Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Jarosław Gomułka [DRAFT]

Shuffla: fast and scalable full-text search engine

Praca magisterska
napisana pod kierunkiem
dr. Pawła Gawrychowskiego

Wrocław 2012

Oświadczam, że pracę magisterską wykonałem samodzielnie i zgłaszam ją do oceny.

Data: ..... Podpis autora pracy: .....

Oświadczam, że praca jest gotowa do oceny przez recenzenta.

Data: ..... Podpis opiekuna pracy: .....

Streszczenie

Celem tej pracy jest stworzenie efektywnego i skalowalnego narzędzia do dokładnego przeszukiwania zbioru użytkowników dużego portalu społecznościowego. Przechowane dane to imię, nazwisko, wiek, oraz miejsce zamieszkania.

Użytkownik portalu może wyszukiwać innych użytkowników poprzez:

1. podanie imienia (jego prefiksu lub podsłowa),
2. podanie nazwiska (jego prefiksu lub podsłowa),
3. podanie wieku (dokładnie bądź przez określenie dolnej i górnej granicy),
4. podanie miejsca zamieszkania (jego prefiksu lub podsłowa).

Kolejnym założeniem jest konieczność personalizacji wyników wyszukiwania. Na przykładu użytkownik z San Francisco szukający Jane powinien dostać w pierwszej kolejności osoby o imieniu Jane z San Francisco, następnie osoby z Kaliforni, Stanów Zjednoczonych, a dopiero na samym końcu z innych części świata.

Jednym z możliwych rozwiązań tak postawionego problemu jest zastosowanie jednej z popularnych relacyjnych baz danych. Inną możliwością jest użycie wyszukiwarki tekstowej takiej jak Apache Solr czy Sphinx Search. Jeszcze innym pomysłem jest zaimplementowanie własnego narzędzia. Celem tej pracy jest przedstawienie Shuffli, wyspecjalizowanej wyszukiwarki stworzonej przeze mnie z myślą o tym konkretnym problemie. Oprócz przedstawienia architektury i szczegółów implementacji, porównam jej efektywność z PostgreSQL, Apache Solr, Sphinx Search.

Pracuję nad portalem nk.pl (który jest największym polskim portalem społecznościowym) od początku roku 2011. W sierpniu 2011 roku zostało mi powierzone rozwiązanie tego problemu.

Słowa kluczowe: bazy danych, nosql

Shuffla: fast and scalable full-text search engine

Jarosław Gomułka

October 3, 2012

Abstract

Given a list of people (their first name, last name, age, home town) we want to provide high-available, scalable and very fast solution that can search through such data. User can narrow search results by:

1. defining first name (its prefix or substring)
2. defining last name (its prefix or substring)
3. narrowing age (user can provide lower or/and upper bound for age)
4. defining city (its prefix or substring)

It would be great if results could be personalized. For example, if user from San Francisco is looking for Jane, top results should show Jane's from San Francisco, then Jane's from California, then rest of U.S. One of possible solution is to use relational database. Other solution is to use full-text search engine like Apache Solr or Sphinx Search. Another solution is to develop very own search engine dedicated to solving such task. In this paper I will describe Shuffla, full-text search engine created by me. I will compare it to Sphinx, Solr, PostgreSQL.

I'm working on nk.pl (which is biggest polish social networking website) since the beginning of 2011. In August 2011 I was appointed to solve the task described above.

Keywords: database, nosql

1 Problem definition

Lets define table definition by combination of

1. table name
2. list of pairs <column name, type of column>

Type of column can be either a string or a number.

1.1 Selecting data

There are different possible conditions for different types. Possible conditions for numbers are comparisons, inequalities and strict inequalities. Strings are compared using lexicographic order so comparisons, inequalities and strict inequalities should be supported. There are two additional conditions

1. Checking if given string is prefix of selected column value.
2. Checking if given string is substring of selected column value.

Presenting results:

1. User can define order in which results will be returned.
2. User can narrow number of results by defining offset and limit.

Narrowing result described in abstract is possible with defined model. Personalizing results can be done by mapping cities to GPS coordinates. GPS coordinates can be added to scheme. For selecting users in some range we can add conditions

1. someLatitiude - radius $\leq$ latitude $\wedge$ latitude $\geq$ someLatitude + radius
2. someLongitude - radius $\leq$ longitude $\wedge$ longitude $\geq$ someLongitude + radius

2 Shuffla's competition

Problem described in this paper is solvable by various existing software. There are two most common kinds of software which can be used:

2.1 Relational databases

In this paper I'm only testing PostgreSQL which is in my opinion best RDBMs nowadays.

2.1.1 PostgreSQL

PostgreSQL has really cool features like: point in time recovery, tablespaces, asynchronous replication, online/hot backups, a sophisticated query planner/optimizer, and write ahead logging for fault tolerance. It supports international character sets, multibyte character encodings, Unicode, and it is locale-aware for sorting, case-sensitivity, and formatting. It is highly scalable both in the sheer quantity of data it can manage and in the number of concurrent users it can accommodate. There are active PostgreSQL systems in production environments that manage in excess of 4 terabytes of data Jarek: Kopiowane ze strony Postgresa, mam nadzieje, że to nie plagiat . PostgreSQL keeps its data on disk. PostgreSQL implements their own master-slave replication, there are also few middlewares which could provide master-master replication [1]. There is a slow log, error log, PostgreSQL performance can be easily monitored by monitoring software like munin, new relic. SQL implies very strong typing system. PostgreSQL processes comparisons and inequalities very fast as long as proper indexes are created. Big disadvantage of using PostgreSQL for this problem is performance on substring queries. B-tree based indexes are not supporting fast processing of substring queries. There are attempts to create full-text indexes (`gist`, `gin`) which could outperform classical index based on B-tree. Example of processing query by B-tree: `first name = John and substring(last name) = qwertyabc and limit = 1`. In this case engine is finding first person with first name John. It takes $O(\log n)$ time. Finding another person with first name John takes $\mathcal{O}(1)$. Unfortunately all these rows must be processed one by one for checking if they contain substring `qwertyabc`. Since there is no one with such name, engine will process all guys named John, and it will find nothing.

2.2 Full-text search engine

Another solution is to use full-text search engine. The most popular full-text search engines are Sphinx search and Lucene Solr. Almost all big players are choosing Sphinx and Solr over others such as Elastic Search, Xapian so I'm going to consider only these two. Both engines are based on similar data structure called inverted index. Inverted index keeps list of every occurring word in database. For each word inverted index remembers where this word occurs. This is similar to glossaries which we can often see at the end of books. Such inverted index is stored on disk. When processing query, engine selects best strategy to follow. Strategy can be seen as procedure

1. Narrow words from inverted index which need to be process as much as possible.
2. Process remaining words.

For example if query is `first name = John and limit 1` then we are only interested in word John from inverted index. Engine finds first occurrence and returns it. If query is `first name = John and substring(last name) = qwertyabc and limit = 1` then it is likely that there are more Johns then all people with last name containing `qwertyabc`. Engine could use suffix tree which contains all words from inverted index. It would find that there is nobody with such last name. Such output would be rendered

very fast, probably even without reading inverted index. In worst case scenario algorithm works in linear time. Example for such a case could be `substring(last name)=a and substring(last name) = b and ... and substring(last name) = z and limit = 1`. Almost whole inverted index must be processed but still result set will be empty.

2.2.1 Solr

Solr is a popular open source search platform from the Apache Lucene project. It has lots of very cool features like faceted searches, geospatial searches, string tokenizers (makes John and Johnny to be interpreted as the same name). Solr has great admin interface with comprehensive statistics on cache utilization, updates, and queries. Jarek: Jedno zdanie zgapione od Solra. It is easily extendable for new features. Solr implements master-slave replication which solves scalability problems. Everything is perfect except one thing.

