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.

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, oraz 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 21, 2012

1 Introduction

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.

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 own specialized 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, and PostgreSQL.

I am 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.

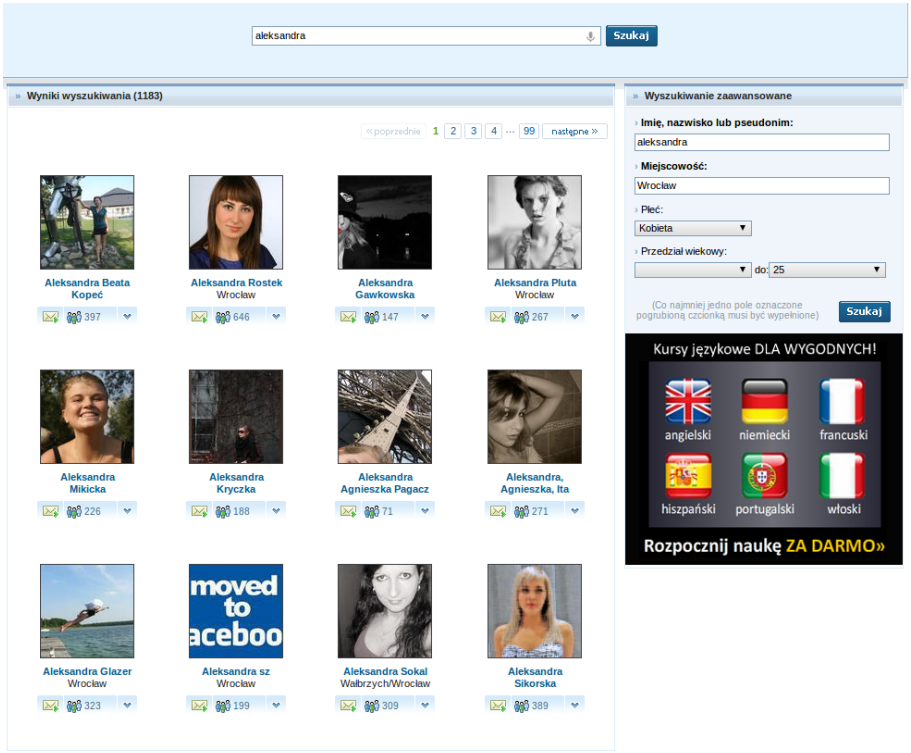


Figure 1: Searching users in nk

2 Problem definition

2.1 Database model

Like many databases, Shuffla provides possibility for storing different types of objects. SQL solves it by giving possibility to create multiple tables and so does Shuffla. Lets define table definition by combination

of table name and list of pairs <column name, type of column>. Fields in out model table are first name, last name, age, home town. They are strings and numbers therefore type of column can be either a string or a number.

2.2 Selecting data

Shuffla like every other database provides possibility for searching data. There are different possible conditions for different types. Possible conditions for numbers are equality check, inequalities and strict inequalities. Strings are compared using lexicographic order so equality check, 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.

User can define order in which results will be sorted. Size of the results may be narrowed by defining offset and limit. These search parameters must work exactly the same as their equivalents in SQL.

Furthermore, it should be possible to personalize the order of of the results. For example, if user from San Francisco is looking for Jane, top results should show Janes from San Francisco, then Janes from California, then rest of U.S. GPS coordinates may be fetched by the modern web browsers [**?**]. 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

3 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 relational databases and full-text search engines.

3.1 Relational databases

Relational Database Management System data is structured in database tables, records and fields. Each table contains multiple rows. Each row contains multiple fields. Relational Databases are storing data in collection of tables, which might be related by common fields. They also provide relational operators to manipulate the data stored in those tables. Most relational databases are using SQL as query language. The most popular relational databases are PostgreSQL, MySQL, MSSQL Server. In this paper I am only testing PostgreSQL which is in my opinion best relational database nowadays.

3.1.1 PostgreSQL

PostgreSQL has really cool features like: point in time recovery, asynchronous replication, online/hot backups, a sophisticated query planner/optimizer, and write ahead logging for fault tolerance. It supports international character sets, unicode, and it is locale-aware for sorting and case-sensitivity. 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: Plagiacik . 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]. PostgreSQL performance can be easily monitored by software like Munin and 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. Here is brief 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 $\mathcal{O}(\log n)$ time where n is number of rows in the table. 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.

3.2 Full-text search engine

In text retrieval, full text search refers to techniques for searching document or a collection in a full text database. It refers to searches based on parts of the original texts represented in databases. The most popular full-text search engines are Sphinx Search and Apache Solr. Almost all big players are choosing Sphinx and Solr over other engines like Elastic Search and Xapian so I am going to consider only the first 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 as much as possible words from inverted index which needs to be processed.
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. Ineqalities, prefix queries and substring queries are treated similarly to equality checks. They are narrowing number of words in dictionary that algorithm needs to process. 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`. By using sophisticated data structure, engine may find out that the are not many people with name containing `qwertyabc`. Such output is rendered very fast[1] In worst case scenario algorithm works in linear time. Example for such 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 and result set will be empty.

3.2.1 Solr

Solr is a 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 which provides extensive statistics on queries, updates and cache usage. 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 few entries, commit takes more than 0.1 sec. Time for commit grows with growing size of data. Instance of Solr in nk contains +20 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. First queries after commit can run slowly. After number of processed queries, engine is past warming stage and queries are handled much faster.

