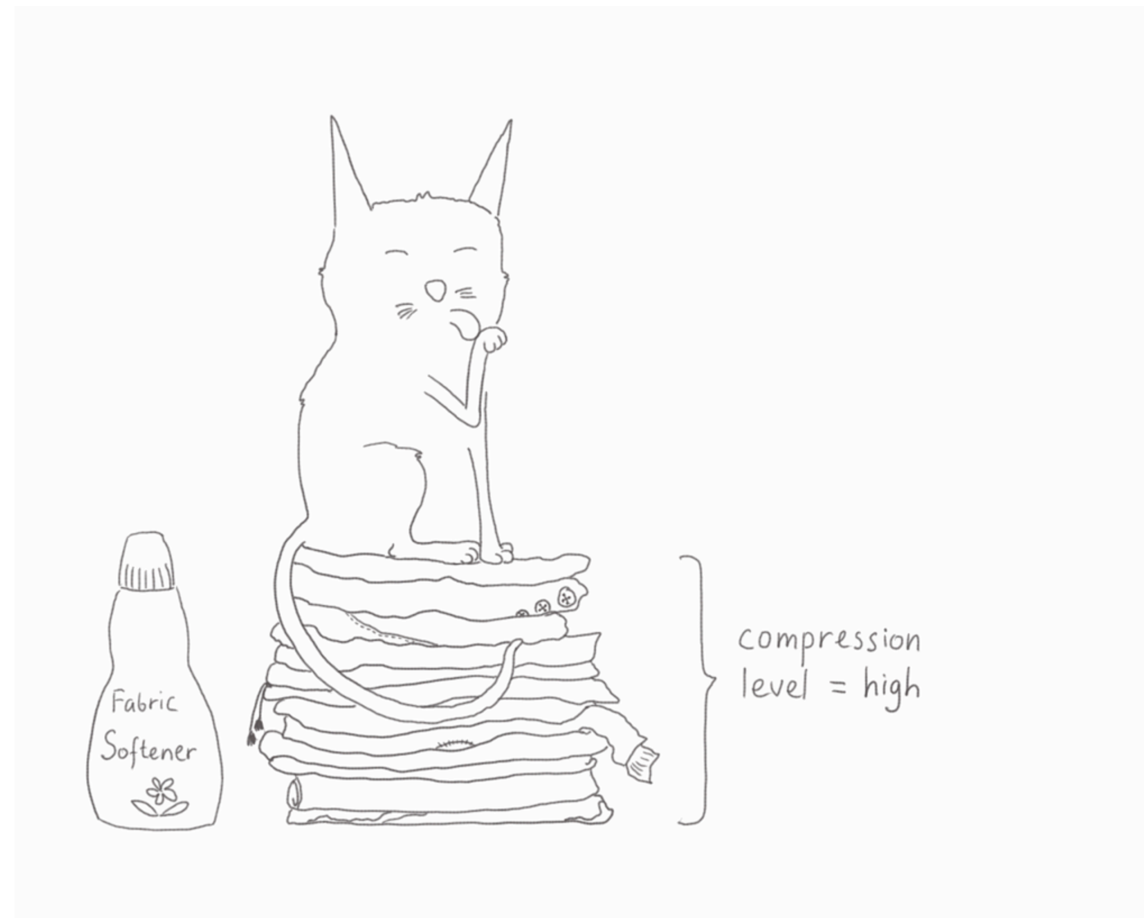
# Storage format: long format

- fairly compact
- simple, easy to implement! (almost as simple as writing a log)
- however slower to read: no column skipping, branching

# Storage: compression

lz4

- fast: faster than snappy and lzo (decompression rate ~1.5 GB/second)
- high-compression mode: better than gzip (6:1 on our data)
- winning points: simple, comes with command-line tool and framing format (takes care of storing checksums, compression levels, buffer sizes)
-