Solr does not modify database in real-time. Data is not added immediately to database after insert/update commands. Data is actually inserted after calling commit command. Commit is very costly operation because it requires all caches to be invalidated. Even for table with a few entries, commit takes more than 0.1 sec. Time for commit grows with growing size of data. Instance of Solr in nk contains 40 million entries and commit takes about 20 seconds. Commit does not block database which is very important. After commit, search engine is in warming stage. It means that all caches are flushed so first queries can run slowly. After number of processed queries, engine is past warming stage and queries are handled much faster.

2.2.2 Sphinx search

Sphinx search does not have replication. It is unacceptable for big social website. There is a way to simulate replication. Inverted index is stored on disk. Directory containing index could be replicated by file system (for example network file system). But if index is big than syncing such directories would take too much time and would be inefficient and problematic.

There are several high-traffic websites with Sphinx search. They horizontally partition searched data across search nodes and then process it in parallel [2].

Replication is very important to us because in case of server failure we want to avoid downtime. This eliminated Sphinx Search from our list of potencial search engines.

Except of replication sphinx has another problem. There are two different kinds of indexes in sphinx.

1. Real-time index - enables inserts
2. Normal index - modifications possible only by rebuilding whole index

Real-time index has many caveats [**?**]. Prefix and substring queries are not supported yet. Periodical rebuild of index with more than 40 million elements is problematic. Apache Solr and Sphinx search are using similar data structures. Since number of features implemented by Solr is much greater than in Sphinx search I decided to look into only Solr.

It seems that none of existing solutions are processing search queries very fast. In worst case scenario it is always $\mathcal{O}(n)$. All solutions are working on data stored on disk. In age when RAM is cheap and servers with +32GB RAM are considered average, solutions working in RAM memory are more than welcomed. Although data stored on disk could be transfer to RAM by ramdisks, it is not primary target to software creators. Solr has designated class `solr.RAMDirectoryFactory` which keeps all data in RAM memory but it does not support replication. For me it seems that this area has a room for improvement. That is why I decided to create my own search engine.

3 Algorithm description

We can clearly see analogy to multidimensional range searches. All conditions (except substring condition) can be represented as half-plane (or half-planes) which narrows the search space. There are three data structures for performing efficient searches in a multidimensional space:

1. Range Tree [10]

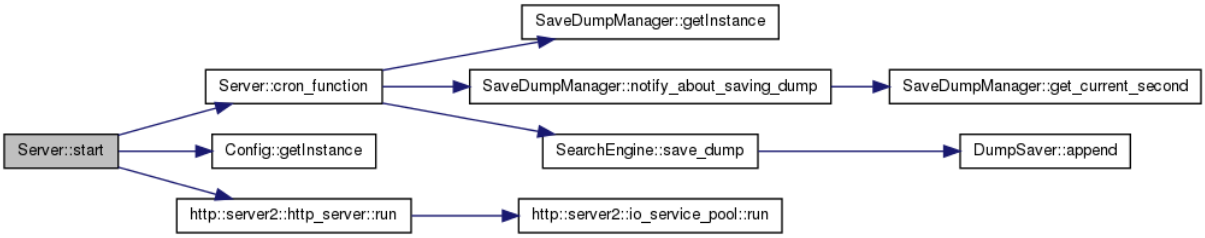


Figure 1: Processing request begins

2. Inverted Index [10]
3. K-D tree [10]

Complexities (where $d >= 2$)

Algorithm	Insert (without balancing)	Delete (without balancing)	Search	Space
K-D tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n^{1-1/d} + k)$	$\mathcal{O}(n)$
Inverted index	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Range tree	$\mathcal{O}(\log^{d-1} n)$	$\mathcal{O}(\log^{d-1} n)$	$\mathcal{O}(\log^{d} n)$	$\mathcal{O}(n \log^{d-1} n)$

Since both Apache Solr and Sphinx Search are using inverted index I decided not to follow this direction and try something new. It seems that K-D tree could outperform inverted index on search queries.

I chose to implement K-D trees over Range trees because:

1. They are only worse in processing search requests. Operations which modify data i.e. insert/delete are more important. If they are very fast, scaling can be easily achieved (for example by implementing master-slave replication). Moreover, while the search bounds are worse in theory, they seem to be significantly better in practice.
2. Space complexity of range trees is unacceptable for this kind of problem. With 5 columns and 10 million rows, algorithm would require more than 100GB of RAM.

We also have queries with substring conditions. In each node of K-D tree we could store structure which efficiently process substring queries. We could use suffix trees, for example Ukonnen's algorithm [15] .

Complexities:

1. Adding text to tree: $\mathcal{O}(|text|)$
2. Removing text from tree $\mathcal{O}(|text|)$
3. Searching text in tree $\mathcal{O}(|text|)$

Unfortunately adding linear structure to every node of K-D tree increases the space complexity to $\mathcal{O}(n \log n)$

4 Implementation

4.1 HTTP Service

(Figure 1).

I decided that it would be best if search engine could mimic RESTful service. Every search/insert/delete request should be sent as a HTTP request. Server works in the multi-threaded environment. As a base I used `example2` [3] from `boost::asio` documentation which fulfills my requirements. Threads are managed by `boost::io_service` [4]. There is another thread for periodical functions. Currently its invoking only saving snapshots.

(Figure 2).

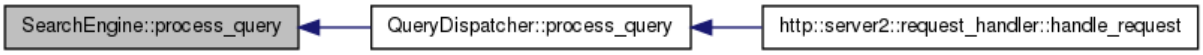


Figure 2: Passing control to SearchEngine class

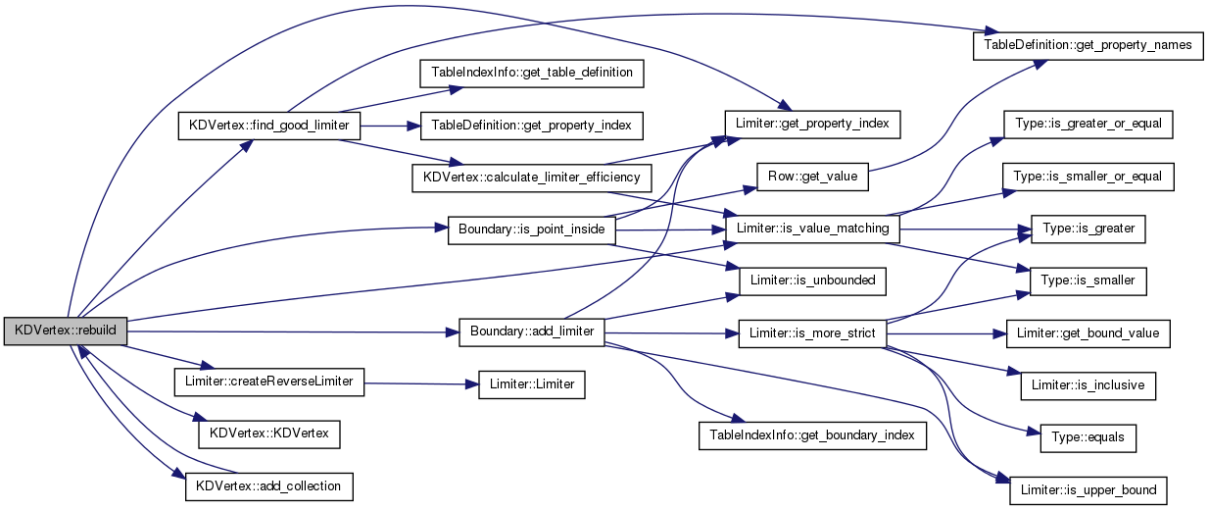


Figure 3: Rebuilding process

4.2 Validation and passing queries to SearchEngine class

`http::server2::request_handler::handle_request` passes requesting url to `QueryDispatcher`. QueryDispatcher converts every request to corresponding class by using `QueryCreate`. For example, search request is converted to `SearchQuery`, insert is converted `InsertQuery`, and so on. At this level, syntax validation is performed. In case of an error, HTTP error code 400 is returned, and error message is appended to the log file with errors.

