

# Sphinx: High performance full text search for MySQL

Revision 1

May 16, 2008

This article is based on the Sphinx talk from MySQL UC 2008. It is not a verbatim transcription but pretty close to it.

## What's Sphinx?

- FOSS full-text search engine
- Specially designed for indexing databases
- Integrates well with MySQL
- Provides greatly improved full-text search
- Sometimes, can improve *non-full-text* queries
  - By more efficient processing (in *some* cases)
  - By distributed processing on a cluster (in all)
  - Details later in this talk

Sphinx is a free, open-source full-text search engine that was designed from ground up for indexing the content stored in local databases. Sphinx can pull the data from many different databases, but it ties especially well with MySQL. It offers a number of improvements compared to MySQL's built-in full-text indexes; but what's interesting, in some special cases it can also improve general purpose, non-full-text SELECT queries. It manages to do so either by processing the queries more efficiently than MySQL, or by distributing the load across the cluster of Sphinx nodes, or by combining both. We'll return to that in the second part of the talk.

## Why Sphinx?

- Major reasons
  - Better indexing speed
  - Better searching speed
  - Better relevance
  - Better scalability
- “Minor” reasons
  - Many other features
  - Like fixed RAM usage, “faceted” searching, geo-distance, built-in HTML stripper, morphology support, 1-grams, snippets highlighting, etc.

Sphinx improves full-text search speed, relevance, and scalability. It also adds a number of advanced features to full-text search, such as efficiently filtering, sorting and grouping search results, geodistance search, and a number of other minor ones.

## The meaning of “better”

- Better indexing speed
  - 50-100 times faster than MySQL FULLTEXT
  - 4-10 times faster than other external engines
- Better searching speed

- Heavily depends on the mode (boolean vs. phrase) and additional processing (WHERE, ORDER BY, etc)
- Up to 1000 (!) times faster than MySQL FULLTEXT in extreme cases (eg. large result set with GROUP BY)
- Up to 2-10 times faster than other external engines

But let me explain what exactly does that "better" mean for the major features. Speaking of performance figures, Sphinx is usually several times faster than either MySQL built-in full-text index, or than other open-source engines that we're aware of. The figures on the slide are not an exaggeration - we've actually seen one-thousand-fold improvement over MySQL when benchmarking complex queries that return several hundred thousand results from full-text search, and then group those results by some column - for instance, such queries would take over 20 minutes in MySQL, but under 1 second in Sphinx.

### **The meaning of “better” 2.0**

- Better relevancy
  - Sphinx phrase-based ranking in addition to classic statistical BM25
  - Sample query – “To be or not to be”
  - Optional, can be turned off for performance
- Better scalability
  - Vertical – can utilize many CPU cores, many HDDs
  - Horizontal – can utilize many servers
  - Out of the box support
  - Transparent to app, matter of server config changes

Sphinx also tries to improve search relevance by using a ranking method based on query phrase proximity to the matched document text. Most if not all other full-text systems only use so-called BM25 ranking, which only takes keyword frequency into account, and does not care about keyword positions. But that only works well when the keywords are more or less rare. With purely BM25 based system, this sample query will return the documents with lots of to's and be's at the top, but the well-know exact quote will not be placed anywhere near the top. In contrast, Sphinx will place the exact quotes at the very top, partial quotes just below them, and so on.

Finally, if your database is very large, or if your application demands a lot bandwidth, Sphinx can scale easily. Distributing the search across CPUs within a single server or across different physical servers is supported out of the box, and is a matter of several changes in the config file - and these changes will be fully transparent to the calling application.

### **How does it scale?**

- Distributed searching with several machines
- Fully transparent to calling application
- Biggest known Sphinx cluster
  - 1,200,000,000+ documents (yes, that’s a billion)
  - 1.5 terabytes
  - 1+ million searches/day
  - 7 boxes x 2 dual-core CPUs = 28 cores
- Busiest known Sphinx cluster
  - 30+ million searches/day using 15 boxes

Where's the scaling limit? Frankly, we don't know for sure. The biggest search cluster in terms of data size indexes 1 and a half terabytes of data, and the biggest one in terms of bandwidth handles 30 million searches per day over tens of gigabytes of data. There must be a limit at some point, because obviously the software won't scale to half a million machines that Google has, but we're yet to hit that limit.