# Storage: basic write cycle

1. Write records in “long” format to a log until full or time to rotate

2. Rotate log

    1. SHA=`shasum 1.log | cut -d' ' -f1`

    2. lz4 -9 -B4 -BD 1.log $SHA.log.lz4

3. Move rotated and compressed log to the permanent storage

# Query language: example

- wam:

```
.from fieldstats
.select
    device_name
    .hist [ app_launch_t 1000 .min 0 .max 30000 ] ]
.where [
    .eq [ platform s40 ]
    .match [ ts -2d= ]
    .range [ app_version 2.12.11 ]
]
.user-sample 10
```

- sql:

```
select
  device_name,
  cast(((app_launch_t - 0) / ((30000 - 0) / 1000)) as int) *
  ((30000 - 0) / 1000) as bin,
  count(*)
from fieldstats
where
  platform = 's40' and
  app_launch_t >= 0 and app_launch_t <= 30000 and
  ts >= date('-1 day','start of day') and
  ts <= date('-1 day','start of day', ‘+1 day’) and
  app_version_1 >= 2 and app_version_2 >= 12 and app_version_3
  >= 11
group by device_name, bin
```

# Query language: motivation

- language for interactive use!
- concise
- familiar AND expressive == SQL-like
- short syntax for frequent operations, e.g. date selection, resolution for hierarchical fields, ranges and prefix matches for hierarchical fields, quantiles, ability to omit GROUP BY, etc
- custom features:  sampling, histograms
- extensible

# Query system: implementation

query compiler: Schema, Query -> C program

cc: compile, link with C runtime

run N instances in parallel on data stored in files

merge parallel results

write output JSON into a file

# Query compiler

- written in Erlang
- parsing using Piqi serialization system:
  - high-level declarative syntax definition
  - 1 line of code to parse into Erlang representation:

```
wam_piqi:parse_query(QueryString, 'piq',
    _Options = ['piq_frameless_input', 'piq_relaxed_parsing']
).
```

- analyzer and code generation
  - generates a tiny C program

# Query system: runtime

- C and C++ works – single threaded, short lived, no need to call free()

```
$ wc -l *.cc *.c | sort -n -r
    2639 total
     936 wam_query.c        // IO, output JSON formatting,
                               intermediate results serialization,
                               sampling, ...
     810 wam_hashtbl.cc     // std::unorderd_map works as well
     292 wam_hist.cc        // C array
     181 wam_ucount.cc      // hashtbl, HyperLogLog for high cardinalities
     132 wam_bins.cc
     105 wam_ntiles.cc      // TBD: approximate quantiles
      57 wam_random.cc      // std::uniform_int_distribution
      53 wam_orderby.cc     // std::sort(), std::partial_sort()
      41 wam_strtbl.cc      // hashtbl
      32 wam_groupby.cc     // hashtbl
```

- For COUNT DISTINCT use HyperLogLog (hint: there is a good open-source implementation in C)

- Most queries should be able to process input at 1-2GB/second (one thread)

# Query system: running

- single-threaded:

```
ls *.log *.log.lz4 | ./wam-read | queries/wam-query.1 > q1.json
```

- in parallel (using GNU parallel):

```
ls *.log *.log.lz4 | parallel --pipe -N1 -j4 --round-robin \
   './wam-read | queries/wam_query.29 -p - > queries/29/{#}.wmq' && \
./queries/wam_query.29 queries/29/1.wmq queries/29/2.wmq queries/29/3.wmq queries/
29/4.wmq > queries/29.json
```

lz4 -d

# API (protocol)

- JSON-over-HTTP API based on Piqi-RPC
    - define RPC protocol using a high-level definition language
    - on HTTP POST, have your Erlang callback called with JSON converted to Erlang term:

```
query(#wam_query{}) ->
  ...,
  #wam_query_output{...}.
```

# DB shell

- ssh with key-based authentication
- a program behind ssh:
  - read a line
  - make HTTP POST call
  - print output
- rlwrap: line editing, history, tab-completions

