Dumpster Dive your Erlang data!

Tom Szilagyi

Klarna AB

Erlang User Conference 2017

Thank you very much. Hello, everyone!

Scene One

You are the on-call person...

The phone rings at 1 AM — system crashed.

It is 1 AM at night. You are the on-call person. The phone wakes you up. A critical system at your company has just crashed. Now it is your job to fix it before the business opens at 8 AM.

Have you been in this situation? I want to see hands... my dear colleagues, of course... but not only them. I bet you'd like to avoid these events in the future?

Scene Two

I need to change something in the code.

I wonder if this field is always a 2-tuple?

Scene two. You are doing your day job, trying to implement some feature. You need to read a value from a certain field of a record stored in the database.

Now, there is **a lot** of data, and it has been accumulating since the dawn of time when this system started live operations. From what you can see by reading the code, this value you are looking at is always a 2-tuple. But think back to the first scenario: why did that happen? Well, my bet is someone gave the wrong answer to this question! So be careful, you might be planting a booby trap in your system.

Scene Three

I just fixed this bug in the code...

But are there any corrupt records in the live system?

Would be great to see when those were produced
and find their keys so we could fix them manually...

So you found there was a bug in your system, but you have no idea if it actually produced any faulty data. If yes, you want to know when the offending records were created, and what the keys of those records are.

Wouldn't it be great to have something that helps in these scenarios?

Where do these problems come from?

Dynamically typed language

- Supports experimental and bottom-up programming
- Enables seamless business exploration that leads to an organically growing system
- Required for the sake of hot code upgrades

```
{payment, 576}  →  {payment, 576, {pending, <0.1426>}}

        123456  →  {id, 123456}  →  [{id, 123456}, ...]
                                 →  [] ?
```


What are the underlying reasons of these problems? Why do they emerge in complex software systems, more or less as a matter of course?

First of all, Erlang is a dynamic language. It is also strongly typed, which means that each term has a well-defined type. There is a clear hierarchy of types in Erlang and you must always perform explicit conversions to go from one type to another, hence we call Erlang a strongly typed language.

But Erlang is also dynamically typed. Type checking happens at runtime and type errors surface as runtime exceptions. This might seem like a bad thing, and indeed a lot of people will argue that it makes your program more dangerous as opposed to software written in, say, Haskell where the compiler will catch type errors. I am mentioning Haskell because Erlang and Haskell are very close relatives. If you know both, then you know that Erlang and Haskell have basically the same functional core language. If you write sequential, purely functional code, you can go from one to another solely via slight syntactic changes. But Haskell has a static typing system on top of this core, and Erlang has concurrency with dynamic typing.

In fact, dynamic typing is the key to rapid, exploratory, bottom-up programming. The dynamic nature of Erlang makes it easy to plug in new data structures into existing algorithms, to add fields to tuples or to make tuples out of atomic data.

Why is Erlang dynamically typed? Why is Erlang not Haskell? Have you thought about the reason behind that? The reason is that Erlang needs to support **hot code loading** and doing that is impossible with a statically typed language. Think about it: what would be the type signature of a state upgrade function? A function that transforms the state of a server process from an old code version to a new version, that runs when the code is upgraded. Well, you **can not** define this type signature in advance, because when you write your original code version, you don't know what the next version will look like. The whole point of hot code upgrades is to adapt the system to new circumstances **while it is kept online**.

Where do these problems come from?

Highly available systems with lots of data

- On the Internet, you always want to be open for business
- NoSQL databases without a strict schema
  - ... because you want to be future-proof and extensible!
  - Document database: no enforced structure within stored objects
  - Record-based (e.g. Mnesia): each field can contain any valid Erlang term
- Schema changes are hard, esp. for large (> TB) always-online databases
  - Even if possible, we avoid them if we can!
  - Have an `'aux'` field? Great, just add a new `{Key, ...}` tuple!

So we try to keep our system always online. And an always online system also tends to gather and store lots of data.

We are nowadays very commonly using NoSQL-type databases, and this is especially true with Erlang systems. The lure of NoSQL is a bit similar to that of dynamic typing: building on a NoSQL data store makes it possible to quickly come up with a working system, and then keep bending and molding it as we go. No fuss with schemas, no relational wizardry, no data normalization. Just plug what you need to plug into the database, and get on with your life. (You might get other benefits, too, such as gaining availability by accepting eventual consistency, if that is your thing.)

But for the sake of our discussion, with a NoSQL database, the important point is that there is usually only a very lax schema in place. This is true with conventional key-value databases, such as Mnesia, and even more so with so-called document databases.

