Migrating a Full-Text Search

By Andreas Rappold, Barbara Ondrisek, Siegfried Goeschl

Introduction

willhaben.at is a classified ads provider in Austria - according to [1] it is one of the busiest web sites in Austria - the infrastructure manages 1.8 million classified ads, serves 12.9 million page impressions every day, imports 20 new classified ads every minute and runs, on the average, 25 full-text search requests each and every second. *willhaben.at* was happily using Microsoft FAST ESP (Enterprise Search Platform) on Linux until Monday, February 8th, 2010 when the following headline hit the press - "FAST to abandon Linux and Unix" [2]. Running FAST using Microsoft Windows in production - the operations team was not excited. In addition, FAST ESP servers started to slow down and FAST expertise was scarce in general. In the beginning of 2012 product management looked into alternatives and settled for *Apache SOLR* - an open-source search server based on *Lucene* and developed by the *Apache Software Foundation*.

The Impact of Replacing FAST ESP

The first question during the project kick-off meeting - "How big is the impact of replacing FAST with SOLR?" Encouraging answer from software development: "Massive." The term "full-text search" is somehow misleading since many user-driven searches are centred on hierarchical navigation. The term "hierarchical search" would be a better description for navigating classified ads, e.g. starting from "Audio/TV/Phone" to "DSLR" and "Nikon", searching for "D7000" within the price range from 750-900 EUR to be found in Vienna preferably in the 8th or 9th district. In fact the whole classified ads navigation including the site's landing page depends on full-text search. So bringing down the entire site looks uncomfortably possible and provides you with the five minutes of fame you are not looking for.

Risks, Requirements and Implementation

If risk can't be avoided it needs to be managed - the following requirements were defined:

- The SOLR search shall provide at least the same or a better search experience
- The SOLR deployment shall handle more load than existing FAST servers
- Avoid Big Bang Integration and support a stepwise migration instead
- Provide an emergency fallback mechanism from SOLR to FAST
- Implement state-of-the-art server monitoring to detect problems early on

Let's have a closer look at the requirements and how to fulfill them.

Improve Search Experience

The minimum requirement for SOLR was to provide the same or better search experience compared to Microsoft FAST. One important improvement for full-text search was better support of German compound words - that feature was internally called implicit wild-card search. Let's look at an example - the German word "Leinenhose" is a compound word consisting of "linen" and "trousers" and shall be found when searching for "hose" or "leinen" - a plain-vanilla search would miss "leinenhose" when searching for "hose" because these are two completely different tokens as far as full-text search is concerned. Technically there are various approaches such as prefix/postfix wild-card search, N-Grams or a dictionary-based word-splitting algorithm. Project management settled for a modified SOLR *DictionaryCompoundWordTokenFilter* providing a good balance between search speed, index size and matching capabilities as shown below:

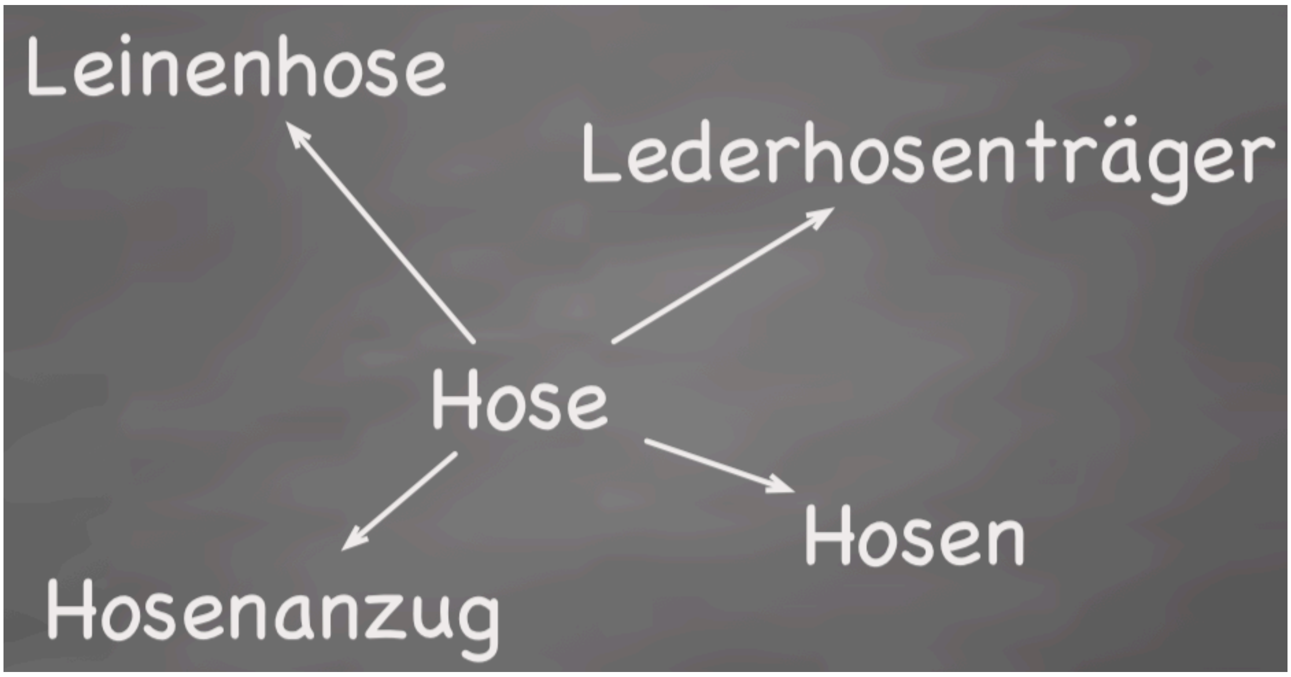


Figure 1: German Compound Words

Performance Requirements

Due to the commercial success and increased traffic the SOLR server deployment needs to handle a significantly higher load than the original FAST installation. During peak hours in spring 2012 the performance requirements exceeded 200 full-text search requests per second - it was assumed that the SOLR deployment needs to handle 300 search requests / second at peak hours.