# Local storage: idea 1

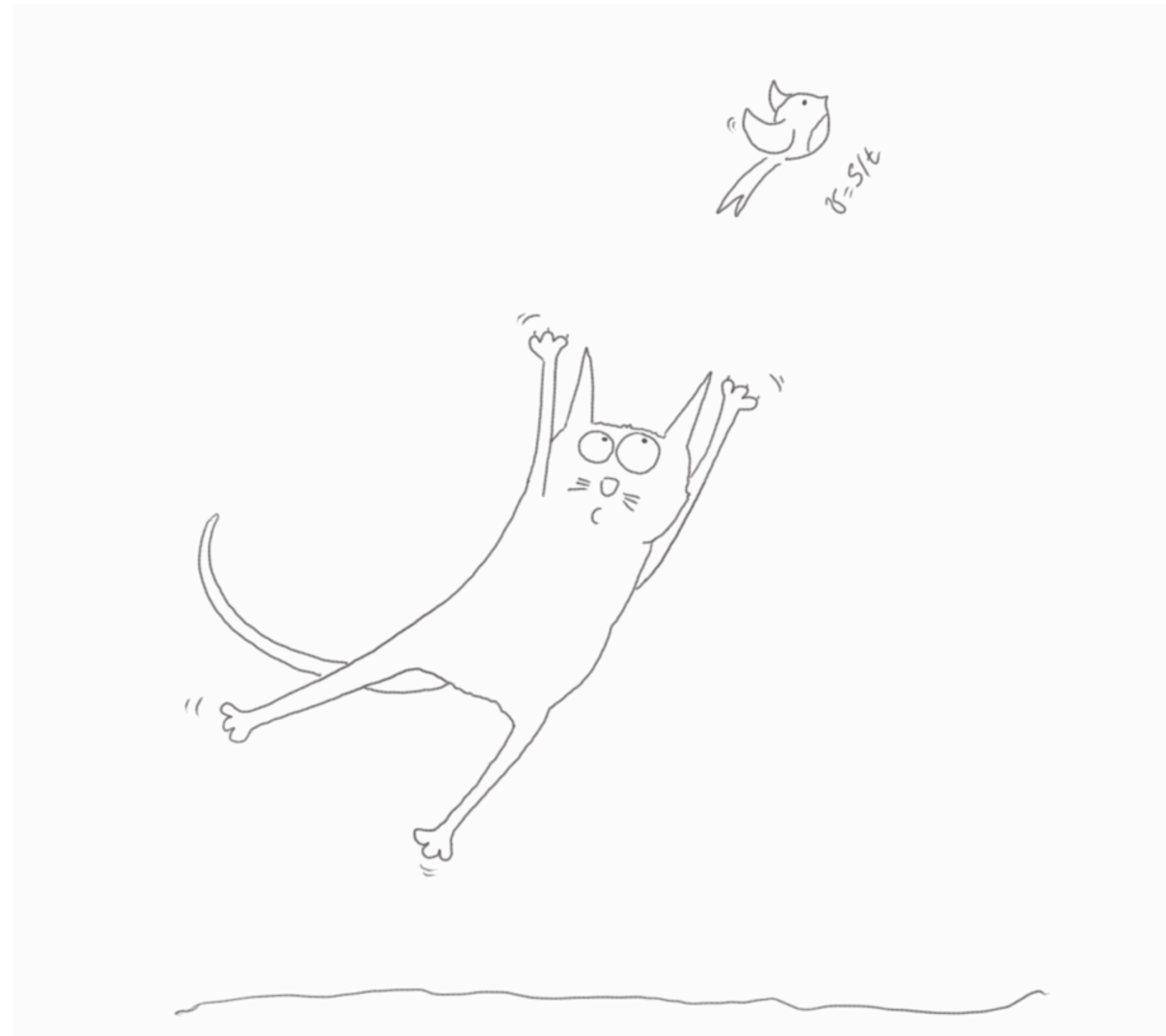
Dump all logs in one directory, e.g. /var/log

```
$ ls
06b00b613d4d2c7c63c4325bf4548c326dd064ad_1389052800-1389139199.log.lz4
07eab0ab576b86d97de0290f0a4e2681830896c8_1389052800-1389115415.log.lz4
0afef2097fcf79fce02fd38bb78495a8cbe3b0ec_1389052800-1389073599.log.lz4
0ca51f3ebdfcd232489df720c00eae908dd5fe19_1389052800-1389089835.log.lz4
0e496d31844713c43ac15ff1873146f17ef4e257_1389052800-1389139199.log.lz4
104f658ae6da49f68efe6874187d31a151e1a748_1389052800-1389139199.log.lz4
11e0a3b145629ecc342df4596549c8365086f67b_1389052800-1389103054.log.lz4
12fdb4b715947c51c9cfb0f000ae91c57c69d4f8_1389052800-1389139199.log.lz4
13e453a6a29518f50b4bf77d11af4534af048b51_1389052800-1389113173.log.lz4
19c3cc412b86e44ff8eb695b952e9ae4e1f62e9c_1389052800-1389086467.log.lz4
1ecc76cfa4581ca6874fd77124ac1eddb09fe931_1389052802-1389139199.log.lz4
2050deb665d6dd12b6237d921ff3da040e181132_1389052800-1389101425.log.lz4
```

