Implementation and Evaluation of CHQL Operators in Relational Database Systems to Query Large Temporal Text Corpora

Bachelor's Thesis of

Albu Dumitru-Cristian

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr.-Ing. Klemens Böhm
Second reviewer:	Prof. Dr.-Ing. Anne Koziolek
Advisor:	Jens Willkomm M.Sc.

08. April 2019 – 07. August 2019

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 07. August 2019

. . . .

(Albu Dumitru-Cristian)

Abstract

Relational database management systems have an important place in the informational revolution. Their release on the market facilitates the storing and analysis of data. In the last years, with the release of large temporal text corpora, it was proven that domain experts in conceptual history could also benefit from the performance of relational databases. Since the relational algebra behind them lacks special functionality for this case, the *Conceptual History Query Language (CHQL)* was proposed. Thus, our objective is to determine to what extent CHQL operators could be implemented in a relational database. The second objective is to evaluate the performance of our implementation. Finally, we investigate in what way can the query optimizer be improved in order to execute efficient plans.

We chose to do the implementation in Oracle and PostgreSQL databases, as they have already shown their capability in critical application throughout the years. Therefore, we can assume their query optimizers are efficient. The first result of this thesis is an original implementation of the CHQL operators in both systems, which is written in both SQL and its procedural extension. Secondly, we improved substantially the performance with the trigram indexes in PostgreSQL. Lastly, the query plan analysis reveals the problem behind the query optimizers choice of inefficient plans, that is the inability of predicting correctly the results from a stored function.

Zusammenfassung

Relationale Datenbank-Management-Systeme spielen eine wichtige Rolle in der Digitalisierung. Ihre Marktveröffentlichung erleichtert die Datenspeicherung und -analyse. Kürzliche Veröffentlichungen von großen temporal Text Corpora haben bewiesen, dass Experten der Begriffsgeschichte vom Gebrauch relationaler Datenbanken profitieren können. Wegen der begrenzten Funktionalitäten der zugrunde liegenden relationalen Algebra, wurde die *Conceptual History Query Language (CHQL)* vorgeschlagen. Unser Ziel ist zu ermitteln, inwiefern CHQL Operatoren in eine relationale Datenbank implementiert werden können. Zweitens müssen wir das Leistungsverhalten dieser Implementierung bewerten. Als Letztes untersuchen wir Verbesserungsmöglichkeiten für einen effizienteren Abfragen-Optimierer.

Wir haben uns für Implementierungen in Oracle und PostgreSQL Datenbanken entschieden, da diese ihre Fähigkeiten mehrmals in unterschiedlichen wichtigen Anwendungen gezeigt haben. Daraus kann auf effiziente Abfragen-Optimierer geschlossen werden. Das erste Ergebnis dieser Arbeit ist eine eigene Implementierung von CHQL Operatoren in beide Datenbank-Systemen. Dies geschieht sowohl in SQL als auch in der entsprechenden prozeduralen Erweiterung. Als nächstes wurde die Performance in PostgreSQL mit Hilfe von trigram Indizes deutlich verbessert. Im Anschluss hat eine Analyse des Abfragen-Ausführungsplans Probleme bei der Wahl ineffizienter Pläne durch den Abfragen-Optimierer aufgezeigt. Er ist nicht fähig die Ergebnisse benutzerdefinierter Funktionen vorherzusagen.

Contents

List of Figures

List of Tables

1. Introduction

Nowadays, almost everything around us can be accessed in a digital form and we contribute to this fact. We produce data by simply making an account on a shopping website or even by making a bank transaction. Yet, we need to have this data always available even if it means to check our financial statements without leaving the house. Even more interesting is that this demand is not new. Over the years, database management systems have been employed extensively to organize data. An interesting kind of database management systems are the ones based on the relational model. After about forty years from their release on the market, the relational database management systems (RDBMS) are still widely-used for more complicated tasks everyday. Due to the longstanding utilization and research, they have become highly efficient and versatile. According to the DB-Engines Ranking[1], they take the top four places as most popular database management systems.

The *Structured Query Language (SQL)* is the language of choice for querying and maintaining relational databases. Even if it is a well-matured and studied query language, SQL lack special functionality in some scenarios. This has led researchers to develop specialized query algebras oriented to solve a specific problem. Take, for example, historians and philosophers that investigate how the frequency-usage of a word has changed in time. One way to accomplish a research in this sense is to query information from large temporal text corpora. Therefore, the *Conceptual History Query Language (CHQL)* has been introduced in 2018 [10]. It is a query algebra designed to fulfill the current and future information needs of conceptual history domain experts.

In order to take advantage of the performance offered by RDBMS, we propose in thesis an implementation of the CHQL operators using the relational algebra. Moreover, we analyze the performance of the implementation and discuss the limiting factors of SQL and implicitly of the query optimizer.

[1] https://db-engines.com/en/ranking

Research Questions

Our work could be divided into three parts: (1) the implementation of the operators, (2) the performance evaluation, and lastly (3) optimization. Consequently, we provide answers to the following research question:

1. To what extent can be the CHQL operators could be translated into relational operators?
   - If they can not be implemented, what functionality is missing and
   - how can we overcome it economical?
2. How efficient executes the query optimizer the CHQL operators?
3. In what way can we improve the query optimizer in order to execute efficient plans?

2. Fundamentals

In this chapter, we present the theory on which our implementation and performance evaluation is based. In the first part, we declare the fundamentals of the CHQL query algebra. This is highly relevant due to the fact that our operators implementation from section 3.1 is build on top of the information presented here. In the last part, we describe the RDBMSs from concept to the execution pipeline. This is important for the chapter 4 where we evaluate the performance metrics of our implementation.

2.1. Conceptual History Query Language

This query language is a joint effort of philosophers working on conceptual history and computer scientists. It provides an analysis tool for conceptual history domain experts [10]. The idea behind it is to retrieve data characteristics from large temporal text corpora. Those characteristics are derived from the information types provided by the work of Reinhart Kosselleck [7]. In the next part, we look at the concept types of conceptual history and their relation to data characteristic. This is one of the two challenges tackled by the query language [10]. Lastly, we dive deeper in the proposed data model and operators of CHQL.

2.1.1. Concepts

For every concept that follows, there is minimum a data characteristic that addresses it[10].

- **Conceptual Design** In texts, words can be used with a different meaning rather than their literal one. That means they can address different topics. The respective data characteristic is defined as the sum of the frequencies of the words that describe the same topic.

- **Context** The context of a word is described by the the neighbouring words and their meaning. For this concept there are two data characteristics proposed. The first one to retrieve the surrounding words for the target one. The second is to match the words to their sentiment.

- **Sentence Structure** This concept refers to the syntactical analysis of a sentence. It provides information about the evolution in structure and meaning. It requires also two data characteristics. First, a filter to separate on the part of speech value. Second, a search function for sentence patterns.

- **Neologism** A neologism is defined as a word introduced later in a language. It has an rising appearance frequency along the years. The data characteristic of this concept represent exactly that.

2.1.2. Query Algebra

The second challenge tackled by the CHQL is *completeness*. That means it can answer to the current and future hypotheses of conceptual history [10]. It accomplishes this with the following data model and operators.

Data Model

A temporal text corpus is composed of single words or word vectors together with part of speech tags and time-stamps. They are defined as ngrams. The formal definition of a ngram is $(\vec{w}, \vec{p}, \vec{ts}) \in W^n \text{ X } P^n \text{ X } \mathbb{Z}^T$. Here, n is the number of the words and T is the time-interval size of the time-series [10].

Operators

In this part, we summarize the CHQL operators. Detailed explications and formal definitions are present in the CHQL paper [10]. Furthermore, the majority of them are designed to be combined in order to obtain complex algebraic expressions.

- **Topic Grouping Operator** It matches every word from a set to its corresponding topics. Then it sumups the frequency of words that describe the same topic.
- **Surrounding Words Operator** For a given target word it returns the set of the neighbouring words.
- **Sentiment Operator** This operator attributes a sentiment value to every word from a set.
- **Text Search Operator** From a large set of words, it retrieves only the ones that corresponds to a given search pattern.
- **Part of Speech Filtering** For a given set of words, this operator filter to keep only the ones that are of a given part of speech.
- **Time Series-based Selection** This operator is also a filter, but this time for the values inside the time-series vector.
- **kNN Operator** It returns the k-nearest neighbours for a target word. That means the top k words with similar usage-frequency in a given search space.
- **Subsequence Operator** For a ngram, it reduce to a time-interval the time-series vector. In comparison, the previous operators work with the whole time-interval.
- **Count Operator** This is nothing else than returning the cardinality of a set.
- **Absolute Value Operator** This operator works on the time-series vector. For the whole range, it returns the vector with absolute value of the elements.

- **Sumup Operator** It has the same mechanism as the topic grouping operator. The only difference is that here the ngrams are not grouped. It returns the resulting time-series vector.
- **Set Operators** These operators comprehend the function for sets. For example, intersection, union and minus.

2.2. Relational Database Management Systems

A relational database management system (RDBMS) is a powerful software designed specifically for handling large amounts of data, e.g. from storing it safely to addressing queries to retrieve it. The first version of such system appeared in the early 1980s [6]. The *relational* term in the name points to the relational model used in these DBMSs.

E. D. Codd was a researcher in the IMB's hard-disk development division. Due to the fact that he was not satisfied with the available navigational model, he proposed in 1970 the relational model [3]. His paper had an enormous influence on the database systems development that followed. The main idea he introduced is that the user should be not concerned with how the data is stored and what perhaps complex data structures are used. The data should be presented to the user organized in tables, i.e. relations [3, 6].

2.2.1. Relational Algebra and SQL

Together with the relational model, E. D. Codd proposed also the relational algebra. This is an algebra on sets of tuples, i.e. relations, used to define queries about those relations [3, 6, 9].

We present here a summary of the most important relational operators for our work [6]. These operators are mostly based on Codd's work, but extended for commercial purposes, such as allowing duplicate tuples or executing operations on all tuples of a relation.

- **Set Operators** These refers to the common operations on sets, e.g. union, intersection and difference.
- **Projection Operator** This operator produce a new relation containing only a subset of the columns of a given table.
- **Selection Operator** Same as the previous operator, this one produce a new relation but with a subset of the tuples present in the given table.
- **Join Operator** Having two relations, this operators combines them on a specified condition to a new table that contains columns from both of them.
- **Aggregation Operators** These are an extension to the initial relational operators. Their main task is to summarize the values in one column of a relation. The standard ones are: *SUM*, *AVG*, *MIN and MAX* and *COUNT*.

- **Grouping Operator** This operator allows us to group the tuples in a relation according to our column criteria. If the grouping is used together with an aggregate operator, than the aggregation takes place inside the groups.
- **Combining Operations to Form Queries** The relational algebra allows us to form expressions of the desired complexity by combining operators. They could be addressed to existing relations or to resulting ones from other operators.

The most commonly used database language to implement the relational operators is the *Structured Query Language (SQL)*. The query functionality of this language is similar to the relational algebra previously mentioned. Due to the fact that it is also capable to declare the database's schema and also to control transactions and privileges, SQL is used to manage a big part of an RDMBS.

2.2.2. Query Processing

In order to understand what happens from the point we send a query written in SQL to the point we receive the results back in our application, we discuss the four main steps of processing and execution from Figure 2.1.