This is compounded by the fact that in complex systems with almost no provisioned downtime, making schema changes is decidedly hard. With storage in the terabyte range, we frequently opt against changing the schema even if we could do it in theory. We prefer to put new data in existing fields instead, extending them in ways that do not require an explicit schema change. For example, we add a new key-value tuple to a dict stored in a field called 'aux'. The operational costs and risks of doing that are much lower than touching the schema.

Where do these problems come from?

Long lived, evolving systems

- A system behind a growing business will run for many years
  - ... doing things it was never originally planned to support
- The problem domain changes and evolves with the business
  - ... making the original design look like a kludge
  - ... resulting in multiple formats of data, all of which must be supported
- Programmers come and go
  - It is very hard to keep things consistently documented
  - Code (and associated data) might end up owned by nobody
  - The original reasoning behind pieces of the design might get lost

They say that successful programs are those that outlive their originally planned lifecycle. Experience shows that they also tend to outlive their original architects and programmers. This poses another set of challenges in the world of successful business software systems.

There are a myriad different methodologies, agile and otherwise, that target this kind of problem. In practice, it is very, very hard to avoid getting hit by some of these. So we tend to live with their effects.

Where do I come from?

- Klarna is taking over the world
  - ... at least in e-commerce payment solutions
  - It takes a lot of data to take over the world!
- Our primary (original) system is built on Erlang/OTP
  - twelve years of development history
  - about 2 TB of live data
  - about 400,000 financial transactions per day
  - more than 400 million transactions since day one
  - about 60 million total end customers
  - about 40% market share in Sweden, 10% across all our markets

So: **Klarna**. I don't think I need to introduce Klarna. The company was founded on a fairly simple business idea, and we are now a major player in the world of e-commerce solutions.

As a Klarna engineer, I work with our single largest software system written in Erlang. This system is the mother of all business applications. It is the backend platform behind most of Klarna's financial services in our biggest markets, mainly the Nordics and Germany. This system grew organically from its first days as a prototype back in 2005 – it has handled Klarna's transactions starting from number one. You can see some current numbers on the slide. Pretty incredible growth for a single system, if you ask me. **It takes a lot of data to take over the world!**

However, the pain points I mentioned to you are from real experience. It has been my main occupation in the past few months to think about them.

It all started about a year ago, in May 2016, when I first-handedly experienced "Scene One."

On a beautiful night in May, 2016:

```
logic24:run aborted
Type=throw
Reason={logic24_phase_crashed,
        [{payment,
          {error,
           {badmatch,
            {aborted,
             {{badrecord,payer_info},
              [{payment_rec,iban,1,
                 [{file,...},{line,...}]},
               {refund_rule,
                 maybe_refund_details_available_policy_t,3,
                 [{file,...},{line,...}]},
               ...]}}},
           [{logic24_payment,'-process_payments/6-fun-3-',4,
             [{file,...},{line,...}]},
            {pulib_parallel,do_while,3,
             [{file,...},{line,...}]},
            ...]}},
```

In this case, something went wrong in our nightly run. This is a large batch job that does a lot: it handles payments and refunds, and closes bookkeeping for the day. Anything preventing it from successful completion is a major offender that needs immediate attention.

(next overlay)

Somewhere deep down, a piece of data that was **not** a #payer_info{} record was treated as such. This triggered a **badrecord** inside the transaction where the code was executing. The transaction was aborted, and so was the nightly run.

On a beautiful night in May, 2016:

New code, meet old data!

```
-module(payment_rec).
...
iban(#payment_rec{payer_info = PI}) ->
    PI#payer_info.bank_account.
```

got called with something like this:

```
payment_rec:iban(#payment_rec{payer_info = []}).
```


This is what happened, more or less.

The code was new. The data was not. Someone thought that the field called payer_info would certainly always contain an instance of the record #payer_info{}.

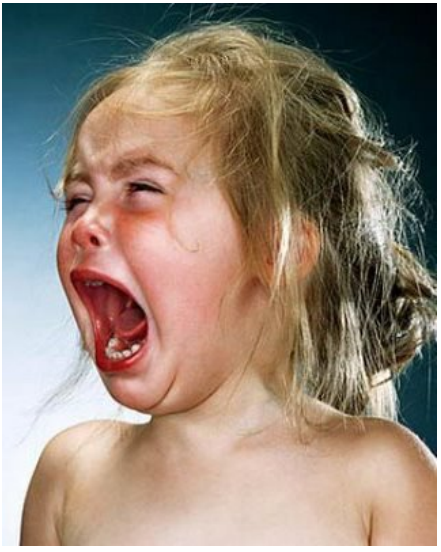
Logical, but naive.