3.2.2 Sphinx Search

Sphinx Search is very good and fast search engine. Sphinx provides excellent relevance ranking (by using multiple factors like phrase proximity and various statistics). It is possible to install Sphinx as MySQL or PostgreSQL plugin. Unfortunately our specification emphasizes Sphinx disadvantages. Sphinx does not have replication. It is unacceptable for any high traffic 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].

[1] with table containing 8 million rows Solr is rendering output for such query in less than 0.01 second

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 (which takes too much time for our requirements)

Real-time index has many caveats [**?**]. Prefix and substring queries are not supported yet. Periodical rebuiling of an index with more than 20 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 I decided to look only into Solr.

3.3 Why am I creating new search engine?!

It seems that none of the 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 very appreciated. Although data stored on disk could be stored in 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 room for improvement. I was also very eager to write my own search engine. I have never implemented such a tool and I was interesting to see how it goes. That is why I decided to create my own search engine.

4 Algorithm description

We can clearly see analogy to multidimensional range searches. All conditions (except substring condition) can be represented as orthogonal range queries. There are three data structures for performing efficient searches in a multidimensional space:

1. Range Tree [11]
2. Inverted Index
3. K-D tree [11]

Complexities of these algorithms are depending on number of elements (n) and number of dimensions (d). In our case number of dimensions is corresponding to number of columns in the table. Here is a table of complexities for listed algorithms.

Algorithm	Insert	Delete	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. K-D tree is worse only in processing search requests. Modifying operations i.e. insert/delete are more important. If these 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.

At the beginning lets start with basic description of K-D tree. Later, in chapter 5, I will describe how this basic version can be improved. K-D tree is a binary tree. Each node contains a pivot, for our purpose we can view this as a condition $row_e \leq C$ where e is pointing to some dimension. In node on level h, pivot is pointing to dimension $h \bmod D$ (where D is number of dimensions). Most important functions of our data structure are insert and search. Both these functions are recursive. Insert call of `row` to `node` is handled by following procedure:

1. If `node` is a leaf then add `row` to the node. Choose a pivot and create `node` sons if necessary
2. Otherwise if `row` is matching condition defined by `node->pivot` then recursively insert `row` to left son of `node`
3. Otherwise recursively insert `row` to right son of `node`

At this point we do not know nothing about how pivots are chosen and how/when rebuilding of a tree is processed. I will get back to this topics later.

In classical K-D tree we are searching points in some orthogonal search region. Parameters for search functions are `node` and `searchRegion`. If `node` is a leaf then we are processing all rows inside `node`. Otherwise we recursively searching through `node` sons with two exceptions. If intersection of region represented by `node` and `searchRegion` is empty then we can stop processing this subtree. If region represented by `node` is fully contained in `searchRegion` then we do not have to process all it's children, we already know that all rows will be inside `searchRegion`.

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 $x \geq c$. 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 $T(n) = \mathcal{O}(n^{\frac{\log 2^{d-1}}{\log 2^d}}) = \mathcal{O}(n^{1-1/d})$.

As I mentioned before almost all conditions can be represented as orthogonal range queries. But we also have to handle 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 [16]. Unfortunately adding linear structure to every node of K-D tree increases the space complexity to $\mathcal{O}(n \log n)$. There are other problems with suffix trees. If we have to return data ordered by some column then all we can do is fetch all rows matching substring condition and sort them. Not mentioning that we can process this way only one substring condition in a query.

There is another possibility. Engine may process search query without substring parts. In the end, if row is matching query without it substring parts then pattern matching is performed. In this solution rows should be provided to pattern matching in order defined within query.

Solution with suffix tree is faster when there are not many rows with given substring. If situation is opposite then pattern matching will work faster. I decided to implemented solution with pattern matching because of its simplicity and superiority in space complexity. As you may see in tests results at the end of this paper, even without suffix trees memory usage is high.

5 Implementation

5.1 HTTP Service

I decided that it would be best if search engine could work as simple HTTP server. Every query should be send as HTTP GET request. As a base of HTTP server I used `example2` from `boost::asio` documentation [3]. In Shuffla configuration file user should define number of available threads. One of the server start procedure is to create all this threads. Every HTTP request is assign to one of these threads by `boost::asio::ip::tcp::acceptor`. There is another thread for periodical functions. Currently its invoking only saving snapshots. All dependencies of server start procedure are presented on Figure 2.

5.2 Validation and passing queries to SearchEngine class

`http::server2::request_handler::handle_request` passes requesting urls to `QueryDispatcher`. `QueryDispatcher` converts every request to corresponding class by using `QueryCreate`. For instance, search request is con-

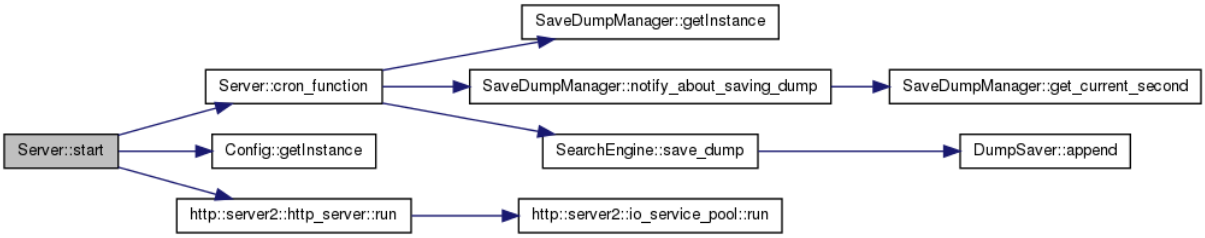


Figure 2: Processing request begins

verted 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 (engine should report queries executing for too much time),
3. locking tables in case of modifying query,
4. passing valid requests to K-D Trees.