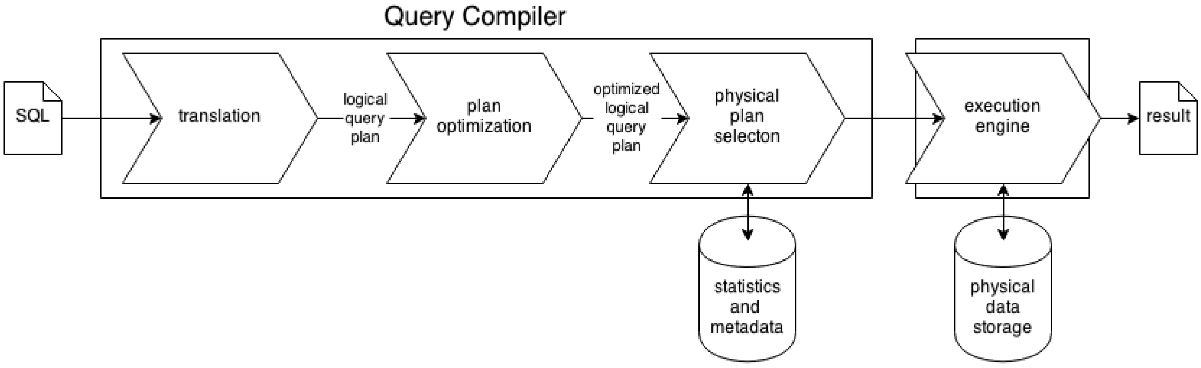


Figure 2.1.: Query Processing Pipeline (Source: mlwiki.org)

1. **Translation** In this step, the SQL query is translated recursively into relational algebra equivalent. Normally, the translation is visually represented with a tree structure.
2. **Plan Optimization** Because the query is translated into an relational algebra expression, there can exist equivalent expressions which are more efficient. Thus, the task of the query optimizer in this step is to find the most efficient expression that yields the same results as the target expression. This problem is undecidable. Therefore, the optimizer runs on a best-effort basis to provide the better plan.
3. **Physical Plan Selection** For this step is also the query optimizer responsible. The effort here consists of assigning for every node in the query plan received from the previous step the efficient algorithm. Its decision making process is helped in this instance by the statistics and the metadata. Because this optimization problem is solved trough a greedy

algorithm, the statistics are an important factor in a query performance, e.g. insufficient statistics lead to suboptimal plans.

4. **Execution Engine** The last step is executed outside the query compiler. Here the physical plan is executed. The database engine has the job of accessing the physical data storage and to correctly apply the algorithms on the retrieved data. After the processing is done, the results are sent back to the user who queried the database.

3. Implementation

In the first part of this chapter, we explain our design choices and we discuss the implementation of the CHQL operators. We build them on top of the formal definitions from section 2.1. Furthermore, this original contribution answers the question of what can be implemented with standard SQL and what are the limitations of this relational algebra. In the second part, we illustrate an use-case scenario provided by a benchmark concept. Moreover, we go over the query groups and explain their scope and their implementation in our work.

We have chosen for our implementation to work with only two RDBMSs, namely PostgreSQL and Oracle. Both of them are well-settled in the market and their optimizer engines present interest for our performance analysis. Therefore, we also show the differences between them regarding our implementation of the operators and respectively the benchmark.

3.1. CHQL Operators

In this section, we explain our process from implementing the data model to the actual implementation of CHQL operators. In the last part, we discuss the particularities we have encountered with both relational database systems.

3.1.1. Data Model

The problem of finding an appropriate data model that suits the CHQL functionality needs has already been tackled from a theoretical standpoint [10]. We have presented it in section 2.1. Therefore, our starting point is the abstract schema derived from the proposed mathematical one. In a formal sense, an **ngram** is defined in a relational database by the tuple `(word, part_of_speech[], timeseries[])`. Here the *word* represents a string, *part_of_speech* is an array of strings and *timeseries* is an arrays of numbers. The vector space from the mathematical definition is expressed through the array cardinality. The exception is the *word* attribute where the cardinality is equal with the number of words separated with space. This straight-forward approach has the advantage of keeping the query algebra unaltered.

The implementation of the abstract data model in different the two databases. Oracle does not support the definition of columns as array data type directly nor does it have the functionality present in PostgreSQL. To overcome this inconvenience we implemented two user-defined-types (UDTs). Moreover the columns in Oracle defined as UDT are not allowed in SQL operations as *GROUP BY* or *UNION*.

The final implementation of the data model in the two RDBMSs is presented in comparison in Table 3.1. *Pos* and *times* are *VARRAYs* of *varchar2* type respectively *double precision* type. Both systems are using for the time series arrays real numbers due to the fact that there are operators that calculate averages and therefore the results are consistent across the system.

	word	part_of_speech	time_series
PostgreSQL	text	text[]	double precision[]
Oracle	varchar2	pos	times

Table 3.1.: Implementation of the data model in PostgreSQL and Oracle

3.1.2. Language

Another important aspect of providing an efficient implementation of the CHQL operators is the language used. The obvious choice in every RDBMS is SQL, due to the fact that it is a declarative language. It has the advantage of taking the optimization away from the user. However, it does not suffice all the functionality needed to implement the operators. This can be easily overcome by using the additional procedural languages (PLs) offered by the vendors.

The SQL Procedural Languages are only available to be used inside stored functions or procedures. They add the features missing from the standard SQL, for example conditional loops, variables or complex conditional statements. Some other key features are the possibility to run dynamic SQL with the help of EXECUTE and caching the generic plans. The main mechanism on which they are working is that when an SQL statement is found inside a procedural language code block it is sent to the SQL engine to be performed. This could prove to be a performance bottleneck when too many of this context switches are occurring.

We have decided to use PL/pgSQL in PostgreSQL and PL/SQL in Oracle, as they benefit from being visible to the database. This means that SQL statements inside the procedural code blocks can be optimized. The main disadvantage would be that even if the SQL statements are optimized inside the code blocks, they are not aware of the other statements in the same block, losing statistics important to query planing. The alternative is to use directly a procedural programming language as `C, Python, et cetera`. Their disadvantage is that the implementation is not visible to the optimizer. This imply that the developer is in full control of the performance, and it can not take advantage of already optimized SQL statements. Another disadvantage is that they are prone to errors which are discovered only at runtime.

Oracle provides two different approaches to avoid the unnecessary context switches, if the task permits. The first one is the *FORALL LOOP* to be used whenever it is wanted to update, insert or delete large collections of rows. The second one is the *BULK COLLECT INTO* which is useful for *SELECT* statements that retrieve a lot of tuples [5]. The problem of statistics is not tackled by neither of them.

3.1.3. Discussion

In this subsection, we look at particularities of the actual implementation of the CHQL operators in the two DBMS. We focus on showing the key differences between PostgreSQL and Oracle. An overview of all the implemented functions and procedures is presented in the section A.1 and section A.2.

Data Partitioning

The CHQL operators *collocation* and *search* have as input parameter the size of ngram they work with. To avoid unnecessary memory usage we split the data accordingly as follows: for every type of *ngram* there is a table with the name *ngram_x* where x ∈ {'one', 'two', 'three', 'four', 'five'}. The tables have their attributes defined by the data model from Table 3.1. This partitioning improves the performance of the previously mentioned operators. In our implementation, the size of ngram represents the table to be used. Therefore we do not need to filter the ngrams based on their size.

Inheritance Concept

This concept, present in object-oriented databases, allows users to design schemes based on the relation *parent table* - *child table* [4]. As in object-oriented programming, child table inherits attributes from the parent table and can also extend the data model. This concept brings new possibilities and advantages, for example to query the parent table to obtain the results from all the child tables and also easier to maintain data partitioning. The only CHQL operator that needs the whole dataset for its logic is *matches word*. It looks for tuples that matches the input pattern irrespective of the ngram size.

Due to the fact that PostgreSQL supports this concept, we have used it to create the inheritance relationship between the ngram tables as in Figure 3.1.

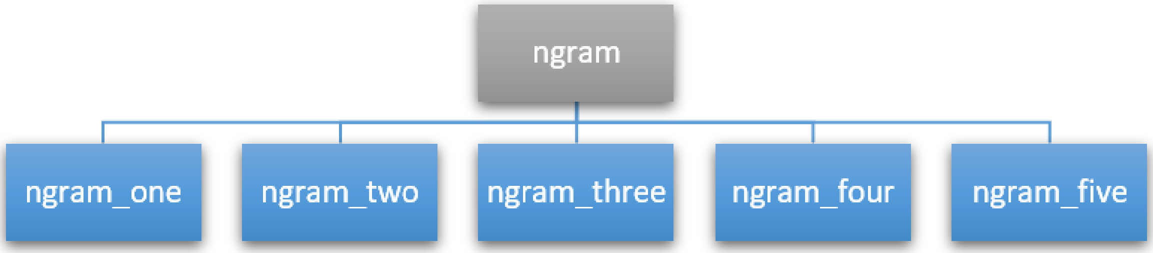


Figure 3.1.: PostgreSQL inheritance concept

On the other hand, Oracle does not implement the inheritance concept between tables, making the querying process of all the tables more complicated. In the Figure 3.2 we observe that we need more SQL SELECT statements in Oracle to accomplish the same goals as in the PostgreSQL implementation of the *matches word* operator. The special data structures used to store the partial results in Oracle are present due to the incapacity of using SQL UNION operator on user defined types.

```
RETURN QUERY SELECT *
FROM ngram
WHERE ngram.word ~* posix_creator(word);
```

(a) PostgreSQL version using inheritance

```
FOR rec IN (
SELECT *
FROM ngram_one
WHERE REGEXP_LIKE(word, posix_word, 'i'))
LOOP
l_ngram := NGRAM_TYPE(rec.word,
rec.part_of_speech, rec.time_series);
result_tbl.extend;
result_tbl(result_tbl.count) := l_ngram;
END LOOP;
...
FOR rec IN (
SELECT *
FROM ngram_five
WHERE REGEXP_LIKE(word, posix_word, 'i'))
LOOP
...
END LOOP;
RETURN result_tbl;
```

(b) Oracle version simplified

Figure 3.2.: Code snippets from *matches word* operator implementation

Pattern Matching

The above mentioned operators *collocation*, *search* and *matches word* have something in common. That is the ability to accept as input parameter either ngram or POSIX regular expressions. To implement this functionality we need more than what SQL LIKE operator offers. That is the reason why in both DBMS we have decided to use the more complex REGEX LIKE operator. To further reduce the complexity of utilizing the CHQL operators by end users, we have implemented a help function that adds to the pattern received as input parameter the logic needed to correctly filter the tuples. For example in *collocation* the user can specify either the target word or a pattern that includes it. With the help function we transform the input into the pattern we need. This works in both cases.

PostgreSQL as well as Oracle implement the POSIX 1003.2 regular expression (RE) standard, but it boost the performance of REGEX by using non-POSIX extensions called advanced regular expressions (AREs). In Figure 3.3 we present the usage of capabilities offered by AREs in comparison to the REs available in Oracle with the clear advantage of using less complicated final REs for filters. The word boundaries are an example of such. The words in our ngrams are separated with spaces, but they are also present at the beginning or the end of a string.

PostgreSQL implements the word boundary escape character as we see in bellow. On the other hand, in Oracle we need to differentiate between word inside the ngram or at the margins.

```
pattern := ’\y(’|| word ||’)\y’;
```

(a) PostgreSQL version using word boundary

```
pattern = ’(^|\s)’|| word ||’(\s|$)’;
```

(b) Oracle version

Figure 3.3.: Code snippets from *posix_creator* help function implementation

Dynamic SQL

There are multiple approaches on how a user can give as an input parameter the table on which an CHQL operator should be executed. For example, the table selection logic could be implemented inside the operator with the help of a conditional statement. In our case this method implies rewriting the SQL SELECT statements in each of the five conditional branches. A better solution in regard of code readability is to use dynamic SQL which allows us to specify the table and further parameters at runtime.

In Figure 3.4a we see the main part of the *search* operator which uses dynamic SQL. The key difference regarding this approach in PostgreSQL is the usage of `pg_typeof()` function. This function returns the type of the input parameter. The trick used in our case is to send an object of the table type we need and then we insert in the dynamic statement the result of the function. This is the only allowed modality to specify the table at runtime, thus protecting the database against *SQL Injection*.

```
 RETURN QUERY EXECUTE format(’
SELECT word, part_of_speech,
time_series[$2 - 1800 : $3 - 1800]
FROM   %s
WHERE  word ~* $1’
,pg_typeof(_tbl_type))
USING  posix_creator(_pattern),
year_from, year_to;
```

(a) PostgreSQL version

```
sql_st := ’SELECT NGRAM_TYPE(word, part_of_speech,
subsequence_array(:1, :2, time_series)) FROM ’
|| tbl_type || ’ WHERE REGEXP_LIKE(word, :3)’;
EXECUTE IMMEDIATE sql_st BULK COLLECT INTO
result_tbl
USING year_from, year_to,
posix_creator(pattern);
RETURN result_tbl;
```

(b) Oracle version

Figure 3.4.: Code snippets from *search* operator implementation

As we see in Figure 3.4b, Oracle allows direct concatenation of the table name in the dynamic query. This implies consequently that the table input parameter is in the form `'ngram_table'`.