Avoiding Big-Bang Integration

Avoiding big-bang integration and emergency fall-back becomes second nature when working with 24x7 server applications - even more so when your annual bonus depends on system downtime (aka feedback loop in management terms). The classified ads are divided into three so-called *verticals*.

- Bits And Pieces (BAP, used goods)
- Estate
- Motor

SOLR to FAST Fall-back

Looking at those verticals, it was decided to migrate one vertical at a time while using a *Minimally Invasive Integration Pattern*. A custom query parser was written to translate a *FAST query language* (FQL) query into a corresponding SOLR query on the fly. Consequently, the emergency fall-back from SOLR to FAST was straightforward to implement - instead of translating a FQL statement into a corresponding SOLR query and hitting the SOLR server, the FQL query was simply processed by the FAST ESP server.

Server Monitoring

Quite a few production and performance problems do not arise out of the blue but could be caught early on, assuming that comprehensive server monitoring is straight-forward to use and available for everyone. The worst case is a flood of customer complaints and high-level management showing up at your desk while you are the very last one finding out about the problem. Consequently, keeping an eye on server monitoring tools is encouraged and possible from within the whole company starting from the browser to dedicated TV screens hooked up to *Raspberry PIs* located in the offices and even in the kitchen. Considering the corporate culture, it is encouraged to setup server-monitoring tools before running the first performance test.

The Testing Approach

At this point we have a project, risks, and requirements - time to think about quality assurance and testing. At the company *willhaben.at* agile rules - QA, DEV & OPs are co-located in the same room and work closely together. Hence the following QA approach was chosen:

Activity	**Team**
Unit tests	Dev
Functional tests	QA
Performance tests	Dev & Ops
Integration tests	Dev & QA
Acceptance tests	QA & PM
Pre-production tests	Dev & Ops

Working together tightly has additional advantages such as a better understanding of each other's work and its challenges. For that reason the topic of testability came up early in the development cycle because testability was considered as a critical success factor in delivering 24x7 server-side software.

Striving for Testability

Testability is in many ways a crosscutting concern and comes in many flavours ranging from repeatable system setup to custom monitoring components.

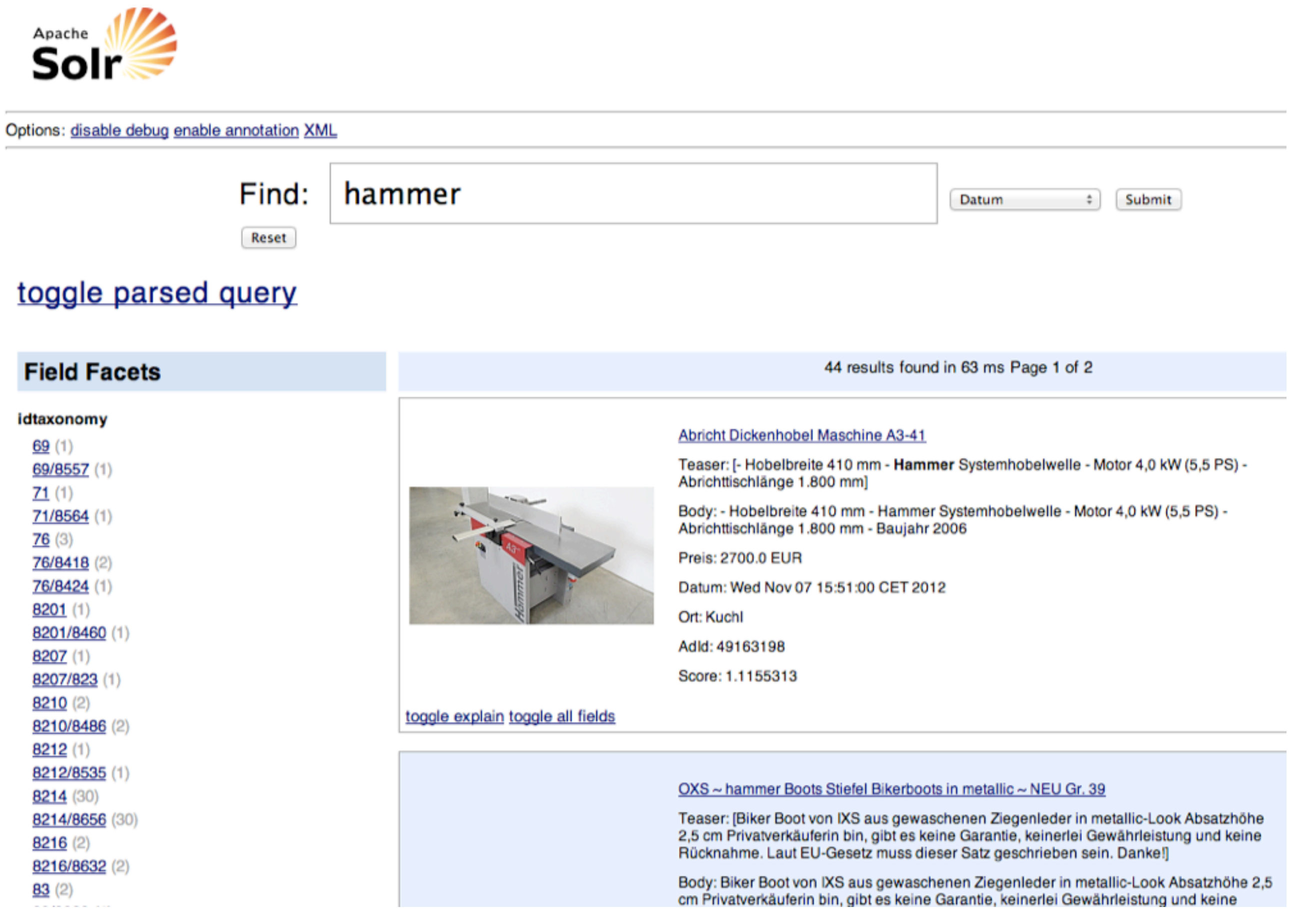
In our case we settled for the following action items to improve testability