Instances of query classes are sent to `SearchEngine` class. `SearchEngine` class is responsible for:

1. data validation (types, correspondence to table definition),
2. measuring running time (slow log purpose),
3. locking tables in case of modifying query,
4. passing valid requests to K-D Trees.

4.3 Processing insert/delete requests

Inserting/deleting without tree balancing is pretty straight-forward. Just simple recursion. Balancing is a little bit tricky.

(Figure 3).

Balancing can be done using any linear algorithm for finding median in set. I've decided to take different approach. In `KDVertex::fing_good_pivot` I'm selecting randomly about 30 rows belonging to node and I'm iterating over them. Based on coordinate value in this row I'm creating a `Pivot`. `Pivot` is simple class which takes `Row` and `property_name`. For any `Row` pivot can check if given Row satisfy condition defined inside `Pivot`. Algorithm iterates over all rows in node and in linear time calculates pivot efficiency. If pivot efficiency is good enough (See definition of well-balanced tree) than I'm stopping calculations. I've got a pivot and I'm recursively rebuilding two subtrees. I've decided to use this solution because of space efficiency which is $\mathcal{O}(1)$. Space complexity is crucial because if it reaches linear size than rebuilding root may cause memory usage to double. Such behaviour is very unwelcome in server environment. Implemented solution is $\mathcal{O}(n^2)$ but in average case it works well because $\frac{1}{3}$ of rows are creating good pivots.

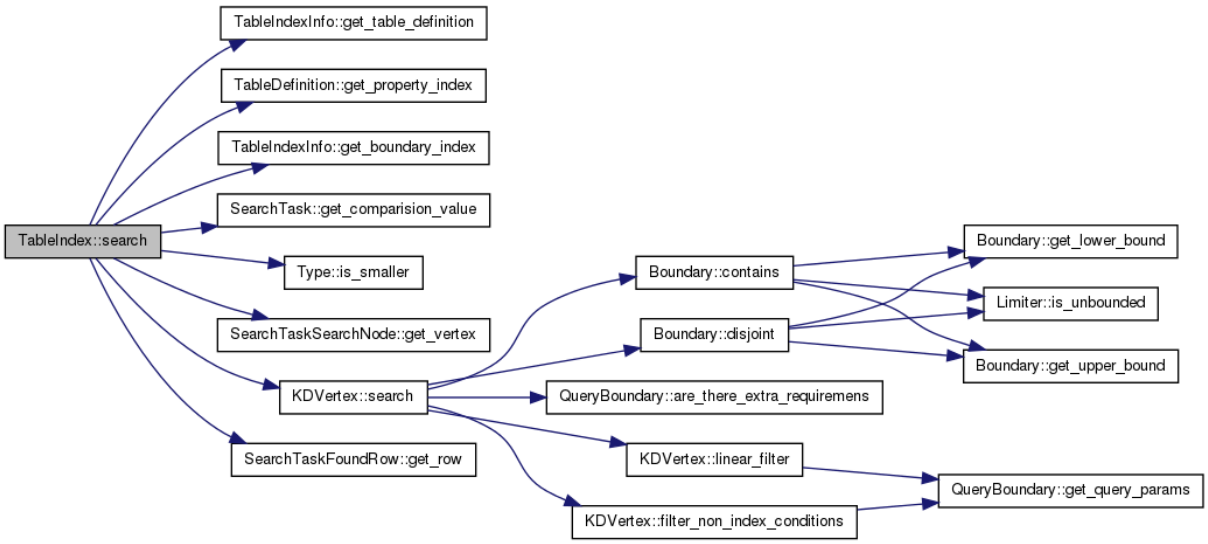


Figure 4: Processing search request

4.4 Processing search requests

(Figure 4).

When request data is validated, control is passed to `TableIndex` class. Actual computations begins here. Lets look at processing search request. Method `TableIndex::search` contains priority queue which processes all search events. There are different event objects for different tasks. There is `SearchTaskSearchNode` which contains node to process. There is also `SearchTaskFoundRow` which always contains one row. This row may be inserted to task queue if and only if it satisfy all conditions defined in search request. Both these classes extends abstract class `SearchTask`. When processing `SearchTaskFoundRow` queue just adds row to result set. When processing `SearchTaskSearchNode` queue passes control to `KDVertex::search` which returns all events that should be added to queue. `KDVertex::search` checks if boundary defined by query contains/is disjoint with current node's boundary. If it's disjoint there is nothing more to do. If boundary defined by query contains current node's boundary than every row from this node should be added to queue. When none of these conditions are true, algorithm simply adds two sons of current node to queue. There was idea to add possibility for multiple index creation. Such index could be created for some subset of table's properties. Queries with conditions containing only properties from such subset would run much faster. For this feature I introduced two functions `KDVertex:filter_non_index_conditions QueryBoundary::are_there_extra_requirements`. Possibility of creating additional indexes is not implemented yet.

4.5 Locking tables

1. any number of read-only queries can be run concurrently as long as no write query is being performed at the moment,

2. write query can be performed if no other query is being performed at the moment.

This locking schema has the advantage that it can be implemented using only `mutexes` and `locks` from `boost`. A `mutex` object facilitates protection against data races and allows thread-safe synchronization of data between threads. A thread obtains ownership of a mutex object by calling one of the lock functions and relinquishes ownership by calling the corresponding unlock function. Mutexes may be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads. `Boost.Thread` supplies recursive and non-recursive mutexes with exclusive ownership semantics, along with a shared ownership (multiple-reader or single-writer) mutex. Jarek: 3 ostatnie zdania są przeklejone z dokumentacji boosta, nie wiem czy ta
There are three most common kinds of locks. All locks are working in similar fashion. Mutex is passed as constructor parameter to lock. Ownership of mutex is acquired in constructor. Ownership of mutex is released when lock object is destroyed.

1. `unique_lock` - When mutex is acquired by `unique_lock`, then no one can acquire this lock until unique lock will release ownership of mutex.
2. `shared_lock` - Multiple shared locks can acquire the same mutex.
3. `upgrade_lock` - Acquires upgradable ownership. Upgradable ownership may be at any time upgraded from shared lock to unique lock and vice versa.

My solution requires every table to have is own mutex. Read-only request use `shared_lock` and write request use `unique_lock`.

4.6 How data is stored

Requirements:

1. all data can be easily serializable,
2. it should somehow force type safety,
3. very good space efficiency,
4. it must be fast!

Values can be strings or numbers, therefore I created 3 classes:

1. abstract class Type,
2. class TypeString which extends Type,
3. class TypeNumber which extends Type.

Both TypeString and TypeNumber are implementing all supported functions. TypeNumber implements comparisons and inequalities. TypeString implements comparisons, inequalities, substring and prefix function. Pattern matching is done by function strstr from cstring [5].

Single row stores:

1. pointer to the table definition,
2. for each column I store pointer to Type which contains value of this column.