Function Nesting

Some of the CHQL operators do not run on the stored tables, but they take as input the resulting set from the other operators in a plug-and-play manner. To achieve this functionality we need to implement them so they can be nested, which translates formally to $f(g(x))$. The mathematical form is correct only when the functions are defined on the right domains.

In our case, the PostgreSQL implementation works as follows: first, the function $g(x)$ is executed to completion and the resulting set is fed row-wise into the outer function f. Moreover, nested CHQL operators are not allowed in the *from* clause. On the other hand, to achive the same functionality in Oracle we use *nested tables* which are a collection data type. Therefore, the working mechanism is: first, function $g(x)$ is executed to completion and the resulting set in form of a nested table is sent into f; afterwards f iterates through the entire input table and calculates the results and stores them in a table. In comparison to PostgreSQL, nested functions can be called only in the *from* clause. In the Figure 3.5 we see the differences in the implementation complexity of the same CHQL operator *has counts larger than* in both DBMSs.

```
SELECT _ngram.*
WHERE counts < ANY (
SELECT _ngram.time_series[s]
FROM (SELECT
generate_subscripts(_ngram.time_series,1)
as s) subscripts
)
```

(a) PostgreSQL version

```
FOR j IN 1..in_table.count LOOP
    rec := in_table(j);
    result_value := 0;
    count_el := rec.time_series.count;
    FOR i IN 1..count_el LOOP
      IF rec.time_series(i)>counts THEN
          result_value := 1;
          EXIT;
      END IF;
    END LOOP;
    IF result_value = 1 THEN
      result_tbl.extend;
      result_tbl(result_tbl.count):=rec;
    END IF;
END LOOP;
RETURN result_tbl;
```

(b) Oracle version

Figure 3.5.: Code snippets from *has counts larger than* operator implementation

Array and String Handling

The majority of the CHQL operators have filters that apply on the time-series attribute, which in our case is an array. PostgreSQL stands out because of its more advanced and native supported array manipulation functions. By contrast Oracle offers minimum functionality for array handling, therefore the implementation of help functions is needed for more complex uses, e.g. returning subseries or iterating through an array directly in an SQL statement. In Figure 3.6 we present an example of how to implement the subseries returning function in Oracle. Firstly, an output array is constructed with the desired subseries length, then proceeding to copy elements from the input array into it. In contrast, the same functionality is obtained in PostgreSQL by writing $array[startPosition : endPosition]$. The advantage is not just syntactical, but also in terms of performance. In this implementation, the database engine choose how to retrieve the subseries. Therefore every update regarding this type of handling is automatically affect the query performance without user intervention.

From all the CHQL operator, one that stands out is *sumup*. In a relational database it can be translated only to an aggregate function. Even though the standard SQL supports neither array nor user defined types, both RDBMSs offer interfaces for custom agregates implementation.

```
create or replace FUNCTION SUBSERIES_ARRAY
(
  YEAR_FROM IN INTEGER
, YEAR_TO IN INTEGER
, IN_ARRAY IN TIMES
) RETURN TIMES AS
    result_arr TIMES := TIMES();
    count_el INTEGER := year_to-year_from + 1;
    pos INTEGER := 0;
BEGIN
    result_arr.extend(count_el);
    FOR i IN year_from..year_to LOOP
        pos := pos + 1;
        result_arr(pos) := in_array(i-1800);
    END LOOP;
    RETURN result_arr;
END SUBSERIES_ARRAY;
```

Figure 3.6.: Oracle implementation of *subseries* help function

We encounter similar difficulties when using string manipulation operations, needed mostly for *collocation*. In this operator, we replace the search pattern with a placeholder, then we split the resulting ngram into an array. Oracle and PostgreSQL are comparable in what they offer in this sense, the only difference we encountered being the *split_to_array* function. However, there is no equivalent in Oracle and we have implemented it as in Figure 3.7.

```
create or replace FUNCTION TRANSFORM_POS
(
  input_text IN VARCHAR2
) RETURN POS AS
    l_string VARCHAR2(32767) := input_text || ',';
    l_comma_index PLS_INTEGER;
    l_index PLS_INTEGER := 1;
    l_count PLS_INTEGER := 1;
    out_array POS := POS();
BEGIN
  LOOP
    l_comma_index := INSTR(l_string, ',', l_index);
    EXIT WHEN l_comma_index = 0;
    out_array.extend();
    out_array(l_count) := SUBSTR(l_string, l_index, l_comma_index - l_index);
    l_count := l_count + 1;
    l_index := l_comma_index + 1;
  END LOOP;
  RETURN out_array;
END TRANSFORM_POS;
```

Figure 3.7.: Oracle implementation of split to string help function

kNN Operator

The CHQL operator *kNN* search for the k ngrams that have most similar timeseries evolution as the target word. Furthermore the search space could be restricted to the result set of another operator such as *search* or *collocation*. We have discussed the importance of *kNN* operator for conceptual history in section 2.1.
Due to the fact that *kNN* operator is itself a problem field, it has been already researched on its own. As stated, the operator is nothing else than a specialization of the *k-nearest Neighbours* fundamental problem in computer science. In the CHQL case, the elements are timeseries and the similarity metrics are given by the *dynamic time warping* algorithm.
The optimization of *kNN* queries for the special case of ngrams has already been tackled by itself in a master thesis [1]. Because our work is not focused on providing optimization for this operator, we decided not to carry out with it.

3.2. Benchmark

In this final section, we go over the benchmark implementation in PostgreSQL and Oracle. This set of queries is conceived to reproduce the normal usage of an end user interested in the field of conceptual history [8]. It consist of 68 information needs addressed to the DBMS with the help of the implemented CHQL operators. In Table 3.2 we present their distribution in four different categories according to the main question asked in each case. Next, we regard each query group separately and compare their translation from the abstract concept to the corresponding implementation in PostgreSQL and Oracle. Moreover, we divide each group in several ranges, each with a slightly different implementation.

Group	Range	#Queries
Matches Word	1.1.1-1.1.5 1.2.1-1.2.5	10
Search Word	2.1.1-2.1.5 2.2.1-2.2.4	9
Collocation	3.1.1-3.1.5 3.2.1-3.2.5 3.3.1-3.3.5 3.4.1-3.4.6 3.5.1-3.5.5 3.6.1-3.6.5 3.7.1-3.7.5	36
Polysemy	4.1.1-4.1.5 4.2.1-4.2.4 4.3.1-4.3.4	13

Table 3.2.: Overview of the query groups and their scope

3.2.1. Matches Word Query Group

This group contains general inquires that return the frequency of an ngram, searching for it in all the tables.

Range 1.1.*

The queries in this range use the subsequence filter to limit the years in the time series.

- CHQL:

  ```
  matches word(^Computermaus$)subsequence(1801,2000)
  ```

- PostgreSQL:

  ```
  SELECT (subsequence(1801, 2000, matches_word('^Computermaus$'))).*;
  ```

- Oracle:

  ```
  SELECT *
  FROM subsequence(1801, 2000, matches_word('^Computermaus$'));
  ```

Range 1.2.*

The queries in this range use patterns that include wildcards.

- CHQL:

```
matches word(^Computer \w+$)
```

- PostgreSQL:

```
SELECT (matches_word('^Computer \w+$')).*;
```

- Oracle:

```
SELECT *
FROM matches_word('^Computer [[:alnum:]_]+$');
```

3.2.2. Search Word Query Group

This group is a specialization of the precedent one, the difference being that the subsqeuence filter and the table to be searched are input parameters.

Range 2.1.*

The queries in this range show usage of different tables as input parameter.

- CHQL:

```
search(Computer \w+,1801,1950,5G)
```

- PostgreSQL:

```
SELECT (search_word('Computer \w+',1801,1950,NULL::ngram_five)).*;
```

- Oracle:

```
SELECT *
FROM search_word('Computer [[:alnum:]_]+',1801,1950,'ngram_five');
```

Range 2.2.*

The queries in this range use the selection filters on frequency values.

- CHQL:

```
search(Emanzipation,1801,2000,5G) has_counts_larger_than(15)
```

- PostgreSQL:

```
SELECT (has_counts_larger_than(15,
search_word('Emanzipation',1801,2000,NULL::ngram_five))).*;
```

- Oracle:

```
SELECT *
FROM has_counts_larger_than(15,
search_word('Emanzipation',1801,2000,'ngram_five'));
```

3.2.3. Collocation Query Group

This group deals with the queries regarding the surrounding words for a pattern. An input parameter is also the function type to be used on the results : Frequency, Closeness or Cohesion.

Range 3.1.*

The queries in this range represent the standard version of the group.

- CHQL:

```
collocation(Maus,FREQ,1801,2000,2G)
```

- PostgreSQL:

```
SELECT (collocation('Maus','FREQ',1801,2000,NULL::ngram_two)).*;
```

- Oracle:

```
SELECT *
FROM collocation('Maus','FREQ',1801,2000,'ngram_two');
```

Range 3.2.*

The queries in this range also add a selection filter on the frequency.

- CHQL:

```
collocation(Emanzipation,FREQ,1801,2000,5G)
            has_counts_larger_than(7)
```

- PostgreSQL:

```
SELECT (has_counts_larger_than(7,collocation('Emanzipation','FREQ',
        1801,2000, NULL::ngram_five))).*;
```

- Oracle:

```
SELECT *
FROM has_counts_larger_than(7,
     collocation('Emanzipation','FREQ',1801,2000,'ngram_five'));
```

Range 3.3.*

The queries in this range use a selection filter on part of speech.

- CHQL:

```
collocation(Emanzipation,FREQ,1801,2000,5G)
                is_part_of _speech(NOUN)
```

- PostgreSQL:

```
SELECT (pos_filter('NOUN',
        collocation('Emanzipation','FREQ',1801,2000,
                    NULL::ngram_five))).*;
```

- Oracle:

```
SELECT *
FROM pos_filter('NOUN',
     collocation('Emanzipation','FREQ',1801,2000,'ngram_five'));
```

Range 3.4.*

The queries in this range explore different functions as input parameter.

- CHQL:

```
collocation(Maus,COHESION,1801,2000,3G)
```

- PostgreSQL:

```
SELECT (collocation('Maus','COHESION',
        1801,2000,NULL::ngram_three)).*;
```

- Oracle:

```
SELECT *
FROM collocation('Maus','COHESION',1801,2000,'ngram_three');
```

Range 3.5.*

The queries in this range search for a pattern inside the surrounding words set.

- CHQL:

```
search(Frau,1801,2000,
       collocation(Emanzipation,FREQ,1801,2000,2G))
```

- PostgreSQL:

```
SELECT ROW(result.word, result.part_of_speech,
           result.time_series[1801-1800:2000-1800])::ngram
FROM collocation('Emanzipation','FREQ',1801,2000,
                 NULL::ngram_two) result
WHERE result.word ~* posix_creator('Frau');
```

- Oracle:

```
SELECT *
FROM collocation('Emanzipation','FREQ',1801,2000,'ngram_two')
WHERE REGEXP_LIKE(word,posix_creator('Frau'),'i');
```

Range 3.6.*

The queries in this range use the *conflate* operator on the result set.

- CHQL:

```
conflate(collocation(Maus,FREQ,1500,2000,5G),SUM)
```

- PostgreSQL:

```
SELECT (conflate(collocation('Maus','FREQ',1801,2000,
                             NULL::ngram_five),'SUM')).*;
```

- Oracle:

```
SELECT *
FROM conflate(collocation('Maus','FREQ',1801,2000,
                          'ngram_five'),'SUM');
```

Range 3.7.*

The queries in this range show the *topicgrouping* operator implementation.

- CHQL:

```
collocation(Emanzipation,FREQ,1801,2000,5G)topicgrouping
```

- PostgreSQL:

```
WITH result AS(
 SELECT (collocation('Emanzipation','FREQ',1801,2000,
                     NULL::ngram_five)).*)
```

```
SELECT category.topic,result.part_of_speech,
       sumup(result.time_series)
FROM result
JOIN category ON result.word = category.word
GROUP BY category.topic, result.part_of_speech;
```

- Oracle:

```
WITH result AS(
 SELECT *
    FROM collocation('Emanzipation','FREQ',1801,2000,
                     'ngram_five'))
SELECT category.topic,sumupagg(result.time_series)
FROM result
JOIN category ON result.word = category.word
GROUP BY category.topic;
```

3.2.4. Polysemy Query Group

This group handles the questions regarding the polysemy and synonyms. The queries in ranges 4.2.* and 4.3.* have not been implemented in Oracle. The reason behind our decision is the inability of using SQL statements equivalent to those from PostgreSQL on UDTs. An implementation of those statements in PL/SQL implies a non-dynamically approach on the algorithms used. Thus it presents no interests for our performance evaluation.