- Use well-known production test data for regression testing
- Provide a simple SOLR-based GUI to cater for prototype testing
- Determine the most frequent search terms

When tweaking the SOLR server configuration it becomes obvious that there are many configuration options and equally many ways of causing undesired side effects and errors. Therefore it is mandatory to implement a comprehensive set of full-text searches based on well-known test data. The well-known tests data consisted of 10,000 production adverts per vertical ("Motor", "Estate" & "Bits and Pieces"). When using the Maven build tool, it only takes three command line invocations from a clean SVN checkout to an up & running local SOLR server instance with 30,000 ingested documents.

```
trunk> mvn clean install
trunk/solr-server> mvn clean jetty:run
trunk/sol-server/src/test/bin> ./import-all.sh
```

With a SOLR server and test data at your fingertips it would be very convenient to run full-text search queries and view the results in a web browser. The out-of-the-box SOLR Admin GUI is not user friendly so the *VelocityResponseWriter* [3] was used which in turn utilizes *Apache Velocity* for rendering a HTML page with search results. After a little bit of tinkering with Velocity templates and the Maven build, the GUI is automatically deployed with the SOLR server. In other words - when executing the three command line invocations above, the GUI is ready to use.



idtaxonomy
69 (1)
69/8557 (1)
71 (1)
71/8564 (1)
76 (3)
76/8418 (2)
76/8424 (1)
8201 (1)
8201/8460 (1)
8207 (1)
8207/823 (1)
8210 (2)
8210/8486 (2)
8212 (1)
8212/8535 (1)
8214 (30)
8214/8656 (30)
8216 (2)
8216/8632 (2)
83 (2)

44 results found in 63 ms Page 1 of 2

Abricht Dickenhobel Maschine A3-41

Teaser: [- Hobelbreite 410 mm - **Hammer** Systemhobelwelle - Motor 4,0 kW (5,5 PS) - Abrichttischlänge 1.800 mm]

Body: - Hobelbreite 410 mm - Hammer Systemhobelwelle - Motor 4,0 kW (5,5 PS) - Abrichttischlänge 1.800 mm - Baujahr 2006

