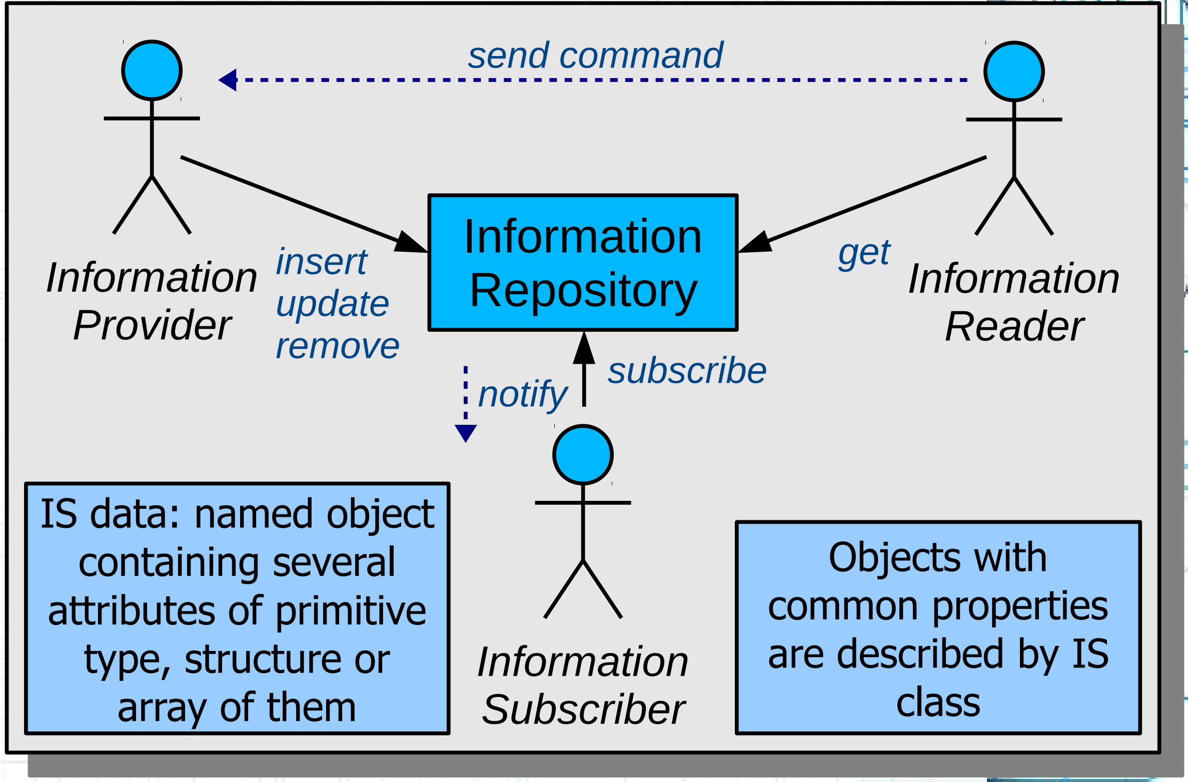
The evolution of Persistent Back-End for the ATLAS information System of TDAQ (P-BEAST)

Igor Soloviev, UCI

IS Introduction

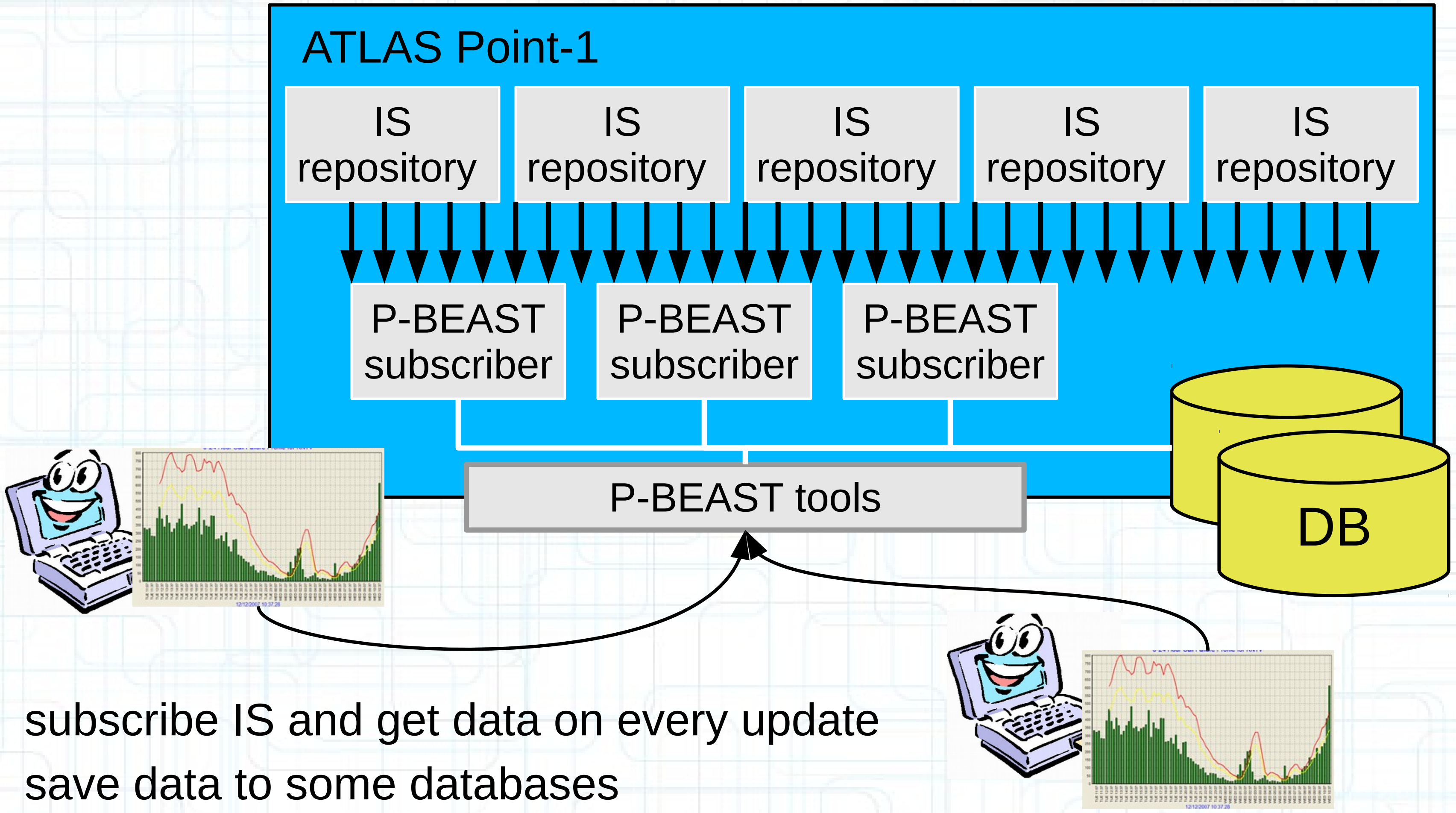
- In ATLAS online world the Information Service (IS) provides a means of sharing information between software applications in the distributed environment
- Is widely used for online operational monitoring by DAQ, trigger and detectors
- Information is not persistent and exists while service is running


send command
Information Provider
insert
update
remove
Information Repository
get
Information Reader
notify
subscribe
Information Subscriber
IS data: named object containing several attributes of primitive type, structure or array of them
Objects with common properties are described by IS class

P-BEAST Requirements

- Main use cases: post mortem analysis of IS data by experts and online dashboards for shifters
- Archive *any* type of raw IS data including various numeric types, strings, date/time, structures and arrays of them
- Support schema evolution (i.e. changes of IS classes)
- Satisfy challenging data input rates and volumes (avoid back-pressure on IS)
  - O(10^2) IS information repositories (servers)
  - O(10^6) variables (attributes) of O(10^5) objects
  - up to 1 Hz refresh rate for some objects, need to absorb peaks of the insertion rates
- Provide API to access archived data
- Provide graphical visualization tools for archived data
- The interfaces need to be available on ATLAS experiment area (P1) and GPN

P-BEAST High Level Design



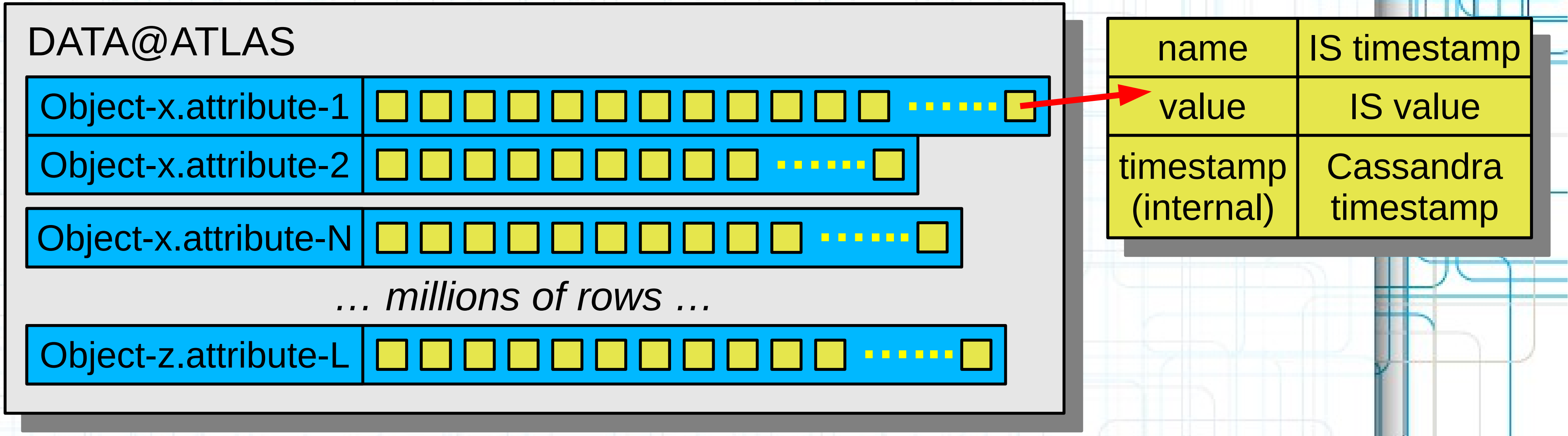
- subscribe IS and get data on every update
- save data to some databases
- make data available inside P1 and on GPN
- provide visualisation tools

