

Data Aggregation System, an information retrieval on demand over relational and non-relational distributed data sources

Valentin Kuznetsov (Cornell, USA), Dave Evans (FNAL, USA), Simon Metson (Bristol, UK), Gordon Ball (Imperial, UK)

Motivations ...

- A user wants to query different data services without knowing of their existence
- A user wants to aggregate information from different data services

DAS in nutshell

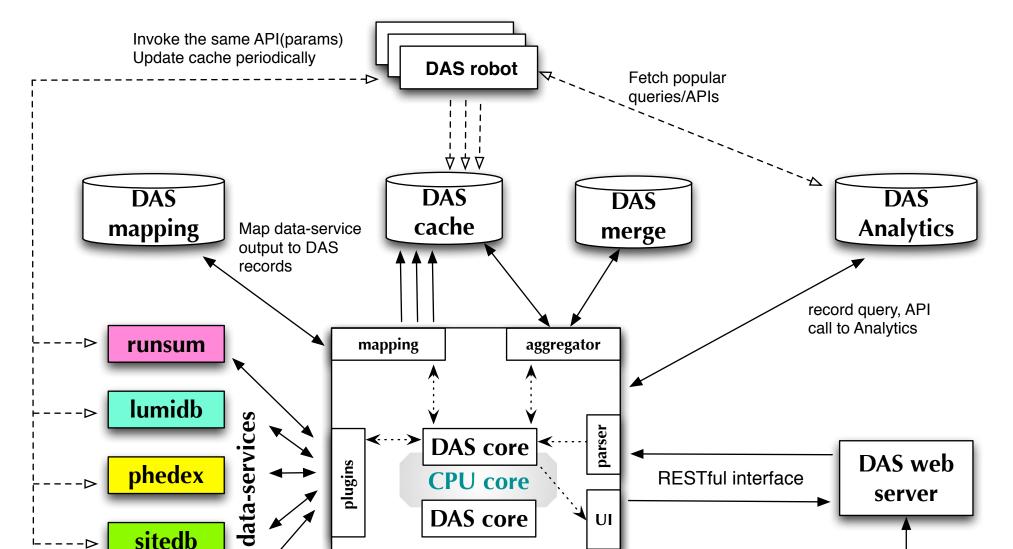
DAS provides a novel approach to aggregating data from multiple sources without applying any requirements on data providers. Once data is accessible on a web DAS can handle the rest.

DAS leaves data management up to the data-providers. It is true that they know better how to handle, preserve and secure their data.

DAS is agnostic to data content. Thanks to NoSQL document-based database MongoDB we're able to store any type of meta-data documents provided by data-providers.

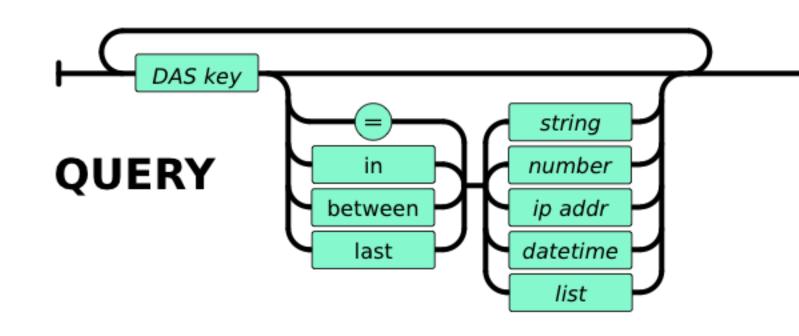
DAS provides a free text-based query language to ease data-lookup. It