4.7 Database persistence

Guaranteeing persistence of database which holds data in RAM is difficult (as you can see by example of Redis [6]). After every modifying query, engine should not only write such data to disk but ensure that OS will actually write such data to disk. It is a very costly operation which decrease speed of the application drastically. Instead of such solutions, preferred way is to have "almost durable" system. Nowadays there are two most popular ways of having almost durable database in RAM:

1. AOF - append only file, every modifying query is saved to disk (without forcing OS to actually do it),
2. Snapshotting - in constant time intervals, snapshots of database are saved to disk.

I implemented both possibilities. I tried to find a way to combine AOF method with snapshotting so in case of an crash, database could be automatically restored as best as possible. I think the best way is to have a basic bash script that loads a snapshot and executes queries in AOF that were executed after saving last snapshot.

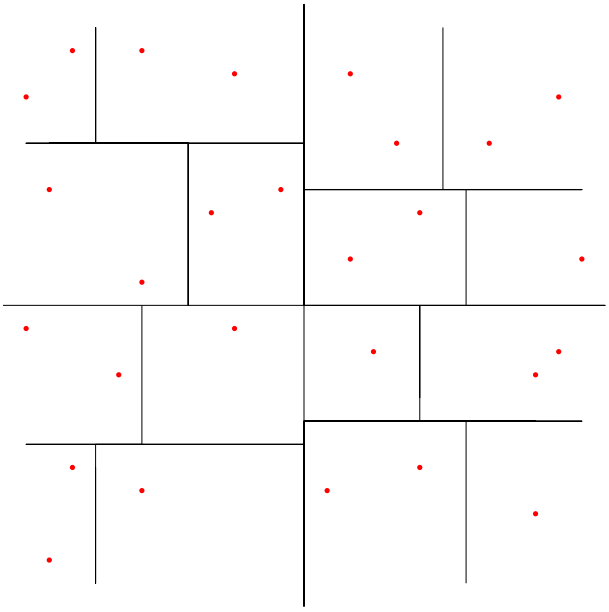


Figure 5: Storing whole area covered by nodes

4.8 Presenting results

Modern databases and search engines have to support most popular formats. Nowadays these are JSON, XML and (maybe) YAML. Chosen library should provide the same interface for feeding data without any assumptions of the selected output format. Chosen library should be providing interface for implementing new formats (so if a new format becomes popular, our software can support it right away). Support for unicode characters is necessary. For some simple queries rendering response could be a bottleneck. That is why I chose `boost::property_tree`. Unfortunately this library does not support unicode by default. It is possible to fix it, by following [7] .

5 Implemented optimizations

5.1 Queue event

Web search-engines usually shows maximum of 10-20 results per page. High percentage of users would not look at more results anyway. Lets input random english word into google search engine. Google shows, that they have found millions of results. But they actually shows us only 10 first results. Such cases are very typical and they must be processed very fast, i.e. $\mathcal{O}((\text{offset} + \text{limit}) * ?)$ instead of $\mathcal{O}(\text{number of results} * ?)$.

In classical K-D tree, search is performed using a simple recurrence. Lets define a single recurrence call during the search as a search event.

Each search event is performed on a single node of K-D tree. For each node we could store boundary of its nodes. Such boundary should store the values of lower and upper bound for each column.

Instead of running search events right away we could insert such events into a priority queue. Consider case when user requested data ordered by value of column X. In such case, priority queue should always return events with minimal lower bound of value X. As soon as sufficiently many results have been found, engine can stop processing search events from the queue. Hence we can hope to substantially restrict the number of actually visited nodes of the K-D tree.

5.2 Defined boundary and real boundary

Imagine that we want to find point with smallest value of the x coordinate. We need to process all nodes that do not have a lower bound on x. Lets count them:

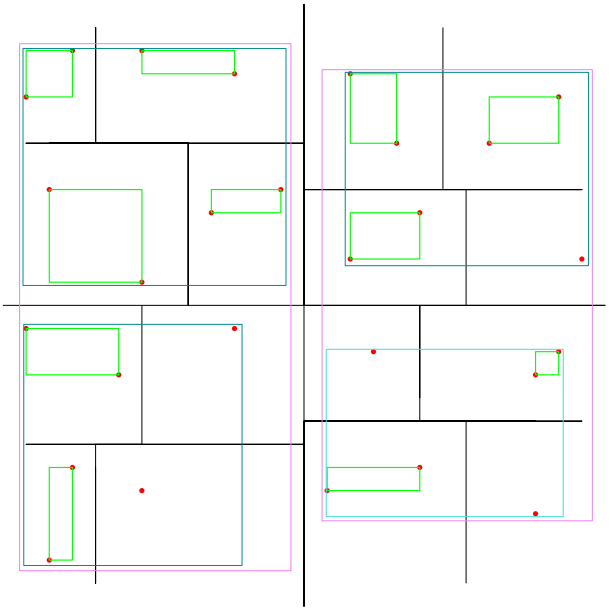


Figure 6: Area covered by points inside nodes

$T(1) = 1$
$T(n) \geq 2^{d-1} * T(n/2^d)$

There are $\Omega(n^{1-1/d})$ such nodes. Instead of storing whole area covered by the node (see Figure 5), we could calculate area covered by points inside each node.

(Figure 6).

Lets revised this idea on example where point with minimal value of coordinate x is searched. First event contains root of a tree. It is processed. It causes to push two of its sons to the queue. Then we chose node with smaller minimum value of coordinate x. This event is processed and another two nodes are added. Point with minimal value of coordinate x belongs to exactly one of these nodes, and this node is processed. This situation goes on until leaf containing point with minimum value of coordinate x is processed. At this point we found point with minimal value of coordinate x. Search process can be stopped and results may be rendered. Summarizing $\mathcal{O}(\log n)$ nodes are added to queue, and $\mathcal{O}(\log n)$ are processed. Finding minimum value of coordinate x of node takes $O(1)$ since we are storing this information in boundary. Processing one node takes $\mathcal{O}(1)$ time. Bootleneck of this solution is priority queue. Priority queue may be implemented as a binary heap. We processed $\mathcal{O}(\log n)$ nodes so complexity of this solution is $\mathcal{O}(\log n \log \log n)$. It can be improved by using binomial heaps [12]. Binomial heap has an insert in amortized time of $\mathcal{O}(1)$ and finding minimum in $\mathcal{O}(1)$ which reduces complexity to $\mathcal{O}(\log n)$.

5.3 Raw pointers instead of shared pointers

In modern C++ pointers should not be used, and shared pointers are recommended instead. Idea behind shared pointer is to automate releasing resources. Basically shared pointer keeps counter representing the number of copies of current pointer. When shared pointer is created, counter is incremented. When shared pointer is deleted, then counter is decremented. When counter is down to zero, then resource is released. With such a tool, developer is free from most common cause of memory leaks. Unfortunately such tool comes with time and space overhead. I decided to use raw pointers which give better running and space time. Right now I think it was biggest mistake in the project. Source code became really messy, even small change in legacy code could create bugs.

5.4 Balancing K-D tree

K-D tree has similar problem to binary trees. Binary tree can be unbalanced and so can K-D tree. My idea for balancing K-D tree is very simple. When number of nodes in one subtree is 5 times larger that

number of nodes in opposite subtree, then I'm rebuilding their parent subtree from the beginning. This is known as partial rebuilding, a simple yet very powerful paradigm often used for making data structures dynamic, for example in weight-balanced trees [13]. While the worst case complexity of a single operations can be large because of this rebalancing, the amortized complexity [14] is actually rather good.

6 Complexities

Nomenklatura:

1. left(θ) - left subtree of vertex θ
2. right(θ) - right subtree of vertex θ

Let n be number of points in the tree. Tree is X-balanced if and only if $\forall\ \theta$: X * |left(θ)| >= |right(θ)| $\wedge$ X * |left(θ)| >= |right(θ)|. Tree is well-balanced if it is 2-balanced. Tree is barely-balanced if it is 5-balanced. Height of both well and barely balanced trees are $\mathcal{O}(\log n)$. Every point belongs to $\mathcal{O}(\log n)$ nodes.