5.3 Processing insert/delete requests

Inserting/deleting without tree balancing is pretty straight-forward. Just simple recursion described in previous chapter. Balancing is tricky as you may see in dependencies visualisation on Figure 3.

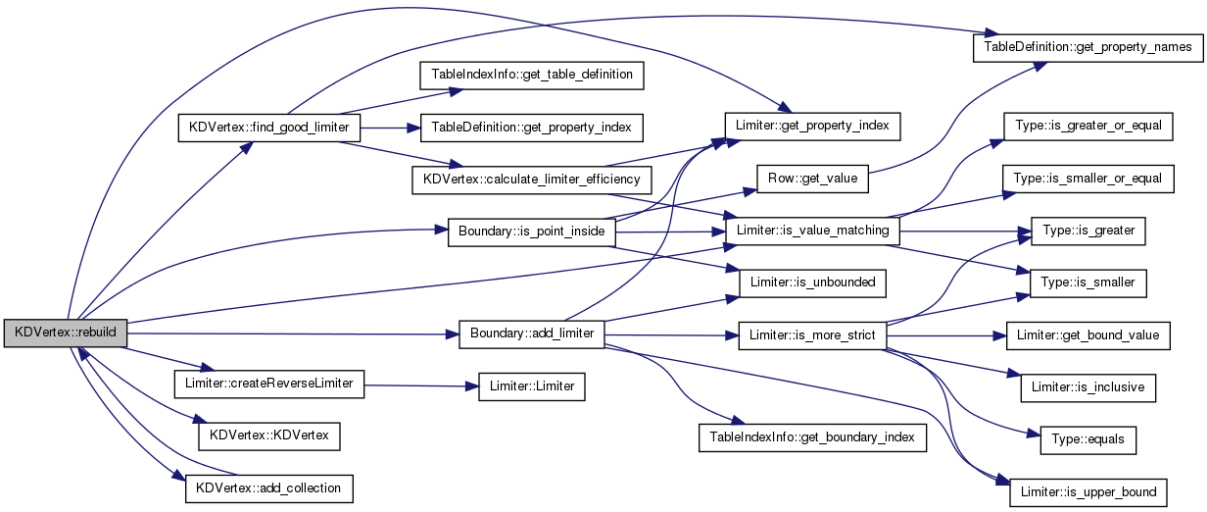


Figure 3: Rebuilding process

Balancing can be done using any linear algorithm for finding median in set. I have decided to take different approach. In `KDVertex::find_good_pivot` I am iterating over all points. Based on coordinate value in currently processed row I am 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 am stopping calculations. I have got a pivot and I am recursively rebuilding two subtrees. I have 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 $\frac{1}{5}$ of rows are creating good pivots. Therefore expected complexity of rebuilding subtree with n nodes is $\mathcal{O}(n \log n)$.

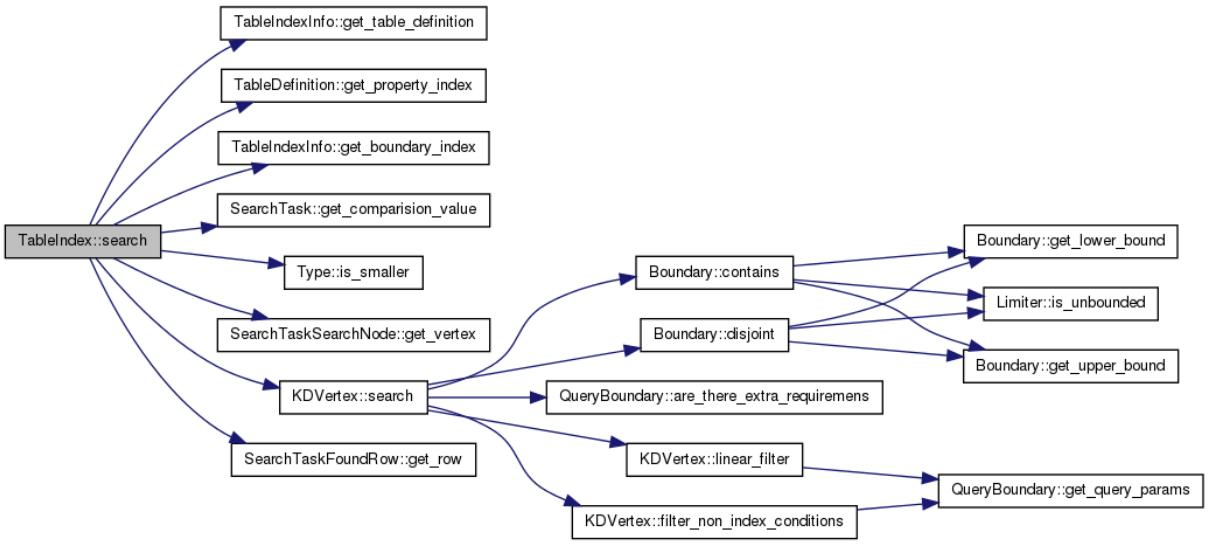


Figure 4: Processing search request

5.4 Processing search requests

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 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 satisfies all conditions defined in search request. Both these classes are extending abstract class `SearchTask`. When processing `SearchTaskFoundRow`, algorithm just adds row to result set. When processing `SearchTaskSearchNode` algorithm passes control to `KDVertex::search` which returns all events that should be added to queue. `KDVertex::search` checks if boundary defined by query is disjoint with current node's boundary. If it's disjoint there is nothing more to do. If not then 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.