### How does it work?

- Two standalone programs
  - **indexer** – pulls data from DB, builds indexes
  - **searchd** – uses indexes, answers queries
- Client programs talk to **searchd** over TCP
  - Via native APIs (PHP, Perl, Python, Ruby, Java)...
  - Via SphinxSE, pluggable MySQL engine
- **indexer** periodically rebuilds the indexes
  - Typically, using cron jobs
  - Searching works OK during rebuilds

Now that we have an idea of what Sphinx is, let's talk about how specifically does it work. Actually, there is no Sphinx. There are two standalone programs. The first one, called **indexer**, can pull the data from the database and create a full-text index over the pulled data. The other one, called **search daemon** or **searchd** for short, can then answer full-text queries using the index built by **indexer**. Client programs can talk to **searchd** over TCP either directly, using native API in one of the listed languages, or through MySQL with Sphinx Storage Engine, which is a kind of a **searchd** API that can be built into MySQL. **Indexer** needs to be run periodically in order to rebuild the indexes. The rebuild uses a shadow copy of the index, so searches work fine during the rebuild anyway, they do not stall.

### Indexing workflow

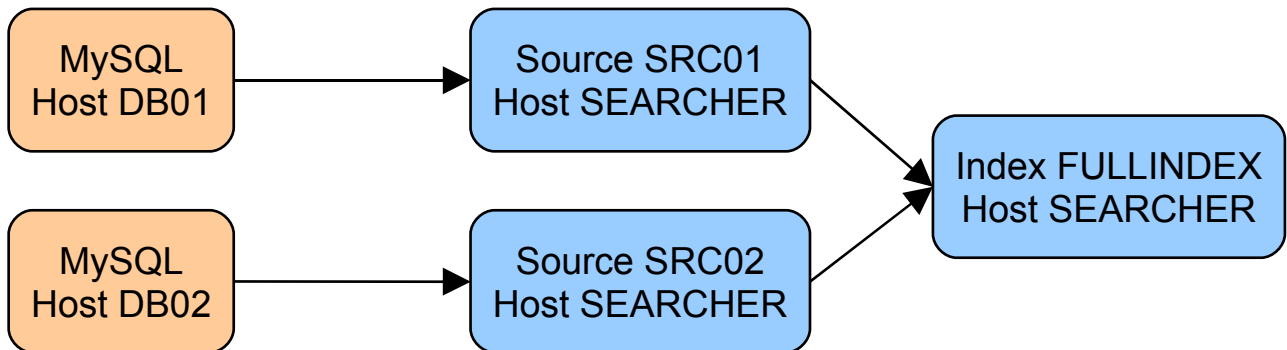
- Data sources – “where to get the data?”
  - MySQL, Postgres, XML pipe...
- Local indexes – “how to index the data?”
  - Also storage location, valid characters list, stop words, stemming, word forms dictionaries, tokenizing exceptions, substring indexing, N-grams, HTML stripping...
- Arbitrary number of indexes
- Arbitrary number of sources per index
  - Can pull data from different DB boxes in a shard

Speaking of indexing workflow, there are two major entities: data sources, and full-text indexes. Sources simply tell the **indexer** where can it get the data from. Index settings also specify where to store the resulting index, what are the valid in-word characters, what stopwords should be ignored, and so on.

Every instance of the search daemon can serve an arbitrary amount of indexes, and every single index can in turn be built from an arbitrary number of sources. So you can for instance pull some of the data from MySQL, some from Postgres, and create a big unified index over that.

## Sample – all eggs in one basket

Combining sharded database data for the ease of use



More practically, you can for instance combine the data from 2 database boxes on a single search box.