Range 4.1.*

The queries in this range return the frequency from the union of the surrounding words of more search patterns.

- CHQL:

```
collocation(Hund|Katze|Maus,FREQ,1801,2000,4G)
```

- PostgreSQL:

```
SELECT (collocation('Hund|Katze|Maus','FREQ',1801,2000,
                    NULL::ngram_four)).*;
```

- Oracle:

```
SELECT *
FROM collocation('(Hund|Katze|Maus)','FREQ',1801,2000,
                 'ngram_four');
```

Range 4.2.*

The queries in this range show the change between word collocations.

- CHQL:

```
change(collocation(Emanzipation,FREQ,1801,1900,5G),
       collocation(Emanzipation,FREQ,1901,2000,5G),ABS)
```

- PostgreSQL:

```
WITH result1 AS (
 SELECT (collocation('Emanzipation','FREQ',1801,1900,
                     NULL::ngram_five)).*),
result2 AS(
 SELECT (collocation('Emanzipation','FREQ',1901,2000,
                     NULL::ngram_five)).*)
SELECT result1.word,result1.part_of_speech,
       change_array(result1.time_series,result2.time_series,'ABS')
FROM result1
JOIN result2
ON result1.word = result2.word
AND result1.part_of_speech[1] = result2.part_of_speech[1];
```

- Oracle: Not Implemented

Range 4.3.*

The queries in this range deal with set operations.

- CHQL:

```
test(collocation(Bank,FREQ,1801,2000,5G),
     collocation(Sitzbank,FREQ,1801,2000,5G),MINUS)
```

- PostgreSQL:

```
SELECT result1.word,result1.part_of_speech
FROM collocation('Bank','FREQ',1801,2000,NULL::ngram_five) result1
EXCEPT
SELECT result2.word, result2.part_of_speech
FROM collocation('Sitzbank','FREQ',1801,2000,
                 NULL::ngram_five) result2;
```

- Oracle: Not Implemented

4. Optimization

4.1. Benchmark

In the first part of this section we go over the environments on which the benchmark presented in section 3.2 has run, the data used and how it was partitioned in the tables. At the end we present the initial timings and their meaning for further optimization described in the next sections.

4.1.1. Technical Data

The performance of the CHQL operators implemented in relational databases was tested on the systems described in Table 4.1 and Table 4.2. We have decided to run the queries in a single thread for two reasons. Firstly, the servers have different architecture. Secondly, it is not essential the processing time, but rather the query plans executed for our implementation.

	PostgreSQL Server
OS	Ubuntu 16.04.6 LTS
CPU	Intel(R) Xeon(R) E5-2630 v3
CORES	2.40 GHz 8c/16t
RAM	4x 32GB PC4-17000R
HDD MEMORY	4x 4TB (RAID5, Software)
DBMS VERSION	11.4 (Ubuntu 11.4-1.pgdg16.04+1)

Table 4.1.: Technical specification of PostgreSQL server

	Oracle Server
OS	Oracle Linux Server release 7.4
CPU	AMD Opteron (tm) Processor 4234
CORES	3.1 GHz 6c
RAM	4x 16GB PC3-12800R
HDD MEMORY	8x 1TB RAID6 Symbios Logic MegaRAID SAS 2108
DBMS VERSION	12.2.0

Table 4.2.: Technical specification of Oracle server

Dataset

The temporal text corpus we use for our data set is the one for the German language. It consists of ngrams of maximum size five. Additionaly it has for each ngram the part of speech tagging and the time series. The latter represents the usage frequency of the corresponding ngram between the years 1500 and 2008. We have chosen the Google Ngram Corpus version 20120701. To make it compatible with our data model, we preprocess the corpus and group the equal ngrams, thus resulting the timeseries array.

To construct our dataset used for performance evaluation we have uploaded the whole preprocessed corpus in our partitioned tables. About 10% of the tuples from an ngram table have been aleatory selected. The timeseries array has been restricted to the frequencies in time interval 1800-2009. In Table 4.3 we describe how many tuples are in every table used in the benchmark process and their size on the disk. Even if the data model implemented in PostgreSQL and Oracle is logically the same, the first one takes more than twice as much disk space as the latter to store the tuples. This fact could be attributed to the internal implementation of arrays.

Table	Tuples	Size (MB)	
		PostgreSQL	Oracle
ngram_one	9,237,177	18,432	5,858
ngram_two	14,706,800	28,672	9,292
ngram_three	31,485,364	61,440	36,664
ngram_four	25,327,612	49,152	16,007
ngram_five	13,638,730	26,624	8,611
category	24,201	1	0.75
sentiment	1,694	0.07	0.0625
Total	94,421,578	184,321	76,533

Table 4.3.: Data distributions over the tables

4.1.2. Non-optimized Results

In order to have an accurate overview of the implementation performance on both systems, we have tested them using neither indexes nor any form of parallel processing. In Figure 4.1 we see the comparison between the two systems for every query in the benchmark. A more detailed view of the results is in section A.3.

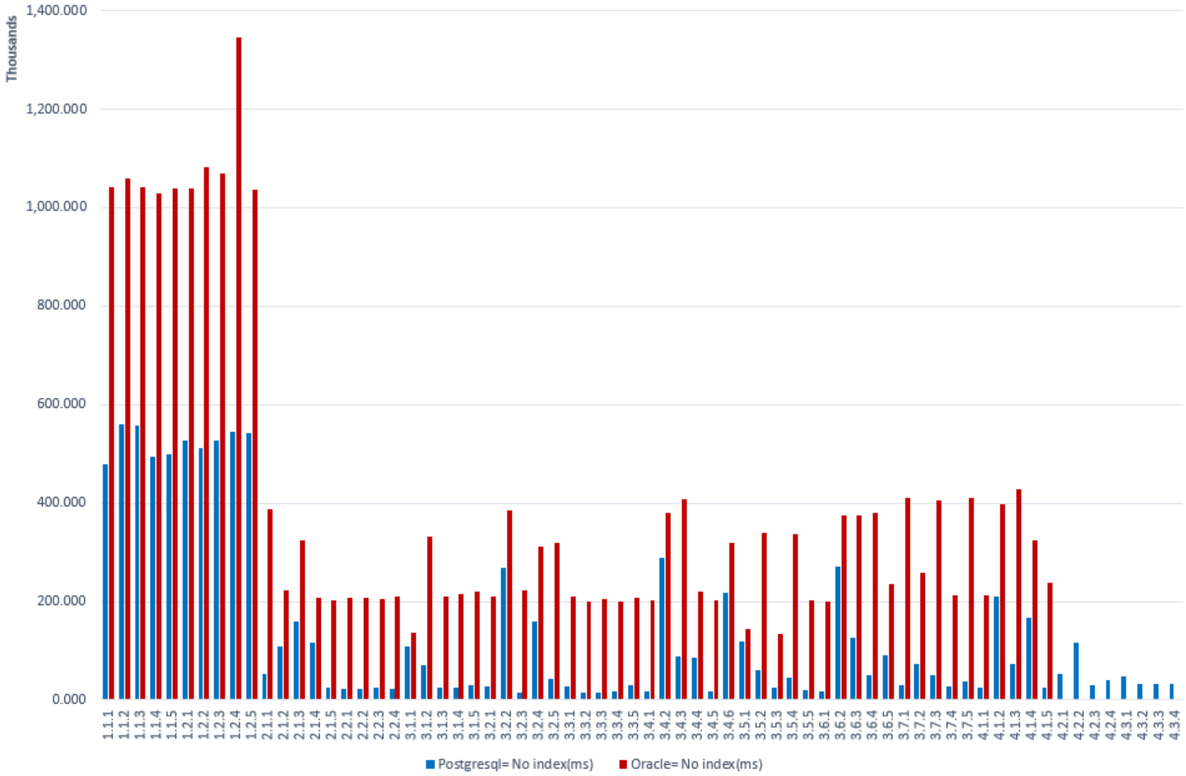


Figure 4.1.: Comparison between Oracle and PostgreSQL without indexes

Due to the fact that the RDBMSs are running on two different servers, we have looked at the query plans. This has showed us that for the same plan Oracle took on average two times more to process the query. If we combine these results with the fact that Oracle requires less than half of disk memory to store the dataset, we obtain that the Oracle server described in Table 4.2 performs slower by an order of approximately four. This is the reason why we compare further the PostgreSQL timings with only the quarter of the initial time needed in Oracle. The results can be seen in Figure 4.2.

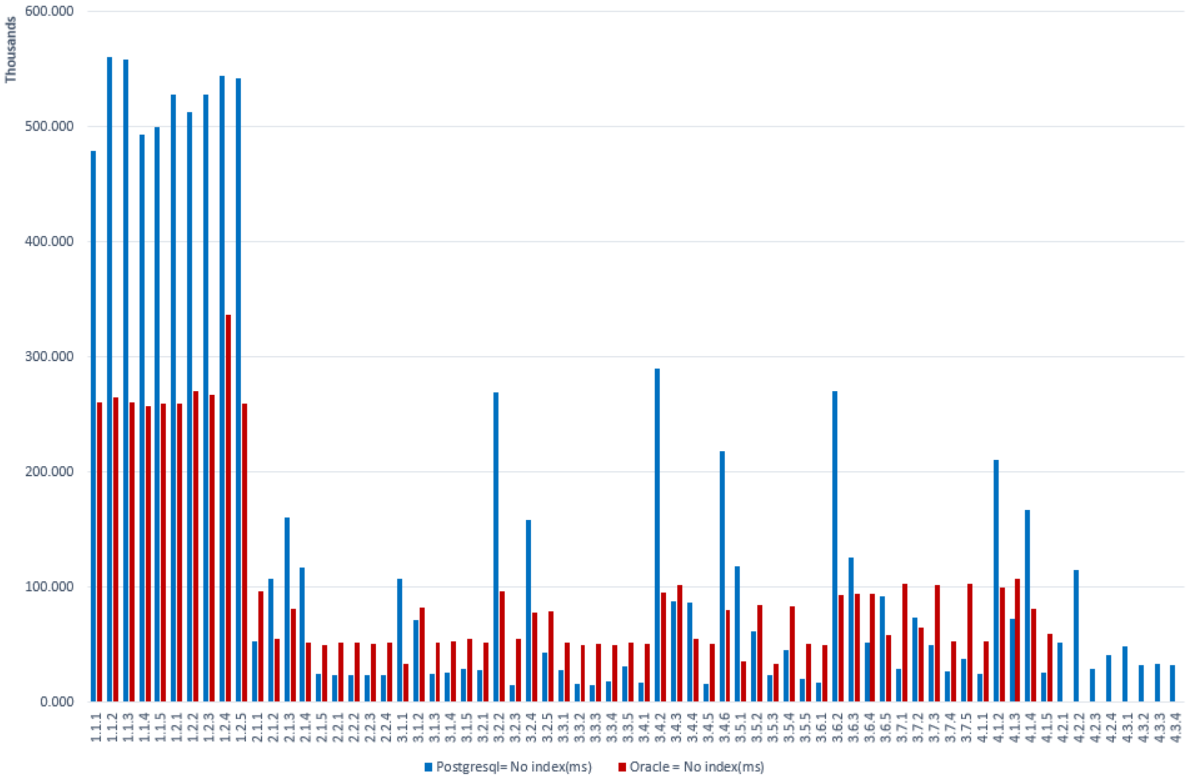


Figure 4.2.: Adjusted comparison between Oracle and PostgreSQL without indexes

The most time-consuming queries are the first ten because they execute a full table scan on all of the ngram tables, the bottleneck is the read from the disk operation. Another interesting remark is the almost constant time across a range of similar queries in Oracle, the number of tuples returned having moderate to no influence on the performance metrics. The same could not be said about the PostgreSQL implementation, where there are significant jumps in timings across the same query groups as before. We have looked in particular on the execution plans of the queries 3.2.2, 3.4.2 and 3.6.2 where the implementation of the CHQL operators in Oracle take more than half of the time for the respective query in PostgreSQL. The execution plans have not been conclusive as why those queries are special cases. For a better understanding of the performance metrics of individual CHQL operators, we look at the average time per query group and also the average time per query across the whole benchmark. The query groups are described in section 3.2. As we have seen in the previous graph, the operator which takes the longest is the *matches word* in both implementations. The *collocation* operator uses in the implementation a function call to the *search* operator in order to retrieve the ngrams that match the search pattern. It is useful to know this aspect, because the operations on the data returned by the *search* operator are mostly procedural. This provides an insight on how long are taking the procedural operations that cannot be optimized.