First Prototype (2009-2013)

- Considered several approaches. The NoSQL technology (*Cassandra*, *Hbase* and *MongoDB*) and time-series oriented tools (*RRDTool*, *Graphite*, *OpenTSDB* and *KDB*) looked like most suitable candidates.
- Performed evaluation https://cds.cern.ch/record/1402973 and selected Cassandra:
  - designed to handle high write load from multiple clients and can absorb peaks in the write rates
  - scale horizontally, easy to increase performance by just adding more nodes (scales linearly assuming that data keys are randomly distributed)
  - schemaless allowing evolution of IS classes
  - data model naturally supports time-series data (timestamp as key in key:data pairs), no loss of data granularity in time
  - easy deployment (fast prototyping)
  - low maintenance requirements
  - high data availability (replication, no single point of data failure)

IS Data in Cassandra

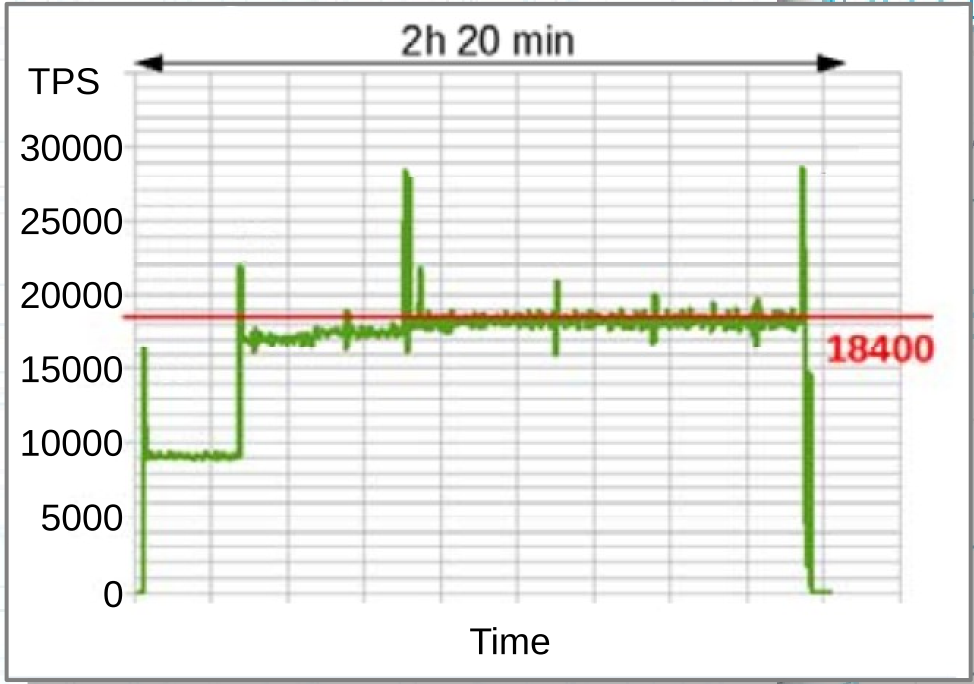
- Create DATA column family (table) per run config (ATLAS, initial)
- Concatenate IS class/object ID with attribute name to identify row in above column family
- Each row contains *name:value* pairs, where the *name* is data modification timestamp and the *value* is a value of attribute data



- Create few extra column families to describe attributes and objects per IS class, etc. to be able to perform data queries

Cassandra HW and Performance Results

- Evaluated several models for deployment. Finally installed P-BEAST on 3 powerful (year 2012) nodes on Point-1 (dual CPU Xeon X5650 @ 2.67GHz, 24 GB RAM, 4x1TB RAID)
- Use data replication factor 2 (and raid 0)
- Subscribe subset of *DAQ* IS servers and classes (below 1/4 of total IS information)
- 6 MB/s writing aggregate performance (0.5 TB / day!) and 18K updates per second
- Introduce 5% data smoothing (skip "similar" numeric data) to reduce writing to 0.8 MB/s



Problems with Cassandra

- Significant changes between major versions, some bugs
- For our case the data storage cannot keep more than two months of raw data, so need to remove old data (move to EOS) and provide yet another mechanism to access them
  - i.e. can only use Cassandra as an intermediate buffer
- Problems with data compaction (for removal and maintenance)
  - compaction requires 50% of free disk space (i.e. use <50%) in addition to inefficient storage (e.g. extra 8 bytes for every data point)
  - compaction has problems (may never finish with random errors even on powerful node), if data row is large (above 60 MB); querying of such data also is very inefficient => no vertical scaling
- Forced to redesign DB schema adding time buckets to row ID (one row keeps data for few hours)