Well, in this case it was trivial to analyze the problem and deploy a proper fix even in the middle of the night.

However, at the time my daughter was really small. She is still very little, just 15 months right now, but at the time she was a newborn baby, only about 10 weeks old.

On a beautiful night in May, 2016:

I just learned what Daddy does for a living!



And she was not happy.

So I started thinking about this problem. What could we do?

How could we have more insight into our data?

DUMPSTER DIVE

your Erlang data!

My proposal is to actually **take a look at the data**.

I know, it does not sound like a revolutionary idea.

But it works.

Dumpster diving:

"Because you never know what you might see"

- Ad-hoc open-ended data inspection
- Discover the type structure of data
  - Categorize alternatives
  - Count occurrences
  - Estimate cardinalities
  - Collect samples
- A human-machine hybrid approach
  - Machine: crunch through the pile
  - Human: "drive" and make sense of it

Why do I call this "dumpster diving?"

Well, it's a way of life... :)

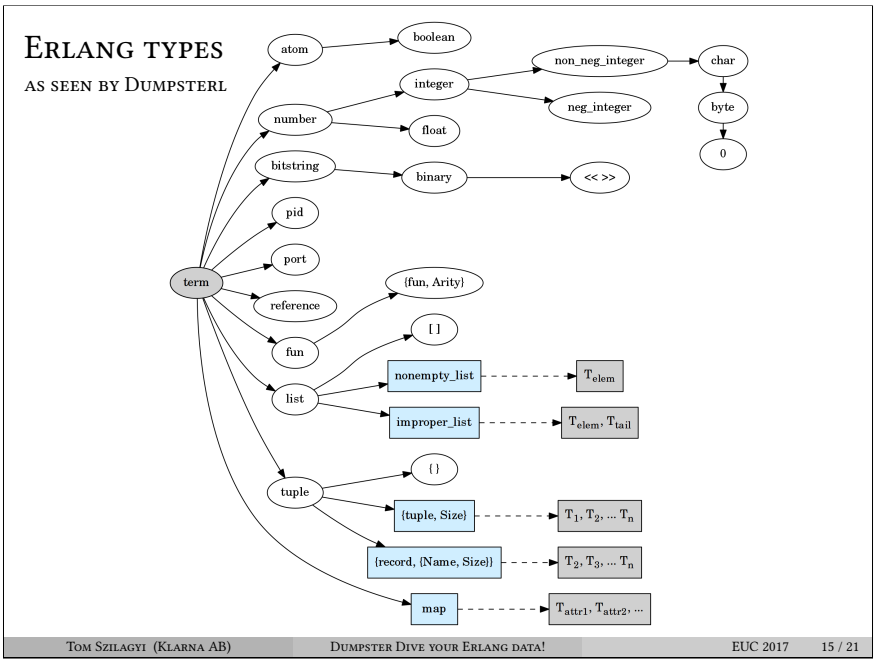
But seriously, I think it nicely fits the nature of the job. You don't exactly know what you are looking for. The task is fairly open-ended.

How do we approach this task?

Agent Smith said: "Never send a human to do the machine's job!" So we need a tool, a machine. What is the machine's job? The heavy lifting. To crunch through the heaps and piles of data, and do whatever sorting and aggregation is feasible to help us make sense of it. It is also the machine's job to present the results in digestible form. All of this is done by **dumpsterl**, which is the subject of this talk.

On the other hand, the human is still needed. It is us humans who possess the powers of comprehension. It is us who need to make sense of what we see. The machine will help us, but we need to stay in the driver's seat.

All right, enough talk. I think it's time for you to see a demo of dumpsterl.



Now that you've seen dumpsterl in action, let's talk about how it works. In particular, I will speak about the **type system** of dumpsterl, and how it is implemented. This is the core of dumpsterl and the only essential topic that you need to understand to be an enlightened user.

Dumpsterl implements **predicate typing**. This simply means that for each term considered, it follows a decision tree of predicate functions to arrive at the most specific type that fits the term. The graph shows this decision tree but not the predicate functions.

The boxes marked with light blue are **generic types**. They are types that are parameterized by further types. We call these parameters **attributes**. These are shown in the grey boxes connected to them. For example, the list type has a single attribute, specifying the type of the items of the list. Other generic types have other attributes. Each generic type has a certain "attribute signature" that specifies the number and semantics of all the attributes.

From a strict type hierarchy point of view, generic types are leaf nodes, since their children belong to a different type domain. They do not specify the type of the whole term, rather they specify the type of an attribute. The attribute's type spec is again rooted at the node **term**, so the tree is recursive.

