Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Jarosław Gomułka

Shuffla: a 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

Mając dany zbiór użytkowników dużego portalu społecznościowego chcielibyśmy dysponować efektywnym i skalowalnym narzędziem, które umożliwi nam jego dokładne przeszukiwanie.

Naturalnym rozwiązaniem tak postawionego problemu jest użycie relacyjnej 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 zadaniu. Oprócz opisania architektury i szczegółów implementacji, porównam jej efektywność z PostgreSQL, Apache Solr, oraz Sphinx Search.

Jestem członkiem zespołu rozwijającego portal nk.pl (który jest największym polskim serwisem społecznościowym) od początku roku 2011. W sierpniu 2011 roku zostało mi powierzone rozwiązanie tego problemu.

Shuffla: a fast and scalable full-text search engine

Jarosław Gomułka

November 26, 2012

1 Introduction and overview of the content

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

One possible solution is to use a relational database. Other solution is to use a full-text search engine like Apache Solr or Sphinx Search. Another solution is to develop our own specialized search engine dedicated to solving such tasks. In this thesis I will describe Shuffla, a full-text search engine created by me and compare it to Sphinx, Solr, and PostgreSQL.

I have been working at nk.pl (the biggest Polish social networking website) since the beginning of 2011. In August 2011 I was appointed to solve the task described above.

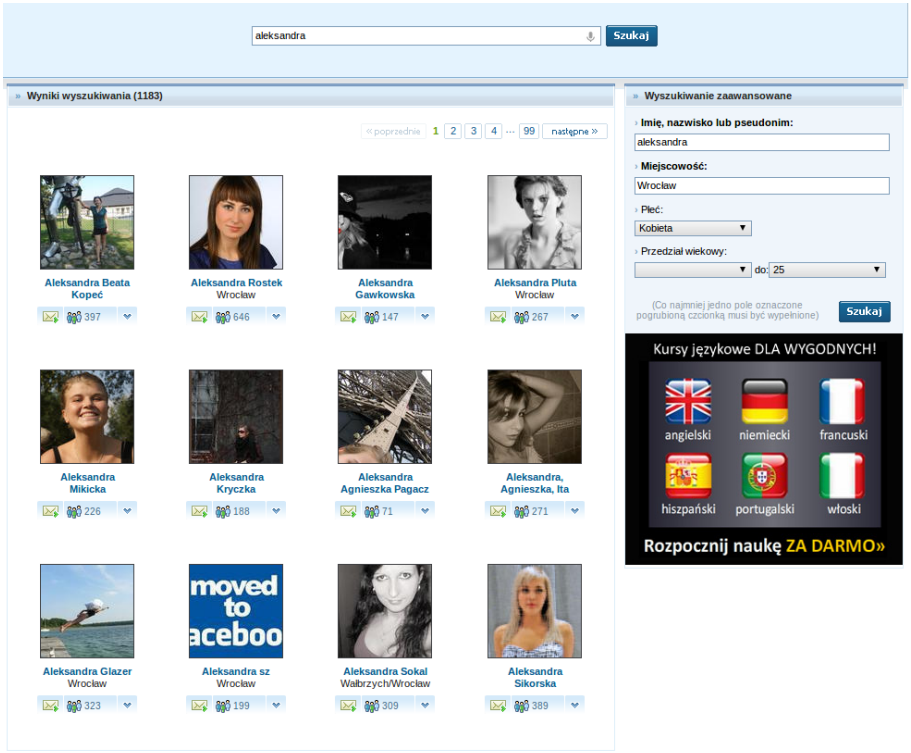


Figure 1: Searching users in nk

In Section 2 I formally define the problem. Then in Section 3 I present how it can be solved using already existing software tools and justify writing a new search engine. The core of my implementation is a data structure for efficient query handling, which I present in Section 4. Then I describe optimizations used to speed-up this data structure in Section 5 and architecture of my implementation in Section 6. In Section 7 I formally analyze the complexity of the resulting solution, and in Section 8 present the experimental results. Finally, in Section 9 I present the architecture and a few statistics of the search engine actually used in nk and summarize in Section 10.

2 Problem definition

2.1 Database model

As with many databases, Shuffla allows you to store different types of objects. Like SQL, Shuffla does so by offering possibility of creating multiple tables. A table is defined by specifying its name and a list of columns, where each column has a name and a type. In our example use case the columns are: first name, last name, age, and home town. They are strings and numbers, therefore a type of column can be either a string or a number.

2.2 Selecting data

Shuffla like every other database offers possibility of 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 possible conditions are (again) equality check, inequalities and strict inequalities. Additionally, the following two conditions should be supported:

1. checking if a given string is a prefix of the selected column value,
2. checking if a given string is a substring of the selected column value.

Users can define `order_by` column. If the `order_by` clause is specified, the returned rows are sorted in the specified order. Size of the results may be narrowed by defining `offset` and `limit`. These search parameters must work exactly in the same way as their equivalents in SQL.