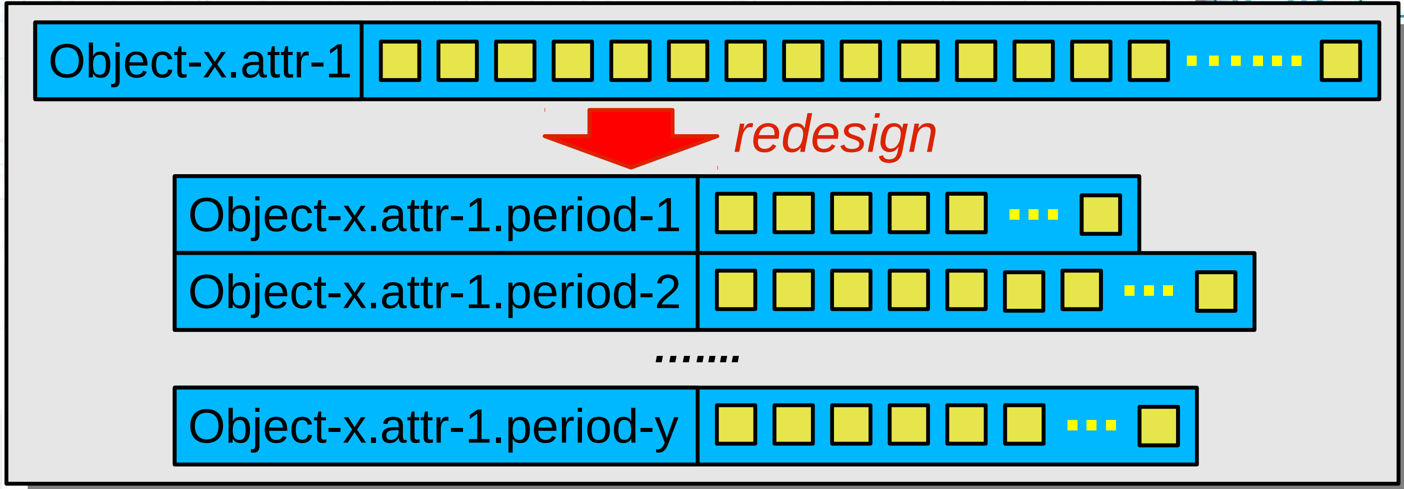
raw product

lacking required self-sufficiency

space inefficient

append-only design

problems only become visible during real use



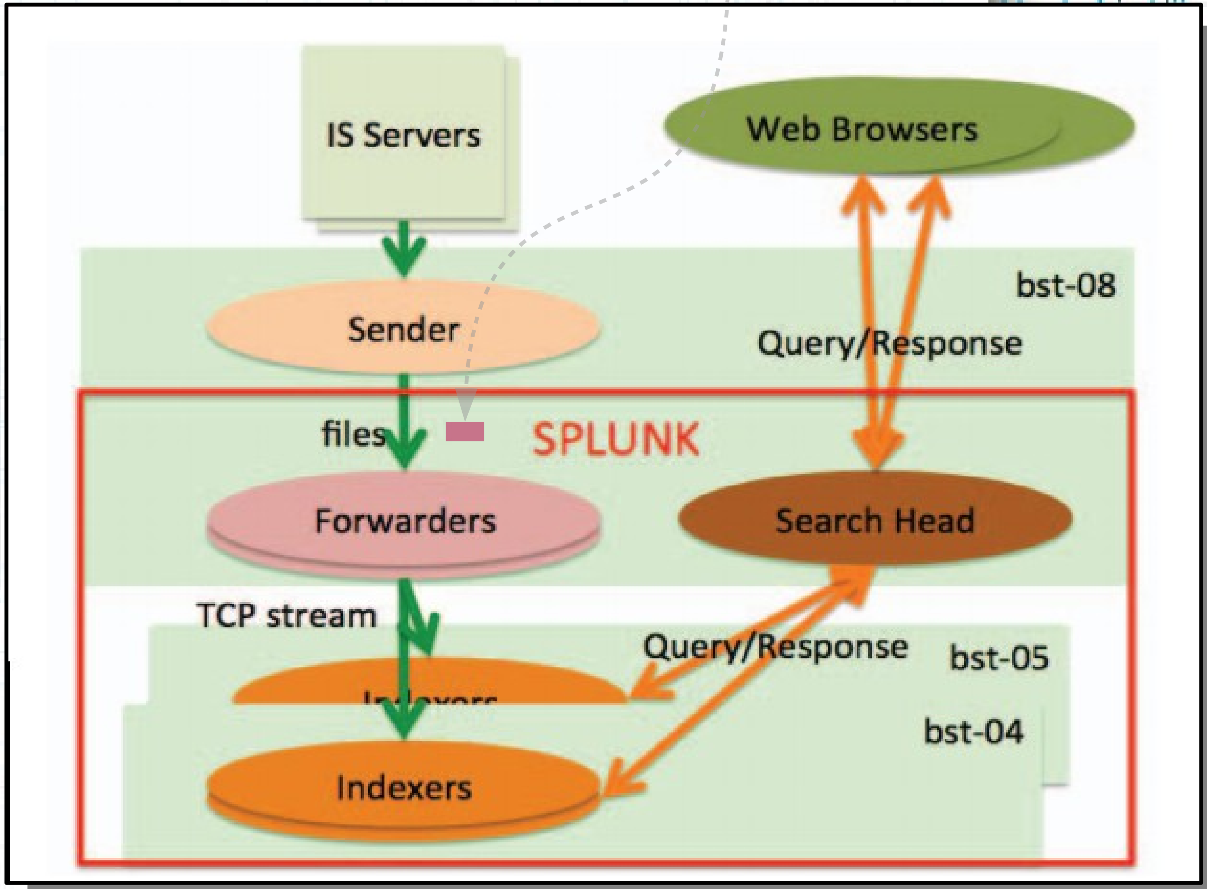
Splunk - Data Storage and Visualisation Prototype (2013)

- Cassandra does not provide any data visualization tools
- Splunk was advertised by IT and was evaluated as both the P-BEAST Web visualisation back-end and the database itself
  - http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7097473
- Splunk:
  - is a search engine for machine data (log files, configuration, monitoring metrics, etc.)
  - collects and indexes data to enable:
    Searching • Reporting • Correlating • Trending • Alerting
  - has been built to scale (up to 100 GB / node / day)
  - provides searches (>120 commands for data manipulation)
  - allows to configure Web dashboards
- Splunk is a commercial product with limited free license

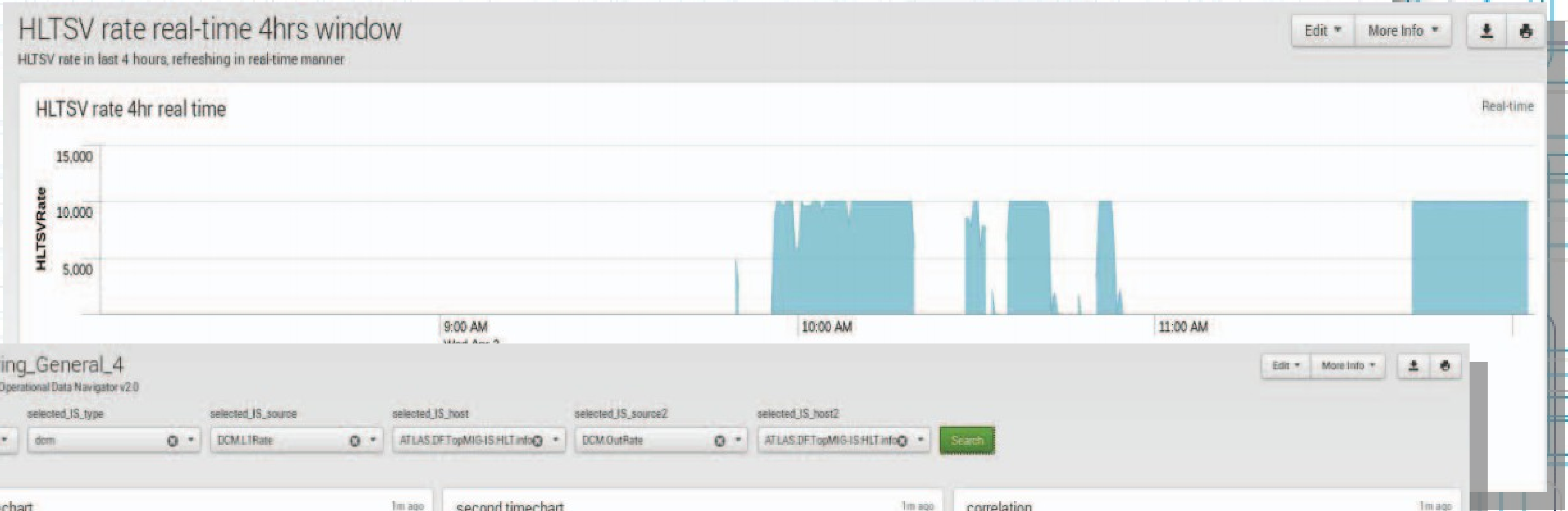
Splunk and P-BEAST Architecture

- Forwarders, Search Head and Indexers are components of Splunk running on different hosts for maximum performance
- P-BEAST receiver gets IS data and stores them into a file (stream) in a text format optimised for Splunk
- The text format uses ~200 bytes per IS attribute value
- The goal of evaluation was to find optimal configuration of Splunk components from performance and disk utilization points of view

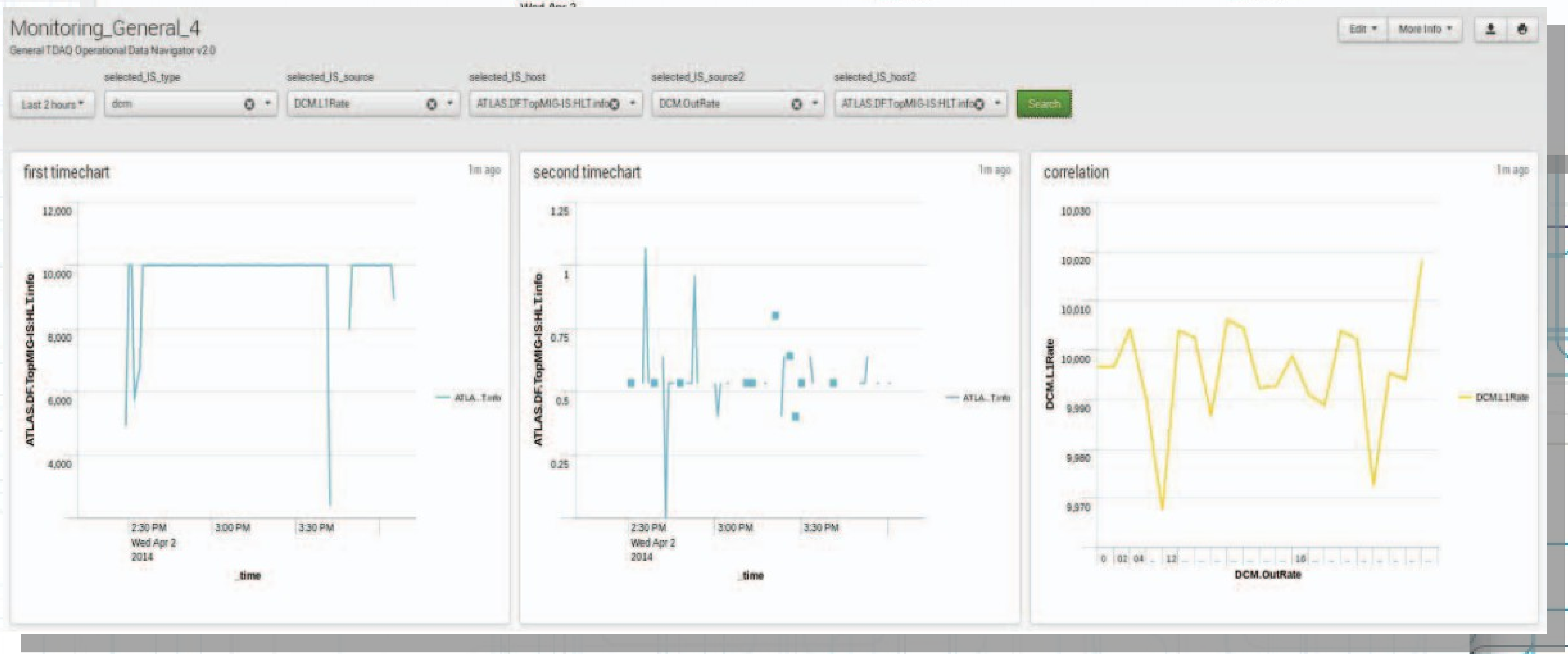
```
***SPLUNK*** sourcetype=is index=HLTSV
host=TDAQ.DF.HLTSV.Events source=HLTSV.LVL1Events
timestamp=1395131133784309 value=505358093
```



