Word Evaluator Research

VocabVersus

Thomas van der Molen

|  |  |
| --- | --- |
| **Project Information** | |
| Project members | Thomas van der Molen |
| Project Name | VocabVersus |

Table of Contents

[Topic 3](#_Toc132383750)

[Goal 3](#_Toc132383751)

[Summary 3](#_Toc132383752)

[Questions 3](#_Toc132383753)

[Word Set Storage 4](#_Toc132383754)

[Persistence 4](#_Toc132383755)

[Storage Options 4](#_Toc132383756)

[Relational Database 4](#_Toc132383757)

[Loosely Structured Database 5](#_Toc132383758)

[Word evaluation 5](#_Toc132383759)

[Database Query 5](#_Toc132383760)

[Apache Lucene 5](#_Toc132383761)

[Cloud Search 6](#_Toc132383762)

[Evaluator Real World Performance 7](#_Toc132383763)

[SQL Database 7](#_Toc132383764)

[Performance 7](#_Toc132383765)

[Data Management 8](#_Toc132383766)

[Apache Lucene 9](#_Toc132383767)

[Performance 9](#_Toc132383768)

[Data Management 9](#_Toc132383769)

# Topic

Vocab Versus requires a service where **correct words can be stored** and later be checked against to **see if a word submitted by a player is correct**.

For this service, large amounts of data in the form of word lists (which contains all correct words for a given topic e.g. English, periodic table, etc.) have to be persistently stored to be re-used in games. Secondly, the service will have to be able to look through a given word list and find if a word submitted by a user is contained within the list.

This whole process will have to happen with many player submissions and return the result in real-time, as to give the player feedback on their submission.

# Goal

The goal of this research will be to explore the different methods of **storing large amounts of text** in the form of word lists, with the ability to (very) **quickly search** through the text and find a match for a given word.

# Summary

This research document explores multiple solutions for storing VocabVersus word sets, evaluating user word submissions using research and POC applications.

Based on the research, the approaches were chosen to explore further for usage in VocabVersus, these being a [relational database with SQL queries](#_SQL_Database) and a document based querying structure with the [Apache Lucene framework](#_Apache_Lucene).

Both Apache Lucene and SQL Database solutions had promising results; having acceptable response times within the scope of a real-time web game when configured properly.

However both also showed limitations attached to this, such as the SQL Database requiring extra storage space for indexing and limitations with the EFCore ORM, while Apache Lucene requires significantly more set-up and research for the required implementation.

Based on the findings summarized above, the VocabVersus Word Evaluator will use Lucene.NET for its storage and evaluation as it showed the best results within the VocabVersus scope, ignoring the more convoluted implementation required.

# Questions

There are multiple questions that will guide the research, these questions are based on the topics explained in the [introduction to this topic](#_Topic_1).

For the Sub Questions, certain Research Methods that are expected to be used have been documented, this does not include all methods that will be used, but does cover the [Library](https://ictresearchmethods.nl/Category:Library), [Workshop](https://ictresearchmethods.nl/Category:Workshop) and [Showroom](https://ictresearchmethods.nl/Category:Showroom) research strategies from the [DOT framework](https://ictresearchmethods.nl/The_DOT_Framework).

|  |  |
| --- | --- |
| **Main Question** | |
| What is the best method of storing and evaluating large amounts of grouped words to use for a game of Vocab Versus | |
| **Sub Questions** | **Research Methods** |
| What storage solutions exist for storing large amounts of textual data | [Literature study](https://ictresearchmethods.nl/Literature_study)  [Community research](https://ictresearchmethods.nl/Community_research) |
| What specialized methods exist for finding matching parts of text within a large amount of text | [Available product analysis](https://ictresearchmethods.nl/Available_product_analysis) |
| What combination of persistent storage and word matching technique gives the most optimal execution time | [Prototyping](https://ictresearchmethods.nl/Prototyping)  [Benchmark test](https://ictresearchmethods.nl/Benchmark_test) |

# Word Set Storage

The VocabVersus game relies on word sets to validate that a given word submission by a player is a valid word. This word set must be stored somewhere so it can be referenced in the future during a game.

## Persistence

Due to the same word set being allowed to be used in many different games, the data will have to be stored persistently and cannot be coupled, for example to a single game instance.

Due to the large nature of a word set and the ability to use different word sets depending on the game rules, the data size and storage capacity will have to be kept in mind when determining how to persistently store these datasets.