1TB / 64MB per log = 32K files...

# Summary so far

- data model
- schema
- storage format
- query language, compiler, runtime
- API
- shell



# Local storage: use filesystem as index

Add some structure – e.g. partition your data by UTC days:

```
$ find .
./2014-01-07/a2ebef4ea9702f317fe593a06eab3792c057b542_1389052800-1389139199.log.lz4
./2014-01-07/a9666d376ece42e54348df134e6f111f373a8722_1389052800-1389119064.log.lz4
./2014-01-07/aa8b729c49cafe5e12edc989e9db9350bb5d4377_1389052800-1389139199.log.lz4
./2014-01-07/aae36826f4ba98a2a0f115a8de35447ddc7f3e04_1389052800-1389123949.log.lz4
./2014-01-07/ab6e300afee3e11b33024bb86c84e05b76755eae_1389052800-1389139199.log.lz4
```

# Local storage: application-level indexing

Add some more indexing (application specific), .e.g index by platform:

```
$ find .
./android/2014-01-07/06b00b613d4d2c7c63c4325bf4548c326dd064ad_1389052800-1389139199.log.lz4
./android/2014-01-07/07eab0ab576b86d97de0290f0a4e2681830896c8_1389052800-1389115415.log.lz4
./android/2014-01-07/0afef2097fcf79fce02fd38bb78495a8cbe3b0ec_1389052800-1389073599.log.lz4
./android/2014-01-07/0ca51f3ebdfcd232489df720c00eae908dd5fe19_1389052800-1389089835.log.lz4
./android/2014-01-07/0e496d31844713c43ac15ff1873146f17ef4e257_1389052800-1389139199.log.lz4

...

./s40/2014-01-07/ea5228abff04cd02164d5b7cf66cf66c936add9a_1389052800-1389139199.log.lz4
./s40/2014-01-07/ecbd7812186fa3fda180acf371c7722f0c23ac77_1389052800-1389067722.log.lz4
./s40/2014-01-07/f0efd941422e725926569e399cd92bbd041cb5a5_1389052800-1389139199.log.lz4
./s40/2014-01-07/f3f958055a598c307190c9e97bc3659ff3d8ecc0_1389052800-1389139199.log.lz4
```

Caveats:

- it is not automatic – the query system needs to understand this indexing structure
- you have to decide upfront