Dependencies of search process are visualized on Figure 4.

5.5 Locking tables

To correct work in multi-threaded environment we must ensure thread safety. It requires us to eliminate all data races, especially simultaneous writes which can make our data inconsistent. That is why I introduced locking. Basically by acquiring lock we are restricting access to some part of code.

Our goal is to create following schema: Any number of read-only queries can be run concurrently as long as no write query is being performed at the moment, write queries 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 to kopia z boost doca 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 its own mutex. Read-only request use `shared_lock` and write request use `unique_lock`. With such solution write queries cannot be executed along with other queries. Using `shared_lock` allows simultaneous execution of other read-only queries. So everything works exactly as it suppose to.

5.6 How data is stored

Architecture of data storage in databases is very important. It influences application performance and architecture of all other components. I defined requirement for data storage as:

1. all data must be serializable,
2. chosen architecture should 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`.

If we take a closer look at specification we may notice that we are never changing values of existing numbers and strings. Therefore these classes may be immutable. Immutable object is an object whose state cannot be modified after it is created. Immutability trivially guarantees thread-safety and allows for some storage optimization. Lets take a look at `TypeString`. Storing text values in `std::string` would be inefficient because of overhead caused by `std::string` being not immutable. Text is actually stored in `char*`. This array is created in `TypeString` constructor because at this point we know text length.

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

Single row stores pointer to the table definition. Besides that there is a `std::vector<Type>` which stores values of all fields.

5.7 Database persistence

In case of a crash there must be a way to restore database. In this chapter I will describe how Shuffla preserves data. 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 drastically decreases application speed. Instead of such solution, 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.

5.8 Presenting results

Presenting results may sound like a trivial issue. Actually it is not and not many libraries would fit for requirements presented below. 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 to render, without any assumptions of the selected output format. Support for unicode characters is necessary. For some simple queries rendering response could be a bottleneck. That is why I chose `boost::property_tree`. This library does not support unicode by default. It is possible to fix it by following [7] .

6 Implemented optimizations

What makes an algorithm useful in practice are optimizations. In this chapter I am presenting description of core optimizations implemented in Shuffla. Most important are first two optimizations which are making Shuffla few times faster.

6.1 Queue event

Web search-engines usually shows maximum of 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. Rather than showing all results they are showing us only 10 most relevant results. Such cases are very typical and they must be processed very fast, i.e. $\mathcal{O}((\text{offset} + \text{limit}) * ?)$ instead of $\mathcal{O}(\text{total number of results} * ?)$.

In classical K-D tree implementations, search is performed by using a simple recurrence. Lets define a single recurrence call 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. This optimization alone may seem like big improvement. It actually is not. Number of nodes added to the queue may be large.

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

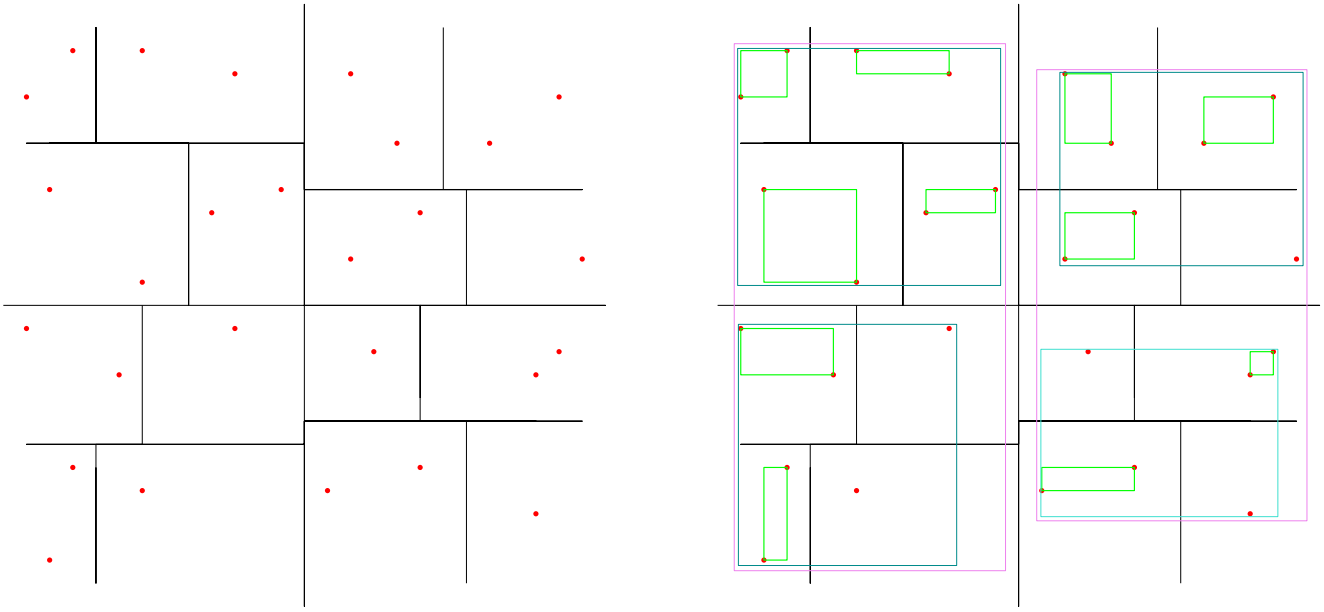
$$T(1) = 1$$
$$T(n) \geq 2^{d-1} * T(n/2^d)$$