## Distributed indexes

- Essentially, lists of local and **remote** indexes

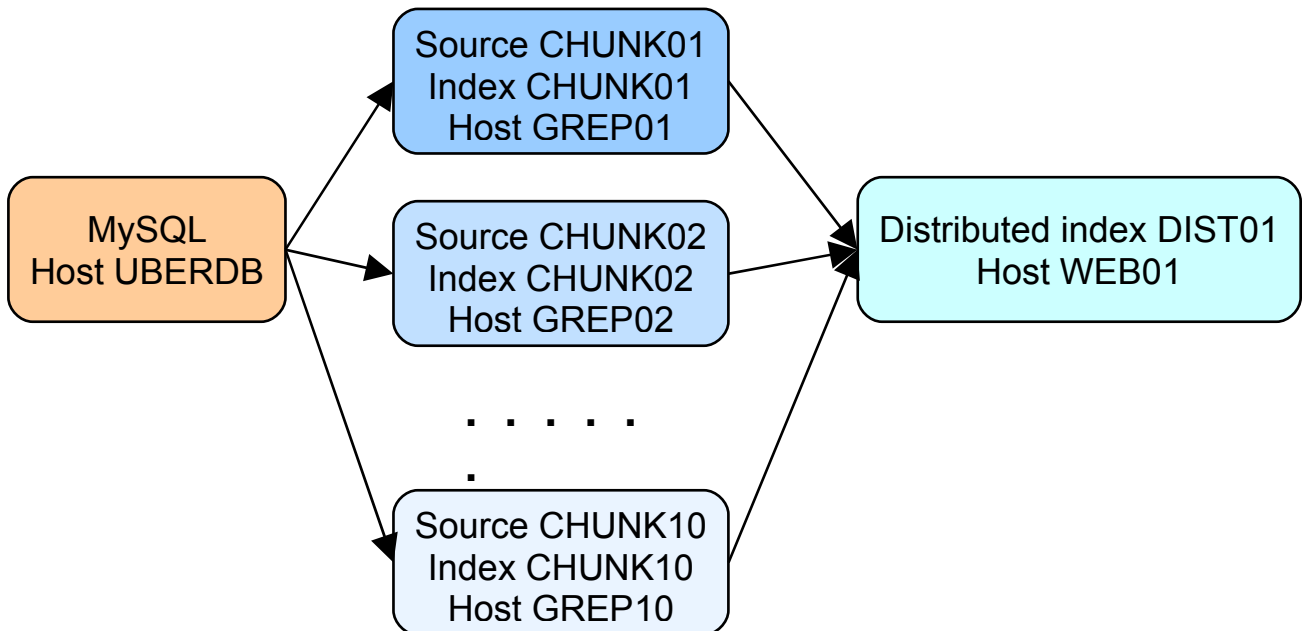
```
index dist1
{
    type = distributed
    local = chunk1
    agent = box02:3312:chunk02
    agent = box03:3312:chunk03
    agent = box04:3312:chunk03
}
```

- All local indexes are searched sequentially
- All remote indexes are searched in parallel
- All results are merged

Besides local indexes, Sphinx also supports so-called distributed ones. Essentially the distributed index is simply a list of several local and several remote indexes that all will be searched when the distributed index is queried. The nice thing about distributed indexes is that all the remote searches are carried out in parallel. This obviously helps to reduce query latencies.

## Sample – divide and conquer

Sharding full-text indexes to improve searching latency



Here's the sample setup where the huge database is split across 10 search servers, each carrying 1/10th of the documents. Those search servers are in turn queried by the distributed index set up on the web frontend box. The searchd instance on web frontend does not add much impact because the aggregation of the results is pretty fast. And, of course, the queries execute several times - not 10, but maybe 8 or 9 times faster than they would on a single search server.

### Searching 101 – the client side

- Create a client object
- Set up the options
- Fire the query

```
<?php
include ( "sphinxapi.php" );
$cl = new SphinxClient ();
$cl->SetMatchMode ( SPH_MATCH_PHRASE );
$cl->SetSortMode ( SPH_SORT_EXTENDED, "price desc" );
$res = $cl->Query ( "ipod nano", "products" );
var_dump ( $res );
?>
```

Okay, but how do we search those magic indexes? Pretty simple. Here's a 6-line sample in PHP that will search for iPod Nano as an exact phrase in the index called products, and sort the results by the price. The required lines are in red. So the simplest possible query only takes 3 lines.