Furthermore, it should be possible to personalize the order of the results. For example, if a user from San Francisco is looking for Jane, the top results should show Janes from San Francisco, then Janes from California, then the rest of U.S. GPS coordinates of the user can be fetched by the modern web browsers [1], and GPS coordinates of each home town can be added to the scheme. Then to select results in some range we use the following 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

The problem described in this thesis may be solved by various existing software tools. There are two most common kinds of such tools 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 store 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 use SQL as query language. The most popular implementations are PostgreSQL, MySQL, MSSQL Server. In this thesis I will only test PostgreSQL which is in my opinion best relational database nowadays.

3.1.1 PostgreSQL

PostgreSQL has really useful features such as: 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 is locale-aware for sorting and case-sensitivity. It is highly scalable both in the quantity of data it can manage and in the number of concurrent users it can accommodate. PostgreSQL keeps its data on a disk and it implements their own master-slave replication. There are also few middlewares which could provide master-master replication [2]. 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. Serious disadvantage of using PostgreSQL for this problem is performance on substring queries, as B-tree based indexes do not support 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 a brief example of processing query by B-tree: `first name = John and substring(last name) = qwertyabc and limit = 1`. In this case engine finds 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 to check if they contain substring `qwertyabc`. Since there is no one with such a name, engine will process all men named John and 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. The searches are 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 word occurring 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 a disk. When processing query, engine selects best strategy to follow. Strategy can be seen as the following procedure:

1. narrow as many words as possible from inverted index which needs to be processed,
2. process remaining words.

For example, if a query is `first name = John and limit = 1`, I am only interested in word `John` from inverted index. The engine finds the first occurrence and returns it. Inequalities, prefix queries and substring queries are treated similarly to equality checks. They narrow the number of words in the dictionary that algorithm needs to process. If a query is `first name = John and substring(last name) = qwertyabc and limit = 1`, it is likely that there are more Johns than all people with last name containing `qwertyabc`. By using sophisticated data structure, engine may find out that the are not many people with a name containing `qwertyabc`. Such output is rendered very fast[1] In worst case scenario algorithm works in linear time. An 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 the result set will be empty.

3.2.1 Solr

Solr is an open source search platform from the Apache Lucene project. It has lots of very handy features like faceted searches, geospatial searches, string tokenizers (makes John and Johnny to be interpreted as the same name). Solr has a 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 to database immediately after insert/update commands. The data is actually inserted after calling commit command. Commit is very costly operation because it requires all caches to be invalidated. Even for the table with a few entries, commit takes more than 0.1 sec. Time for commit grows with a growing size of data. An instance of Solr at 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 a warming stage. It means that all caches are flushed. First queries after commit can run slowly. After a number of processed queries, engine is past warming stage and queries are handled much faster.

3.2.2 Sphinx Search

Sphinx Search is a 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

[1] With table containing 8 million rows Solr renders output for such query in less than 0.01 second.

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 by storing the inverted index on disk and replicating the directory containing the index (for example using a network file system). But if the index is big then syncing such directories would take too much time and would be inefficient and problematic.

There are several high-traffic websites using Sphinx Search. They horizontally partition searched data across search nodes and then process it in parallel [3].

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 potential search engines.

Apart from replication Sphinx has another problem. There are two different kinds of indexes possible:

1. real-time index, where inserts are possible,
2. normal index, where modifications possible only by rebuilding the whole index (which takes too much time for our requirements).

Real-time index has many caveats [4]. Prefix and substring queries are not supported yet. Periodical rebuilding of an index with more than 20 million elements is problematic. Apache Solr and Sphinx search use 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 process search queries very fast. In worst case scenario it is always $\mathcal{O}(n)$. All solutions work on data stored on a disk. Nowadays when RAM is cheap and servers with +32GB RAM are considered average, solutions working in RAM memory are very appreciated. Although data stored on a disk could be stored in RAM by ramdisks, it is not a primary target for 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 there is room for improvement in this area. I was also very eager to write my own search engine. I have never implemented such a tool and I was interested to see what are the challenges involved. 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 standard data structures for performing efficient searches in a multidimensional space:

1. range tree [12],
2. inverted index [14],
3. K-D tree [12].

Complexities of these algorithms depend on the number of elements n and the number of dimensions d. In our case, number of dimensions corresponds to number of columns in the table. Here is a table of complexities for those three data structures.

Data structure	Insert time	Delete time	Search time	Space usage
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 use 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.

I will start with a basic description of K-D tree. In Section 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 it as a condition $x_i \leq C$ where x_i denotes the i-th coordinate. Insert and delete are similar to their equivalents in classical binary tree. Each function is recursive. Based on condition in given node algorithm processes either left or right subtree. To achieve best complexity, in node on level h, pivot splits according to the h mod d-th coordinate, where d is number of dimensions.

Insert call of `row` to `node` is handled by the following procedure:

1. if `node` is a leaf then add `row` to the node. Choose a pivot and create sons of `node` if necessary,

2. Otherwise, if `row` matches condition defined by pivot, recursively insert `row` to left son of `node`,

3. Otherwise, recursively insert `row` to right son of `node`.

At this point let us assume that splitting leaf into two nodes occurs when leaf contains at least 30 rows. Pivot should be chosen as the median of the corresponding coordinates of all points in the subtree.