Assumptions:

1. all values of all columns are unique
2. rebuilding subtree with root θ causes subtree with root θ to be well-balanced.

6.1 Inserting/deleting point X

Lemma 1. *Rebuilding tree of size n takes $\mathcal{O}(n \log n)$.*

Proof. We can find the median in linear time [11]. Hence in order to rebuild a tree, we first find the median, and then recursively rebuild two subtrees half the size. Complexity of such procedure can be described by $T(n) = 2T(n/2) + \mathcal{O}(n)$, which solves to $T(n) = \mathcal{O}(n \log n)$. □

Lemma 2. *Amortized time of insert/delete is $\mathcal{O}(\log^2 n)$.*

Proof. We will give $3 \log n$ credits to each node visited during insert/delete. As the depth of the whole tree is $\mathcal{O}(\log n)$, the total number of credits allocated during a single insert/delete operation is just $\mathcal{O}(\log^2 n)$. The goal is to make sure that the following invariant holds: a node such that the left subtree is of size n_l and the right subtree is of size n_r has at least $3|n_l - n_r| \log n$ credits available (observe that because the tree is well-balanced, this is at most $n \log n$). We must prove that the invariant still holds after each insert/delete and that we always have enough credits to amortize the rebuilding. We consider those two parts separately.

1. Each node visited during an insert/delete operation gets $3 \log n$ credits. Notice that after the visit we either increase or decrease n_l or n_r. Hence we increase $3|n_l - n_r| \log n$ by at most $3 \log n$, and so we can afford to pay for this increase using the new credits allocated to the node (it can also happen that we decrease $|n_l - n_r| \log n$, which is even better).
2. We rebuild a node as soon as one subtree is five times larger than the other, say $n_l = 5n_r$. Then the number of credits accumulated at the node is at least $(n_l - 2n_r) \log n = 3n_r \log n \geq n \log n$, where $n = n_l + n_r$. After we reconstruct the tree, the required number of credits will be at most $\log n$. Hence we can use $(n-1) \log n$ credits to pay for the reconstruction, and by Lemma 1 this is enough.

There is one additional details. The value of $\log n$ can (and will) actually change during the execution of the algorithm. We actually care about its integer part, tough. In order for it to increase, we need to perform n insert/delete operations. Hence if we charge each such operation with additional $c \log^2 n$ credits, whenever the integer part of $\log n$ increases, we will have $cn \log^2 n$ credits available.

Now, for each node of the tree we need to add $n \log n$ credits, hence the total required number of credits is described by the recurrence $T(n) = n \log n + T(\alpha n) + T((1-\alpha)n)$, where $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$. The recurrence solves to $T(n) = \Theta(n \log^2 n)$. This can be proved by drawing recurrence call tree. At each level work done sums to $\mathcal{O}(n \log n)$. There are $\mathcal{O}(\log n)$ levels therefore the complexity is $\mathcal{O}(n \log^2 n)$.

We have enough credits to pay for the increase. □

Pawel: Tutaj fajnie byłoby skonstruować przykład, że ta złożoność to faktycznie $\log^2 n$, tzn ciąg n insertów, które w sum

Jarek: Ostatnio po raz drugi poświeciłem więcej czasu temu problemu, bez sukcesu;/

6.2 Search

Recall that the complexity of searching in a static K-D tree is $\mathcal{O}(n^{1-1/d}+k)$. The usual proof of such bound is by analyzing a query of the form $x >= c$. We want to bound the number of nodes whose cells intersect the boundary of such query. Because every d-th level of the tree partitions the points according to the x coordinate, we can write the recurrence $T(n) = 1 + 2^{d-1}T(\frac{n}{2^d})$, which by the master theorem solves to $T(n) = \mathcal{O}(n^{\frac{\log 2^{d-1}}{\log 2^d}}) = \mathcal{O}(n^{1-1/d})$.

Unfortunately, such worst case bound seems difficult to achieve in our dynamic settings, where the tree is just almost balanced. Nevertheless, a slightly weaker bound holds under the assumption that a child of a node corresponding to n points corresponds to at most $\frac{2}{3}n$ points itself.

Jarek: Troche to będę musiał porozbijać, bo za dużo rzeczy na raz tu się dzieje

Lemma 3. *The recurrence describing the worst-case scenario is* $T(n) = 1+2^{d-1}T(\frac{\frac{2}{3}n}{2^{d-1}})$ *which solves to* $\mathcal{O}(n^{1-\log_{3*2^{d-2}}\frac{3}{2}})$.

Proof. K-d tree is a binary tree. Root has height 0, its children have height 1, and so on. Lets count maximal possible number of visited nodes on level d by a recursive search algorithm where region is defined by $X_e \geq C$. Node will be recursively processed iff it intersects search region but is not fully contained in it. At one of these levels there will be a pivot on dimension e. Such line intersects either the region represented by left son or right son, not both. Therefore only one of its sons will be recursively called. Then the upper bound for processed nodes is 2^{d-1}. It can be easily generalized to every region. Adding another condition may only reduce number of processed nodes. Hence upper bound for number of processed nodes on level d is 2^{d-1}.

The total number of points on all nodes on level d is n. Moreover, because the tree is well-balanced, the total number of points on the processed nodes on level d is at most $\frac{2}{3}n$ Pawel: why?. Hence the average number of points inside a processed node on level d is $\frac{\frac{2}{3}n}{2^{d-1}}$. I will prove that the worst case scenario occurs when all subproblems are the same size. Then the complexity will be $T(n) = 2^{d-1}T(\frac{\frac{2}{3}n}{2^{d-1}})$.

Our goal will be to prove that $T(n) \leq cn^a$ for some constant $a < 1$ using induction on n. For small values of n it is true because we can choose constant c. Now lets prove that $T(n) \leq cn^a$ assuming that $T(n') \leq cn'^a$ for all $n' < n$. We have:

$$T(n) = \sum_{i=1}^{2^{d-1}} T(x_i) \leq \sum_{i=1}^{2^{d-1}} c(x_i)^a \quad \text{where} \quad \sum_{i=1}^{2^{d-1}} x_i = \frac{2}{3}n$$

As we are interested in upperbounding $T(n)$, we are interested in the maximum possible value of the sum of all $(x_i)^a$ under the condition that all x_i sum up to $\frac{2}{3}n$. Because the function $f(x) = x^a$ is convex for $a < 1$, from Jensen's inequality we get that the sum is maximized when all x_i are equal. Hence:

$$\begin{aligned} T(n) &= \sum_{i=1}^{2^{d-1}} T(x_i) = 2^{d-1}T\left(\frac{\frac{2}{3}n}{2^{d-1}}\right) \leq 2^{d-1}c\left(\frac{\frac{2}{3}n}{2^{d-1}}\right)^a \\ &= c2^{d-1}\left(\frac{2}{3}\right)^a 2^{-a(d-1)}n^a = 2^{(1-a)(d-1)}\left(\frac{2}{3}\right)^a cn^a \end{aligned}$$

Hence $T(n) \leq cn^a$ as long as $2^{(1-a)(d-1)}(\frac{2}{3})^a \leq 1$.

$2^{(1-a)(d-1)}(\frac{2}{3})^a \leq 1$
$2^{(1-a)(d-1)}2^a \leq 3^a$
$2^{d-1-ad+2a} \leq 2^{a\log_2 3}$
$d-1-ad+2a \leq a\log_2 3$

$d-1 \leq a(\log_2 3 + d - 2)$
$a \geq \frac{d-1}{\log_2 3+d-2}$
$\frac{d-1}{\log_2 3+d-2} < 1$ so for all d exists a < 1.