### Searching 102 – match contents

- Matches will always have document ID, weight
- Matches can also have numeric attributes
- No string attributes yet (pull ‘em from MySQL)

```
print_r ( $result["matches"][0] ):

Array (
    [id] => 123
    [weight] => 101421
    [attrs] => Array (
    [group_id] => 12345678901
    [added] => 1207261463 ) )
```

What is the format of results returned in the previous sample? Sphinx will always return the document identifier that you provided when building the index, and the match weight, or relevance rank, that it computed. Depending on the index settings, it can also return user-specified numeric attributes. These attributes in Sphinx correspond to table columns in MySQL. There can be an arbitrary number of attributes. Not that neither the full original row nor any of its text fields are currently stored in Sphinx index; only explicitly specified numeric attributes are stored and can be retrieved from Sphinx. Other columns will have to be pulled from MySQL.

### Searching 103 – why attributes

- Short answer – efficiency
- Long answer – efficient filtering, sorting, and grouping for big result sets (over 1,000 matches)
- Real-world example:
  - Using Sphinx for searching only and then sorting just 1000 matches using MySQL – up to **2-3 seconds**
  - Using Sphinx for both searching and sorting – improves that to **under 0.1 second**
  - Random row IO in MySQL, no row IO in Sphinx
- Now imagine there’s 1,000,000 matches... ☺

One might ask, why introduce numeric attributes to the full-text search system at all? The answer is efficiency. Using full-text engine for full-text search only and the database for everything else works fine as long as your result sets are small – say, up to 1 thousand matches. However, imagine that there's 100 thousand matching rows that you want sorted by product price. Pulling those 100 thousand rows from the full-text engine and passing them to the database for sorting is going to be now just slow but extremely slow – and in fact, it's the database part which is going to be slow, not the full-text search. On the other hand, if the full-text engine does have the price data stored in its index, it can quickly sort the matches itself, and return only those first 100 that you're really interested in. That's exactly what Sphinx attributes are for.

### Moving parts

- SQL query parts that can be moved to Sphinx
  - Filtering – WHERE vs. SetFilter() or fake keyword
  - Sorting – ORDER BY vs. SetSortMode()
  - Grouping – GROUP BY vs. SetGroupBy()
- Up to 100x (!) improvement vs. “naïve” approach

- Rule of thumb – move everything you can from MySQL to Sphinx
- Rule of thumb 2.0 – apply sacred knowledge of Sphinx pipeline (and then move everything)

More formally, the attributes can be used to efficiently filter, sort, and group the full-text search matches. The filters are similar to WHERE clause in plain SQL; sorting is similar to ORDER BY; and grouping is similar to GROUP BY. Having all this supported on the engine side lets us move all the additional search results processing from MySQL to Sphinx. And the savings can be huge. Remember that 20-minute MySQL query versus 1-second Sphinx query benchmark? That's exactly about it.

But in order to keep Sphinx-side queries efficient as well, we'd want to understand how they work internally. So let's proceed to the internal searching workflow.

### Searching pipeline in 30 seconds

- Search, WHERE, rank, ORDER/GROUP
  - “Cheap” boolean searching first
  - Then filters (WHERE clause)
  - Then “expensive” relevance ranking
  - Then sorting (ORDER BY clause) and/or grouping (GROUP BY clause)

Searching works as described on the slide – everything begins with looking up the next document that satisfies the full-text query; then the document is checked against the specified filtering conditions, if there are any; then if it still matches, we compute the relevancy; and pass the document to the sorting queue.

### Searching pipeline details

- Query is evaluated as a boolean query
  - CPU and IO,  $O(\text{sum}(\text{docs\_per\_keyword}))$
- Candidates are filtered
  - based on their attribute values
  - CPU only,  $O(\text{sum}(\text{docs\_per\_keyword}))$
- Relevance rank (weight) is computed
  - CPU and IO,  $O(\text{sum}(\text{hits\_per\_keyword}))$
- Matches are sorted and grouped
  - CPU only,  $O(\text{filtered\_matches\_count})$

Every operation in the pipeline has an associated cost. The boolean query part is essentially reading several per-keyword document ID lists and intersecting those lists, this puts some impact both on CPU and disk IO depending on how much documents there are per each keyword. Then the filtering happens, and it eats some CPU depending on how much matches the full-text query retrieved. Then the relevance ranking performs some disk IO again, it reads keyword positions and computes just how close the phrases in the document are to the query phrase. The amount of CPU and IO consumed by relevance ranking is now proportional not to the amount of full-text matches but to the amount of filtered matches. Finally, the filtered and ranked matches are sorted. This only needs CPU time, also proportional to the amount of filtered matches.