There are $\Omega(n^{1-1/d})$ such nodes. This number can be decreased by optimization described in next chapter.

6.2 Defined boundary and real boundary

Instead of storing whole area covered by the node (see Figure 5a), we could calculate area covered by points inside each node (see Figure 5b).

Lets revised this idea on example where k points with minimal values of coordinate x are requested. 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 first point with minimal value of coordinate x. Then all nodes with lower bound equal to second minimal value of coordinate x are processed. There are again $\mathcal{O}(\log n)$ such nodes, all of them are processed and one row is found. The situation goes on until we find k rows. After finding k points, search process can be stopped and results may be rendered. Summarizing $\mathcal{O}(k \log n)$ nodes are added to queue, and $\mathcal{O}(k \log n)$ nodes are processed. Finding minimum value of coordinate x



(a) Storing whole area covered by nodes

(b) Area covered by points inside nodes

of node takes $\mathcal{O}(1)$ 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 $\mathcal{O}(k \log n \log(k \log n))$.

6.3 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 decreased to zero then resource is released. With such a tool developer is free from most common cause of memory leaks. Unfortunately shared pointersare comming with time and space overhead. I decided to use raw pointers which gives better running time and space usage. 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.

6.4 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 2 times larger that number of nodes in opposite subtree then I am 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 [14]. While the worst case complexity of a single operations can be large because of this rebalancing, the amortized complexity is actually rather good. Amortized complexity describes average time required to perform a sequence of related operations. Detail description with examples of amortized complexity can be found in [15].

7 Complexities

Terminology:

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

Let n be number of points in the tree. Tree T is α-balanced if and only if

$$\bigvee_{v \in T} \quad \frac{1}{\alpha} \leq \frac{|\operatorname{left}(v)|}{|\operatorname{right}(v)|} \leq \alpha$$

Tree is well-balanced if it is 2-balanced. Tree is super-balanced if it is $\frac{3}{2}$-balanced. Height of both well and super 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 super-balanced.

7.1 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 [12]. Hence in order to rebuild a tree, we first find the median, and then recursively rebuild two subtrees. Complexity of such procedure can be described by $T(n) = \mathcal{O}(n) + T(\alpha n) + T((1-\alpha)n)$, where $\frac{2}{5} \leq \alpha \leq \frac{3}{5}$. The recurrence solves to $T(n) = \Theta(n \log n)$. This can be proved by drawing recurrence call tree. At each level work done sums to $\mathcal{O}(n)$. There are $\mathcal{O}(\log n)$ levels therefore the complexity is $\mathcal{O}(n \log n)$. □

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

Proof. We will give $6 \log n$ 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 $6|n_l - n_r| \log n$ 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 $6 \log n$ credits. Notice that after the visit we either increase or decrease n_l or n_r. Hence we increase $6|n_l - n_r| \log n$ by at most $6 \log n$, 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 two times larger than the other, say $n_l = 2n_r$. Then the number of credits accumulated at the node is at least $6(2n_r - \frac{3}{2}n_r) \log n = 6\frac{1}{2}n_r \log n = n \log n$, 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. □

7.2 Search

Recall that the complexity of searching in a static K-D tree is $\mathcal{O}(n^{1-1/d} + k)$. Unfortunately, such worst case bound seems difficult to achieve in our dynamic settings, where the tree is just well-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.

Lemma 3. *Upper bound for number of processed nodes on level d is* 2^{d-1}.

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}. □

Lemma 4. *Upper bound for average number of points inside a processed node on level d is* $\frac{\frac{2}{3}n}{2^{d-1}}$.

Proof. The total number of points on all nodes on level d is n. The total number of points in processed nodes on level d is at most $\frac{2}{3}n$. Worst case scenario occurs when number of processed nodes is maximal. From proof of lemma 3 we know that it is the scenario with search region defined as $X_e \geq C$. For all nodes with pivots on dimension e it is true that nodes from one son will be processed and nodes from other son will not be processed. Tree is well-balanced so ratio between processed points and not precessed points is not greater than 2:1. Hence the upper bound for 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 $T(n) = 2^{d-1}T(\frac{\frac{2}{3}n}{2^{d-1}})$.

Lemma 5. *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. 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 $2^{(1-a)(d-1)}(\frac{2}{3})^a \leq 1$.

Simple transformations of $2^{(1-a)(d-1)}(\frac{2}{3})^a \leq 1$ gives $\frac{d-1}{\log_2 3+d-2} < 1$ 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	$\mathcal{O}(n^{0.939})$

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 without substring conditions algorithm behaves like $\mathcal{O}(S \log n)$ where S = offset + limit.