In Figure 4.3 we analyze first the PostgreSQL averages per query group. We see that the queries in the *Collocation* group take more time to finalize in comparison with the ones from *Polysemy*, which is surprising given the fact that the latter uses also the *collocation* operator. The observations for the PostgreSQL environment are not necessary valid for Oracle. Here the *Polysemy* group takes longer on average than the others, but this could be also attributed to the missing queries which couldn't be implemented (see section 3.2). Analyzing the averages in parallel we observe further that not only the disk reads are a disadvantage in PostgreSQL, but also the procedural part of the implementation is slower. The difference between the *Collocation* and *Search* query groups shows that the PL/SQL takes on average 30% less time to execute the same tasks in comparison to the PL/pgSQL. This remark reefers to the difference between the time needed from the point where the results of *search* are retrieved and the point when *collocation* finishes working on them. We can state that based on the difference of the two query groups due to the fact that they use approximately the same target words. We have decided to look for ways to optimize the CHQL operators only on the PostgreSQL server. Firstly, the fact that both query optimizers execute the same plans. We do not see a reasons to double the work. Secondly, in practice the timings were better on the PostgreSQL server.

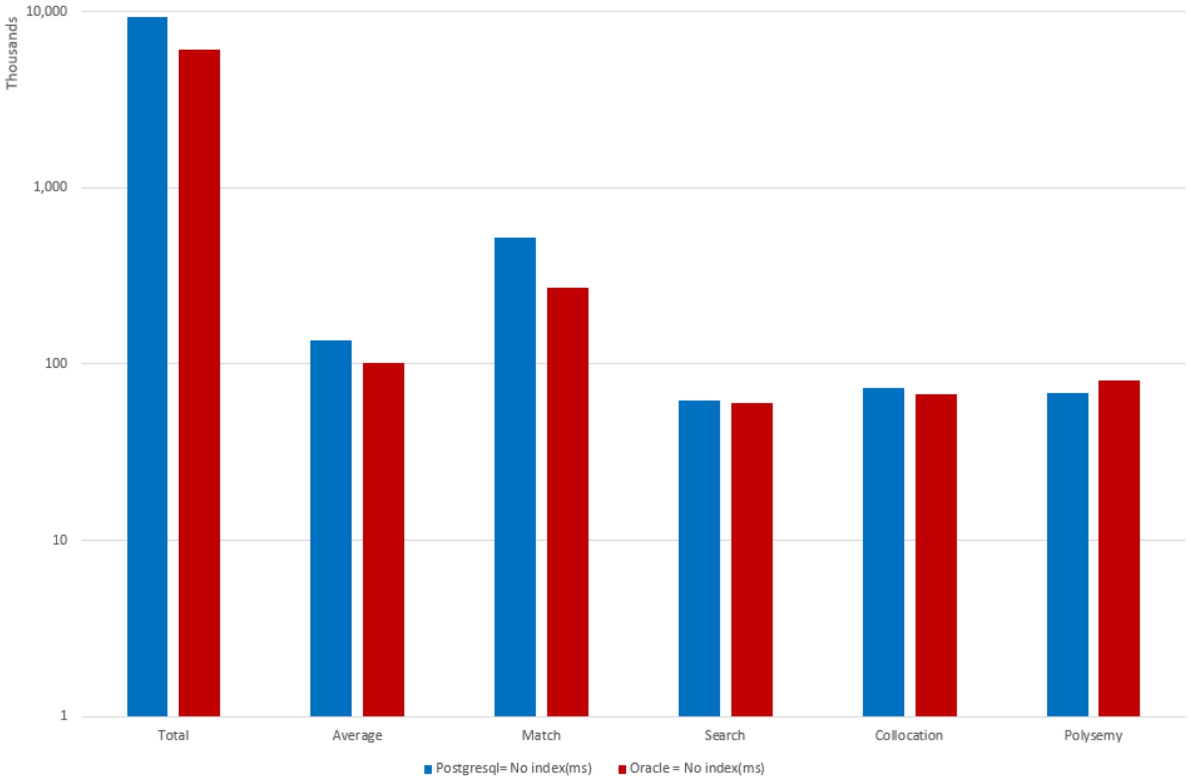


Figure 4.3.: Average time per query group. Comparison between Oracle and PostgreSQL

4.2. General Optimization

When we talk about optimizing a database solution, the first step is always to see if we can improve the implementation with general strategies. In the first part of this subsection we go over the different indexing possibilities and their outcome. In the second part we discuss one of the most important factor in efficient database solutions, statistics.

4.2.1. Indexes

Index strategies are very popular in database performance optimization because they can speed up searches on tables if they are chosen according to the task. Every one of them is based on a different data structure or implementation, this leading to distinct maintenance or creation cost. We have considered two indexes for our implementation, B-Tree and Trigram.

B-Tree Indexes are the most used and they come as standard choice in PostgreSQL. They are based on a self-balancing tree data structure and offers logarithmic time per operation if used correctly [4]. In Figure 4.4 we observe that this index type has obtained worse performance metrics across the whole range of queries. This can be attributed to the fact that they are not suited for pattern matching with a leading wildcard.

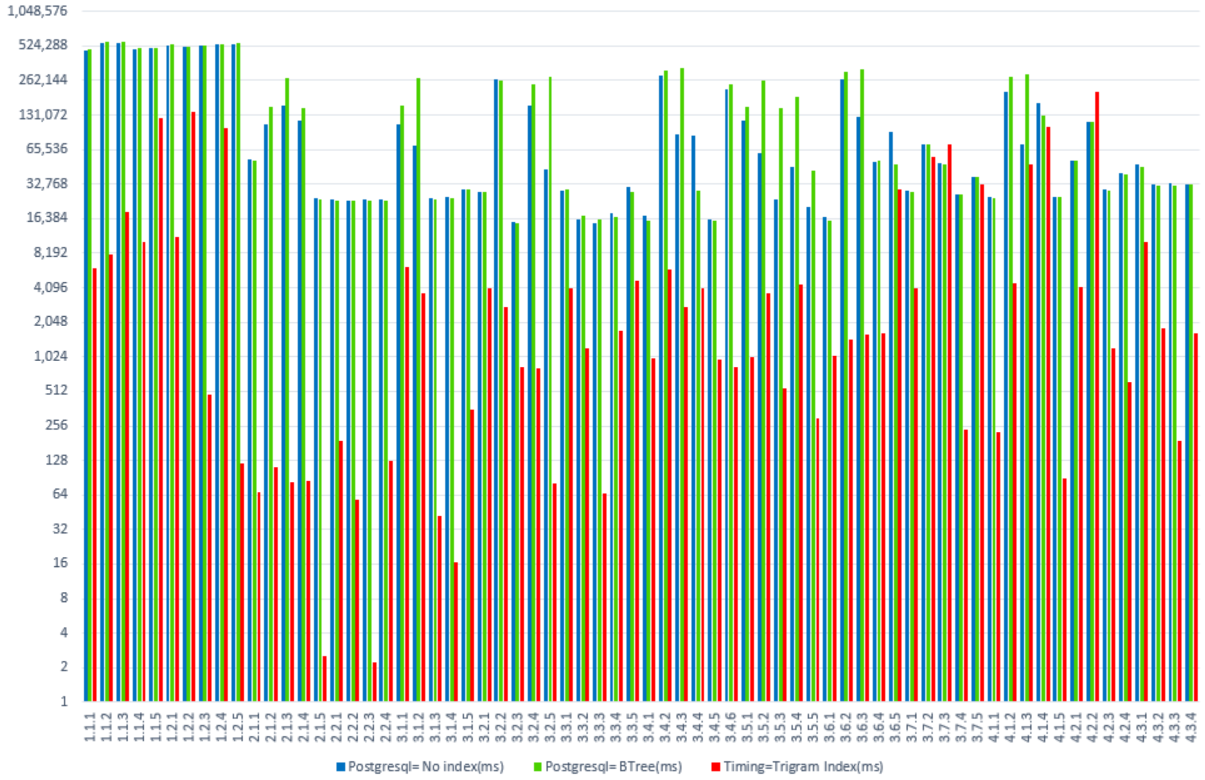


Figure 4.4.: Comparison between the index strategies

Because the pattern matching is a key factor in the functionality of the CHQL operators, we have studied indexes that could speed up this kind of scans. The one that fulfills this requirement is the *Trigram* index. They are supported in PostgreSQL and they can be implemented with the help of the *GIN* or *GiST* index interfaces. Their mechanism is to split up a text in trigrams, i.e. words of only three letters [4]. One of the big advantages is that they work well with all the implemented pattern matching functions. In our case *REGEXP LIKE*. Therefore, queries do not require any modification.

The Figure 4.4 and Figure 4.5 allow us to see how these two index strategies performed in comparison to the implementation without any index. The standard *B-Tree* indexes have showed no usability for the purpose of speeding up any query group. On the other hand, the *trigram* indexes have obtained superior performance across all the query groups. The overall average improvement is of roughly order ten. Thus, they remain part of the optimization.

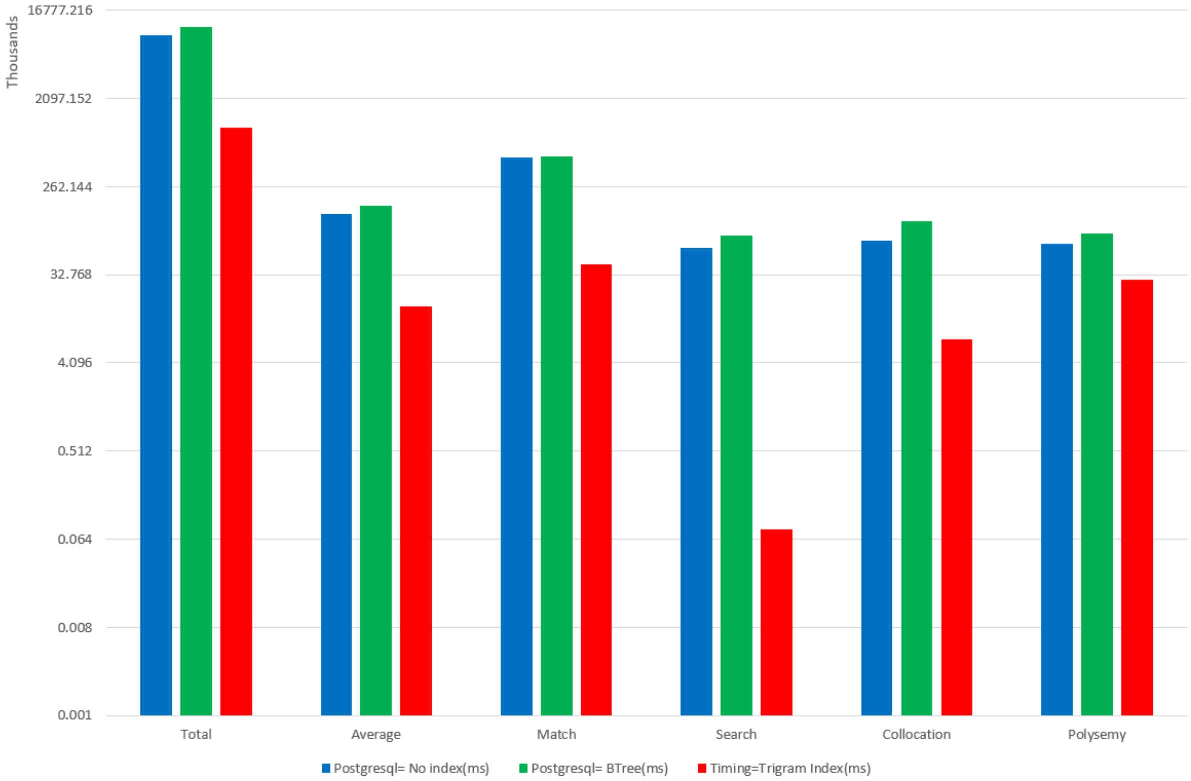
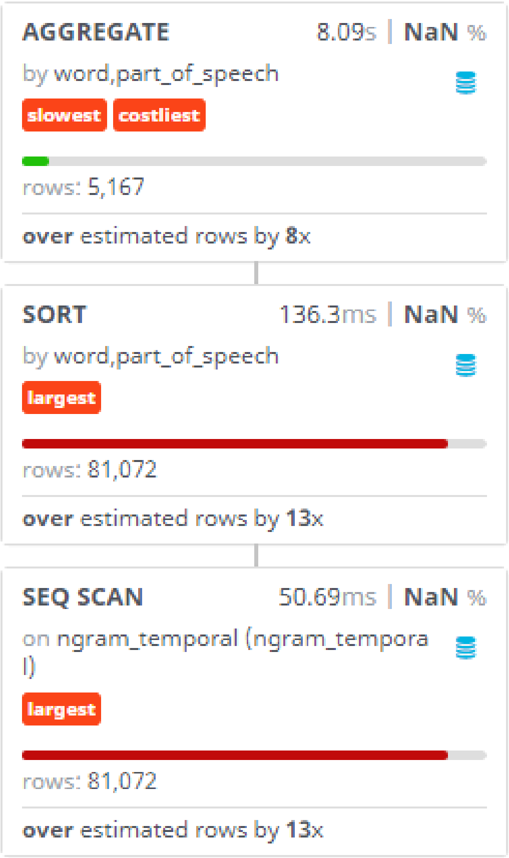