Splunk-based P-BEAST Dashboard Examples


HLTSV rate real-time 4hrs window

HLTSV rate in last 4 hours, refreshing in real-time manner

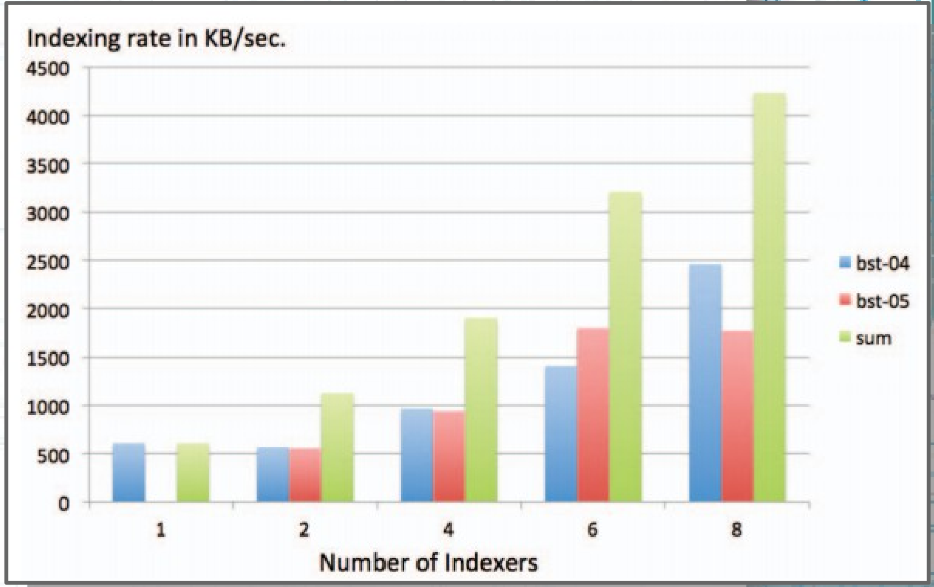

Monitoring_General_4

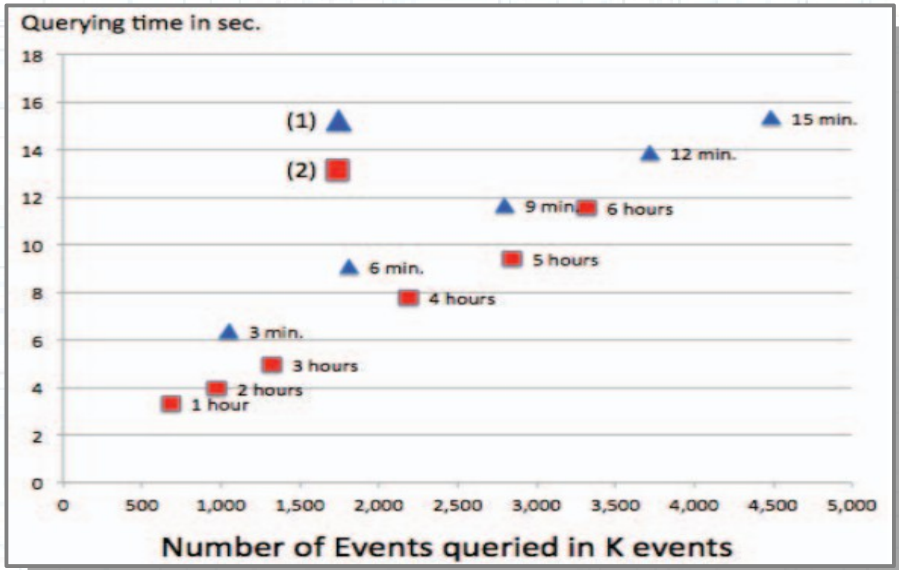
General TDAQ Operational Data Navigator v2.0

selected_IS_type	selected_IS_source	selected_IS_host	selected_IS_source2	selected_IS_host2
dcm	DCM.L1Rate	ATLAS.DF.TopMIG-IS:HLT.info	DCM.OutRate	ATLAS.DF.TopMIG-IS:HLT.info

Splunk Performance

- When inserted to Splunk, the input text is internally compressed in ~5 times
- Insertion rate scales proportionally to number of indexers and reaches 2 MB/s (~10000 IS updates/s) per single node (*measure results on the HW used for Cassandra deployment, see slide 7*)
- In accordance with Splunk experts, it will be better, if real-time (non historical) data are inserted





- Querying performance depends on many indexing parameters and in general case allows to scan about 100,000 values per second and per indexer node
- Insertion of realistic data for a large time interval on request (weeks, months) results big latency (minutes) before data become visible in browser

Problems with Splunk

- Large overhead when store data
  - Original non-optimized text format goes into internal Splunk database; little user control on internal database structure
  - Even compressed, the overhead is ~5 times vs. raw binary data
- Impossibility to delete old data by time period
  - If we cannot keep all data, we will need to remove old ones
- Inefficient graphical interface for large datasets
  - Downsampling is performed on client side (i.e. by JavaScript running inside user's Web browser). Takes minutes to display result containing O(10^6) or more

space inefficient

append-only architecture

lacking required self-sufficiency

slow GUI

Motivations for development of new P-BEAST (2013)

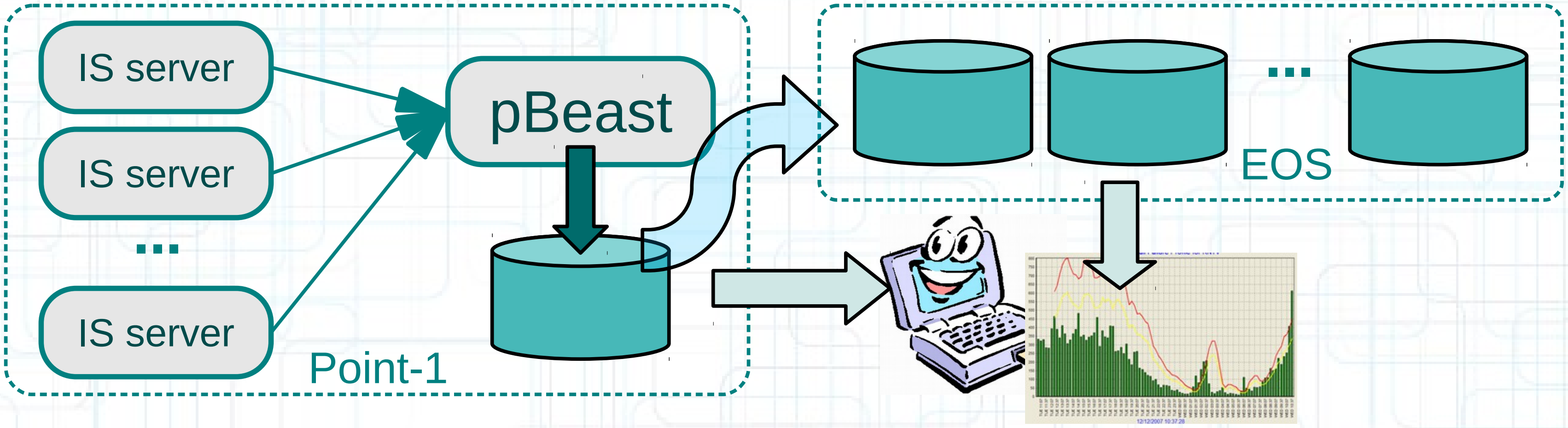
- Using available HW (3 nodes, 12 TB) we can build a system to archive small fraction of IS data (<1/4) using Cassandra and Splunk
  - To be able to get all data from IS we need an order of 10-20 powerful nodes
  - To keep all IS data for Run 2 we need more than 500 TB of disk space
- If all data cannot be kept on our storage, with Cassandra we can export them to EOS / Castor, but then we need to develop extra software to access the data, e.g. re-import when needed, or use some other DB tools

Disclaimer

Above conclusions are based on the state of the Cassandra and Splunk in 2013. They may be significantly improved since then.

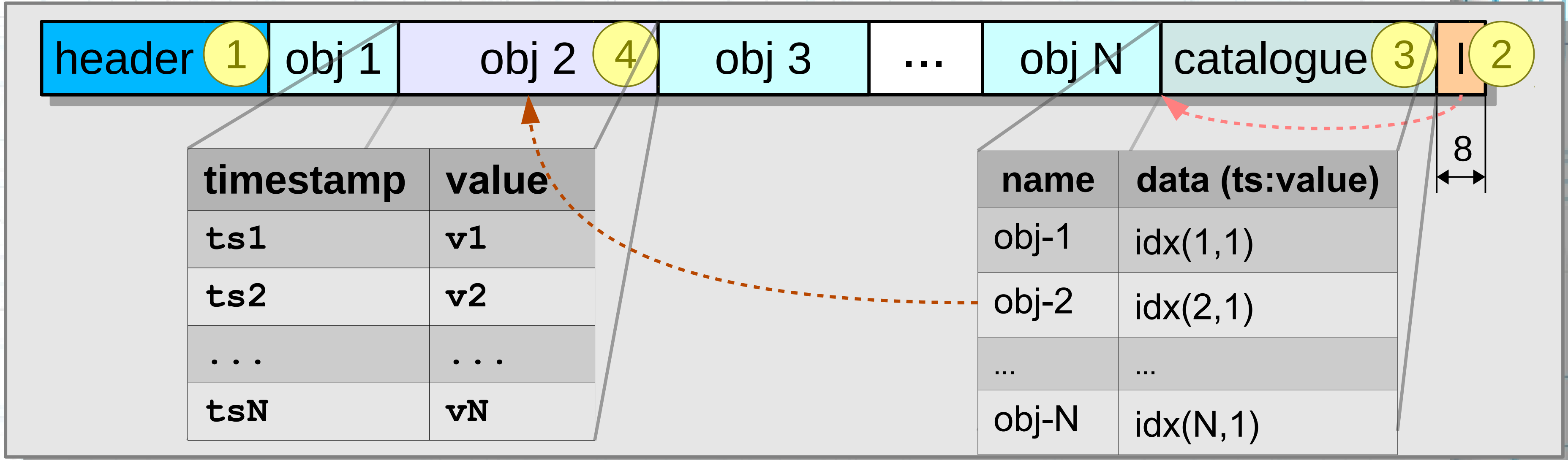
Ideas About New P-BEAST DB

- We need something working and do not like to restart evaluation cycle without 100% confidence. Try to make own simple solution.
- Get IS data and save into some files
  - Use language and platform neutral format for data serialization
  - Read/write efficiency (binary format parsing can be 100 times faster and several times smaller than text; random access to data inside file is mandatory requirement for remote file systems)
  - Moderate number of files is tolerated (O(10^4) per directory is OK to keep good read performance using *ext4* file system)
- Export such files on EOS for long-term storage
- Provide interfaces for data retrieval and visualisation