Preis: 2700.0 EUR

Datum: Wed Nov 07 15:51:00 CET 2012

Ort: Kuchl

AdId: 49163198

Score: 1.1155313

toggle explain toggle all fields

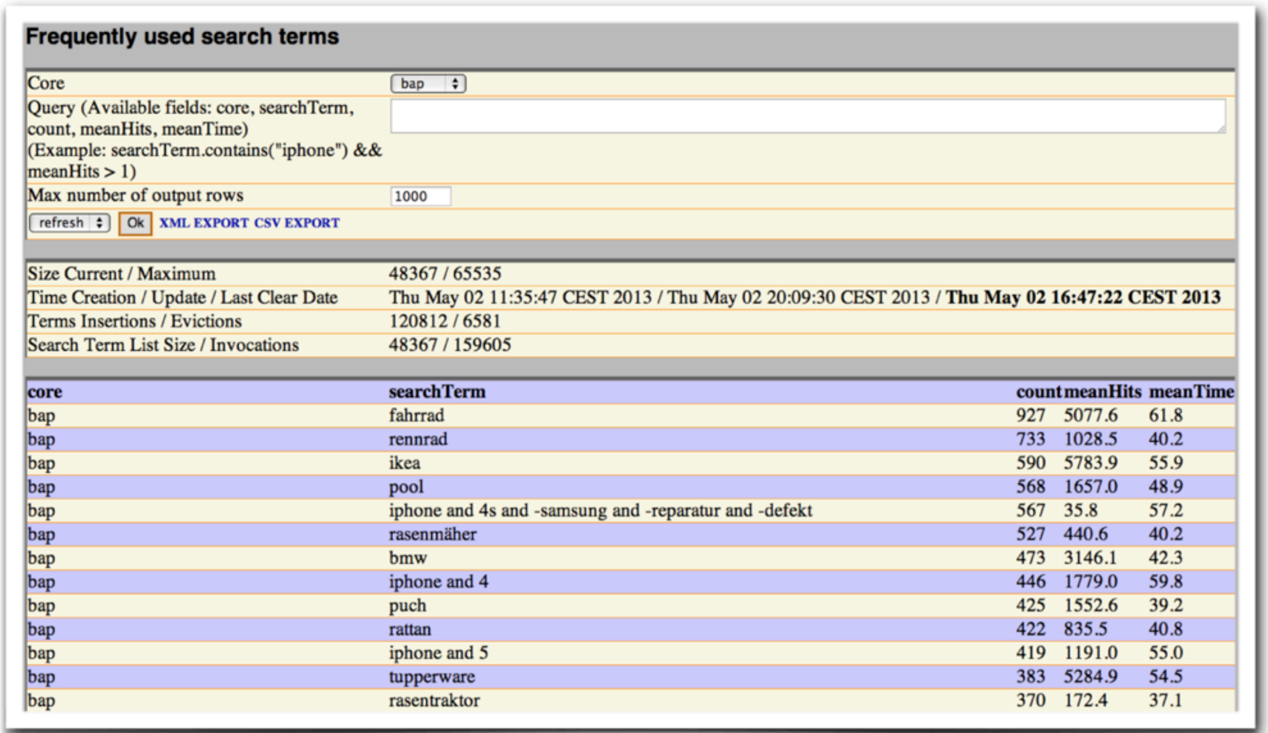
OXS ~ hammer Boots Stiefel Bikerboots in metallic ~ NEU Gr. 39

Teaser: [Biker Boot von IXS aus gewaschenen Ziegenleder in metallic-Look Absatzhöhe 2,5 cm Privatverkäuferin bin, gibt es keine Garantie, keinerlei Gewährleistung und keine Rücknahme. Laut EU-Gesetz muss dieser Satz geschrieben sein. Danke!]

Body: Biker Boot von IXS aus gewaschenen Ziegenleder in metallic-Look Absatzhöhe 2,5 cm Privatverkäuferin bin, gibt es keine Garantie, keinerlei Gewährleistung und keine

Figure 2 : Solritas GUI

Functional tests were defined according to *Risk-based Testing* [4] and the test team asked - "What are the most frequent search terms in production so we can focus on them?!" The intention was clear; only answering proved difficult. The list of most frequent search terms was to be extracted from the FAST server's query log that in turn required some sophisticated scripting. Looking at the question in more detail it becomes obvious that the most frequent search terms change over time, e.g. "windsurfing equipment" is not hugely popular during winter in Austria. Also the on-line advertising department showed great interest in the list of most frequent search terms which brings back testability as a cross-cutting concern - during the course of the project a custom SOLR component was implemented to retrieve and display the most frequent search terms in real-time over the browser as shown below



Frequently used search terms

Core	bap
Query (Available fields: core, searchTerm, count, meanHits, meanTime) (Example: searchTerm.contains("iphone") && meanHits > 1)	
Max number of output rows	1000
refresh Ok XML EXPORT CSV EXPORT	

Size Current / Maximum	48367 / 65535
Time Creation / Update / Last Clear Date	Thu May 02 11:35:47 CEST 2013 / Thu May 02 20:09:30 CEST 2013 / **Thu May 02 16:47:22 CEST 2013**
Terms Insertions / Evictions	120812 / 6581
Search Term List Size / Invocations	48367 / 159605

core	searchTerm	count	meanHits	meanTime
bap	fahrrad	927	5077.6	61.8
bap	rennrad	733	1028.5	40.2
bap	ikea	590	5783.9	55.9
bap	pool	568	1657.0	48.9
bap	iphone and 4s and -samsung and -reparatur and -defekt	567	35.8	57.2
bap	rasenmäher	527	440.6	40.2
bap	bmw	473	3146.1	42.3
bap	iphone and 4	446	1779.0	59.8
bap	puch	425	1552.6	39.2
bap	rattan	422	835.5	40.8
bap	iphone and 5	419	1191.0	55.0
bap	tupperware	383	5284.9	54.5
bap	rasentraktor	370	172.4	37.1