Figure 4.5.: Comparison between average times per query group with different index strategies

4.2.2. Statistics

In section 2.2 we have explained why good statistics are one of the key factors in developing a perfomant database solution. The CHQL operators are using the original tables stored on the disk only to perform searches, the rest of operation being performed on the result retrieved. We take the *collocation* operator and analyze a part of it's execution plan to show the importance of statistics. In Figure 4.6 we observe that the query optimizer has chosen to execute a sort aggregate on the temporal table inside the *collocation* function, due to the fact that it had over estimated the number of the rows inside the temporal table.



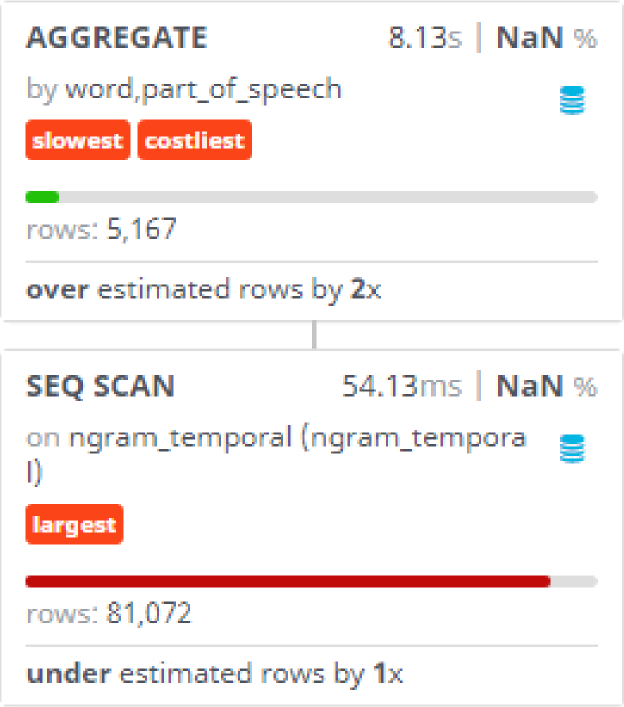
AGGREGATE 8.09s | NaN %
by word,part_of_speech
slowest costliest
rows: 5,167
over estimated rows by 8x

SORT 136.3ms | NaN %
by word,part_of_speech
largest
rows: 81,072
over estimated rows by 13x

SEQ SCAN 50.69ms | NaN %
on ngram_temporal (ngram_tempora
l)
largest
rows: 81,072
over estimated rows by 13x

Figure 4.6.: Execution plan for a sort aggregate operation on temporal table

The problem with estimating the wrong number of rows could lead to inferior execution plans, i.e. slow queries. Fortunately PostgreSQL allows user to execute *ANALYZE* on temporal tables inside an user-defined-function. This gather statistics about the table that is called onto and as we can observe in Figure 4.7 the engine now predicts correctly and chooses an hash aggregate which is for smaller tables faster. Additionally there is also an 130 millisecond improvement over the previous execution plan for this aggregate.



AGGREGATE 8.13s | NaN %

by word,part_of_speech

slowest costliest

rows: 5,167

over estimated rows by 2x

SEQ SCAN 54.13ms | NaN %

on ngram_temporal (ngram_tempora l)

largest

rows: 81,072

under estimated rows by 1x

Figure 4.7.: Execution plan for a hash aggregate operation on temporal table

4.3. In Depth Exploration of Execution Plans

In this section, we examine slow queries and try to identify possible hints for the query optimizer. Where that is not possible, we make an effort to understand why it can not be optimized.

4.3.1. Variables

After implementing the *trigram* indexes, we have observed that the time needed to execute the queries in the *Match* group is same as before. The solution in case of slow queries is to retrieve the execution plan and to identify the problem. This further shows the estimated values for rows and cost. Thus, this allows us to understand the decision process. Therefore, we have looked at the execution plan of one of the queries in this group. The function call to `posix_creator()` is repeated for every tuple in each ngram table. This leads to the indexes not being used and time wasted retrieving the same result from the help function. It has been clear that the implementation presented in Figure 3.2a is the not optimal. To resolve the issue, we use the PL/pgSQL language. This allows us to store in a variable the result of the `posix_creator()`. We have obtained the implementation displayed in Figure 4.8 for the *matches word* operator. This fix resolved both problems identified in the execution plan. In conclusion, it reduces the times with around three minutes for the benchmark testing without any index or with *b-tree* index, and with more than five for *trigram* indexes. The timings presented in subsection 4.1.2 are adjusted with these findings.

```
CREATE OR REPLACE FUNCTION matches_word(word text)
RETURNS SETOF ngram
AS $$
DECLARE
 pattern text := posix_creator($1);
BEGIN
   RETURN QUERY SELECT *
 FROM ngram
 WHERE ngram.word ~* pattern;
END
$$ LANGUAGE plpgsql;
```

Figure 4.8.: Improved implementation of *matches word* with variables

4.3.2. Join Operation

After we have analyzed the timings of each query, the one we list in Figure 4.9 is of great interest to us. The reason behind this affirmation is that the timing in the *trigram* indexed run is close to the implementation without any indexes. Thus, inferring that the query might process data which can be discarded earlier or the chosen join algorithm is not optimal.

```
WITH result AS(
SELECT (collocation('Krieg', 'FREQ', 1801, 2000, NULL::ngram_five)).*)
SELECT category.topic, result.part_of_speech, sumup(result.time_series)
FROM result
JOIN category ON result.word = category.word
GROUP BY category.topic, result.part_of_speech;
```

Figure 4.9.: Query 3.7.2 from the *Collocation* group

Inspecting the query execution plan, we have seen that nothing can be further optimized inside the *collocation* operator. The motive is that the costliest operations inside it are not in SQL. In regard to our second supposition, we spot that the join algorithm used for the topic grouping is *merge join*. Based on the fact that the number of rows returned by *collocation* operator is small enough, we explore the possibility of using a *hash join*. In comparison to *merge join* this sould be faster. The first step we have taken is to modify the `work_mem` from `4MB` to `64MB`. The `work_mem` is responsible to limit the memory that a hash table or a in-memory sort operation could use. After this limit, the operation is required to write in a temporary table on the disk [4]. We obtain the execution plan from Figure 4.10 for the query listed above.

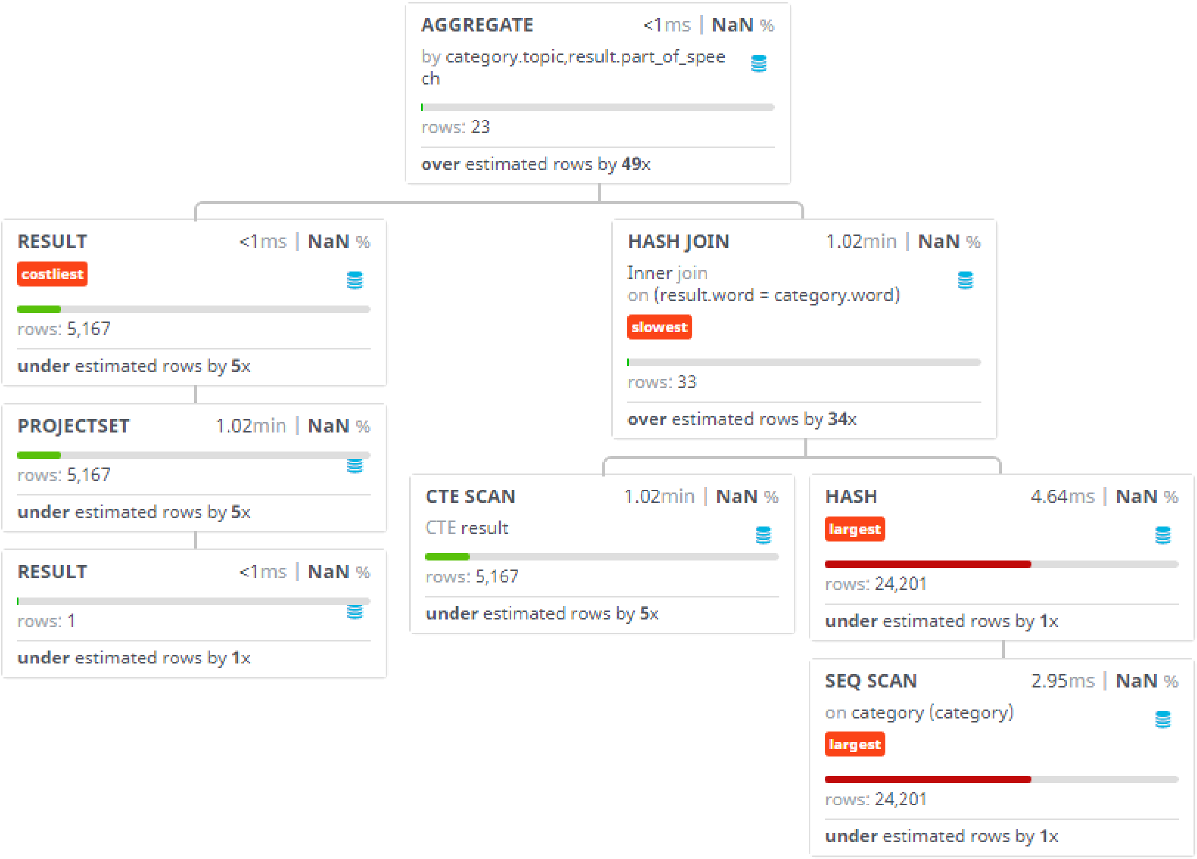

AGGREGATE <1ms | NaN %
by category.topic,result.part_of_spee ch
rows: 23
over estimated rows by 49x
RESULT <1ms | NaN %
costliest
rows: 5,167
under estimated rows by 5x
PROJECTSET 1.02min | NaN %
rows: 5,167
under estimated rows by 5x
RESULT <1ms | NaN %
rows: 1
under estimated rows by 1x
HASH JOIN 1.02min | NaN %
Inner join
on (result.word = category.word)
slowest
rows: 33
over estimated rows by 34x
CTE SCAN 1.02min | NaN %
CTE result
rows: 5,167
under estimated rows by 5x
HASH 4.64ms | NaN %
largest
rows: 24,201
under estimated rows by 1x
SEQ SCAN 2.95ms | NaN %
on category (category)
largest
rows: 24,201
under estimated rows by 1x

Figure 4.10.: Execution plan for topic grouping query with hash join on Category Table

We notice that the algorithm builds in this case a hash table for the larger relationship, respectively the *category* table. Firstly, we have tried to change the ordering of tables, as the *hash join* in PostgreSQL loads the right relation in the hash table. This attempt proves to be unsuccessful even with the engine hints listed below:

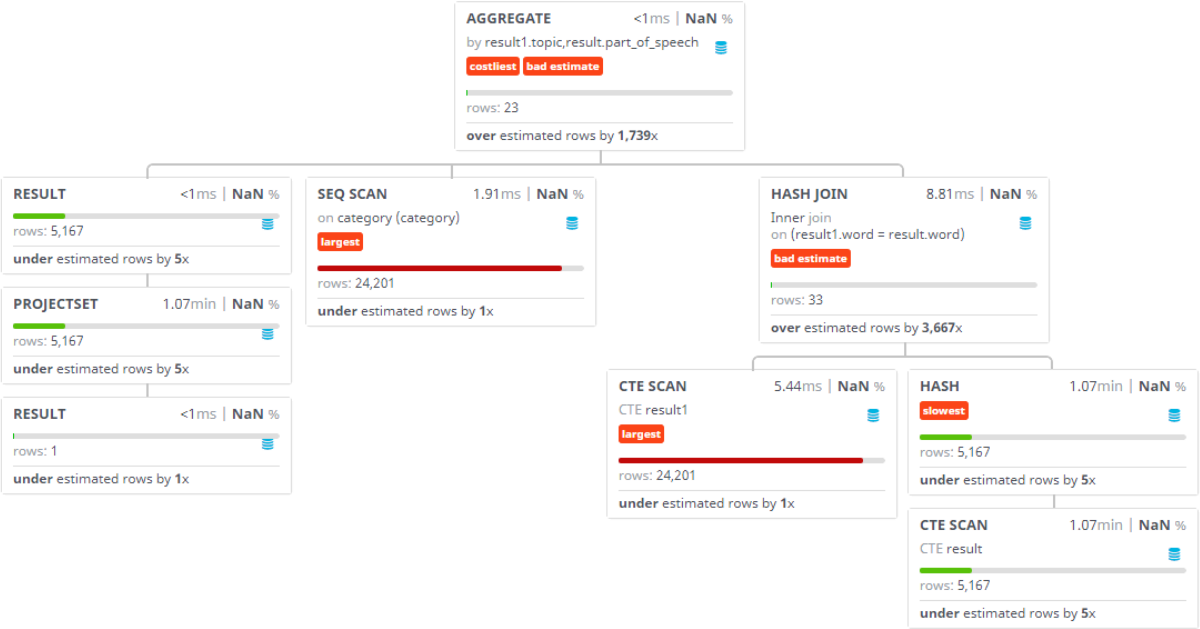
- `SET join_collapse_limit = 1`
- `SET from_collapse_limit = 1`

Both hints constrain the ability of the query planner to modify the *join* and *from* table ordering. Furthermore, to take advantage of these hints, we also change the query to the version below:

```
WITH RESULT AS (
SELECT (COLLOCATION('KRIEG','FREQ', 1801, 2000,NULL::NGRAM_FIVE)).*),
RESULT1 AS (
SELECT * FROM CATEGORY)
SELECT RESULT1.TOPIC, RESULT.PART_OF_SPEECH,SUMUP(RESULT.TIME_SERIES)
FROM RESULT1
JOIN RESULT ON RESULT.WORD = RESULT1.WORD
GROUP BY RESULT1.TOPIC, RESULT.PART_OF_SPEECH;
```

Figure 4.11.: Modified version of the query 3.7.2. Here both table used in join operation come from *Common Table Expressions (CTEs)*

This query with both *collocation* operator result set and the *category* table as *common-table-expressions* (CTEs) is forcing the query planner to hash the smaller table as we see in Figure 4.12.



AGGREGATE <1ms | NaN %
by result1.topic,result.part_of_speech
costliest bad estimate
rows: 23
over estimated rows by 1,739x

RESULT <1ms | NaN %
rows: 5,167
under estimated rows by 5x

PROJECTSET 1.07min | NaN %
rows: 5,167
under estimated rows by 5x

RESULT <1ms | NaN %
rows: 1
under estimated rows by 1x

SEQ SCAN 1.91ms | NaN %
on category (category)
largest
rows: 24,201
under estimated rows by 1x

HASH JOIN 8.81ms | NaN %
Inner join
on (result1.word = result.word)
bad estimate
rows: 33
over estimated rows by 3,667x

CTE SCAN 5.44ms | NaN %
CTE result1
largest
rows: 24,201
under estimated rows by 1x

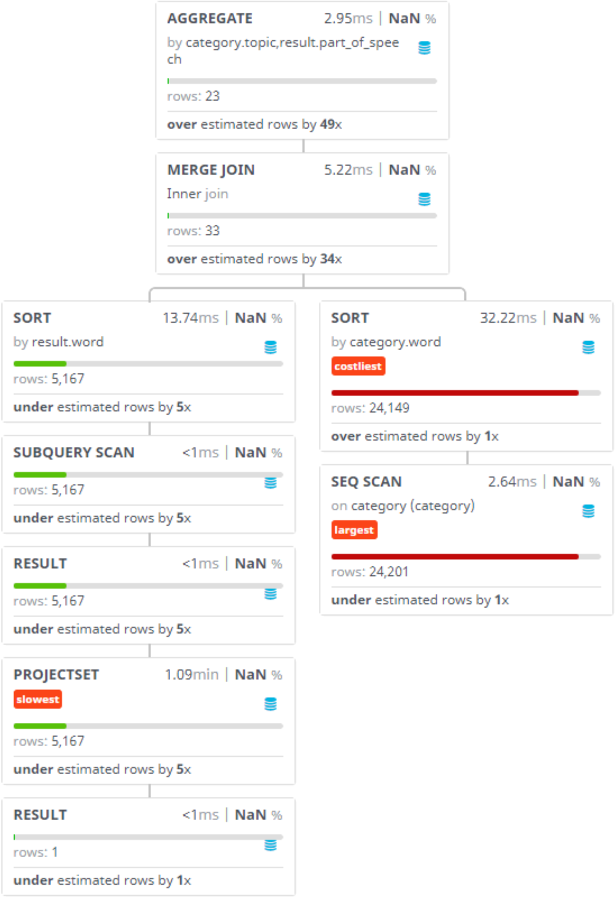
HASH 1.07min | NaN %
slowest
rows: 5,167
under estimated rows by 5x

CTE SCAN 1.07min | NaN %
CTE result
rows: 5,167
under estimated rows by 5x

Figure 4.12.: Execution plan for topic grouping query with hash join on collocation results in CTE

Due to our success in changing the ordering, we have now the reason why the planner did not choose as standard to hash the result of the *collocation* operator. Even if the statistics for the CTE's results are equal to the ones in the previous plan, the engine overestimates the resulting rows for the *hash join* and also for the *aggregate* by more than 1.500x. The reason behind this overestimation is the inability of the optimizer engine to store statistics during the run. As we have discussed in section 2.2, there is no possibility for the optimizer to adapt the plan at runtime.

In a last attempt to obtain a performance gain for this query, we change once again the ordering of the tables. In this case, both tables are in the *from* clause. This statement turns into to the plan in Figure 4.13 where we notice that the statistics are back to normal but once again the engine chooses a *merge join*. This also brings no difference in timings.



AGGREGATE 2.95ms | NaN %
by category.topic,result.part_of_speech
rows: 23
over estimated rows by 49x

MERGE JOIN 5.22ms | NaN %
Inner join
rows: 33
over estimated rows by 34x

SORT 13.74ms | NaN %
by result.word
rows: 5,167
under estimated rows by 5x

SUBQUERY SCAN <1ms | NaN %
rows: 5,167
under estimated rows by 5x

RESULT <1ms | NaN %
rows: 5,167
under estimated rows by 5x

PROJECTSET 1.09min | NaN %
slowest
rows: 5,167
under estimated rows by 5x

RESULT <1ms | NaN %
rows: 1
under estimated rows by 1x

SORT 32.22ms | NaN %
by category.word
costliest
rows: 24,149
over estimated rows by 1x

SEQ SCAN 2.64ms | NaN %
on category (category)
largest
rows: 24,201
under estimated rows by 1x

Figure 4.13.: Execution plan for topic grouping query with merge join

In conclusion of this subsection, we could clearly propose the modification of `work_mem` as the only interesting optimization we found. This allows the query optimizer to choose faster algorithms where it is possible, without other engine hints. On the other hand, the unsuccessful attempts to improve the timings by modifying the query structure has shown us important facts about the optimizer. Firstly, as we have seen in Figure 4.10, the optimizer chooses to hash the table stored on the disk, even if it has more tuples. The reason could be the assertion that the data is evenly distributed. Secondly, when we force it to execute the join in the hard coded order, the statistics are completely useless. Therefore, we propose only the change of the memory limit. Other hard-coded modification could lead to worse performance metrics with a larger dataset.

4.3.3. Except and Union Operators

Based on our observations from the previous subsection, we have decided to inspect also the SQL operators *Except* and *Union*. Both are responsible with set operation. The first one is equivalent with $A \setminus B$, the latter for $A \cup B$. Queries in the *Polysemy* group use them. Therefore, we analyzed the execution plans for this query types.

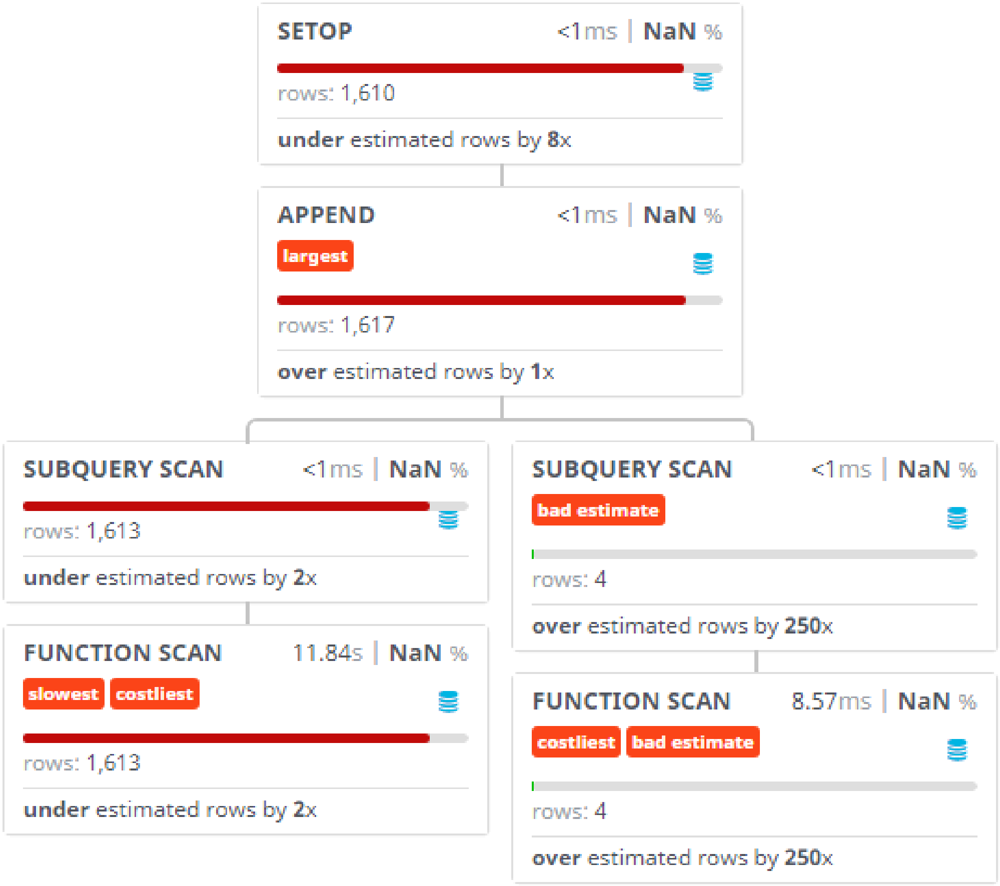


Figure 4.14.: Execution plan for query 4.3.1

We observe in Figure 4.14 the same behavior from the query optimizer. Even if here the algorithm for the `SETOP` is based on hashing, the statistics are useless. The optimizer uses for its estimations in both `Subquery Scan` a constant value of `1024` tuples. This practice could not be altered. It only shows us that it can not predict anything that comes from a user-defined-function.