In classical K-D tree algorithm searches points in some orthogonal search region. The parameters for search functions are `node` and `searchRegion`. If `node` is a leaf, algorithm processes all rows inside `node`. Otherwise, algorithm recursively searches through `node` sons with two exceptions. If intersection of region represented by `node` and `searchRegion` is empty, I can stop processing this subtree. If the region represented by `node` is fully contained in `searchRegion`, I can assume that all rows will be inside `searchRegion` (but still these rows must be collected).

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 $x \leq C$. I 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, I can write the recurrence $T(n) = 1 + 2^{d-1}T(\frac{n}{2^d})$, which by the master theorem solves to $T(n) = \mathcal{O}(n^{\frac{\log 2^{d-1}}{\log 2^d}}) = \mathcal{O}(n^{1-1/d})$. I will repeatedly use the master theorem in our complexity analysis, hence I state it below.

Theorem 1 (Master theorem)**.** *Assume that $a \geq 1, b > 1$ and for all $n \geq 1$:*

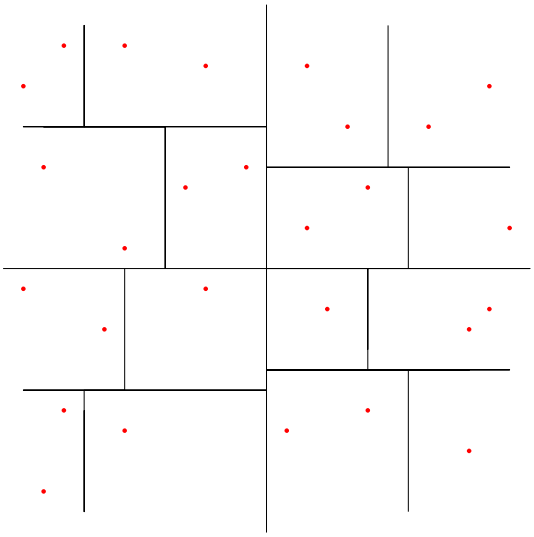
$$T(n) = \begin{cases} \Theta(1) & \text{if } 1 \leq n < b \\ aT(\frac{n}{b}) + f(n) & \text{if } b \leq n \end{cases}$$

Then, if $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant ϵ, $T(n) = \Theta(n^{\log_b a})$.

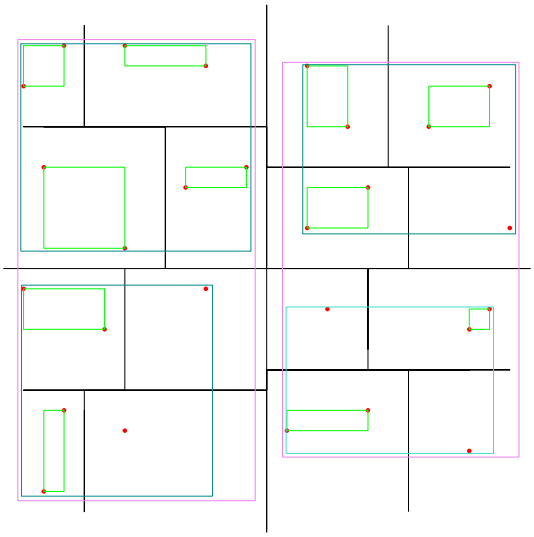
As I mentioned before almost all conditions can be represented as orthogonal range queries. But the data structure also has to handle queries with a substring conditions. In each node of K-D tree I could store a structure which efficiently processes substring queries. I could use suffix trees, for example Ukonnen's algorithm [17]. 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 the suffix trees. If data structure has to return data ordered by some column, all it can do is fetch all rows matching the substring condition and sort them. Not to mention that this way I can process only one substring condition in a query.

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

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



(a) Storing whole area covered by nodes



(b) Area covered by points inside nodes

5 Implemented optimizations

What makes an algorithm useful in practice are optimizations. In this section I will present description of core optimizations implemented in Shuffla. The most important are first two optimizations which are making Shuffla few times faster.

5.1 Queue event

Web search-engines usually show maximum of 20 results per page. A high percentage of users would not look at more results anyway. If we input a random English word into Google search engine, the number of results exceeds millions. Nevertheless, instead of showing all of them, Google shows just 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, a 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 I insert such events into a priority queue. Let me consider the case when user requests 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 sufficient results have been found, the engine can stop processing search events from the queue. Hence, I can hope to substantially restrict the number of actually visited nodes of the K-D tree. This optimization alone may seem like a big improvement. It actually is not. The number of nodes added to the queue may be large.

Imagine that I want to find few points with smallest values of the x coordinate. Data structure needs to process at least all nodes that do not have a lower bound on x. Let me compute how many of them we need to process in the worst possible case:

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

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

5.2 Defined boundary and real boundary

Instead of storing whole area covered by the node (see Figure 2a), I calculate area covered by points inside each node (see Figure 2b).