□

$T(n) = 1 + 2^{d-1}T(\frac{\frac{2}{3}n}{2^{d-1}})$ solves to (by the master theorem) $T(n) = \mathcal{O}(n^{\frac{\log 2^{d-1}}{\log 3*2^{d-2}}})$.
$\frac{\log 2^{d-1}}{\log 3*2^{d-2}} = \log_{3*2^{d-2}} 2^{d-1} = \log_{3*2^{d-2}} 2^{d-1} + \log_{3*2^{d-2}} \frac{3}{2} - \log_{3*2^{d-2}} \frac{3}{2} = \log_{3*2^{d-2}} 3 * 2^{d-2} - \log_{3*2^{d-2}} \frac{3}{2} = 1 - \log_{3*2^{d-2}} \frac{3}{2}$. Therefore $T(n) = \mathcal{O}(n^{1-\log_{3*2^{d-2}} \frac{3}{2}})$.
Here is table with a few values of d and complexity associated with this d.

d	Complexity
2	$\mathcal{O}(n^{0.631})$
3	$\mathcal{O}(n^{0.774})$
4	$\mathcal{O}(n^{0.837})$
6	$\mathcal{O}(n^{0.896})$
10	$\mathcal{O}(n^{0.939})$

Jarek: Nie wiem czy ta tabelka ma sens. A może warto to rozszerzyć o jakiś wykresik?

While the worst case bound might seem somehow disappointing when d is large, one should remember that in practice we will observe a much better performance. For average queries algorithm behaves like $\mathcal{O}(S \log n)$ where S = offset + limit.

Let me explain. Lets take for example data with random distribution. When algorithm recursively enters node of K-D tree then with some probability there are some rows fitting query. If we define this probability as a constant then finding single row takes $\mathcal{O}(\log n)$.

7 Tested engines

7.1 Solr

Schema used in testing is:

```
<fields>
  <field name="id" type="uuid" indexed="true" stored="true" default="NEW"/>
   <field name="first_name" type="text_general" indexed="true"
            stored="true" required="true" />
   <field name="last_name" type="text_general" indexed="true"
            stored="true" required="true"/>
   <field name="age" type="int" indexed="true" stored="true" />
   <field name="city" type="text_general" indexed="true" stored="true"/>
</fields>

<uniqueKey>id</uniqueKey>
```

Solr requires unique key. Combining all properties is not enough. For example there are lots of guys named Jan Kowalski in Warsaw. Therefore I created another property named id.

1. Comparisons: `property:value`
2. Inequalities are realized by range queries: `property:[from TO to]`. `From` and `to` can be replaced by *. Number greater or equal to 2 can be searched by `property:[2 TO *]`. `property:[* TO *]` matches everything.
3. Prefix queries: `property:value*`.
4. Substring queries: `property:*value*`.

7.2 PostgreSQL

Schema used in testing is:

```
CREATE TABLE users (
  first_name TEXT not null,
  last_name TEXT not null,
  age INT not null,
  city TEXT not null
);

CREATE INDEX index1 ON users (first_name, last_name, age, city);
CREATE INDEX index2 ON users (last_name, age, city);
CREATE INDEX index3 ON users (city, age, first_name);
CREATE INDEX index4 ON users (age, first_name, last_name);
```

Syntax:

1. Comparisons and inequalities are realized by `=`, `>=`, `>`, `<=`, `<`
2. Prefix queries are handled by `property LIKE "text\%"`
3. Substring queries are handled by `property LIKE "\%text\%"`

For text-search PostgreSQL implemented special type of indexes: GIN [8] and GIST [9]. Unfortunately I was unsuccessful in making them work with polish words. Therefore I'm testing PostgreSQL with default indexes.

7.3 Shuffla

Table is created by sending HTTP request:

`/create_table/test_table/?first_name=string&last_name=string&age=int&city=string`

Shuffla's query format is:

`/search/<table name>/?subquery(&subquery)*`

Where subquery can be:

1. `field`=`value` for equality check
2. `GREATER_THAN`(`field`)=`value` means that given value is greater than value of `field`
3. `SMALLER_THAN`, `SMALLER_OR_EQUAL`, `GREATER_OR_EQUAL` work in similar fashion to `GREATER_THAN`
4. `PREFIX`(`field`)=`value` requires `field` to have prefix `value`
5. `CONTAINS`(`field`)=`value` requires `field` to contain `value` as a substring

8 Tested results

8.1 How results are tested

Testing environment is machine with Intel i5-2500 processor (4 cores, 3.5GHz), 8GB RAM and SSD drive. Comparing engines with default settings is pointless. I've tried to configure them all to production use. Most important change is increased memory limit for Solr and PostgreSQL to 1GB.

In nk we've got several millions registered users. This number does not grow in spectacular fashion. I want to simulate engines behaviour in similar conditions.

At the beginning I'm inserting $8 * 10^6$ random users. Then I'm running a script which creates random insert/search queries. It sends created queries for few hours and calculates statistics.

Inserted data is generated in a very simple way. I've gather lists of most popular first names and last names in Poland (with counts). Probability of selecting name is proportional to it is count. Age is selected with linear distribution from set $[5, 100]$.

Search queries are generated by this procedure:

1. Pick N - number of expressions in search query
2. Column in expression is selected from set [first name, last name, age, city]. Each of them has 25 percent chance to be chosen.
3. If selected column has int type (only age is an integer) then possible queries are comparison and inequalities. There are 1 comparison and 4 inequalities. Each of them has 20 percent chance to be chosen.
4. If selected column has string type then possible queries are comparison, inequalities, prefix queries and substring queries. Prefix and substring queries are having 25 percent chance to be chosen. Each of the comparison and inequalities has 10 percent chance to be chosen. Selected column is always compared to another string. This string is selected by selecting random word with 1 to 5 characters. First character is uppercase others are lowercase characters.

Then I'm selecting random column and I'm setting it as order by column. Then I'm choosing offset and limit from range $[0, 50]$.

8.2 Inserting 8 million rows

8 million rows. Average string contains 7 characters. It means that size of inserted data is about 250 MB. There is always some overhead in keeping data structures. It causes shuffla to use 6 GB RAM.

Inserting 8 million rows is not most important part of testing. It is performed before production deployment and we are doing this part only once. I'm happy as long as indexing time does not last longer than a few hours.

Shuffla inserts data with 1000 request per second. Using single inserts with other engines is not so efficient. All considered engines are able to read csv file and import data from such file. This is the only way to import data within few hours.

8.3 Statistics

For purpose of testing in true multi-threaded environment I used tsung. Tsung is a distributed load testing tool written in erlang. It is protocol-independent and can be used to stress HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP, and Jabber/XMPP servers.

I'm using 4 core processor for testing so I decided to test all engines using 4 threads. Each thread is sending 10.000 randomly created queries.

Solr and PostgreSQL are using less than 1 GB of RAM. It is possible because both these engines are keeping data on a hard drive. Shuffla stores everything in RAM and it is consuming 6GB of RAM.

Engine	Req / sec	Memory consumption
Solr	6.89	1GB
Shuffla	3.86	6GB
PostgreSQL	2.55	1GB

Solr is superior over Shuffla in every way. Solr is faster. Solr consumes less memory. It is more configurable, for example we could increase Solr memory limit. Solr is just better choice, especially for big social website purposes.

9 Summary

After this discussion you may think that Shuffla is useless. I do not think it is useless. Shuffla fits great for smaller problems like searching through internet forum or blog. Solr or PostgreSQL are requiring not easy installation and configuration. Installation of shuffla takes 2 minutes, creating table takes 10 seconds and searching is very easy. In short/simple projects having possibility of integrating search engine in less than 5 minutes is more than welcome.