5. Discussion

The scope of this thesis is to enhance an existing relational database management system with the CHQL operators and to analyze their performance. The results indicate that the operators are implementable and no further rules are needed by the optimizer.

Firstly, in order to obtain the CHQL operators implementation in a RDBMS we have chosen a combination of standard SQL and procedural language SQL/PL, an extension to the standard language. In this way, we have reproduced the CHQL query algebra in an relational database system. Because we have worked with two RDBMSs to implement the operators, we have encountered different difficulties in implementation. For example, the need to use user-defined-types for arrays in Oracle, whereas PostgreSQL offers them as standard along with functions for manipulating them. Due to this reasons, only few operators have been implemented in standard SQL and that only in PostgreSQL. All in all, we have succeeded to translate the logic of the CHQL operators in a relational database system, along with a benchmark to test their performance.

Secondly, we have tested both implementations against the typical use scenario provided by the benchmark. After we have analyzed the timings of both Oralce and PostgreSQL, we have continued the performance evaluation and optimization only on the latter. The motive behind this decision is the fact that both servers executed the same query plans and PostgreSQL is faster. Major contribution to the timings are the indexes. Contrary to our expectations, the trigram indexes have speed up significantly all the queries that requested data directly from the disk stored tables. Furthermore, they allowed us to see the internal optimization mechanism, for example the usage of buffers, thus reducing the processing time from first run to the second. After the general optimization strategies, we have experimented with different query implementations and engine hints. They have proved not to bring any performance gain. For example, we have tried to reproduce the mechanism of Oracles *BULK COLLECT INTO* in functions where we need to retrieve and then iterate through a large set of tuples [5]. Another example is the modification we have made to a query statement in order to change the execution plan. Despite the fact that the timings remained the same, we understand now the decisions made by the optimizer. Our analysis results are in line with the argument that statistics are crucial for the query optimizer.

The generalization of the results in the optimization part is limited by the early choice of using only a subset of around 10% of the whole German Text Corpora available. Most of the optimizations we have experimented with do not bring any improvements in the resulting processing time. This does not imply that they will behave in the same manner for larger data sets. Nonetheless, the results are valid for the scope of this thesis, the query plan analysis providing the necessary informations about the CHQL operators. Another limiting factor have been the necessity of using two different servers, thus making the comparison between the timings in PostgreSQL and Oracle harder to examine. A final factor to be described here is the inability of relational database systems to produce a plan for the whole query when functions and procedures are involved [2]. This along with the fact that statistics gathered at runtime are not used to dynamically adjust the next query plans are the reasons why we have observed wrong executions plans.

These results should be taken into account when considering how to implement a scalable solution of a CHQL query type system. They offer the necessary insight for an optimized implementation of the operators and highlights the weak spots of RDBSs regarding this operators.

6. Conclusion

This thesis has explored the feasibility of implementing CHQL operators in a relational database and has investigated their performance. Based on our original implementation in two different relational systems we can conclude that the declarative programming paradigm of standard SQL is insufficient. Fortunately, the procedural language extension proves to be enough to overcome this lack of functionality, enabling us to efficiently implement all CHQL functions (all but the kNN search on ngrams, which is a stand-alone research question). Analyzing the query plans has offered some meaningful insights about the behavior of the query optimizer. They enabled us to draw the inference that the optimizer is not capable to plan the whole query at once and therefore approximates falsely the number of records returned by a function. This behavior leads to an inefficient execution plan. As the results presented in this work were only tested with around 10% of the German Language Temporal Text Corpus[1] available, further research is needed in order to determine how the optimization strategies proposed here perform on larger datasets.

[1] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

Bibliography

[1] Janek Bettinger. “Efficient k-NN Search of Time Series in Arbitrary Time Intervals”. MA thesis. Karlsruhe Institute for Technology, Feb. 2018.

[2] Surajit Chaudhuri. “An Overview of Query Optimization in Relational Systems”. In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS ’98. Seattle, Washington, USA: ACM, 1998, pp. 34–43. ISBN: 0-89791-996-3. DOI: `10.1145/275487.275492`. URL: `http://doi.acm.org/10.1145/275487.275492`.

[3] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: `10.1145/362384.362685`. URL: `http://doi.acm.org/10.1145/362384.362685`.

[4] *Documentation: 11*. URL: `https://www.postgresql.org/docs/11/index.html`.

[5] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming*. O’Reilly Media, Inc., 2014. ISBN: 1449324452, 9781449324452.

[6] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. ISBN: 9780131873254.

[7] Niklas Olsen. *History in the Plural: An Introduction to the Work of Reinhart Koselleck*. 1st ed. Berghahn Books, 2014. URL: `http://www.jstor.org/stable/j.ctt9qd7pr`.

[8] Markus Raster. “Erstellung eines Benchmarks zum Anfragen temporaler Textkorpora zur Untersuchung der Begriffsgeschichte und historischen Semantik”. MA thesis. Karlsruhe Institute for Technology, Feb. 2019.

[9] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN: 0-13-861337-0.

[10] Jens Willkomm et al. “A Query Algebra for Temporal Text Corpora”. In: *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries*. JCDL ’18. Fort Worth, Texas, USA: ACM, 2018, pp. 183–192. ISBN: 978-1-4503-5178-2. DOI: `10.1145/3197026.3197044`. URL: `http://doi.acm.org/10.1145/3197026.3197044`.

A. Appendix

A.1. PostgreSQL Appendix

Name	Type	Output
absolute	help function	double[]
absolute	CHQL	set of ngram
all_counts_larger_than	CHQL	set of ngram
all_counts_smaller_than	CHQL	set of ngram
appears_in_year	CHQL	set of ngram
average	help function	double[]
change_array	help function	double[]
cohesion	CHQL	double[]
collocation	CHQL	set of ngram
conflate	CHQL	set of ngram
freq_closeness	CHQL	double[]
has_counts_larger_than	CHQL	set of ngram
has_counts_smaller_than	CHQL	set of ngram
matches_word	CHQL	set of ngram
pos_filter	CHQL	set of ngram
posix_creator	help function	text
search_word	CHQL	set of ngram
subsequence	CHQL	set of ngram
sum_array	aggregate function	double[]
transform_binary	CHQL	double[]
sumup	CHQL aggregate	double[]

Table A.1.: Functions implemented in PostgreSQL

A.2. Oracle Appendix

Name	Type	Output
absolute	CHQL	ngram_table
all_counts_larger_than	CHQL	ngram_table
all_counts_smaller_than	CHQL	ngram_table
appears_in_year	CHQL	ngram_table
average	CHQL	ngram_table
change_array	help function	TIMES
closeness_func	CHQL	TIMES
cohesion	CHQL	TIMES
collocation	CHQL	ngram_table
conflate	CHQL	ngram_table
freq_closeness	help procedure	TIMES
has_counts_larger_than	CHQL	ngram_table
has_counts_smaller_than	CHQL	ngram_table
matches_word	CHQL	ngram_table
pos_filter	CHQL	ngram_table
posix_creator	help function	varchar2
search_word	CHQL	ngram_table
split_to_array	help function	SPLIT_TYPE
subsequence	CHQL	ngram_table
subsequence_array	help function	TIMES
sum_array	help procedure	double
sumupagg	CHQL aggregate	TIMES
transform_binary	CHQL	TIMES
transform_POS	help function	POS
transform_TIMES	help function	TIMES

Table A.2.: Functions implemented in Oracle

A.3. Timing Appendix

Query No	Oracle	PostgreSQL			Rows
	No index (ms)	No index (ms)	B-Tree (ms)	Trigram (ms)	
1.1.1	1040499.0	479520.274	483699.721	6076.72	2
1.1.2	1058607.0	560638.689	569279.955	7959.411	0
1.1.3	1041657.0	558238.666	562881.055	18889.644	6
1.1.4	1029419.0	493247.841	498384.191	10282.512	10
1.1.5	1038005.0	499536.737	502432.962	121582.708	2
1.2.1	1039234.0	527514.065	535551.658	11479.277	1836
1.2.2	1082058.0	512942.546	518204.309	141278.491	2947
1.2.3	1068555.0	527552.385	529901.336	481.152	541
1.2.4	1346042.0	544038.167	545611.905	100091.071	1957
1.2.5	1037693.0	542284.330	551315.770	120.446	767
2.1.1	388151.0	53205.460	52685.394	68.26	0
2.1.2	223468.0	107777.949	155278.637	112.385	1370
2.1.3	323928.0	160204.027	271572.073	82.635	1
2.1.4	206439.0	116813.750	152290.347	85.841	233
2.1.5	201721.0	24662.678	24252.347	2.499	0
2.2.1	208012.0	23823.723	23699.491	187.754	194
2.2.2	206488.0	23509.505	23293.422	58.137	0
2.2.3	203986.0	24339.010	23528.926	2.23	0
2.2.4	209427.0	23826.107	23554.354	128.135	1368

Table A.3.: Timings Part 1

Query No	Oracle	PostgreSQL			Rows
	No index (ms)	No index (ms)	B-Tree (ms)	Trigram (ms)	
3.1.1	135829	107960.721	159424.823	6286.806	201
3.1.2	331363	71565.74	272026.077	3634.175	882
3.1.3	209693	24934.286	24073.294	41.927	0
3.1.4	214271	25469.656	24659.77	16.627	0
3.1.5	219374	29612.852	29393.823	357.651	88
3.2.1	209937	28226.624	28023.254	4058.986	599
3.2.2	384654	269377.013	260141.929	2780.822	0
3.2.3	222896	15231.314	15039.031	839.868	469
3.2.4	312281	158298.674	240269.301	812.075	322
3.2.5	318738	43462.904	282126.904	80.975	0
3.3.1	210320	28367.683	29097.706	4009.805	198
3.3.2	198888	16146.632	17301.461	1202.849	4
3.3.3	203887	15084.358	16180.114	66.733	2
3.3.4	201108	18066.669	17068.884	1736.064	40
3.3.5	208441	30964.944	28217.178	4712.216	30
3.4.1	202074	17423.95	15779.914	993.352	90
3.4.2	380117	289843.848	318657.589	5868.83	320
3.4.3	406908	87824.139	337951.391	2813.689	951
3.4.4	220731	87241.246	28398.878	4054.654	787
3.4.5	201808	16310.187	15873.205	971.955	226
3.4.6	319780	218208.388	241461.58	837.399	305
3.5.1	143580	118094.305	155042.684	1030.925	0
3.5.2	339798	61303.039	260582.436	3692.665	2
3.5.3	134822	24190.165	150592.114	538.372	0
3.5.4	335980	45758.592	190924.672	4325.587	4
3.5.5	201776	20723.139	42908.465	296.638	0

Table A.4.: Timings Part 2

Query No	Oracle	PostgreSQL			Rows
	No index (ms)	No index (ms)	B-Tree (ms)	Trigram (ms)	
3.6.1	200080	16867.717	15823.068	1037.164	226
3.6.2	375440	270779.284	312851.462	1442.847	951
3.6.3	375979	125525.606	331137.518	1583.022	951
3.6.4	379845	51713.362	52425.852	1623.333	951
3.6.5	234356	92262.988	48417.34	29071.527	4255
3.7.1	411329	29068.763	27804.37	4028.416	5
3.7.2	258459	73441.603	72715.316	55980.701	23
3.7.3	406025	49790.361	49199.573	72330.512	25
3.7.4	211643	26562.542	26506.617	235.419	0
3.7.5	411329	38038.997	37597.539	32279.926	6
4.1.1	213977	25274.891	24699.633	224.319	62
4.1.2	398734	210905.645	278998.761	4533.389	834
4.1.3	429098	72542.829	298398.362	48840.908	4629
4.1.4	324685	167386.585	127952.677	102522.517	7195
4.1.5	236960	25452.903	25627.451	88.784	4
4.2.1		52189.995	52814.299	4144.531	787
4.2.2		115559.063	112804.098	206315.347	10242
4.2.3		29532.366	28852.625	1207.556	469
4.2.4		40926.164	39652.468	623.082	0
4.3.1		48372.793	46816.35	10208.359	1610
4.3.2		32485.644	31821.171	1819.127	75
4.3.3		33643.194	32077.438	191.314	1
4.3.4		32756.533	32242.072	1655.701	75

Table A.5.: Timings Part 3