New P-BEAST DB Files

- New P-BEAST file storage is based on low-level primitives of Google Protocol Buffers (see addendum) with data compaction (extend IoV for unchanged data, binary format, integer *variants*) and compression (zip)
- Create files per time buckets and arrange by IS classes and attributes
- Support any IS data including arrays / nested types and end-of-validity (none implemented in Cassandra/Splunk); support schema evolution
- Microseconds precision for timestamps relative to base value in header
- Write at once, sequential data access (no file size limits), zip per object
- Random read access (see yellow numbers for order), efficient on EOS

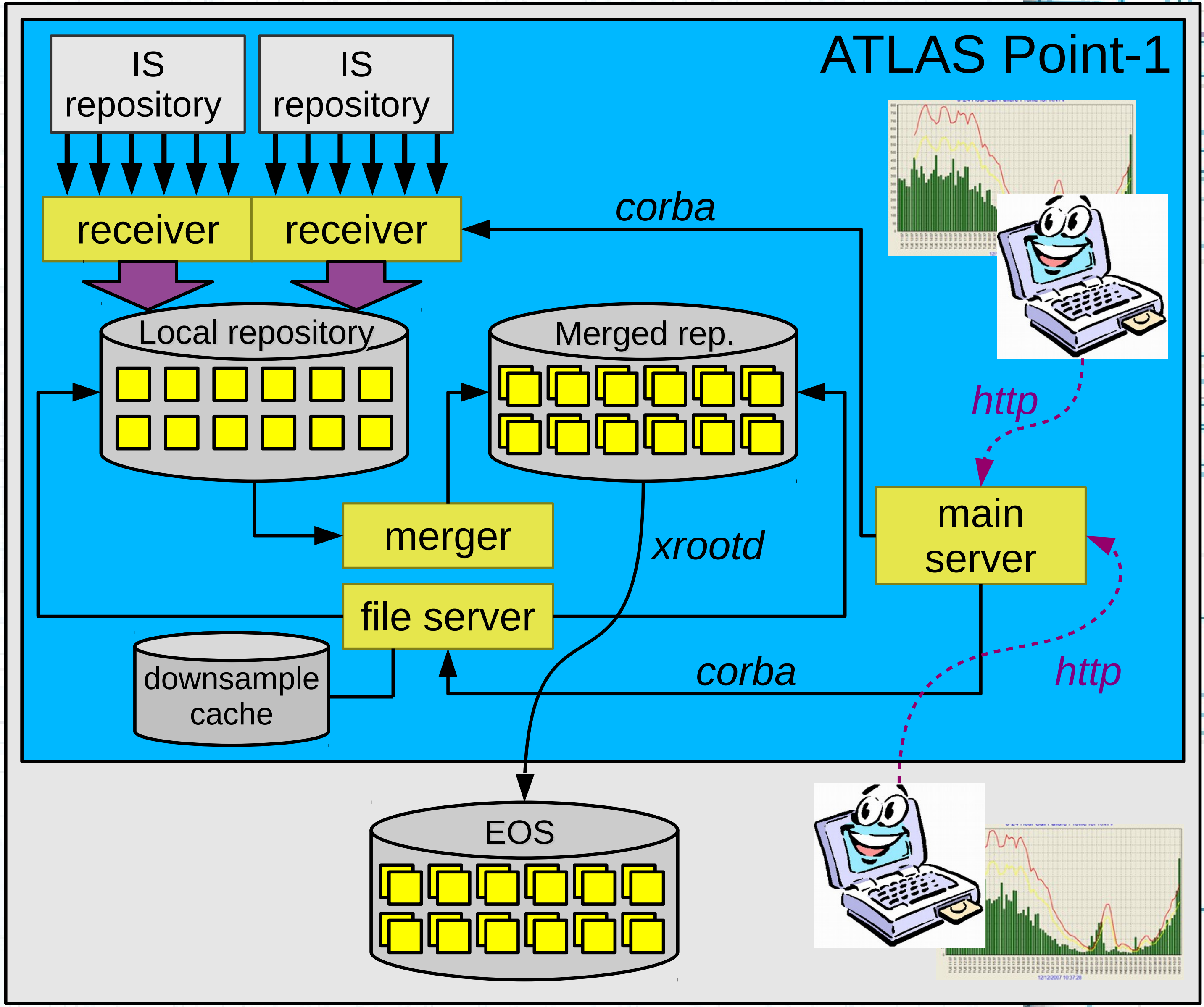


timestamp	value
ts1	v1
ts2	v2
...	...
tsN	vN

name	data (ts:value)
obj-1	idx(1,1)
obj-2	idx(2,1)
…	…
obj-N	idx(N,1)

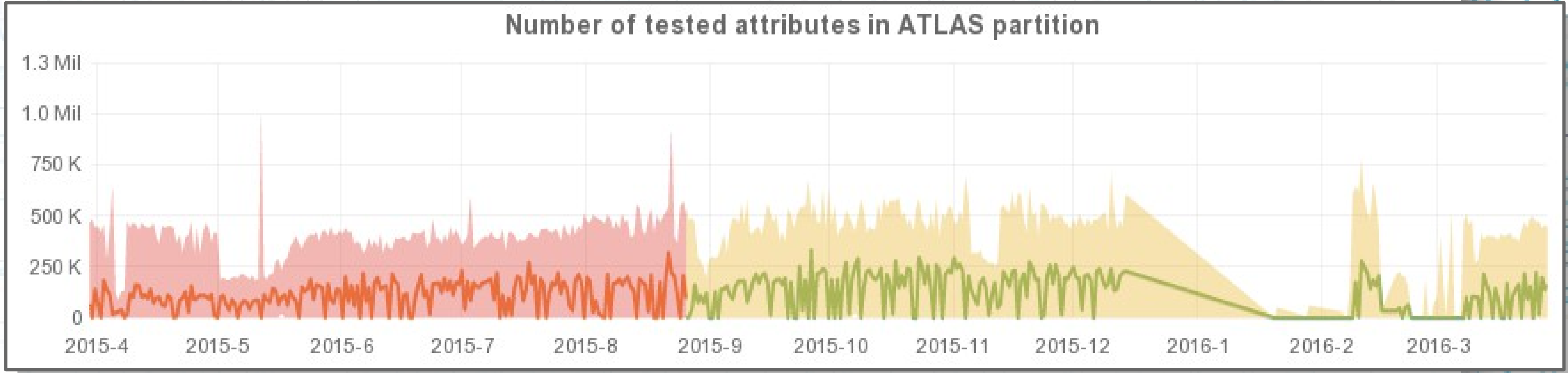
New P-BEAST Data Flow

- The receivers get IS data, keep in-memory, reorder, compact and store to local repository containing many small files
- The merger combines small files above certain age and stores to merged repository
- The file server accesses repository data; performs downsampling and keep results in cache
- The main server accepts client requests via *CORBA* and *REST* API; it talks with receivers and file servers and combines their responses to answer user
- The files appeared in merged repository are synchronized with EOS

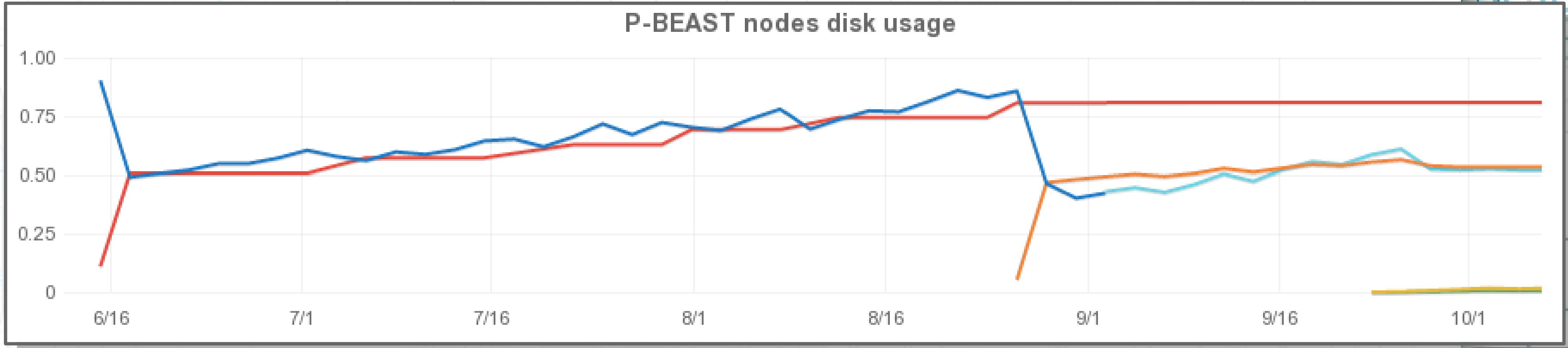


P-BEAST DB Performance on old HW (until 2015-09)