Let me elaborate on an example where k points with minimal values of coordinate x are requested. First event contains root of the tree. It is processed. It causes pushing two of its sons to the queue. Next, I choose node with the smaller minimum value of coordinate x. This event is processed and another two nodes are added. The 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 a minimum value of coordinate x is processed. At this point I have found first point with a 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 I find k rows. After finding k points, search process can be terminated and results may be rendered. To sum up, $\mathcal{O}(k \log n)$ nodes are added to queue, and $\mathcal{O}(k \log n)$ nodes are processed. Finding minimum value of coordinate x of node takes $\mathcal{O}(1)$ since we store this information in boundary. Processing one node takes $\mathcal{O}(1)$ time. The bootleneck of this solution is priority queue. Priority queue may be implemented as a binary heap. I processed $\mathcal{O}(\log n)$ nodes so complexity of this solution is $\mathcal{O}(k \log n \log(k \log n))$.

5.3 Raw pointers instead of shared pointers

In modern C++ pointers should not be used and shared pointers are recommended instead. The idea behind shared pointer is to automate releasing resources. Basically, shared pointer keeps the counter representing the number of copies of the current pointer. When the shared pointer is created, the counter is incremented. When shared pointer is deleted, the counter is decremented. When the counter is decreased to zero, the resource is released. With such a tool developer can avoid the most common source of memory leaks. Unfortunately shared pointers come with time and space overhead. I decided to use raw pointers, which gives better running time. In retrospect, I think it was biggest mistake in the project. The 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 the binary trees. A binary tree can be unbalanced and so can K-D tree. My idea for balancing K-D tree is very simple. When the number of nodes in one subtree is 2 times larger than the number of nodes in the opposite subtree, I rebuild their parent subtree from scratch. 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 [15]. 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. A detailed description with examples of amortized complexity can be found in [16].

6 Implementation

6.1 HTTP Service

I decided that it would be best if the search engine could work as a 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 [5]. In Shuffla configuration file user should define the number of available threads. One of the server's starting procedure is to create all these threads. Every HTTP request is assigned to one of these threads by `boost::asio::ip::tcp::acceptor`. There is another thread for periodical functions. Currently it is invoking only saving snapshots. All dependencies of server starting procedure are presented in Figure 2.

6.2 Validation and passing queries to SearchEngine class

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

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

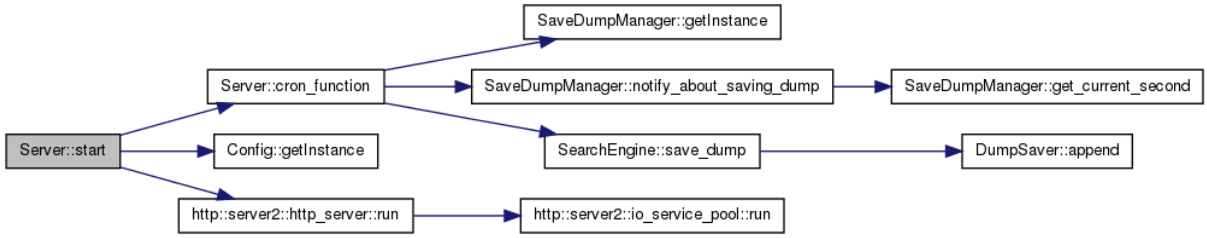


Figure 2: Processing request begins

1. data validation (types, correspondence to table definition),
2. measuring running time (the engine should report queries being executed for too long),
3. locking tables in case of modifying query,
4. passing valid requests to K-D Trees.

6.3 Processing insert/delete requests

Inserting/deleting without tree balancing is pretty straightforward. It is just a simple recursion described in the previous section. Balancing is tricky as you may see in dependencies visualization in Figure 3.

Balancing could be done using any linear algorithm for finding median in the set. I have decided to take a different approach. In `KDVertex::find_good_pivot` I am iterating over all points. Based on coordinate value in currently processed row I create a `Pivot`. `Pivot` is a 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 20 randomly selected rows in node and in linear time calculates pivot efficiency. If pivot efficiency is good enough (see definition of a super-balanced tree in Section 7) I stop calculations. If none of these rows create good pivots, I iterate over all rows and check them one by one if they creating good pivot. After selecting pivot 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, the rebuilding root may cause memory usage to double. Such a behaviour is very unwelcome in server environment. The implemented solution is $\mathcal{O}(n^2)$ but in average case it works well because $\frac{1}{5}$ of rows create good pivots. Therefore, the expected complexity of rebuilding subtree with n nodes is $\mathcal{O}(n \log n)$.