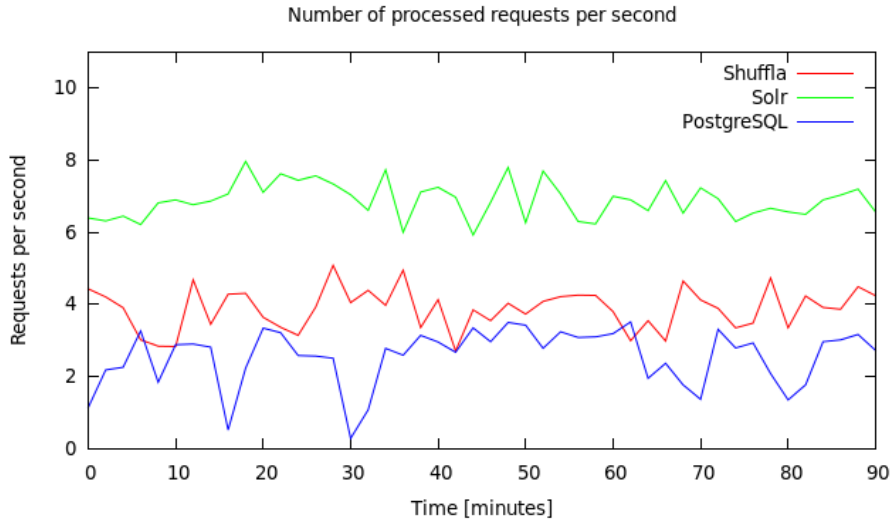


Figure 7: Number of search requests per second

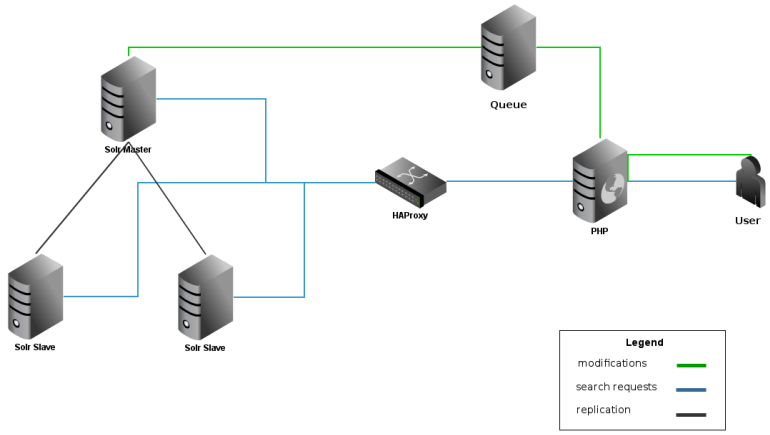


Figure 8: Architecture of searching

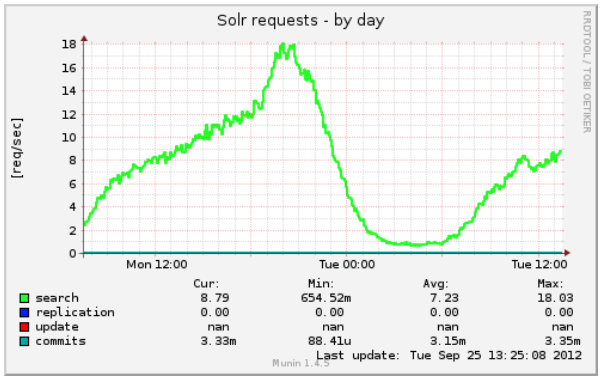
10 Few words about Solr in nk

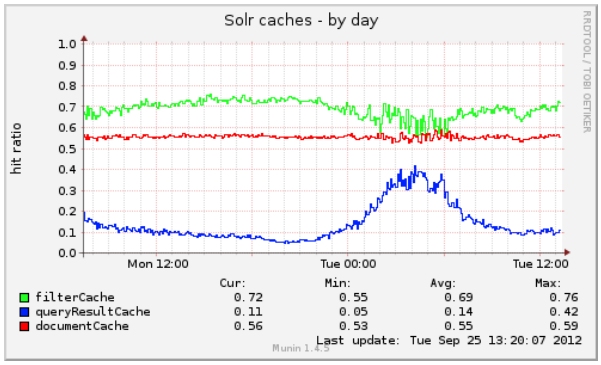
10.1 Search architecture in nk

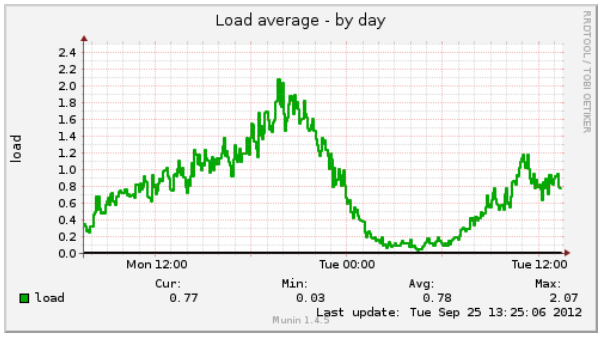
In nk we have 3 servers with Solr. There is master-slave replication between them. One server is a master another two are slaves. Only master processes modifying queries. When user does anything related to searching, PHP code handles it.

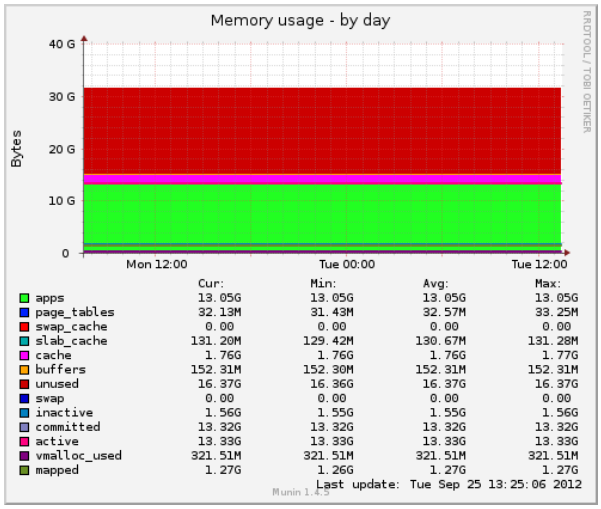
When some action requires modifying data in Solr, PHP code is creating task for it. This task it inserted into a queue. There is a background job which processes all tasks one by one from queue. We need queue to avoid unexpected spikes. For example uncaught spammer attack may cause 1000 modifying requests per second(TODO: to powinno być jako gwiazdka na dole strony *Actually this is highly unlikely, nk crashes spammers activity extremely well). Anyway if this happens master could easily crash or at least experience some kind of slowness due to big load. We want to avoid such situations.

In case of search request, we need to distribute requests between servers. For this purpose search requests are going through HAProxy TODO: reference. This load balancer is distributing requests between our 3 servers.









10.2 Master server statistics from munin

TODO: Jak munin generuje wykresy

TODO: To chyba wykres z slave'a, gdzie inserty?! At night Solr does not do much. Load grows as the day progress. At the peek our master server handles 18 search requests per second (about 8 PM). Do not forget that we have three Solr servers on production and this chart is only from one of them. Apache Solr has advanced cache system. Usually 60-70 percent of requests are rendered based on values from cache.

TODO: Definicja load TODO: Dlaczego używamy tylko 1/3 ramu

TODO: Coś o dyskach na produkcji i ile to jest odczytów na sekunde TODO: Netstat When we are closer to the peek we have more connections to Solr. It is close to 100 at the peek.