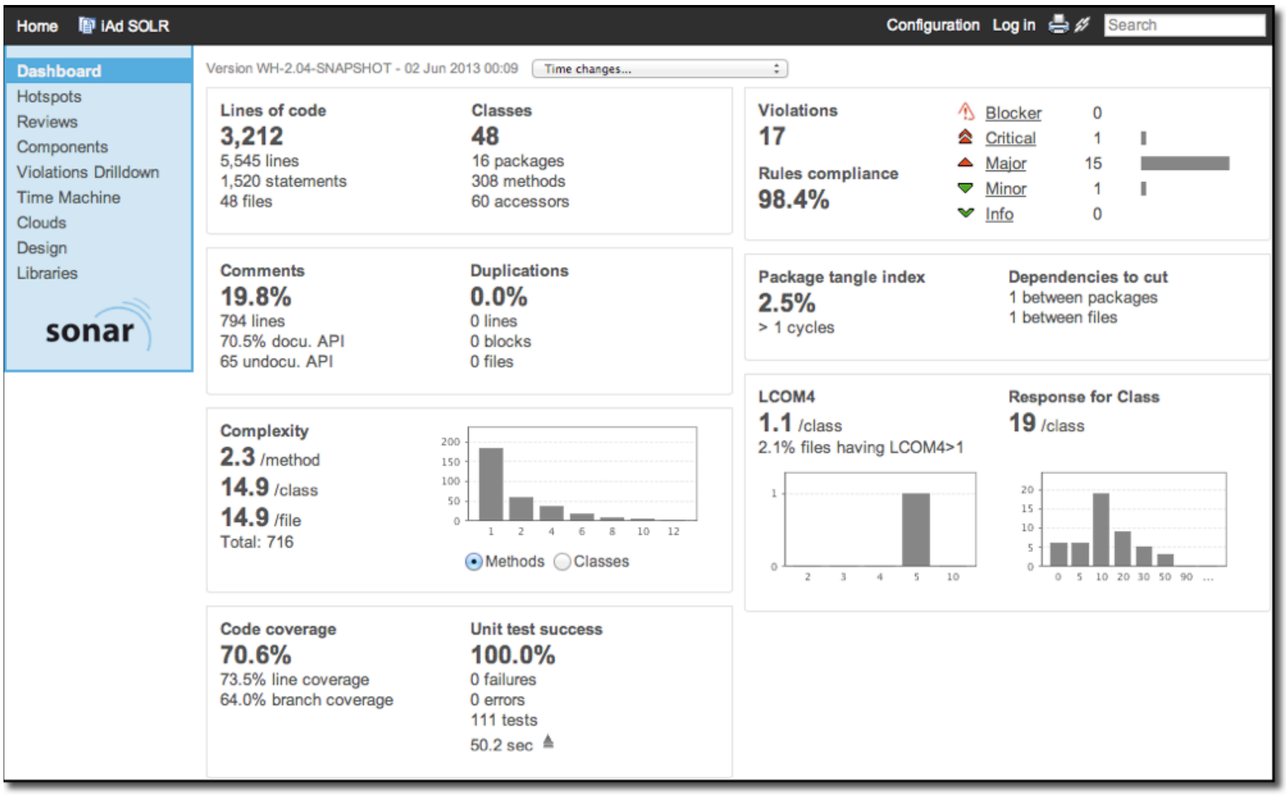
Figure 3: Most Frequent Search Terms

Unit Tests

For unit testing the usual suspects were used such as

- *JUnit* as regression test framework
- *Jenkins* as CI server
- *Sonar* for static code analysis

For the implementation a mostly test driven approach was used - writing unit tests along the actual Java code, while the Sonar reports were used as a quality feedback loop.



Version WH-2.04-SNAPSHOT - 02 Jun 2013 00:09

Lines of code
3,212
5,545 lines
1,520 statements
48 files

Classes
48
16 packages
308 methods
60 accessors

Violations
17

Rules compliance
98.4%

Blocker	0
Critical	1
Major	15
Minor	1
Info	0

Comments
19.8%
794 lines
70.5% docu. API
65 undocu. API

Duplications
0.0%
0 lines
0 blocks
0 files

Package tangle index
2.5%
> 1 cycles

Dependencies to cut
1 between packages
1 between files

Complexity
2.3 /method
14.9 /class
14.9 /file
Total: 716

LCOM4
1.1 /class
2.1% files having LCOM4>1

Response for Class
19 /class

Code coverage
70.6%
73.5% line coverage
64.0% branch coverage

Unit test success
100.0%
0 failures
0 errors
111 tests
50.2 sec

Figure 4: SOLR Sonar Report

Integration Tests

The approach of translating a FAST query to a corresponding SOLR query allows executing the same query using both full-text search servers and comparing the results - in other words a *Gold Standard Oracle* [5] approach can be used. Does a full-text search return the same search result when processed by FAST versus SOLR? The answer is somewhere between "no" and "it depends". Let's have a closer look at comparing the results of two full-text search engines. Full-text search engines differ in their implementation, configuration, and technical capabilities such as:

- Tokenizer
- Stemmer
- Relevance algorithm
- Query boosting
- Stopword list
- Synonym list

Consequently, returned documents will differ to a certain degree depending on the search request. This required a custom JUnit test harness to implement the following features:

- Iterate through a CSV file containing FAST search queries and their confidence thresholds
- Execute the query using FAST and SOLR server
- Intersect the document from both result sets
- Determine at which percentage SOLR documents are matched with FAST documents
- If the percentage falls below the confidence threshold, the test has failed

Let's have a closer look at a line in the CSV file:

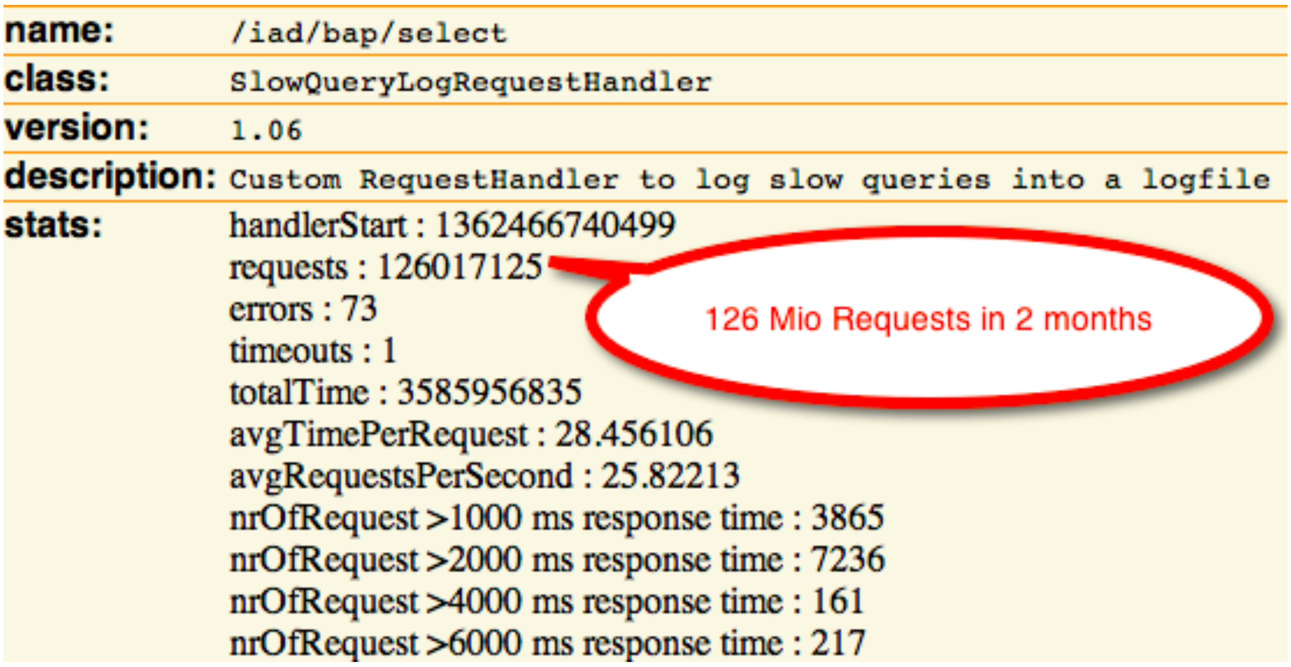
Field	Description	Example
testname	ID to identify the search query	Line-01
query	FAST search query string	and(finntechcomp:string("Guitar", mode="AND", annotation_class="user"))
sortby	Sort order	-fnpublisheddate -fnhwboost
hits	Number of requested documents	25
confidence	The percentage of common documents	85

SOLR Performance Testing

The performance tests were based on *Apache JMeter* - setting up a JMeter test script is straightforward as SOLR exposes a REST interface. The SOLR search requests being executed were derived from the FAST query log representing one hour of peak load of the production system. In addition to the search requests being executed, the JMeter script uploaded two new adverts per second, which were indexed every minute. Adding documents to the SOLR index causes the internal caches to be invalidated resulting in a performance penalty - this penalty is partly compensated by so-called *cache pre-warming queries*.

The result of the first performance test indicated a severe disk I/O bottleneck limiting the throughput to 60 search requests per second. The SOLR index is spread over multiple files totalling 5GB and the hard disks were not capable of delivering the required throughput (mostly random file access). The I/O bottleneck was actually expected and the SOLR index was moved to a ram-disk. Reading the SOLR index file from the ram-disk resulted in a sustained throughput of 200 search requests per second on a single SOLR server instance.

Judging the performance of a full-text search just by looking at the maximum number of processed requests per second and average response time is not good enough - we are also interested in maximum response times. Assuming that only one out of 10,000 search requests takes more than three seconds, there are at least 200 customers every day, thinking that the site is slow today. The original approach determining the distribution of response times, and long-running queries consisted of parsing the FAST query log file. This approach worked but it was clumsy - inspired by the *MySQL Slow Query Log*, a similar functionality was implemented as *SlowQueryLogRequestHandler*. This custom SOLR component logs long-running SOLR queries into a logfile and displays status information in the SOLR Admin GUI as shown below:



name:	/iad/bap/select
class:	SlowQueryLogRequestHandler
version:	1.06
description:	Custom RequestHandler to log slow queries into a logfile
stats:	handlerStart : 1362466740499 requests : 126017125 errors : 73 timeouts : 1 totalTime : 3585956835 avgTimePerRequest : 28.456106 avgRequestsPerSecond : 25.82213 nrOfRequest >1000 ms response time : 3865 nrOfRequest >2000 ms response time : 7236 nrOfRequest >4000 ms response time : 161 nrOfRequest >6000 ms response time : 217

Figure 5: Slow Query Log

Pre-Production Testing

BAP ("Bits and Pieces", used goods) was the first vertical to be migrated - a major risk, considering that it constitutes 90% of the overall number of classified ads and attracts most of the customers and their traffic. So why choose BAP? There was pressure to offload FAST ESP quickly and the two other verticals would have hardly made an impact. In order to minimize the risk the following approach was used:

- The document import was set up to ingest new adverts on both systems simultaneously
- A junction sent the search requests to both full-text search servers
- The FAST search result was used, thereby discarding the SOLR result

This simple approach allowed us to run the SOLR server and document ingestion and performance for one month under production conditions. As an additional QA measure all willhaben.at employees were using one production server wired with SOLR to get more feedback.

Afterwards we had sufficient confidence to switch the junction on the production system, effectively turning SOLR into the primary and FAST into the fall-back system.

The Achieved Quality Level

Taking about QA approaches is academic unless we look at the project's error rate to judge their effectiveness.