Let me explain. Lets take data with random distribution. When algorithm recursively enters node of K-D tree then with some probability there are some rows fitting query. Algorithm will find some results if there is a point on intersection of search region and region representing the node. Probability of such event for search region S and region R representing node can be defined as $P(S,R) = 1 - (\frac{|S \cap R|}{|R|})^p$ where p is number of points inside R. This number may be small only if $|S \cap R|$ is very small compared to $|R|$. If we bound this probability by a constant then finding single row takes $\mathcal{O}(\log n)$.

8 Tested engines

All engines must be tested on the same schema and by the same queries. Yet presented engines have different syntax. In chapters below I am describing table configurations and how queries were translated during their testing.

8.1 Solr

During tests I was using Solr 3.6.1. Tables in Solr are defined via xml file called schema.xml. Part of this file responsible for creating table is listed below.

```
<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.

Below I am presenting syntax for all types of queries:

1. Equality check: `property:value`
2. Inequalities are realized by range queries: `property:[from TO to]`. `From` and `to` can be replaced by `*`. For example searching number greater then or equal to 2 is performed by `property:[2 TO *]`. `property:[* TO *]` matches everything.
3. Strict inequalities are realized by excluding some range `-property:[from TO to]`. For example searching numbers smaller than 2 is performed by excluding range [2,∞]: `-property:[2 TO *]`.
4. Prefix queries: `property:value*`.
5. Substring queries: `property:*value*`.

8.2 PostgreSQL

I am using PostgreSQL 9.1. 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. Equality check, inequalities and strict 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 am testing PostgreSQL with default indexes.

8.3 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 must be greater than value of `field`
3. `SMALLER_THAN`, `SMALLER_OR_EQUAL`, `GREATER_OR_EQUAL` are working 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

9 Tested results

9.1 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 have tried to configure them all to production use. Most important change is increased memory limit for Solr and PostgreSQL to 1GB.

In nk we have 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 am inserting $8 * 10^6$ randomly generated users. Then I am 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 have gather lists of most popular first names and last names in Poland (with counts). Probability of selecting name is proportional to its count. Age is selected with linear distribution from set $[5, 100]$.

Search queries are generated by following 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 equality check, inequalities and strict 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 remaining conditions 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.

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

9.2 Inserting 8 million rows