6.4 Processing search requests

When request data is validated, control is passed to the `TableIndex` class. Actual computations begin here. Let us 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 a node to process. There is also `SearchTaskFoundRow` which always contains one row. This row may be inserted to the task queue if and only if it satisfies all conditions defined in the search request. Both these classes extend abstract class `SearchTask`. When processing

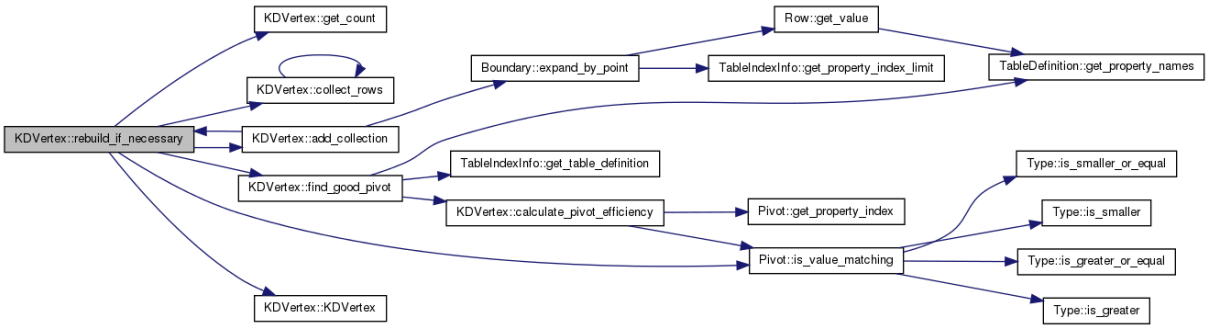


Figure 3: Rebuilding process

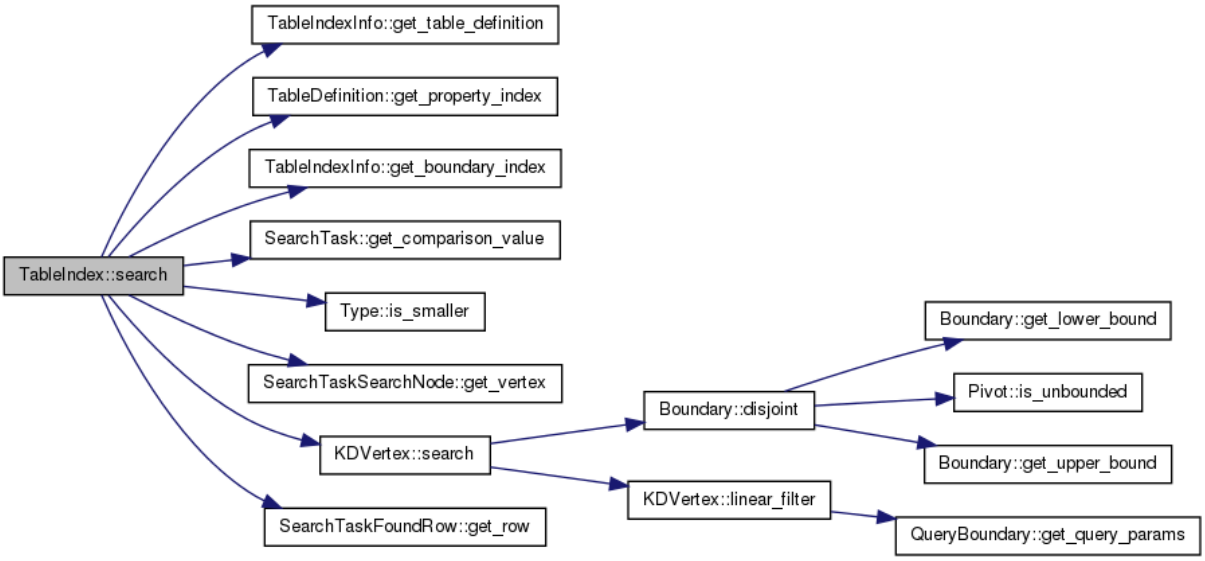


Figure 4: Processing search request

`SearchTaskFoundRow`, algorithm just adds row to the result set. When processing `SearchTaskSearchNode` algorithm passes control to `KDVertex::search`, which returns all events that should be added to the queue. `KDVertex::search` checks if boundary defined by query is disjoint with current node boundary. If it is disjoint, there is nothing more to do. If not, the algorithm simply adds two sons of current node to the queue.

There was an idea to add possibility of multiple index creation. Such index could be created for some subset of the 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 the searching process are visualized in Figure 4.

6.5 Locking tables

To correct work in multi-threaded environment we must ensure thread safety. It requires from 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 the code.

My goal is to create the following schema: Any number of read-only queries can be run concurrently as long as no write query is being performed at the time. Write queries can be performed if no other query is being performed at the same time.

The advantage of this locking schema is 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 mutexes with exclusive ownership semantics, along with a shared ownership (multiple-reader or single-writer) mutex. There are three most common kinds of locks. All locks work in a similar fashion. Mutex is passed as a constructor parameter to lock. Ownership of mutex is acquired in a constructor. Ownership of mutex is released when the lock object is destroyed.

1. `unique_lock` - when mutex is acquired by `unique_lock`, no one can acquire this lock until unique lock releases 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 uses `shared_lock` and write request uses `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 supposed to.