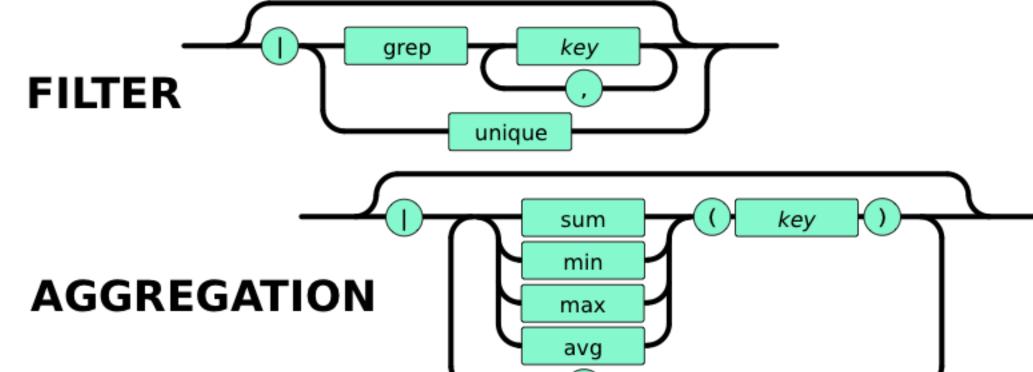
DAS architecture



A user has domain knowledge, but needs to query X services, using Y interfaces and deals with Z data formats to get the data

DAS Query Language





should be as simple as you search on Google.

DAS uses filters and aggregators to help you navigate through your data.

DAS has been developed in CMS to deal with broad variety of existing distributed data services, majority of them are RDMS based.

DAS user interfaces

lata in 🛛 list 🛟 form	mat, 🚺 🛟 results/per	page, records in	browseable 🛟 f	ormat,	
				Search Reset	

⊳ sitedb	р		
dbs	*	DAS Cache server	site=T1_CH_*

• Web server/CLI tool to communicate w/ end-users Cache server to handle requests flow Cache DB to store results from data-providers Merge DB to store aggregated results from DAS cache ◆ Mapping DB to keep info about data-providers (URIs,

URNs, expire timestamps) and DAS keys used by endusers

 Analytics DB to keep track of user requests and query analysis

To simplify data look-up we used

 Presentation maps for data records, which shows a snapshot of data content, e.g. for site record we only show site name and SE info

 Filters in form of standard UNIX pipes, which select a sub-set of data record

	data in list i format, 10 i results/per page, records in browseable i format, site=T1_* grep site.name
data in list \$ format, 10 \$ results/per page, reconsiste=T1_*	Showing 1-10 of 16 rows Filter what you want
Showing 1-10 of 16 rows Query	DAS key: site CMSName: T1_US_FNAL_MSS Record: 4be81f2687eed75f5f000032, <u>show</u> , <u>hide</u>
DAS key: site CMSName: T1_US_FNAL_MSS StorageElement: cmssrm.fnal.gov Record: 4be81f2687eed75f5f000032, show, hide	<pre>{ "das_id": [4be81e7187eed75f5f000002], "_id": 4be81f2687eed75f5f000032, "site": [</pre>



RunSummary n, trigger, detector,

lock, file, block,replica, le.replica, se, node, ... site

site

SiteDB site, admin, site.status, ..

Data Aggregation System

jer, ecal, hcal

run, file, block, site, config, tier, dataset, umi, parameters, ...

Overview country, node, region,

LumiDB mi, luminosity, hltpath

Parameter Set DB CMSSW parameters

MC Id generator, xsection, process, decay, ...

<expressions> | <filters> | <aggregators> or <map-reduce functions>

- <expressions> represented in a form of <*key*> and / or <*key*> <*operator*> <*value*>
- keys: *dataset, block, file,*; operators: =, *in, between, last*; values: *int or string (including patterns)*; filters: *grep, unique, aggregators: sum, count, avg, min, max*

Examples

- * *site=T1_CH_CERN; site=T1_** | *grep site.name*
- *run=20853; run in [20853,20859]; run between [20853,20859]; run last 24h*
- block dataset=/a/b* | grep block.size | sum(block.size)