Understanding this pipeline immediately lets us answer some frequently asked optimization questions, such as the following question.

### Filters vs. fake keywords

- The key idea – instead of using an attribute, inject a fake keyword when indexing

```
sql_query = SELECT id, title, vendor ...  
  
$sphinxClient->SetFilter ( "vendor", 123 );  
$sphinxClient->Query ( "laptop", "products" );  
  
vs.  
  
sql_query = SELECT id, title, CONCAT('_vnd', vendor) ...  
  
$sphinxClient->Query ( "laptop _vnd123", "products" );
```

What's better for performance, using attributes and filters, or using so-called fake keywords?

The key idea behind fake keywords is as follows – instead of storing an attribute along with each document, we replace its value with a specially constructed keyword, and add this keyword to the full-text query where needed. This way we replace a condition with an equivalent full-text query.

As we can see from the pipeline, it all depends on the keyword and filter selectivity.

### Filters vs. fake keywords

- Filters
  - Will eat extra CPU
  - Linear by pre-filtered candidates count
- Fake keywords
  - Will eat extra CPU and IO
  - Linear by per-keyword matching documents count
  - That is strictly equal (!) to post-filter matches count
- Conclusion
  - Everything depends on selectivity
  - For selective values, keywords are better

If the query returns many pre-filter matches, say 1 million, and then filters select only 1 thousand of those and throw away the other 999 thousand, it's more efficient to have a fake keyword – because it's faster to batch intersect 1 million document ID list produced by the keyword with 1 thousand entries long one produced by the fake keyword than to lookup 1 million matches one by one, check each one, and reject most of them. On the other hand if the keywords return only a handful of matches, but each filter value (say, male or female gender) matches millions of records, you should be using filters as is.

So the rule of thumb is to replace filters with fake keywords if they are selective, and keep them as filters if they are not. We can draw an analogy with MySQL here. Think of fake keywords as of indexes. It makes sense to create an index if the column is selective, otherwise you'll be better off with full scan. The general idea stays the same even though the implementation of filters, or conditions, is different.

## Sorting

- Always optimizes for the “limit”
- Fixed RAM requirements, never an IO
- Controlled by **max\_matches** setting
  - Both server-side and client-side
  - Defaults to 1000
- Processes **all** matching rows
- Keeps at most N best rows in RAM, at all times
- MySQL currently does not optimize that well
- MySQL sorts **everything**, then picks up N best

Sorting in Sphinx is also different from MySQL. Basically, it optimizes for the LIMIT clause much better. Instead of honestly keeping and sorting millions of matches, it only keeps a few best matches – at most 1 thousand by default, though that can be easily raised in the config file to 10 thousand, or 100 thousand, or whatever. This is more efficient than MySQL's approach of keeping everything, then disk-sorting everything, and only then applying the LIMIT clause. Keeping track of 1 thousand best matches allows sorting to run in a few kilobytes of RAM even on huge results sets, and also guarantees that there will be no disk IO for sorting. One might argue that 1000 results is not enough, but recent research indicates that most end users won't go to search results page number 17 thousand anyway.

## Grouping

- Also in fixed RAM, also IO-less
- Comes at the cost of COUNT(\*) precision
  - Fixed RAM usage can cause underestimates
  - Aggregates-only transmission via distributed agents can cause overestimates
- Frequently that's OK anyway
  - Consider 10-year per-day report – it will be precise
  - Consider “choose top-10 destination domains from 100-million links graph” query – 10 to 100 times speedup at the cost of 0.5% error might be acceptable

Finally, grouping is also different. What's most important, it's always performed in fixed RAM, just as sorting. But this time the reduced footprint comes at cost: if there's too much rows **after** the grouping, COUNT(\*) values might be off. Sphinx is intentionally trading RAM footprint and query performance for COUNT(\*) precision. However, this trading only happens when there are really many groups – groups, not just matches. A number of practical tasks never suffer from it. Consider, for instance, a query that returns the amount of matching rows for every single day during the last 5 years. After grouping, it will return about 1800 rows, and the per-group counts will be perfectly precise. In the cases when there's many more groups, having the approximate counts might be OK as well. Consider this real-world example query: "process 10's of millions of hyperlinks, and give me top-10 domains that link to YouTube". As long as the figures are noticeably different, there's little difference whether the winner domain had exactly 872,231 or 872,119 links. Not to mention that the query could then be repeated with the filter on top-100 source domains and thus forced to return exact results.