6.6 How data is stored

Architecture of data storage in databases is very important. It influences application performance and architecture of all other components. Here are the requirements for data storage:

1. all data must be serializable,
2. chosen architecture should enforce 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 never change 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. Let us take a look at `TypeString`. Storing text values in `std::string` would be inefficient because of overhead caused by `std::string` being not immutable. A text is actually stored in `char*`. This array is created in `TypeString` constructor because at this point I know the text length.

Both `TypeString` and `TypeNumber` implement 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` [6].

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

6.7 Database persistence

In case of a crash there must be a way to restore database. In this section I will describe how Shuffla preserves data. Guaranteeing persistence of database which holds data in RAM is difficult (as you can see in the example of Redis [7]). 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, a 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.

6.8 Presenting results

Presenting results may sound like a trivial issue. Actually it is not and hardly any libraries would fit all the 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 [8] .

7 Complexities

Let n be the total number of points stored in the whole tree T. For a vertex $v \in T$, I denote by $\text{left}(v)$ and $\text{right}(v)$ its left and right subtree, respectively. T is α-balanced if and only if

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

The tree is well-balanced if it is 2-balanced and super-balanced if it is $\frac{3}{2}$-balanced. The height of both well- and super-balanced trees is $\mathcal{O}(\log n)$. Every point belongs to $\mathcal{O}(\log n)$ nodes.

To simplify the analysis, I assume that the values in all columns are unique. I will ensure that the tree is always well-balanced by rebuilding its subtrees that violate this property. I assume that whenever data structure rebuilds a subtree, it becomes super-balanced. While it is possible to ensure that it becomes perfectly balanced, for the reasons explained in Section 6.3 the actual implementation guarantees a weaker invariant.

7.1 Inserting/deleting point

Lemma 1. *Rebuilding tree of size n can be done in $\mathcal{O}(n \log n)$ time.*

Proof. I can find the median in linear time [13]. Hence, in order to rebuild a tree, I 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 the 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. I 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 $n \log n$). I must prove that the invariant still holds after each insert/delete and that I always have enough credits to amortize the rebuilding. I will consider those two parts separately.

1. Each node visited during an insert/delete operation gets $6 \log n$ credits. Notice that after the visit I either increase or decrease n_l or n_r. Hence, I increase $6|n_l - n_r| \log n$ by at most $6 \log n$ and so I can afford to pay for this increase using the new credits allocated to the node (it can also happen that I decrease $|n_l - n_r| \log n$, which is even better).

2. I 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 I reconstruct the tree, the required number of credits will be at most $\log n$. Hence, I can use $(n-1) \log n$ credits to pay for the reconstruction and by Lemma 1 this is enough.

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

Now, for each node of the tree I 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 the work done sums to $\mathcal{O}(n \log n)$. There are $\mathcal{O}(\log n)$ levels, therefore the complexity is $\mathcal{O}(n \log^2 n)$.

I 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. Let me count maximal possible number of visited nodes on level d by a recursive search algorithm where region is defined by $x_i \leq C$. A 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 splitting according to the the i-th dimension. 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 the number of processed nodes. Hence, the 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 the number of processed nodes is maximal. From proof of Lemma 3 we know that it is the scenario with search region defined as $x_i \geq C$. For all nodes with pivots splitting according to the i-th coordinate it is true that nodes from one son will be processed and nodes from the 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 $T(n) = 1 + 2^{d-1} T(\frac{\frac{2}{3}n}{2^{d-1}})$.*

Proof. My goal is 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 I can choose constant c. Now let me prove that $T(n) \leq cn^a$ assuming that $T(n') \leq c(n')^a$ for all $n' < n$. I 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 I am interested in upperbounding $T(n)$, I am 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:

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

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 there exists $a < 1$.

□

Now, I can solve $T(n) = 1 + 2^{d-1}T(\frac{\frac{2}{3}n}{2^{d-1}})$ by applying the master theorem, which boils down to the following simple computation:

$$\begin{aligned}\frac{\log 2^{d-1}}{\log 3 * 2^{d-2}} &= \log_{3*2^{d-2}} 2^{d-1} = \log_{3*2^{d-2}} 2^{d-1} + \log_{3*2^{d-2}} \frac{3}{2} - \log_{3*2^{d-2}} \frac{3}{2} \\ &= \log_{3*2^{d-2}} 3 * 2^{d-2} - \log_{3*2^{d-2}} \frac{3}{2} = 1 - \log_{3*2^{d-2}} \frac{3}{2}\end{aligned}$$

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 the resulting time complexity.

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 $\mathcal{O}((\text{offset} + \text{limit}) \log n)$.

Let me explain that by taking data with random distribution. When algorithm recursively enters node of K-D tree, 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 the 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, finding single row takes $\mathcal{O}(\log n)$.

8 Experiments

To compare the performance of my implementation, I have tested it on data which was supposed to simulate the example use case. All engines must be tested using the same schema and on the same queries. Yet presented engines have different syntax. Below I describe table configurations and how queries were translated for different engines during their testing. Then I describe the queries themselves, and present the results.