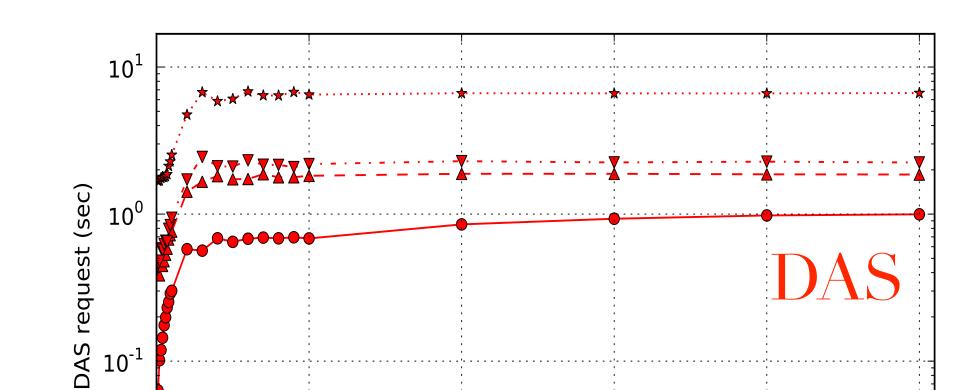
<pre>"site": ["kind": "MSS", "technology": "dCache", "name": "T1_US_FNAL_MSS", "se": "cmssrm.fnal.gov", "id": "10.0" }, "cust_dest_bytes": "2.71767546431e+15",</pre>	<pre>}], "cache_id": [4be81f2687eed75f5f0 "das": { "expire": "1273506945.57", "primary_key": "site.name" } }</pre>	data in list ; format, 10 ; result: site=T1_* sum(site.src_node_files) Aggregate
<pre>"name": "TI US FNAL MSS", "src_node_files": "7524.0", "cust_node_bytes": "2.70470672424e+15", "cust_dest_files": "1503816.0", "nonsrc_node_files": "0.0", "nonsrc_node_bytes": "0.0", "noncust_dest_files": "887650.0", "cust_node_files": "887650.0", "src_node_files": "864485.0", "src_node_bytes": "9.19710486288e+12", "noncust_dest_bytes": "1.83602768692e+15", "noncust_dest_bytes": "1.83602768692e+15", "noncust_node_bytes": "1.82078676773e+15" }], "cache_id": [4be81f268/eed/5f5f000030, 4be81f2: "das": { "expire": "1273506945.57", "primary_key": "site.name" }</pre>	Bleed.2121000001	Showing 1-10 of 16 rows function: sum result: 1808184.0 key: site.src_node_files Record: 0, show, hide { { "function": "sum", "_id": 0, "result": "1808184.0", "key": "site.src_node_files" }

DAS performance

DAS benchmarks has been done on Linux 64-bit 8 core node w/ 16GB of RAM.

We measured separately write and read time for different set of meta-data.

Write time has been driven by data-providers. Due to the fact that data need to be



• Aggregators, e.g. sum, count, which allow get snapshots of data

 Map-reduce functions for more sophisticated data analysis

Data from data-providers were converted into common JSON data-format

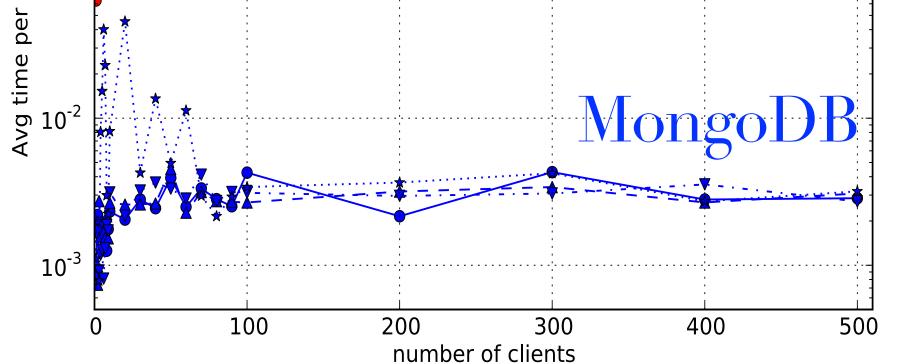
Data notations has been centralized across multiple data-services by using DAS maps, e.g. runNumber, Run were converted into run_number notations.

Access random record using

★ 500M blocks, 100x of projected statistics

shipped over the wire from remote source to DAS we only measured internal insert rates. The MongoDB back-end can handle up-to 20K docs/sec, while DAS merge time was around 5K doc/sec.

The read time benchmark has been done when DAS was populated with 50K dataset and 500K block meta-data information. We used query patterns, e.g. dataset=/A* and measured server response time to find a first matched record.





▲ 500K blocks, current # of CMS blocks