- experienced little problems with insufficient RAM, when archived all IS data from ATLAS and other sources before found optimal configuration
- sustain IS system 500 KHz IS attributes refresh rates (~100K writes/s)

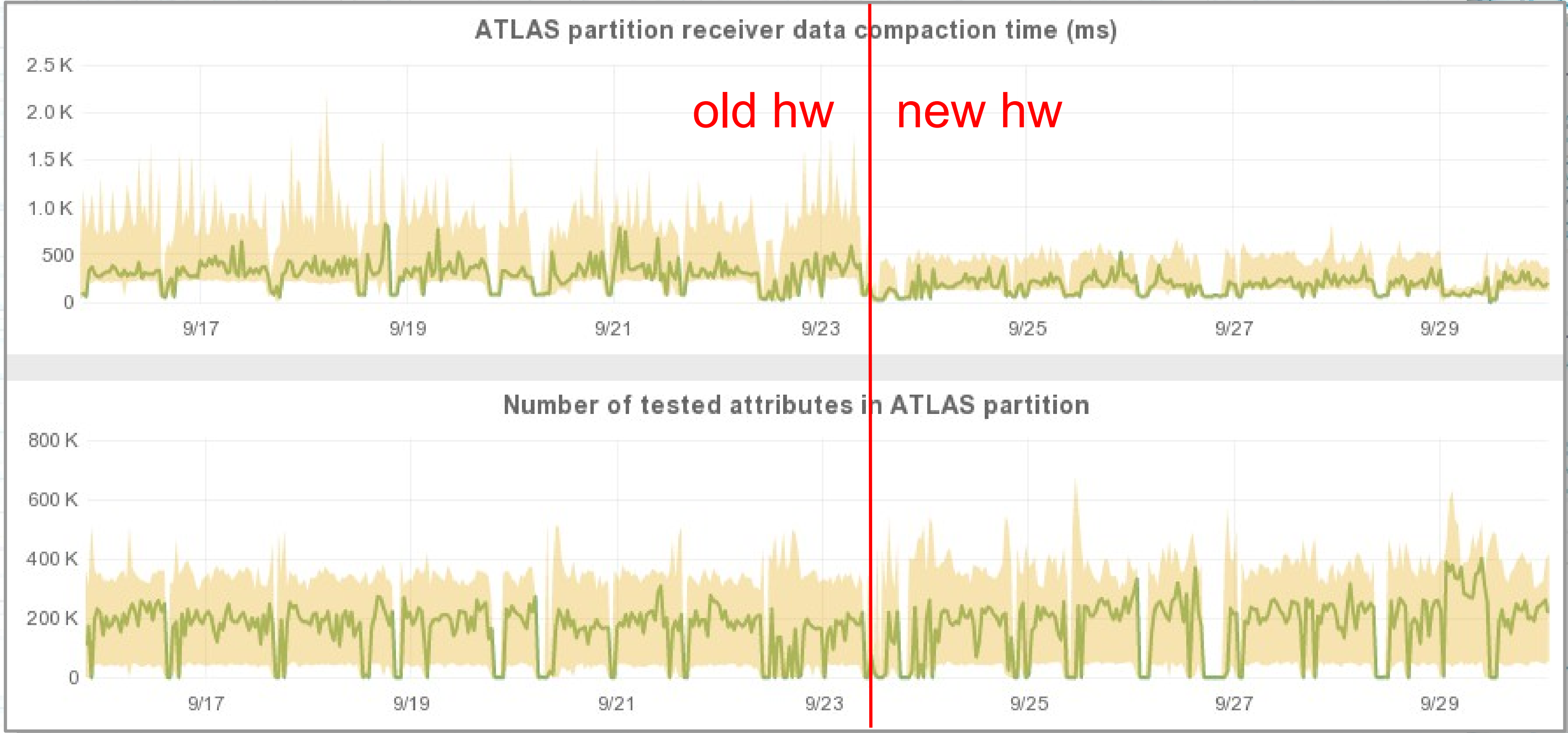


- did not remove raw data since beginning of Run 2 storing 1 TB/month use disk balancing mechanism for merged repositories on 3 nodes



Performance on new HW

- Installed two new powerful nodes: dual CPU Xeon E5-2680V3 @ 2.5GHz, 256 GB RAM, 8x4TB RAID; old nodes are still in use and contributed to performance alongside new ones
- One new node is sufficient to get IS data from all data taking runs on Point-1! Disk space of two nodes is sufficient to keep raw data of Run 2.
- Gain stability, zero problems with P-BEAST software since then



P-BEAST Dashboard

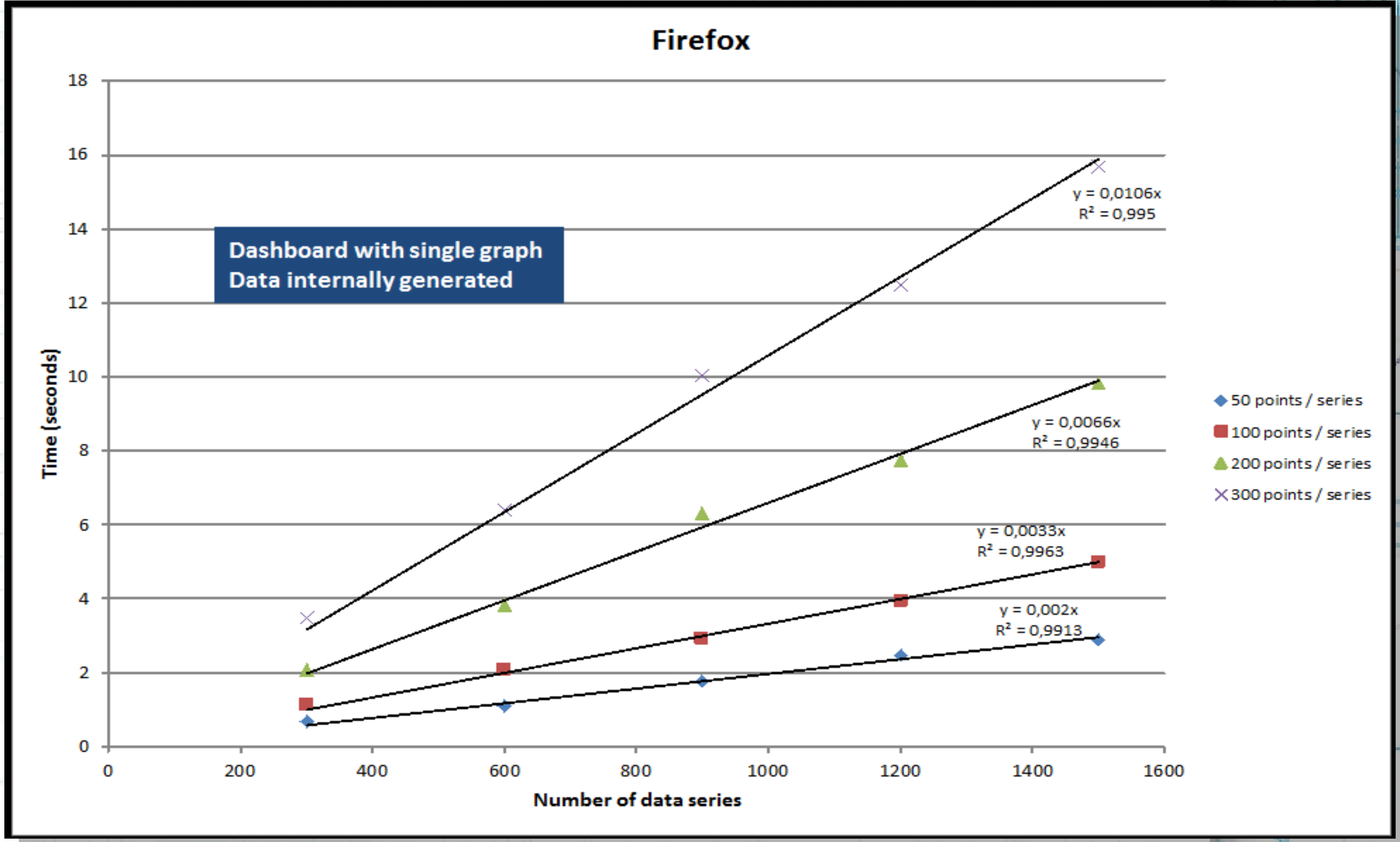
- In 2014 considered several technologies including Kibana and Grafana back-ends for dashboard implementation and various intermediate data source engines including ElasticSearch, *Graphite*, *OpenTSDB and Redis*
- After several prototypes selected Grafana and integrated P-BEAST as it's data source via REST API
  - managed without touching too much the Grafana's code (really 3 or 4 lines of code); the Grafana plugin mode worked pretty well
  - at the same time the documentation to build plug-ins in practice does not exist and some kind of reverse engineering to write some meaningful code (starting from the Graphite data source) was really required