## More optimization possibilities

- Using query statistics
- Using multi-query interface

- Choosing proper ranking mode
- Distributing the CPU/HDD load
- Adding stopwords
- etc.

There are even more optimization possibilities, but the time is limited, so we'll briefly mention only the first two ones – query statistics and multi-queries.

### Query statistics

- Applies to migrating from MySQL FULLTEXT
- Total match counts are immediately available – no need to run 2nd query
- Per-keyword match counts are also available – can be used not just as minor addition to search results – but also for automatic query rewriting

Unlike with MySQL built-in full-text, Sphinx always provides both total matching documents count, that can be used to implement paging – and also per-keyword match counts. So if you're migrating from MySQL full-text search, there's no need to run the 2<sup>nd</sup> query just to get the count any more.

### Multi-query interface

- Send many independent queries in one batch, allow Sphinx optimize them internally
- Always saves on network roundtrip
- Sometimes saves on expensive operations
- Most frequent example – same full-text query, different result set “views”

The idea behind multi-query optimization is as follows – you pass a number of queries to Sphinx in one batch, and it tries to internally optimize the common parts. One frequent case is when you need to perform the very same full-text search but provide several different views on its results, with different sorting or grouping settings. For instance, a products search engine might want to provide per-vendor product counts along with the product search results. Passing both these queries in a single batch will save on the expensive full-text search operation. Internally, it will be only performed once instead of twice, but its results will be sorted and grouped differently.

### Multi-query sample

```

$client = new SphinxClient ();
$q = "laptop"; // coming from website user

$client->SetSortMode ( SPH_SORT_EXTENDED, "@weight desc" );
$client->AddQuery ( $q, "products" );

$client->SetGroupBy ( SPH_GROUPBY_ATTR, "vendor_id" );
$client->AddQuery ( $q, "products" );

$client->ResetGroupBy ();
$client->SetSortMode ( SPH_SORT_EXTENDED, "price asc" );
$client->SetLimit ( 0, 10 );

$result = $client->RunQueries ();

```

Here's the example script. It runs a general relevance sorted search over the products, then another search to pull per-vendor counts, then yet another search to pull 10 cheapest matches. Compared to running these searches one by one, multi-query interface should yield about 2 times better query time.

### Offloading non-full-text queries

- Basic “general” SQL queries can be rewritten to “full-text” form – and run by Sphinx

```
SELECT * FROM table WHERE a=1 AND b=2
ORDER BY c DESC LIMIT 60,20

$client->SetFilter ( "a", array(1) );
$client->SetFilter ( "b", array(2) );
$client->SetSortBy ( SPH_SORT_ATTR_DESC, "c" );
$client->SetLimit ( 60, 20 );
$result = $client->Query ( "", "table" );
```

- Syntax disclaimer – we are a full-text engine!
- SphinxQL coming at some point in the future

And now to finish the talk with something completely different, let's discuss how Sphinx could be used to offload general SQL queries from MySQL and even optimize them.

Well, it can be used to offload general SQL queries from MySQL, and even optimize them.

Because Sphinx provides all that additional filtering, sorting and grouping functionality, it's easy to rewrite basic SELECT queries that do not involve complex JOINS etc in the form that Sphinx can handle. Here's a sample. I know that the resulting syntax is somewhat clunky compared to SQL, but hey, we were starting as full-text engine after all. We'll improve on that in the future.

### Why do that?

- Sometimes Sphinx reads outperform MySQL
  - Sphinx always does RAM based “full scan”
  - MySQL index read with bad selectivity can be slower
  - MySQL full-scan will most likely be slower
  - MySQL can't index *every* column combination
- Also, Sphinx queries are easier to distribute
- But Sphinx indexes are essentially read-only
  - well, almost (attribute update is possible)
- Complementary to MySQL, **not** a replacement

But why would one do that offloading at all? The answer is, as usual, efficiency. It turned out that there are some border cases when Sphinx can outperform MySQL. Even though Sphinx always does a kind of a “full scan”, this scan is RAM based and pretty quick – it will definitely lose to nice MySQL query that properly uses MySQL indexes, but there are cases when it's impossible to build an efficient index in MySQL. Also, it's easier to distribute Sphinx queries.