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

Here is the 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 $[2,\infty]$: `-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 [9] and GIST [10]. Unfortunately I was unsuccessful in making them work with Polish words. Therefore I tested 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` 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

8.4 Testing environment

Testing environment is a 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. The most important change is increased memory limit for Solr and PostgreSQL to 1GB.

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

At the beginning I insert $8 * 10^6$ randomly generated users. Then I will run a script which creates random insert/search queries. It sends created queries for a few hours and calculates statistics.

Inserted data is generated in a very simple way. I have gathered lists of most popular first and last names in Poland (with counts). The probability of selecting a name is proportional to its count. The 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. The column in expression is selected from a 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), the 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, the possible queries are comparison, inequalities, prefix queries and substring queries. Prefix and substring queries have 25 percent chance to be chosen. Each of the remaining conditions has 10 percent chance to be chosen. The selected column is always compared to another string. This string is selected by selecting a random word with 1 to 5 characters.

The average length of combined `first_name`, `last_name`, `city` fields contains 21 characters. It means that the size of inserted data is about 250 MB, but of course keeping data in data structures always incurs some overhead (for example, Shuffla needs 6 GB).

Note that inserting 8 million rows is not the most important part of testing. It is performed before production deployment and I will do this part only once. I am happy as long as indexing time does not last longer than a few hours. While Shuffla handles insert requests with speed of hundreds 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.

After building the database I will select random column and set it as order by column. Then I will choose offset and limit from range $[0, 50]$.

8.5 Statistics

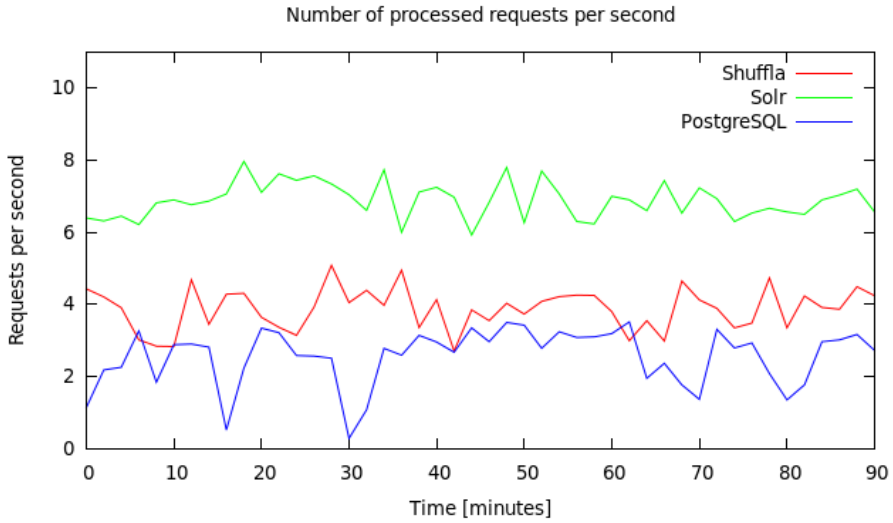
For the purpose of testing in true multi-threaded environment I used tsung. Tsung is a distributed load testing tool written in Erlang. It is protocol-independent and can be used to stress HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP, and Jabber/XMPP servers.

I have used 4 core processor for testing so I decided to test all engines using 4 threads. Each thread sends 10.000 randomly created queries.

Solr and PostgreSQL use less than 1 GB of RAM. It is possible because both these engines keep data on the hard drive. Shuffla stores everything in RAM and consumes 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. It uses less memory. It is more configurable, we could for example increase Solr memory limit. Solr is just better option, especially for large social website purposes.



9 Few words about Solr in nk

9.1 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, see Figure 5.

When some action requires modifying data in Solr, PHP code creates 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, a 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 the search requests go through HAProxy [11]. This load balancer is distributes requests between our 3 servers.