## Storage Options

For this research, database solutions will be discussed as they are the most prominent and accessible method of persistently storing data, the specific database technologies that will be explored are based on the most popular database technologies as explored in the [2022 stackoverflow developer survey](https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-databases).

### Relational Database

[Relational databases](https://www.ibm.com/topics/relational-databases) are a common practice method for storing persistent data, as it allows for large amounts of data to be stored in a predefined structure.

There are many different structured relational databases, such as:

* [MSSQL](https://www.microsoft.com/en-us/sql-server/sql-server-2019)
* [MySQL](https://www.mysql.com/)
* [SQLite](https://sqlite.org/index.html)
* [PostgreSQL](https://www.postgresql.org/)

Most of these databases offer similar functionality based on the SQL query language and therefore also have similar [benefits and tradeoffs](https://databasetown.com/relational-database-benefits-and-limitations/#:~:text=The%20main%20benefits%20of%20using,issue%20of%20speed%20can%20arise.).

|  |  |
| --- | --- |
| **Benefits** | **Limitations** |
| Fast query processing | Harder scalability |
| Data consistency | Complex relational structures |

One other large limitation that is often brought up for SQL databases, is that performance may decrease drastically as datasets grow however, as shortly introduced in [this blog article](https://blog.greglow.com/2018/02/05/sql-server-big-databases-really-slower/) this can often be remedied with modern database technologies via methods such as [indexing or compressing information](https://www.mssqltips.com/sqlservertip/7309/improve-sql-server-performance-when-querying-very-large-log-tables/).

Storing word sets in a structured SQL database could look like the ERD displayed below:

Text

Description automatically generated

### Loosely Structured Database

Loosely structured databases, more often referred to as [NoSQL databases](https://www.ibm.com/topics/nosql-databases#:~:text=NoSQL%2C%20also%20referred%20to%20as,structures%20found%20in%20relational%20databases.), are created to be support more flexible data structures then conventional [SQL based databases](#_Relational_Database). NoSQL databases have several different types, with the 4 primarily used ones being:

* Document
* Key-value
* Column-oriented
* Graph

For a clearer understanding of these 4 types I highly recommend the [introduction of these types by MongoDB](https://www.mongodb.com/scale/types-of-nosql-databases).

|  |  |
| --- | --- |
| **Benefits** | **Limitations** |
| Data flexibility | Less extensible querying |
| Significant modern growth support | No guaranteed data consistency |

# Word evaluation

Besides storing the word sets, word sets will have to be searched through to find a match with a player submitted word, this can be done in many ways with the primary goal being to find a match as fast as possible.

## Database Query

When using an [SQL based database](#_Relational_Database) to store word sets, the SQL language can be used to make queries directly to the database.

An SQL query to find the first occurrence of a word inside a wordlist, utilizing the database structure as [presented during the relational database](#_Relational_Database) could look like the following.

SELECT **COUNT(1)**

FROM **words**

WHERE **setId = wordsetid** AND **word = playersubmittion**;

This query will return either 0 if no instance was found or 1 if a match was found. An issue with this approach, however is that with normal behavior all word with the given wordsetid will be looked through individually until a match is found, this process can however be improved by using [indexing methods](https://www.sqlshack.com/top-five-considerations-for-sql-server-index-design/) on the word value.

One security risk to keep in mind when generating a database query with user input, is to ensure no [SQL injection](https://portswigger.net/web-security/sql-injection) happens by sanitizing the request for which there are many different techniques; one common method is by escaping any input thereby ensuring it is not read as valid code/queries.

## Apache Lucene

[Apache Lucene](https://lucene.apache.org/) is a specialized library for text search, search engines such as [ElasticSearch](https://www.elastic.co/elasticsearch/) and [Solr](https://solr.apache.org/) are based on Apache Lucene.

Lucene works similar to a [NoSQL document based database](#_Loosely_Structured_Database), with the ability to serialize stored text specifically for future search, speeding up this process significantly along with other built-in features such as [fuzzy search](https://www.ibm.com/docs/en/informix-servers/12.10?topic=modifiers-fuzzy-searches).

Due to Lucene being an open source java based library, it has been ported to many other languages and frameworks such as [.NET](https://lucenenet.apache.org/), [python](https://lucene.apache.org/pylucene/), [etc](https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=119540992).

## Cloud Search

There are several enterprise level cloud solutions built specifically for text search, while these could solutions might not be as versatile as solutions previously mentioned, the easy of use along with the reliability of large scale cloud solutions could be a valuable trade off.