Dashboard Details

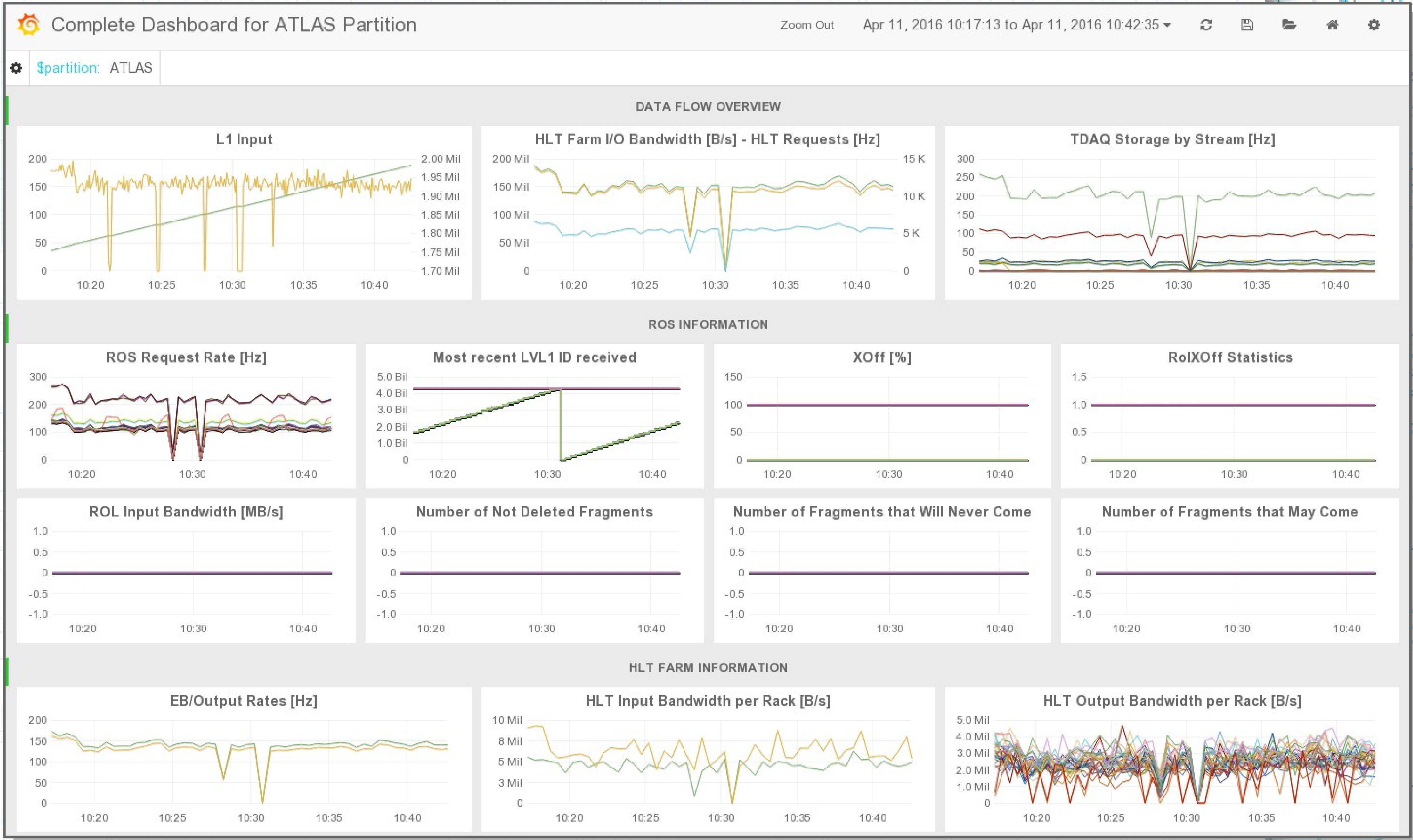
- The biggest challenge in the interaction between Grafana and P-BEAST was to define a data format that could minimize the amount of data to be transferred and (at the same time) required a very little post-processing in the browser
  - put an effort on representation of arrays (each index may have special meaning in user data)
  - perform downsampling on P-BEAST server side
  - add error bands option allowing to show sampled, maximum and minimum values by time interval
- Browsers demonstrate very different performance (next slide):
  - Firefox could hang forever when receiving a lot of data
  - Chrome is 3-4 times faster, but not available* in SLC6
- Query constructor guide for dashboard configuration helps user to navigate through available classes, attributes, IS repositories and object names

Dashboard GUI Performance

- Since downsampling is performed on the server side, there is no need to investigate huge numbers of data points per serie and such numbers are naturally limited by screen resolution
- Varying number of series and data points per serie
- Scales linearly
- For an average CERN desktop computer and SLC6's Firefox 50 to 100 point per serie look like acceptable even for thousand series on the same dashboard (1600 ROS Robin channels)



Dashboard Example


Complete Dashboard for ATLAS Partition
Zoom Out
Apr 11, 2016 10:17:13 to Apr 11, 2016 10:42:35
$partition: ATLAS
DATA FLOW OVERVIEW
L1 Input
HLT Farm I/O Bandwidth [B/s] - HLT Requests [Hz]
TDAQ Storage by Stream [Hz]
ROS INFORMATION
ROS Request Rate [Hz]
Most recent LVL1 ID received
XOff [%]
RoIXOff Statistics
ROL Input Bandwidth [MB/s]
Number of Not Deleted Fragments
Number of Fragments that Will Never Come
Number of Fragments that May Come
HLT FARM INFORMATION
EB/Output Rates [Hz]
HLT Input Bandwidth per Rack [B/s]
HLT Output Bandwidth per Rack [B/s]

Summary and Plans

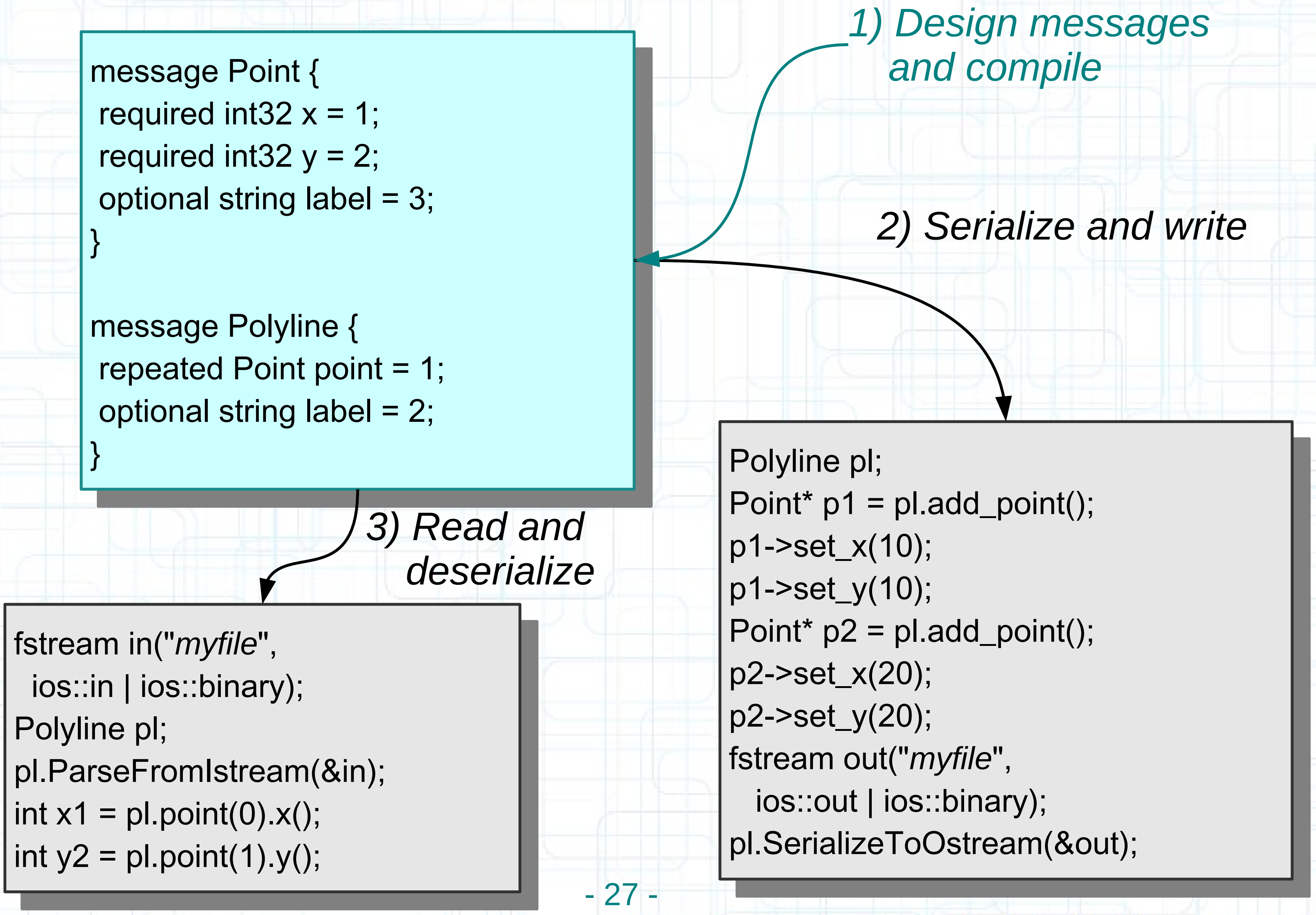
- Have a product satisfying our requirements, no known performance and functional issues
  - based on low level DB implementation adopted for our needs, required a bit more programming, but time spent for design & implementation is comparable with time spent for learning Cassandra and work around found issues
  - gives an order of magnitude of space utilisation and performance efficiencies vs. generic DB solutions
- Consider a possibility to use for other types of time-series data: DCS conditions data / required for offline reconstruction, network and farm monitoring
  - collect new requirements
- Have a stable solution, can reconsider available technologies and make improvements with no hurry

Addendum

Candidate Technologies for own storage implementation (2013)