# Local storage: tables

Table level, e.g. for “fieldstats” and “props_log” tables:

```
$ find .
./fieldstats/android/2014-01-07/104f658ae6da49f68efe6874187d31a151e1a748_1389052800-1389139199.log.lz4
./fieldstats/android/2014-01-07/11e0a3b145629ecc342df4596549c8365086f67b_1389052800-1389103054.log.lz4

...

./props_log/android/2014-01-07/12fdb4b715947c51c9cfb0f000ae91c57c69d4f8_1389052800-1389139199.log.lz4
./props_log/android/2014-01-07/13e453a6a29518f50b4bf77d11af4534af048b51_1389052800-1389113173.log.lz4
./props_log/android/2014-01-07/19c3cc412b86e44ff8eb695b952e9ae4e1f62e9c_1389052800-1389086467.log.lz4
```

# Hash-based partitions as replication unit

In preparation of distributing data, partition your data by some key (e.g. user_id) or randomly:

```
./0/fieldstats/android/2014-01-07/...
./1/fieldstats/android/2014-01-07/...
./2/fieldstats/android/2014-01-07/...
...
./N/fieldstats/android/2014-01-07/...
```

# Distributed storage: simple approach

- Key idea: keep the number of partitions low and:

  #partitions (N) = 2 * #storage devices (D)

- Distribute partitions across storage devices
- Store association between partitions and storage devices in a config file, load config on all nodes
- This is all non-local distributed storage metadata you need
-

| partition | server, device |
|---|---|
| 0 | s1, d0 |
| 1 | s1, d0 |
| 2 | s1, d1 |
| 3 | s1, d1 |
| 4 | s2, d0 |
| 5 | s2, d0 |
| 6 | s2, d1 |
| 7 | s2, d1 |

Example: 8 partitions, 2 servers with 2 devices each

# Distributed storage: replication

- Add replicas to your partition -> storage device association
- Add as many replicas as your need: data is read-only

| | replica1 | replica2 |
|---|---|---|
| partition | server, device | server, device |
| 0 | s1, d0 | s2, d0 |
| 1 | s1, d0 | s2, d0 |
| 2 | s1, d1 | s2, d1 |
| 3 | s1, d1 | s2, d1 |
| 4 | s2, d0 | s1, d0 |
| 5 | s2, d0 | s1, d0 |
| 6 | s2, d1 | s1, d1 |
| 7 | s2, d1 | s1, d1 |

-

# Distributed storage: operation

- reconciliation, restoring replica counts: periodically rsync-push a partition to the configured replicas – see OpenStack Swift design for details
- expiration: expire locally – trivial rm -r <date directory>

# Distributed storage: expansion by doubling storage

- Step1: double storage capacity: D' = D * 2
- Step2: create a new config' with N' = N * 2 partition count
- Step3: keep the previous config! just change it to redistribute N old partitions to N = D' = D * 2 new devices
- Step4: start writing according to the new config
- Step5: for reads, use both configs for calculating cluster coverage
- Key points:
  - by keeping the old config, no need to reshuffle data across partitions
  - you always end up with uniform device utilization
  - conceptually similar to hash tables with monotonic keys: once once table is full, start writing to another one
- can't double? run out of partitions? skewed device utilization is OK as long as you have overprovisioned storage – will go away on its own once you expire old data
- there are alternative yet still rather simple distributed storage approaches, e.g. Disco Distributed File System

# UI and post-processing: tools

- Probably the most difficult part: HCI rather than technical
- IPython notebook – web-based interactive computing environment. The tool is fantastic, use it
- Great Python libraries but too low-level to be used by “casual” users:
  - Pandas – in-memory data post-processing and ad-hoc transformations: pivoting, slicing and dicing, time-series, ...
  - matplotlib – plotting
  - seaborn – aesthetics, 2d linear models, plotting

# UI and post-processing: essentials

Visualization:

- tabular
- bar plots, line plots
- histograms
- time-series

Post-processing:

- pivoting
- ordering
- look at Pandas for inspiration

# TODO: query system

- distributed queries
- materialized views
- columnar storage format
- fast approximate quantiles
- query results resampling (zooming in)
- more query language features, e.g. expressions

# TODO: UI and post-processing

- More interactivity:
  - brushing
  - tooltips
  - panning, zooming
- More high-level data visualization interfaces:
  - scatter plots
  - grid plots
  - more ways to visualize distributions, e.g. 2d histograms
- Machine learning for analyzing high-dimensional data, e.g. PCA, clustering, ...