Pre-Release versus Post-Release Bugs

Personally we make the following distinction regarding bugs:

- Good bugs are found by developers during on-going development and never make it into the bug tracking
- Bad bugs are found by QA when the code is finished, causing entries in the bug-tracking tool

- Really bad bugs slip past development and the QA team, causing highly visible entries in the bug-tracking tool

So let's have a look at the bad and really bad bugs found during the rollout of the three verticals

Vertical	Pre-Release Bugs	Post-Release Bugs
BAP	31	19
ETSTATE	13	3
MOTOR	12	1
SUM	56	23

As shown above we had a lot of post-release bugs found in BAP due to the following reasons:

- BAP was the first vertical being migrated and a lot of lessons were learned improving the quality
- BAP contains 1.6 million adverts with diverse content, which makes processing much harder than the other verticals

Making mistakes teaches valuable lessons, which resulted in a significant lower error rate for the other verticals as shown below:

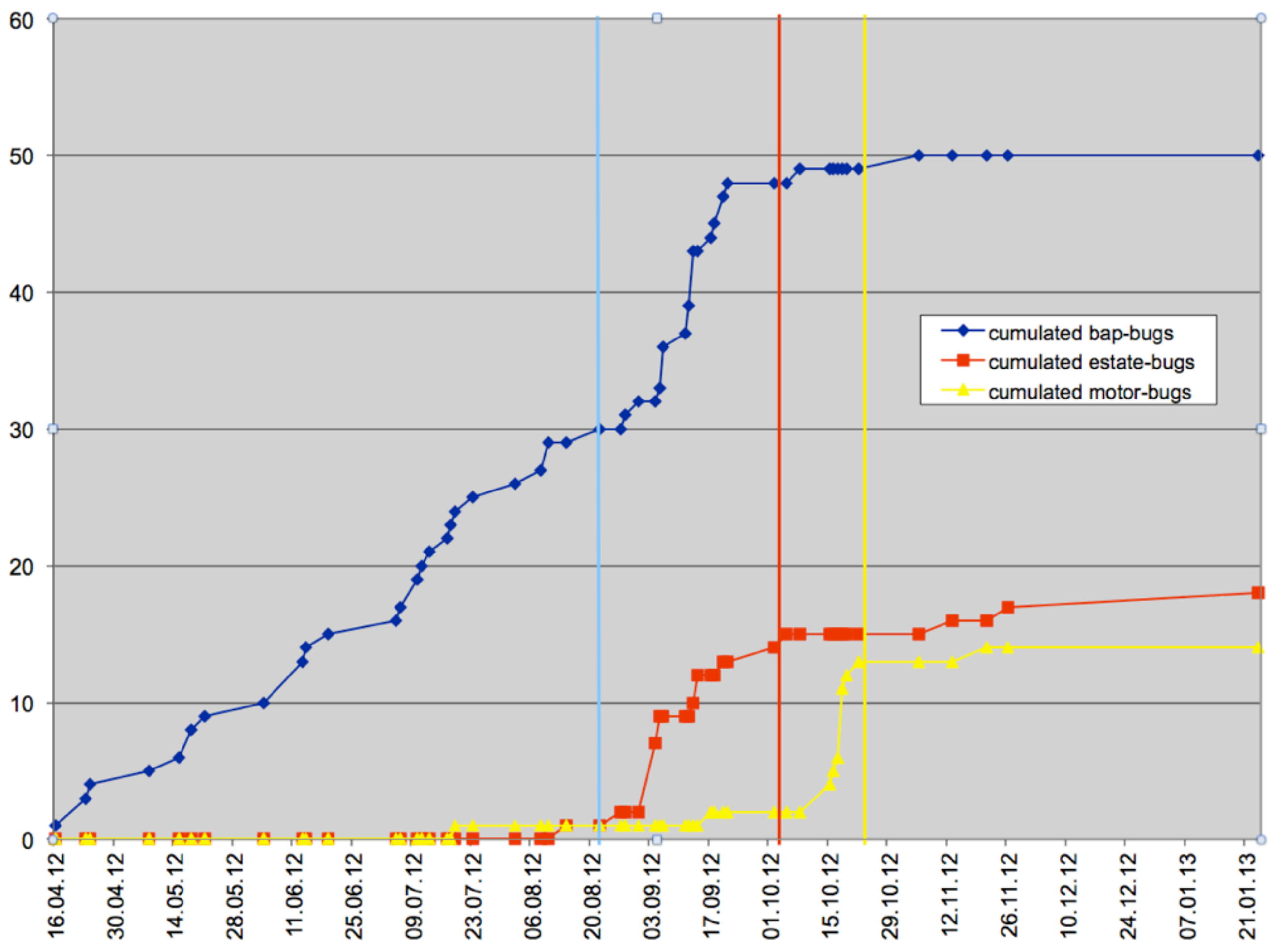


Figure 6: SOLR Cumulative Bugs

Let's have a closer look at bugs slipping past development and the QA team

- Problems with brand names and *WordDelimiterFilterFactory* - tokenizing "B & O", "B&O" and "Bang & Olufson" yielded completely different search results while the customer expects all three variations to find exactly the same classified ads
- Accidentally triggering wildcard search - Alfred Hitchcock's books about the "Three Investigators" are known as "Die Drei ???" in German but unfortunately "???" serves as a wildcard expression to match three arbitrary consecutive characters which in turn returns 1.6 million results
- Support of German compound words and stemming caused interesting search results and customer complaints, e.g. the city "Klosterneuburg" matches the

three German words "Kloster", "neu" & "Burg" but no customer would expect "Klosterneuburg" to be returned when searching for "neu"