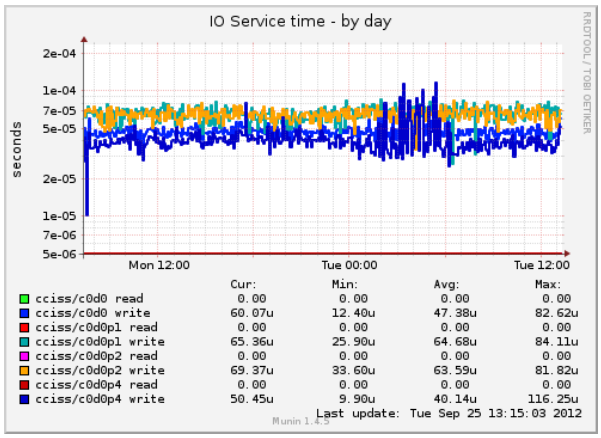
Thread pool contains more than 300 threads. Usually only small amount of this number is used. Java on our production environment is executed with `-Xms24G -Xmx24G` which gives 24GB of RAM for heap and stack (memory used by head + memory used by stack <= 24GB). You may find our heap usage very strange. Most of heap memory is used by caches. We are committing changes every 15 minutes which causes all caches to be flushed by every 15 minutes. For another 15 minutes cache is filled and then flushed again.

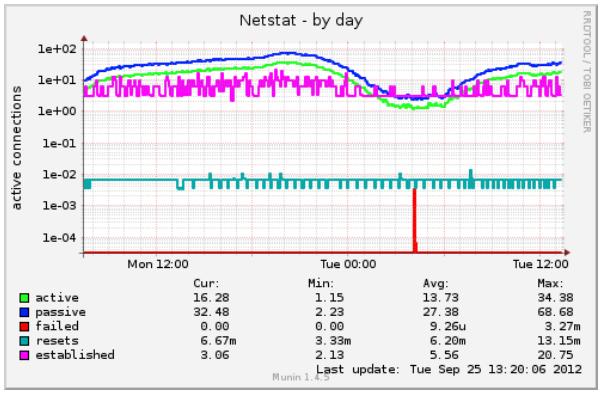
References

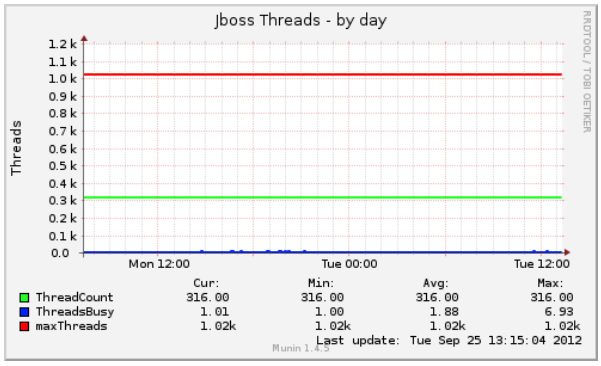
[1] `http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling`.

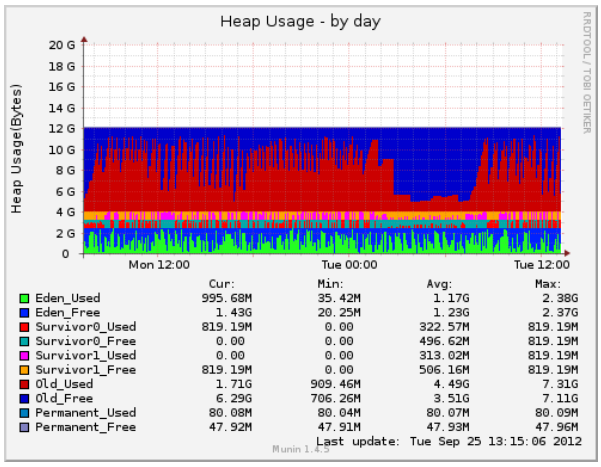
[2] `http://sphinxsearch.com/docs/2.0.1/distributed.html`.

[3] `http://www.boost.org`.


IO Service time - by day
seconds
2e-04
1e-04
7e-05
5e-05
2e-05
1e-05
7e-06
5e-06
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
cciss/c0d0 read 0.00 0.00 0.00 0.00
cciss/c0d0 write 60.07u 12.40u 47.38u 82.62u
cciss/c0d0p1 read 0.00 0.00 0.00 0.00
cciss/c0d0p1 write 65.36u 25.90u 64.68u 84.11u
cciss/c0d0p2 read 0.00 0.00 0.00 0.00
cciss/c0d0p2 write 69.37u 33.60u 63.59u 81.82u
cciss/c0d0p4 read 0.00 0.00 0.00 0.00
cciss/c0d0p4 write 50.45u 9.90u 40.14u 116.25u
Last update: Tue Sep 25 13:15:03 2012
Munin 1.4.5
RRDTOOL / TOBI OETIKER


Netstat - by day
active connections
1e+02
1e+01
1e+00
1e-01
1e-02
1e-03
1e-04
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
active 16.28 1.15 13.73 34.38
passive 32.48 2.23 27.38 68.68
failed 0.00 0.00 9.26u 3.27m
resets 6.67m 3.33m 6.20m 13.15m
established 3.06 2.13 5.56 20.75
Last update: Tue Sep 25 13:20:06 2012
Munin 1.4.5
RRDTOOL / TOBI OETIKER


Jboss Threads - by day
Threads
1.2 k
1.1 k
1.0 k
0.9 k
0.8 k
0.7 k
0.6 k
0.5 k
0.4 k
0.3 k
0.2 k
0.1 k
0.0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
ThreadCount 316.00 316.00 316.00 316.00
ThreadsBusy 1.01 1.00 1.88 6.93
maxThreads 1.02k 1.02k 1.02k 1.02k
Last update: Tue Sep 25 13:15:04 2012
Munin 1.4.5
RRDTOOL / TOBI OETIKER


Heap Usage - by day
Heap Usage(Bytes)
20 G
18 G
16 G
14 G
12 G
10 G
8 G
6 G
4 G
2 G
0
Mon 12:00
Tue 00:00
Tue 12:00
Cur: Min: Avg: Max:
Eden_Used 995.68M 35.42M 1.17G 2.38G
Eden_Free 1.43G 20.25M 1.23G 2.37G
Survivor0_Used 819.19M 0.00 322.57M 819.19M
Survivor0_Free 0.00 0.00 496.62M 819.19M
Survivor1_Used 0.00 0.00 313.02M 819.19M
Survivor1_Free 819.19M 0.00 506.16M 819.19M
Old_Used 1.71G 909.46M 4.49G 7.31G
Old_Free 6.29G 706.26M 3.51G 7.11G
Permanent_Used 80.08M 80.04M 80.07M 80.09M
Permanent_Free 47.92M 47.91M 47.93M 47.96M
Last update: Tue Sep 25 13:15:06 2012
Munin 1.4.5
RRDTOOL / TOBI OETIKER

[4] http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio/reference/io_service.html.

[5] http://www.cplusplus.com/reference/clibrary/cstring/strstr/.

[6] http://redis.io/topics/persistence.

[7] http://stackoverflow.com/questions/10260688/boostproperty-treejson-parser-and-two-byte-wide-char

[8] http://www.sai.msu.su/~megera/wiki/Gin.

[9] http://www.postgresql.org/docs/9.1/static/gist-implementation.html.

[10] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[11] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, Aug. 1973.

[12] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 1990.

[13] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[14] R. E. Tarjan. Amortized Computational Complexity. *Siam Journal on Algebraic and Discrete Methods*, 6, 1985.

[15] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.