### SELECT war story

- Searches on [Sahibinden.com](http://Sahibinden.com)

- Both full-text and not
- “Show all auctioned items in laptops category with sellers from Ankara in \$1000 to \$2000 range”
- “Show matches for ‘ipod nano’ and sort by price”
- Many columns, no way to build covering indexes
- Sphinx full scans turned out being 1.5-3x better than MySQL full scans or 1-column index reads
- Also, code for full-text and non-full-text queries was unified

Here’s a live, bleeding example of the offloading technique. Sahibinden is a leading Turkish auction website – yes, it’s their eBay, with millions of page views daily. Initially they’ve approached us because they had a showstopper problem with full-text searches. However, in fact their non-full-text item searches also started to be several – maybe 2 or 3 times – faster when we moved them to Sphinx. Of course, there was an index on every column, but single-column index chose too many matches and MySQL queries were slow. And they had many columns, so it was simply impossible to build an efficient covering index for every possible column combination. As a nice side effect, both full-text and non-full-text search code was unified, I was told that their programmers liked that a lot.

### **GROUPBY war story**

- Domain cross-links report on BoardReader.com
  - “Show top 100 destination domains for last month”
  - “Show top 100 domains that link to YouTube”
  - ~200 million rows overall
- Key features of the report queries
  - They always group by domain, and sort by counts
  - The result sets are small
  - Approximate results are acceptable – don’t care whether there were exactly 813,719 or 814,101 links from domain X

And here’s another example of a bigger scale that also included distributing the load. Boardreader, the forum search engine that indexes a billion of forum posts using Sphinx, was adding a feature called domain profile. This profile needs to include a report on the hyperlinks, that is, the amount of the inbound links to this domain, top-10 destination domains for outbound links, and so on. It also needs to be dynamic – that is, we might want to view the results for the last week only, or for the weeks during the last month, etc. And the underlying source data is a 200 million row collection of links that’s regularly updated by the crawlers.

All the required link report queries are pretty similar in nature. They all take many source rows, group them by domain or a subdomain, and extract a few best rows sorted by per-domain links count. Also, it’s obviously okay to sacrifice perfect per-domain count precision as long as the order is correct.

### **GROUPBY war story**

- MySQL prototype took up to 300 seconds/query
- Sphinx queries were much easier to distribute
- URLs are preprocessed, then full-text indexed
  - <http://test.com/path/doc.html> → test\$com, test\$com\$path, test\$com\$path\$doc, ...
- Queries are distributed over 7-machine cluster

- Now takes within 1-2 seconds in the worst case
- This is **not** the main cluster load
- Main load is searching 1.2B documents

We started with prototyping the thing with MySQL. However, while prototype MySQL queries were fine for the smaller domains, they slowed down to a crawl for the bigger ones. Extreme cases such as building a report on top-10 domains that link to Google or YouTube took up to 300 seconds to complete. The only possible route of optimizing that to a reasonable value was distributing the queries across a cluster.

In theory it could have been possible to distribute MySQL queries and manually aggregate their results. However, parallelizing the queries using Sphinx was much simpler to implement. Also, Sphinx was a bit faster on a single node as well – not really much, maybe 1.5 or 2 times faster.

Because indexing all the possible URL prefixes was a waste of disk space and indexing time, we also had to create a simple MySQL UDF function that extracted those parts of the URL that were interesting to us. That UDF is used at indexing time. Note that the original data is not altered at all, it gets preprocessed on the fly during the data pull query that the indexer runs.

The queries were then distributed across the same search cluster that handles the main load of searching through a billion of documents. And despite that was an additional load on an already busy cluster, queries suddenly started to complete in a few seconds anyway.

### **Summary**

- Discussed Sphinx full-text engine
- Discussed its pipeline internals – helps to optimize queries
- Discussed how it can be used to offload and/or optimize “general” SQL queries
- Got full-text queries? Try Sphinx