Kinds of types

Kinds, a.k.a. "meta-types"

- **generic**
  - Parameterized by further types
  - list, tuple, record, map
- **union**
  - Cannot be described with a single Erlang type, only as a union.
  - Each actual term belongs to exactly one of its subtypes.
- **ordinary**
  - Can have subtypes, but a term might not belong to any of them.
  - `non_neg_integer()` → `char()` → `byte()`

Each type belongs to one of three kinds, or meta-types.

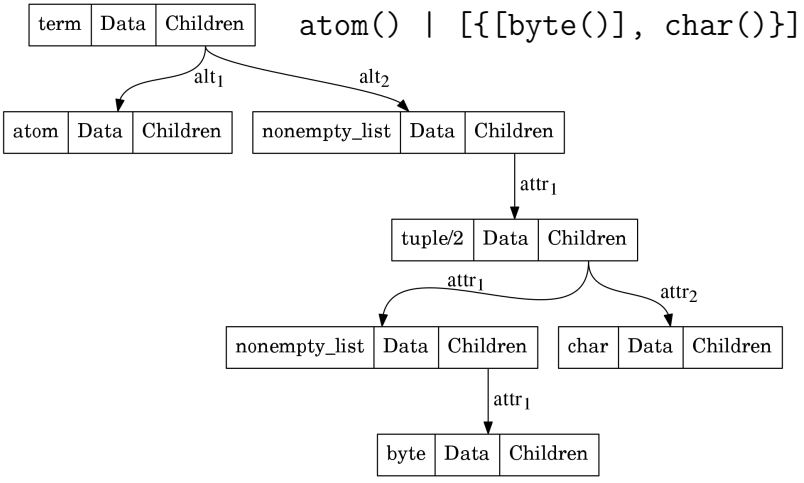
A **generic** type is parameterized by further types, called attributes. A list is really a list of something, a list of its elements, so the attribute of a list is the type of its elements. Generic types include lists, tuples, records and maps.

A **union** type is, by its nature, a union of subtypes. Such a type does not fit into a single node of the type hierarchy, it can only be represented as a union of types. Each actual term belongs to exactly one of the subtypes, which are mutually exclusive.

An **ordinary** type is similar to a union, but it may or may not have further sub-types. Any actual data term either belongs to one of those sub-types, or, if neither of their predicates is true, is absorbed by this type.

The Spec

Tree nodes: `{Class, Data, Children}`



All the information is represented by this data structure called **spec** that is central to dumpsterl.

As the probe goes through the terms, each one is added to the spec. The spec is a tree structure that mirrors the type hierarchy. The tree is formed by nodes that are 3-tuples. Each tree node is decorated with meta-data stored in the `Data` field, more on that later.

For types that are not generic, the child nodes correspond to subtypes. These are marked with **alt** ("alternative"). The children of generic types specify the types of each attribute. The list of attributes is specific to each generic type. For example, a list has one attribute: the type of its items. A tuple has an attribute for each field. These are marked with **attr** ("attribute").

Data in the Spec

Each tree node's `Data` contains:

- Count of terms absorbed
- A statistical sampling of the terms seen
  - each value is either always sampled, or never
  - so we can infer statistics of their properties
- A cardinality estimator
  - Hyperloglog: very efficient algorithm
  - estimate the number of unique values with a few % error
- "Points of interest"
  - interesting points of the term's value domain
  - currently only `min` and `max`
  - for each point, store value, count and key-timestamp range
- Type-specific extra data
  - record attributes
  - list length and binary size histogram data

In each type node of the spec tree, we maintain a bunch of statistics about the data at that point. We've seen this in the demo, so just a quick recap.

First of all, we count the terms, so we can report the occurrences at all levels and for all alternatives.

We also have a sampling of terms. The implementation ensures that the samples are statistically representative.

The cardinality estimate is very useful. We can tell if a certain field has a unique value in each term, or only takes a limited set of values. We implement the Hyperloglog algorithm, which is the state of the art in cardinality estimation.

Dumpsterl tries to present the interesting stuff, and the edges of the value domain are definitely interesting. These are implemented with possible future extension in mind; currently limited to `min` and `max` by value.

Finally, there is extra data that is specific to each type. For example, that is where the record attributes discovered by the probe are stored in the spec.