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

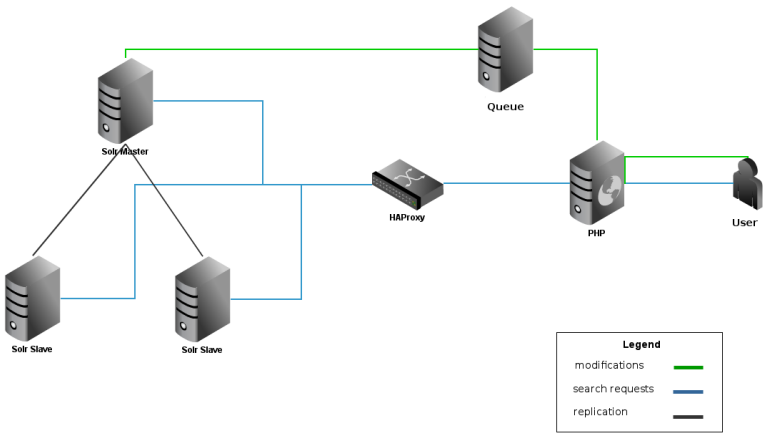


Figure 5: Architecture of searching

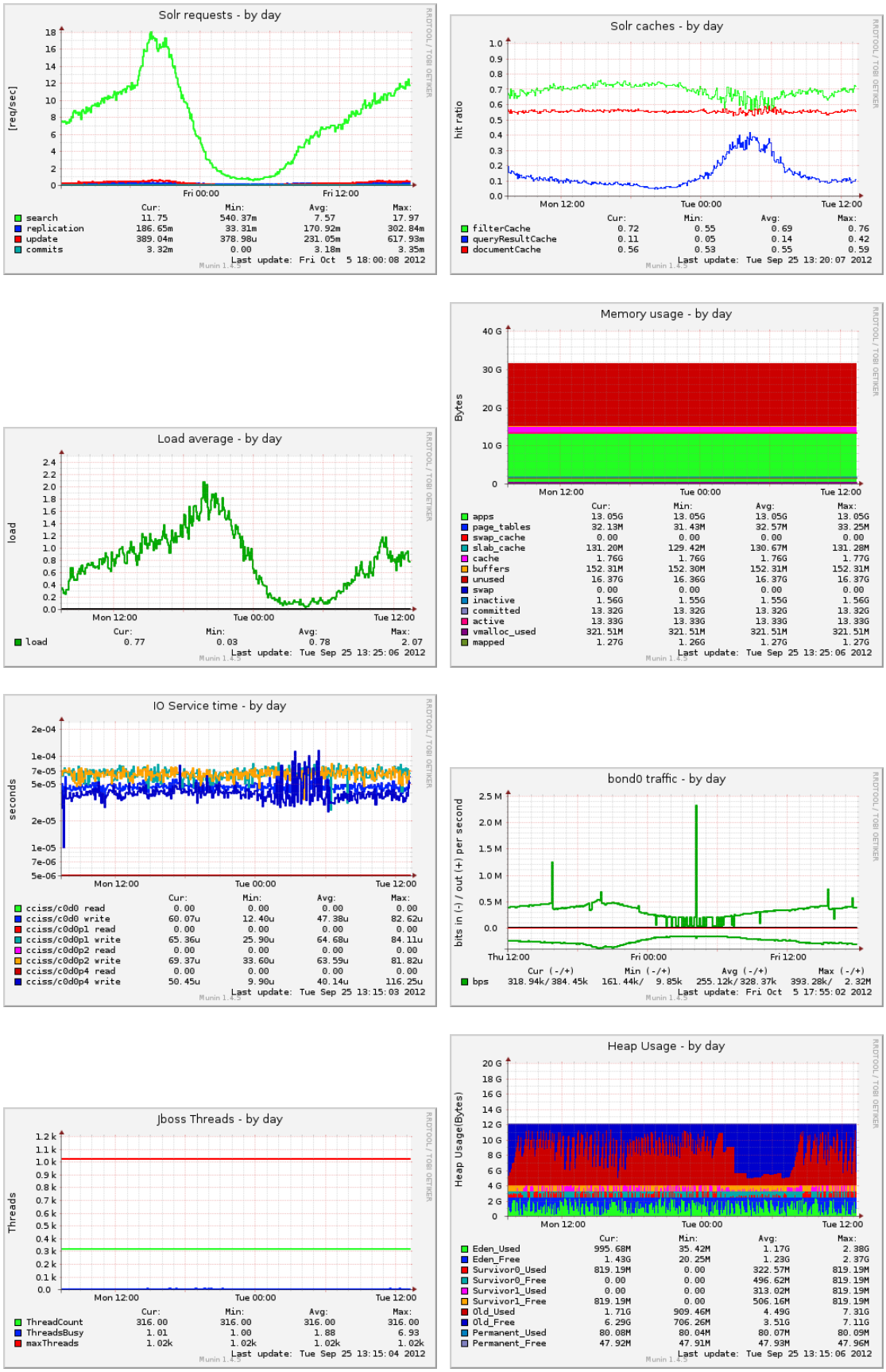


Figure 6: Statistics from nk

9.2 Statistics from the master server

Figure 6 shows a few statistics from the master server.

Number of requests, load - by day

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

At nk we use half of available memory because the other half is reserved for unexpected events, for example, a 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 commit 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 a disk works during a second. It shows very small values. On production all Solr data is kept on ramdisks. It is not zero because of the work done by operating system. Solr network traffic is not huge, it rarely reaches 1 megabit. In the evening, we reach a 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 chose 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 a small number of requests and Solr is unable to get passed warming stage before next commit. The hit ratio is slightly smaller, about 60 percent at night.

10 Summary

After this discussion it might seem that Shuffla is useless. I have to disagree. Shuffla fits great for smaller projects like searching through internet forum or a blog. Both Solr and PostgreSQL require quite difficult installation and configuration. Installation of shuffla takes 2 minutes, creating a 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://dev.w3.org/geo/api/spec-source.html.

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

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

[4] http://sphinxsearch.com/docs/manual-2.0.5.html#rt-caveats.

[5] http://www.boost.org.

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

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

[8] http://stackoverflow.com/questions/10260688.

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

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

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

[12] 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.

[13] 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.

[14] A. F. Cárdenas. Analysis and performance of inverted data base structures. pages 253 – 263, 1975.

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

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

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