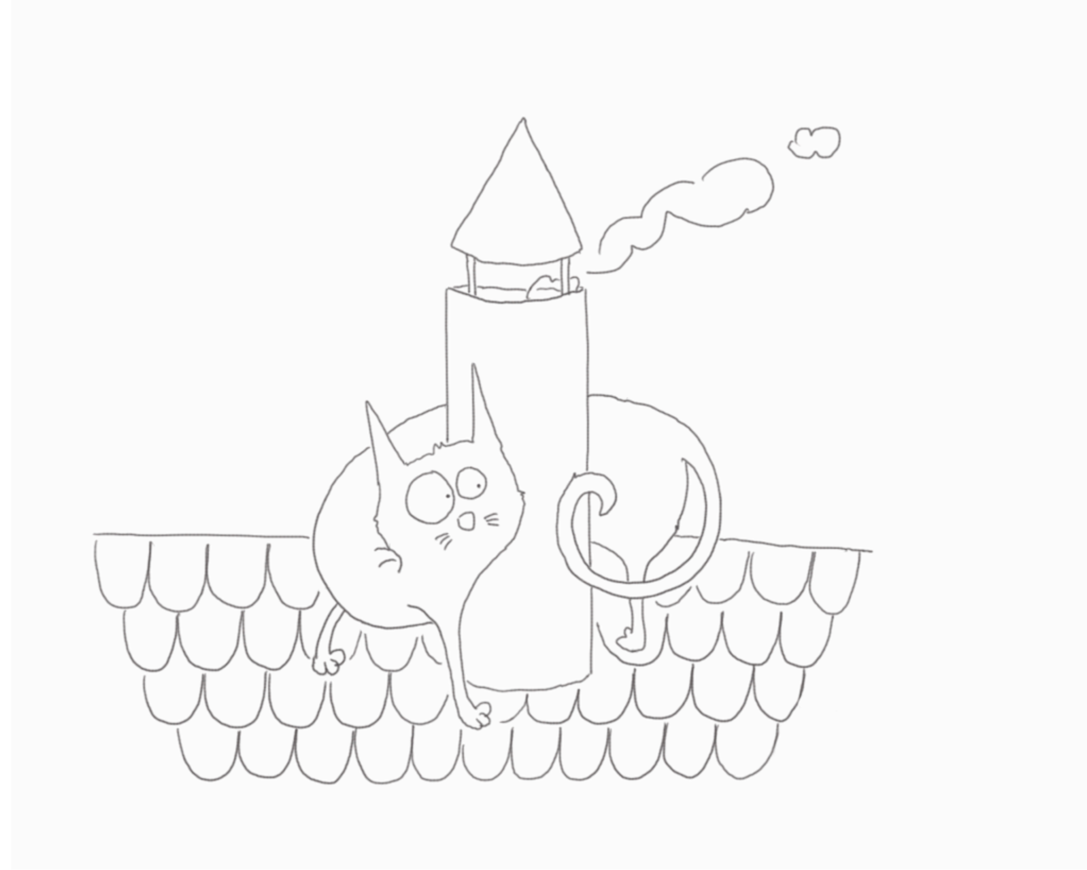
# References

Similar systems and sources of inspiration:

- Google Power Drill
- Facebook Scuba
- The Mature Optimization Handbook (relevant chapters)

Tools:

- Erlang – query compiler, super glue :)
- C, C++ – query runtime
- Piqi, Piqi-RPC – query language, API
- GNU Parallel – parallel job execution
- lz4 – compression
- ssh – remote access and authentication
- IPython Notebook, Python, pydata stack – interactive UI and post-processing
- HyperLogLog (https://github.com/armon/hlld)
- rsync – replication
- git – schema repository



Thank you!