Design Goals and Features

- Handle highly complex data
- Handle large amounts of data
  - Hyperloglog scales up to 1G records (32 bit hash)
  - The rest of the spec has no known scalability issue
  - 100M records should be comfortable; < 3 hours @ 10k records/sec
  - parallelized probe execution
- Support common Erlang database formats
  - mnesia, ets, dets (.DAT), disk_log (.DCD)
  - Easy to extend — just provide an iterator!
- Non-intrusive, minimal impact
  - No registered process, no ets table
  - Can be loaded into the host node from `/tmp`
- Keep it simple, clear and easy to use!
  - Detached GUI, clear API

I have been collecting my ideas for the past several months, and spent the last two months coding the initial release of dumpsterl. I can proudly state that the initial goals have been achieved.

We've seen that dumpsterl handles highly complex, non-uniform data. No surprises here, that is why it was created in the first place. There is no limit on data complexity, apart from your RAM.

There is no theoretical limit to the amount of data you can process with dumpsterl, apart from your ability to wait for the results. However, the hyperloglog algorithm relies on a 32 bit hash function. Due to collisions, the precision of cardinality estimates will slightly degrade above one billion unique values. But the rest of the spec remains fully accurate. Regarding speed, it should be "comfortable" to process a hundred million records within the day. In fact, with a moderately large amount of cores, you should be able to do it under three hours.

I haven't shown this in the demo, but you can read raw mnesia table data with dumpsterl. Raw table file access comes with a few caveats (see the Dumpsterl User Guide), but it is the fastest method.

It is very easy to add support for new data sources, you basically only need to provide the functions to iterate through the data. If you can iterate it, dumpsterl can process it!

Finally, dumpsterl was designed to be simple; as simple as possible, but not simpler. I would like to keep it that way.

Dumpsterl is out!

Released as open source at:

- `https://github.com/tomszilagyi/dumpsterl`
- `https://tomszilagyi.github.io/dumpsterl`

"In God we Trust,

ALL OTHERS BRING DATA!

I hope you are as excited about dumpsterl as I am! The project is open sourced under the same license as Erlang itself, the Apache 2.0 license.

The source is on Github, and there is a modest site dedicated to dumpsterl under my personal webpage, hosting all documentation.

The project is currently considered to be in alpha status, mainly because there is still so much stuff to explore that I feel this is only the beginning. And also because some things might change a bit in the future. But as I showed you in the demo, dumpsterl is in fact quite usable today.

So please check it out and start using it! I would love to hear your feedback. So feel free to open issues and pull requests via GitHub.

(next overlay)

I am leaving you with this famous quote. I hope dumpsterl will help you with that.

If you can take away only one thing from this talk, I recommend the following thought: **Never assume that which you are able to know.**

Thank you. I am now happy to take questions.

Random Fun Facts

- Whatever happened to `pos_integer()`?
  - Dropped in favour of `non_neg_integer()`, `char()`, `byte()` and `0`
- The spec can be converted to Erlang abstract format
  - ... so we can use Erlang's pretty-printer to print our type specs!
- The GUI uses a zipper tree to navigate the spec
- The hash function needed an upgrade
  - 32 bit hash needed for the Hyperloglog algorithm
  - Initial:

    ```
    Hash = erlang:phash2(Term, 1 bsl 32).
    ```

    → 600k unique refs estimated as 200k (collisions!)
  - Only marginally slower, but much better:

    ```
    <<Hash:32, _/binary>> =
    crypto:hash(md4, term_to_binary(Term)).
    ```


So what happened to the `pos_integer()` type? Why is it missing from the type hierarchy? I wanted to keep `non_neg_integer()`, `char()`, `byte()` and `0`. `pos_integer()` does not fit in this scheme, because `char()` and `byte()` contain `0` and `pos_integer()` does not. So I decided to drop `pos_integer()` to keep a clean and consistent type system.

The spec is a complete specification of the type hierarchy. It is quite natural that it can be converted to an Erlang abstract form for the equivalent type definition. That is indeed what happens when dumpsterl prints the type definitions near the top of the report. The spec is converted to Erlang abstract form, which is fed into the Erlang pretty-printer.

If you don't know what a zipper tree is, I recommend that you look it up and study it. It is a very cool and useful technique in pure functional programming. Mastering it will make you a better programmer. I mention this only because the GUI is built on a zipper tree, that is why the spec tree can be seamlessly navigated.

The statistical sampler and the cardinality estimator both rely on hashing the incoming terms to 32 bits. In the beginning, I used `erlang:phash2` but that proved to be lacking with large amounts of data. Due to collisions, it made the hyperloglog algorithm severely underestimate the cardinality of a known set, one with guaranteed-to-be-unique elements. What I found interesting is that the replacement I found, while much better, is almost just as fast.