While there are many text search cloud solutions some popular ones include: [Azure Cognitive Search](https://azure.microsoft.com/en-us/products/search/?ef_id=_k_Cj0KCQjwxMmhBhDJARIsANFGOStIUCOOE99ZIdHGoTfI7hNZkJXyeNsWVsp4-Pm0QsnRaGXG6pUYs20aAm4iEALw_wcB_k_&OCID=AIDcmmy4pl1olr_SEM_k_Cj0KCQjwxMmhBhDJARIsANFGOStIUCOOE99ZIdHGoTfI7hNZkJXyeNsWVsp4-Pm0QsnRaGXG6pUYs20aAm4iEALw_wcB_k_&gclid=Cj0KCQjwxMmhBhDJARIsANFGOStIUCOOE99ZIdHGoTfI7hNZkJXyeNsWVsp4-Pm0QsnRaGXG6pUYs20aAm4iEALw_wcB). [Amazon CloudSearch](https://aws.amazon.com/cloudsearch/) and [Algolia](https://www.algolia.com/).

While these cloud search engines are often very powerful in applications such as in-website search or general web indexing, they often sadly do not provide much flexibility for specific data list search.

# Evaluator Real World Performance

Different methods explored previously will be investigated using real world POC’s, these POC projects will be [ASP.NET](https://dotnet.microsoft.com/en-us/apps/aspnet) WebAPI based applications and will be testing against performance of the provided insert and evaluate endpoints and data management.

For testing data, a [list of English words](https://raw.githubusercontent.com/sindresorhus/word-list/main/words.txt) will be used containing 274.411 total words and will be referenced as a single word set.

## SQL Database

The project used during this investigation can be found [here](WordEvaluatorMSSQL) in the file structure or [online](https://github.com/Thomas-Molen/FHICT-S6-S/tree/main/Documentation/Research/WordEvaluatorMSSQL).

The WordEvaluatorMSSQL web API uses the [ASP.NET 6.0](https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0) framework with [EFCore 7.0.5](https://learn.microsoft.com/en-us/ef/core/) connected to a locally hosted [MSSQL database](https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16).

It should be noted that for the context of VocabVersus I would not recommend using a large ORM like EFCore as these tend to limit the full usage of a database of possibly negatively impact performance with redundant functionality in specific use-cases as discovered during [Data Management](#_Data_Management). However, EFCore gives an extremely easy-to-use system for creating and querying databases allowing for fast POC development time.

### Performance

#### Word Set Insert

For inserting word sets, the following curl request was used with the Words array containing the word list referenced in the [introduction](#_Evaluator_Performance).



|  |
| --- |
| **250.000 Records** |
| 11,82 s |

The 11,82s measured for adding what is considered a regularly sized dataset is very long, this can however be explained to the use of EFCore, as the version used does not yet support [Bulk inserts](https://entityframework-extensions.net/bulk-insert) natively, this causes the insertion of records to be [batched](https://learn.microsoft.com/en-us/ef/core/performance/efficient-updating) per ~40 records (this could be done due to a possibly similar limitation [within SQL databases themselves](https://stackoverflow.com/questions/5940225/fastest-way-of-inserting-in-entity-framework)).

#### Word Evaluation

Below two different database structures were tried, the first attempt was with no extra database setup, and the second approach with indexing the Word column using B-Tree index, as mentioned during the [Relational Database](#_Relational_Database) research.

For evaluating a word, the following curl request was used with the relevant wordSetId and word in the query parameters.



|  |  |  |  |
| --- | --- | --- | --- |
| 1 million total records, 4 word sets, 10 requests | | | |
| Default MSSQL database | | | |
| **Starting Record** | **Middle Record** | **Last Record** | **Missing Record** |
| 4,0 ms | 71,5 ms | 141,0 ms | 149,7 ms |
| Indexed MSSQL database | | | |
| **Starting Record** | **Middle Record** | **Last Record** | **Missing Record** |
| 4,5 ms | 4,1 ms | 3,5 ms | 4,1 ms |

From the results above, it is clearly visible that indexing the word and wordSetId columns gives significant performance increases, with the none indexed table having worse performance the more records have to be looked through.