Average length of combined fields `first_name`, `last_name`, `city` contains 21 characters. It means that size of inserted data is about 250 MB. There is always some overhead of keeping data in 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 am happy as long as indexing time does not last longer than a few hours.

Shuffla handles insert requests with speed of hunderds requests 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. For them it is the only way to import data within few hours.

9.3 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 am 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 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 uses 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.

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 others are slaves. Only master processes modifying queries. When user does anything related to searching, PHP code handles it.

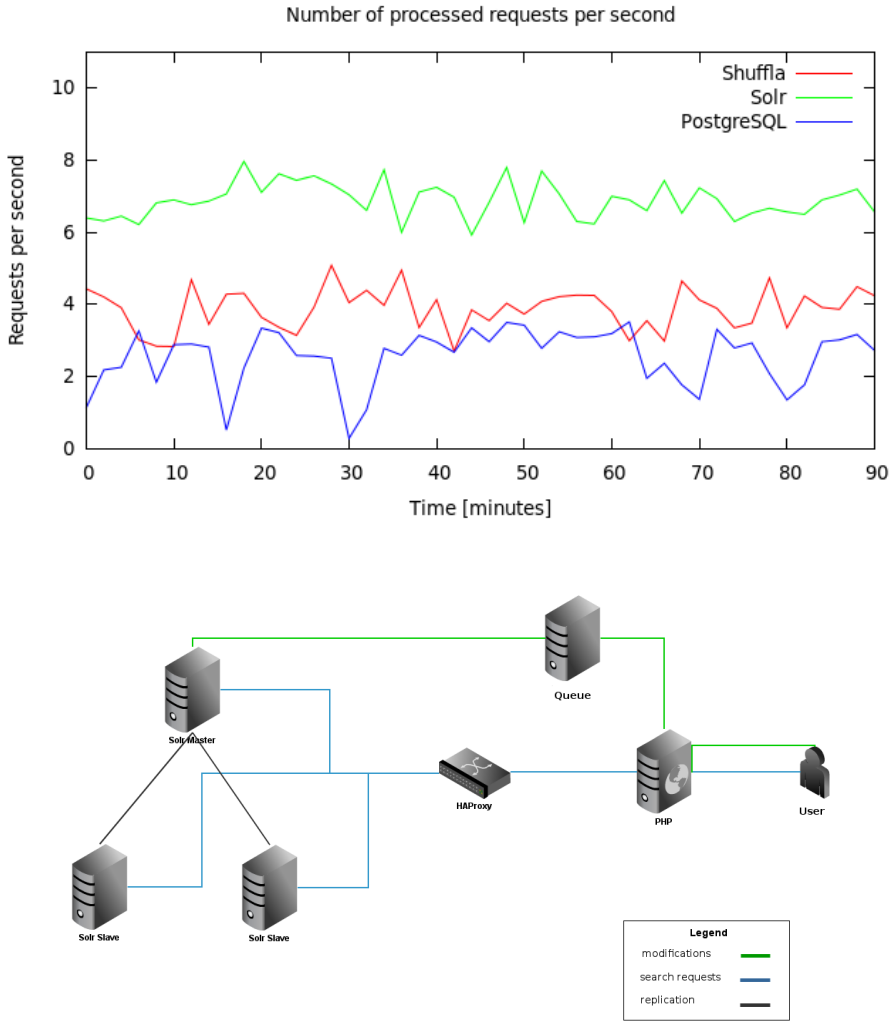


Figure 5: Architecture of searching

When some action requires modifying data in Solr, PHP code is creating task for it. This task is inserted into a queue. There is a background job which processes all tasks from queue one by one. We need queue to avoid unexpected spikes. For example uncaught spammer attack may cause 1000 modifying requests per second [2]. If this happens master could easily crash or at least experience some kind of slowness due to a big load. We want to avoid such situations.

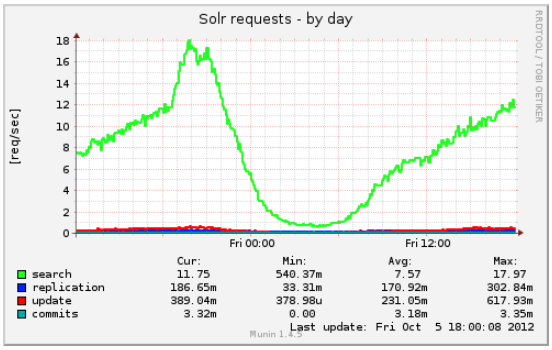
In case of search request we need to distribute requests between servers. For this purpose search requests are going through HAProxy [10]. This load balancer is distributing requests between our 3 servers.

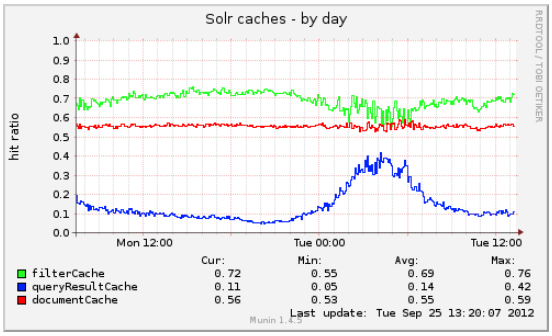
10.2 Statistics from master server

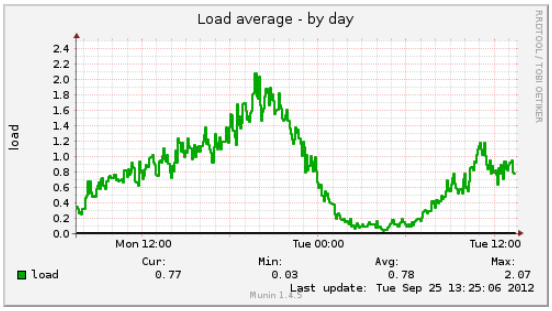
Number of requests, load - by day

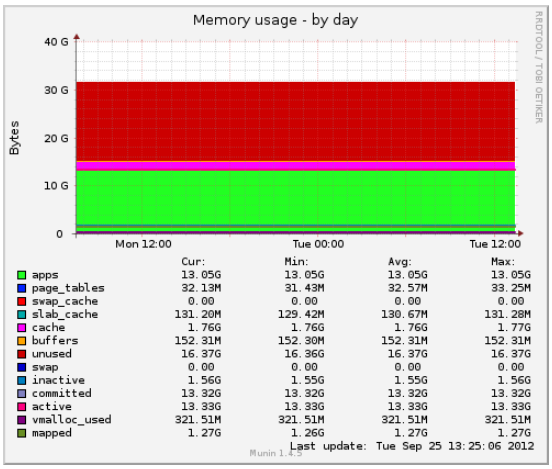
At night Solr does not do much. Load grows as the day progresses. At the peek (about 8 PM) our master server handles 18 search requests per second. Number of other requests like updates/replication/commit is negligible. Do not forget that we have three Solr servers on production and this chart is only from one of them. Load is a measure of the amount of computational work that a computer system performs.

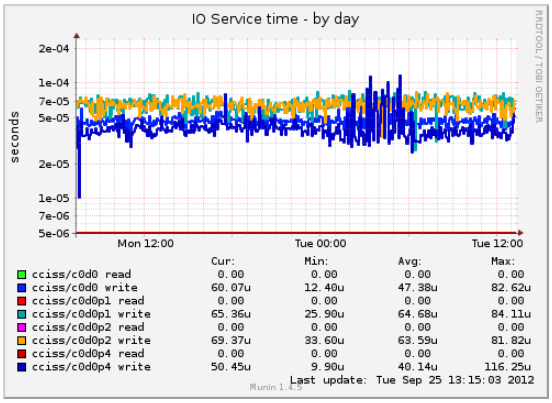
[2] Actually this is highly unlikely, nk crashes spammers activity extremely well

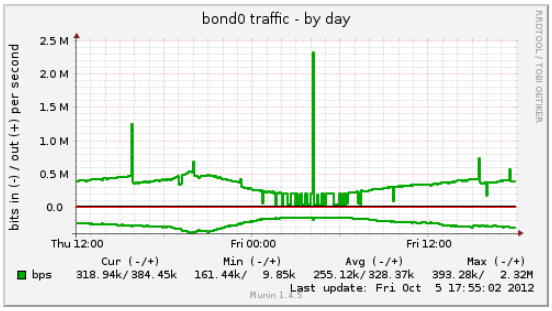

Solr requests - by day
[req/sec]
Fri 00:00
Fri 12:00
search 11.75 540.37m 7.57 17.97
replication 186.65m 33.31m 170.92m 302.84m
update 389.04m 378.98u 231.05m 617.93m
commits 3.32m 0.00 3.18m 3.35m
Last update: Fri Oct 5 18:00:08 2012

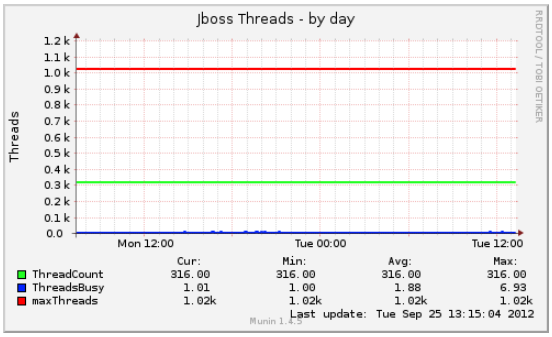

Solr caches - by day
hit ratio
Mon 12:00
Tue 00:00
Tue 12:00
filterCache 0.72 0.55 0.69 0.76
queryResultCache 0.11 0.05 0.14 0.42
documentCache 0.56 0.53 0.55 0.59
Last update: Tue Sep 25 13:20:07 2012

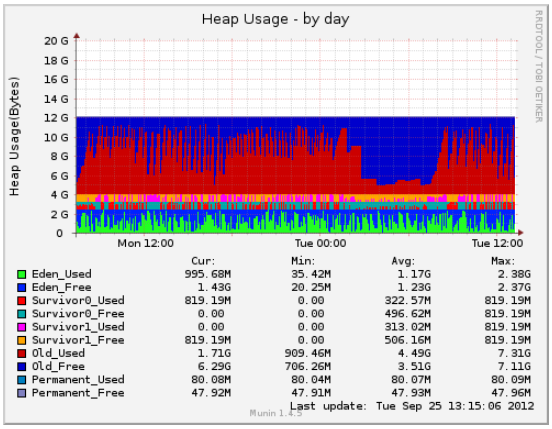

Load average - by day
load
Mon 12:00
Tue 00:00
Tue 12:00
load 0.77 0.03 0.78 2.07
Last update: Tue Sep 25 13:25:06 2012


Memory usage - by day
Bytes
Mon 12:00
Tue 00:00
Tue 12:00
apps 13.05G 13.05G 13.05G 13.05G
page_tables 32.13M 31.43M 32.57M 33.25M
swap_cache 0.00 0.00 0.00 0.00
slab_cache 131.20M 129.42M 130.67M 131.28M