- When searching for motorbikes all of the following search terms are equivalent: "300 ccm","300 cm3","300 cm^3" - initially SOLR had a different understanding
- Poorly understood FAST features caused functional problems when using SOLR
- Document ingestion - the whole document ingestion process (i.e. adding new classified adverts to the SOLR index) is complex and therefore fragile
- General integration problems when working on a large and sometimes alien code base

Performance and Availability Level

More important than bugs were the performance and overall availability of the SOLR server and its infrastructure - downtime is much worse than a few obviously wrong and entertaining search results (remember the management feedback loop). The good news is - the actual performance exceeded the requirements and no downtime happened during nine months of production usage. Having said that the performance and uptime can be mostly attributed to the SOLR server implementation and was not affected by our work - *cui honorem, honorem.*

The following screenshot shows vital server parameters during seven months as recorded by JavaMelody:

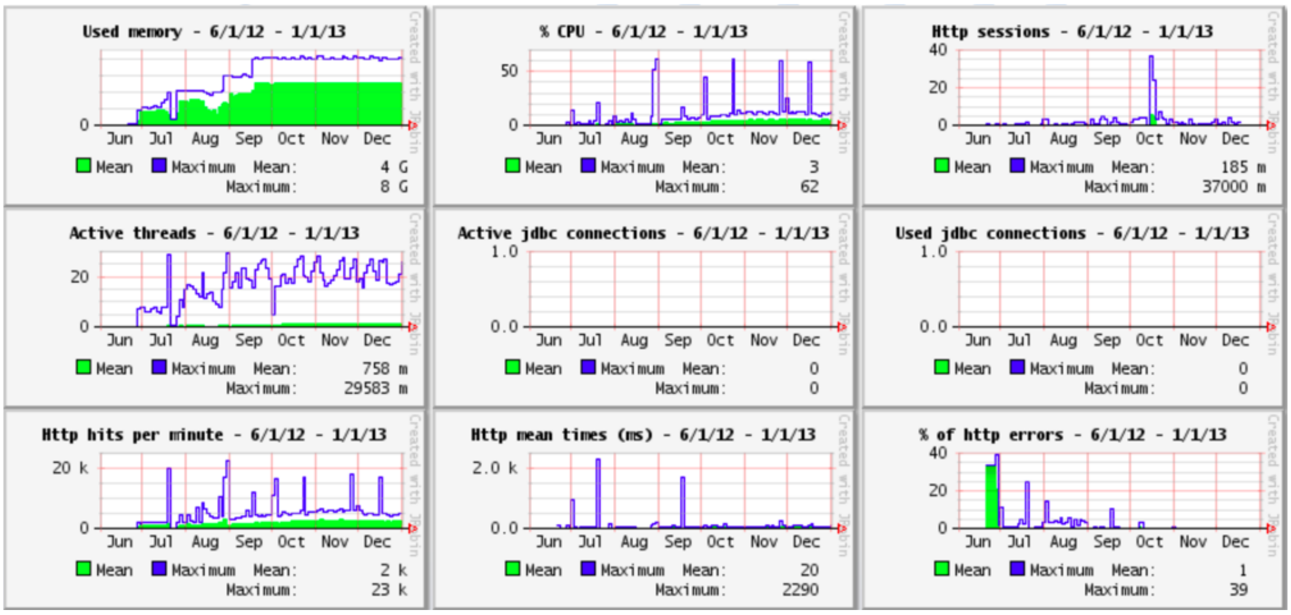


Figure 7: SOLR JavaMelody Report

It can be concluded that

- The SOLR server load never exceeded 62 per cent
- The SOLR server never exceeded 8 GB of memory
- A maximum of number of 23.000 requests were processed per minute
- The average response time for a search request is well below 30 milliseconds

At the first glance the monthly spikes regarding "%CPU" and "Http hits per minute" are a bit puzzling - those spikes were caused by the repeated performance. Running a performance test after each release is good practice to avoid performance degradation caused by newly deployed features.

Conclusion

In our opinion quality assurance goes well beyond testing and requires a cross-functional team. In a perfect world (and fortunately for this project) the team consists of development, testing, operations and product management. In particular the development team has an important role to support testing and operations since many roadblocks can simply be avoided assuming that development team is aware of them and has sufficient resources for the tasks at hand. The migration from FAST to SOLR was a challenging project due to the fact that an existing and mostly well-working system was replaced - replacing a broken system earns more positive feedback. A lot of effort went into the SOLR migration in order to avoid functional degradation, slow response times, production downtime and disgruntled customers. As far as the authors can tell, the project was considered a success, with very few customer complaints, plenty of opportunities to have fun and many lessons to be learned.

References

- [1] http://www.oewa.at/index.php?id=2
- [2] http://blogs.msdn.com/b/enterprisesearch/archive/2010/02/04/innovation-on-linux-and-unix.aspx
- [3] http://wiki.apache.org/solr/VelocityResponseWriter
- [4] Risk management in software quality assurance: Risk-Based Testing, Jan Sickinger 2011
- [5] Robert V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison Wesely Longman, 1999