- Raw binary: performance efficient, but no cross-platform and language compatibility :-(
- Boost serialization: moderate performance and space overhead, but no cross-language (Java) compatibility :-(
- Selected Google Protocol Buffers among similar technologies:
  - Language / platform neutral, extensible way of serializing structured data for use in communications protocols and data storage
  - Serialize data into a binary wire format (compact, forwards and backwards compatible, not self-describing)
  - A developer defines data structures (called *messages*) and services in a *.proto* definition file and compiles that with *protoc*. This compilation generates code that can be invoked by a sender or recipient of these data structures.
  - Protocol compilers for C++, Java and Python available to the public under a free software, open source license and used extensively at Google for almost all RPC protocols, and for storing structured information

ProtoBuf Message Example



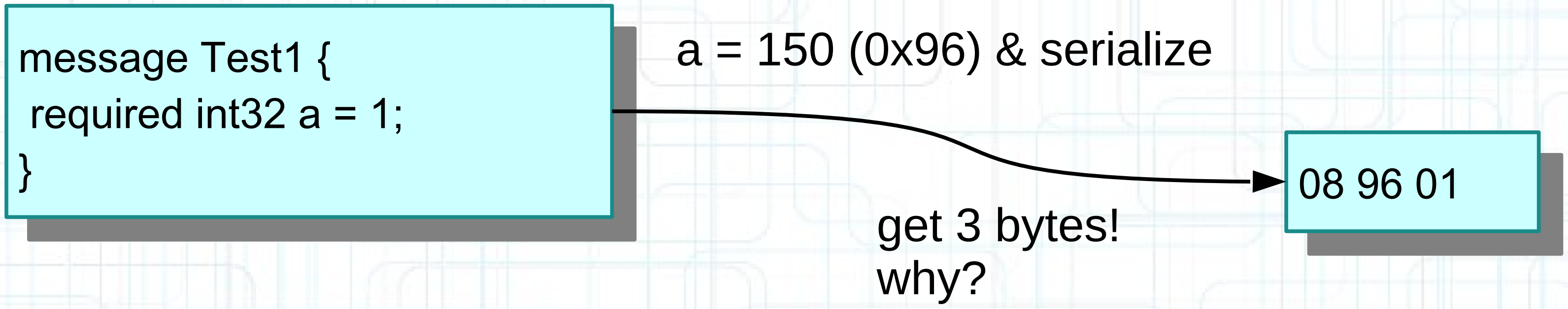
```
message Point {
 required int32 x = 1;
 required int32 y = 2;
 optional string label = 3;
}

message Polyline {
 repeated Point point = 1;
 optional string label = 2;
}
```

```
Polyline pl;
Point* p1 = pl.add_point();
p1->set_x(10);
p1->set_y(10);
Point* p2 = pl.add_point();
p2->set_x(20);
p2->set_y(20);
fstream out("myfile",
   ios::out | ios::binary);
pl.SerializeToOstream(&out);
```

```
fstream in("myfile",
  ios::in | ios::binary);
Polyline pl;
pl.ParseFromIstream(&in);
int x1 = pl.point(0).x();
int y2 = pl.point(1).y();
```

ProtoBuf: messages structure (1/2)



```
message Test1 {
 required int32 a = 1;
}
```

```
08 96 01
```

- 1) Uses variants (a method of serializing integers using one or more bytes; smaller numbers take a smaller number of bytes)
  - Each byte in a varint, except the last byte, has the most significant bit (MSB) set – this indicates that there are further bytes to come. Least significant group comes first.

1 => 0000 0001 (1 byte **01**, MSB = 0)

150 => 1001 0110 0000 0001 (2 bytes **96 01**) how:

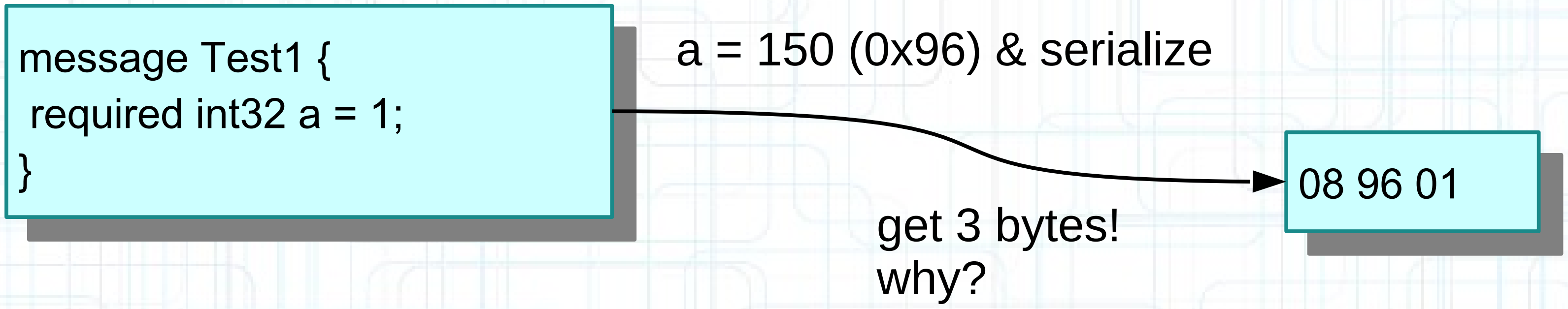
$\cancel{1}$001 0110 $\cancel{0}$000 0001

000 0001 001 0110 (drop MSB and reverse 7bits groups)

“normal” binary 10010110 = 2+4+16+128 = 150

- Signed integers are stored using ZigZag algorithm (no details here...)

ProtoBuf: messages structure (2/2)



```
message Test1 {
 required int32 a = 1;
}
```

- 2) A serialized message is a sequence of keys and pair. Above “08” encodes field number and wire type:

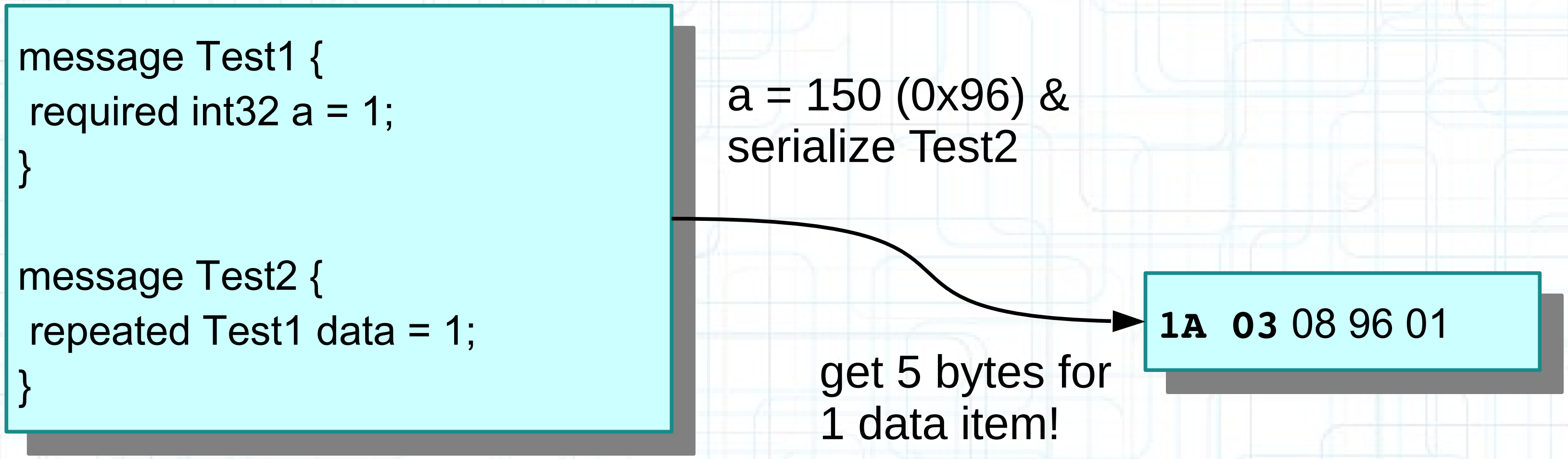
```
((field_number << 3) | wire_type)
```

where available wire types are:

- 0 => varint for [s|u]int32, [s|u]int64, bool, enum
- 1 => fixed64, sfixed64, double
- 2 => length-delimited types (e.g. string) and messages
- 3,4 => deprecated (start/end of groups)
- 5 => fixed32, sfixed32, float

ProtoBuf: sizes of messages

- Because of compatibility across versions and optional fields every field of message has at least 1 byte overhead
- Every embedded message has key prefix as well. When our message is a part of another message, it is encoded as length-delimited type, so:



```
message Test1 {
 required int32 a = 1;
}

message Test2 {
 repeated Test1 data = 1;
}
```

ProtoBuf: issues

- Keys overhead (as described on previous slides)
  - Can be quite significant and visible for small types
- Files Parsing
  - ProtoBuf parser reads a file into buffer (no random access)
  - Has soft limit on buffer size (80 MB by default)
  - Can be enlarged to 512 MB (theoretically up to 2 GB, but strongly not recommended)
  - For DB-like applications it is recommended to read a fraction of data into user buffer and to create a stream from it to be used by ProtoBuf (cumbersome, no thanks!)

File Storage

~~1A 03 08~~ 96 01

- Decided not to use ProtoBuf messages and compiler
- Instead use efficient low-level ProtoBuf methods to serialize basic types
  - Define file structure as needed for random access
  - No message and field key's overhead
  - Of course, requires more development to write code accessing such files
- Implemented for C++
- Create a file per attribute for given number of data items of time interval