### Data Management

#### Storage

Below the storage properties of the two different database structures measured during the [Word Evaluation Performance](file:///F:\School\S6_Software\FHICT-S6-S\Documentation\Research\Word%20Evaluator%20Research.docx#_Word_Evaluation) are displayed

|  |  |  |
| --- | --- | --- |
| Default MSSQL database | | |
| **1 Million Records** | **2 Million Records** | **3 Million Records** |
| 322,34 MB | 441,90 MB | 561,65 MB |
| Indexed MSSQL database | | |
| **1 Million Records** | **2 Million Records** | **3 Million Records** |
| 629,41 MB | 866,91 MB | 1.102,97 MB |

From the results obtained from these tests, it is very noticeable that the indexing of words costs significantly more memory as it is used to [perform more efficient operations](#_Word_Evaluation) traversing a Binary Tree functionally the same length as the longest word in the column.

#### Memory

|  |  |  |
| --- | --- | --- |
| **After adding 250K records** | **After adding 1M records** | **1M records after restart** |
| 1.627,2 MB | 1.843,5 MB | 40,9 MB |

The above table shows the memory usage of the application in 3 different states, the first two being after the process has added 250 thousand and 1 million records respectively, with the third state having had a full restart of the application.

All 3 of the tests have had time to allow the .NET garbage collector to free up all stale memory and having performed several word evaluation requests.

From the results it seems that EFCore does not free up memory of inserted records until a limit (as the increase in memory from 250 thousand to 1 million is much smaller than from 0 to 250 thousand), this does align with the functionality of EFCore’s [entity tracking](https://learn.microsoft.com/en-us/ef/core/change-tracking/) and [identity resolution](https://learn.microsoft.com/en-us/ef/core/change-tracking/identity-resolution) systems which help increase database query performance as (in the case of identity resolution) functioning as an in memory cache.

## Apache Lucene

### Performance

The project used during this investigation can be found [here](WordEvaluatorLucene) in the file structure or [online](https://github.com/Thomas-Molen/FHICT-S6-S/tree/main/Documentation/Research/WordEvaluatorLucene).

The WordEvaluatorLucene web API uses the [ASP.NET 6.0](https://learn.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0) framework with the 4.8.0-beta00016 versions of [Lucene.NET](https://www.nuget.org/packages/Lucene.Net/4.8.0-beta00016/), [Lucene.NET.Analysis.Common](https://www.nuget.org/packages/Lucene.Net.Analysis.Common/4.8.0-beta00016/) & [Lucene.NET.QueryParser](https://www.nuget.org/packages/Lucene.Net.QueryParser/4.8.0-beta00016/) storing the Lucene data in the file system.

#### Word Set Insert

For inserting word sets, the same process as the [SQL Database project](#_SQL_Database) was used.

|  |
| --- |
| **250.000 Records** |
| 0,86 s |

The 0,86 second insert time for a word set is within acceptable process times, with presumably most of this time being spent [analyzing](https://lucene.apache.org/core/8_0_0/core/org/apache/lucene/analysis/Analyzer.html) the word data for faster future querying.

#### Word Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| 1 million total records, 4 word sets, 10 requests | | | |
| Lucene engine | | | |
| **Starting Record** | **Middle Record** | **Last Record** | **Missing Record** |
| 9,0 ms | 8,6 ms | 8,3 ms | 8,3 ms |

Performing single occurrence searches seems to perform very efficient searches into the whole dataset, it should also be mentioned that the request results above had very low standard deviations (<1 ms).

### Data Management

#### Storage

Below the storage property of the Lucene file system used during the [Word Evaluation Performance](#_Word_Evaluation_1) is displayed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Lucene file storage | | | | |
| **1 Million Records** | **2 Million Records** | **3 Million Records** | **4 Million Records** | **5 Million Records** |
| 16,22 MB | 32,44 MB | 21,92 MB | 38,14 MB | 68,18 MB |

From the above data, is seems that Lucene is able to store data fairly efficiently, with the result of 3 Million Records, showing how Lucene tries to optimize this storage whenever possible, even reducing the total size compared to 2 Million Records.

#### Memory

|  |  |  |
| --- | --- | --- |
| **After adding 250K records** | **After adding 1M records** | **1M records after restart** |
| 474,3 MB | 654,1 MB | 41,5 MB |

The above table shows the memory usage of the application in 3 different states, the first two being after the process has added 250 thousand and 1 million records respectively, with the third state having had a full restart of the application.

All 3 of the tests have had time to allow the .NET garbage collector to free up all stale memory and having performed several word evaluation requests.

It seems that Lucene’s Index Writer keeps its memory buffer even after flushing/committing the data, this should not cause any memory leak as the memory buffer is overridden when full in accordance to the given [configuration](https://lucenenet.apache.org/docs/4.8.0-beta00009/api/core/Lucene.Net.Index.IndexWriterConfig.html). However, this internal memory management should be explored further if used in VocabVersus.