cache 1.76G 1.76G 1.76G 1.77G
buffers 152.31M 152.30M 152.31M 152.31M
unused 16.37G 16.36G 16.37G 16.37G
swap 0.00 0.00 0.00 0.00
inactive 1.56G 1.55G 1.55G 1.56G
committed 13.32G 13.32G 13.32G 13.32G
active 13.33G 13.33G 13.33G 13.33G
vmalloc_used 321.51M 321.51M 321.51M 321.51M
mapped 1.27G 1.26G 1.27G 1.27G
Last update: Tue Sep 25 13:25:06 2012


IO Service time - by day
seconds
Mon 12:00
Tue 00:00
Tue 12:00
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


bond0 traffic - by day
bits in (-) / out (+) per second
Thu 12:00
Fri 00:00
Fri 12:00
bps 318.94k/384.45k 161.44k/ 9.85k 255.12k/328.37k 393.28k/ 2.32M
Last update: Fri Oct 5 17:55:02 2012


Jboss Threads - by day
Threads
Mon 12:00
Tue 00:00
Tue 12:00
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


Heap Usage - by day
Heap Usage(Bytes)
Mon 12:00
Tue 00:00
Tue 12:00
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

Jarek: Zdanie z wiki Load X means that we need at least $\lceil X \rceil$ CPUs to run all processes without any of them waiting for a turn.

Memory, heap usage

We are using half of available memory because the other half is reserved for unexpected events, for example crash of one server. It may require increasing memory limits on two other machines. Java on our production environment is executed with `-Xms24G -Xmx24G` which means that memory used by heap plus memory used by stack cannot exceed 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. For another 15 minutes cache is filling (by processing search queries) and then flushed again.

Disk usage and network traffic statistics

IO Service shows average values of how much time disk in working during a second. It shows very small values. On production all Solr data is kept on ramdisks. It is not zero because of work done by operating system. Solr network traffic is not huge, it rarely reaches 1 megabit. In the evening we reach peek with more than a 100 simultaneous connections from PHP servers to Solr.

JBoss threads

Solr must be executed by some application container. For this purpose we choose JBoss. Thread pool contains more than 300 threads. Usually only small amount of this number is used.

Cache usage

Solr has advanced cache system. Usually about 70 percent of requests are rendered based on values from cache. At night there is small number of requests and Solr is unable to get passed warming stage before next commit. Hit ratio is slightly smaller, about 60 percent at night.

11 Summary

After this discussion you may think that Shuffla is useless. I do not think it is useless. Shuffla fits great for smaller projects like searching through internet forum or blog. Both Solr and 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.

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`.

[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] `http://haproxy.1wt.eu/#desc`.

[11] 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.

[12] 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.

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

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

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

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