Exploring Space-Time Trade-Offs in the SAP NetWeaver® Engine TREX

Author: Philippe Masson

Supervisor:
Prof. Donald Kossmann
Dipl.-Ing. Franz Färber

March 3, 2008
Preface

This master thesis has been written in collaboration with the SAP NetWeaver® EIM TREX team at SAP AG, Walldorf. My adviser there was Franz Färber. Help was provided by Uwe Freising who helped me to integrate my prototype implementation into TREX. Marit Rams took the time to support me regarding input data and tests as well as to give me insight into the natural language interface. Holger Schwedes was very helpful regarding different aspects of the join engine and programming within the attribute engine. Johannes Woehler explained both his thesis and aspects of TREX in great detail whenever I was uncertain. Andrew Ross checked the English of this thesis. I would also like to thank Arne Schwarz and Denise Albring for making the administration part of this work fairly easy.
Abstract

In many applications, trade-offs between time and space consumption have to be considered. Those decisions are vital for database-like systems, such as the SAP NetWeaver® backend search and data analysis engine TREX. This master thesis exhibit several points in the application scenarios for TREX where such trade-offs between processing time and memory space may advantageously be shifted toward using more space and gaining additional performance. Some methods are proposed for materializing either all or parts of a join index, which is a construct similar to a logical view and is used frequently in TREX. Various advantages and drawbacks are analyzed and the anticipated performance gain is estimated. Also a solution involving materialization is proposed to speed up the TREX natural language interface (NLI), the response times of which are sometimes unsatisfactory. A prototype of this proposal is implemented and compared against the existing solution concerning performance and memory consumption. We show that most of the queries can be handled in about one third of the time spent by the existing solution, although the numbers depend strongly on the query type. Finally, a glance into the future shows that the end of the rope in exploring trade-offs has not been reached by far, and several interesting generalizations remain to be considered.
Contents

Preface .......................................................... I
Abstract .......................................................... II
Table of Contents ............................................... III
List of Figures ................................................... V
List of Tables .................................................... VI

1 Introduction 1
1.1 Overview and Motivation .................................. 1
1.2 Aim and Approach ........................................... 1

2 TREX Architecture and Data Model 2
2.1 Architecture Overview ..................................... 2
2.1.1 TREX Index Server: Main Engines ...................... 3
2.2 Data Representation ......................................... 4
2.3 Data Distribution ............................................. 5

3 Reducing Joins 7
3.1 Initial State – Status Quo ................................. 7
3.1.1 Data Structures ........................................... 7
3.1.2 Queries in Detail ......................................... 7
3.1.3 Updates in Detail ......................................... 9
3.2 Proposal 1: Finer Grained Filtering ..................... 9
3.2.1 Data Structures ........................................... 10
3.2.2 Queries in Detail ......................................... 10
3.2.3 Updates in Detail ......................................... 10
3.3 Proposal 2: Distributed Materialization ................ 11
3.3.1 Data Structures ........................................... 11
3.3.2 Queries in Detail ......................................... 12
3.3.3 Updates in Detail ......................................... 12
3.4 Proposal 3: Materialized View ............................ 13
3.4.1 Data Structures ........................................... 13
3.4.2 Query in Detail ............................................ 14
3.4.3 Updates in Detail ......................................... 14
3.5 Implementation and Evaluation ............................ 14
4 Improving NLI Performance
4.1 NLI Today .................................................. 16
   4.1.1 Estimations in Detail .......................... 17
   4.1.2 Semantic Reduction in Detail ............... 17
   4.1.3 Range Search in Detail ....................... 17
   4.1.4 Example ............................................. 17
   4.1.5 Estimation Cost Analysis .................... 18
   4.1.6 Limitations ....................................... 18
4.2 Proposal: Faster Estimation via Materialization .... 18
   4.2.1 Queries ............................................. 19
   4.2.2 Estimation Cost Analysis .................... 19
   4.2.3 Updates ............................................. 20
   4.2.4 Security Aspects ............................... 20
4.3 Implementation .......................................... 21
   4.3.1 Collecting Information on Base Data ........ 21
   4.3.2 Integrating the New Search Method .......... 21
   4.3.3 Updates ............................................. 22
   4.3.4 Security Aspects ............................... 22
4.4 Evaluation ................................................ 22
   4.4.1 Data Set ........................................... 22
   4.4.2 Test Method ..................................... 23
   4.4.3 Queries ............................................ 23
   4.4.4 Results ............................................ 25
4.5 Future Work .............................................. 27

5 Summary and Future Outlook ......................... 28
5.1 Global Dictionary ....................................... 28
   5.1.1 Idea Sketch ..................................... 28
   5.1.2 Positive Aspects ............................... 29
   5.1.3 Issues ............................................. 29

A Evaluation Results in Detail ....................... 31
List of Figures

2.1 Overview of the TREX architecture ........................................ 2
2.2 Business object with two base tables .................................... 5

3.1 Proposal 0 – Data Model ..................................................... 8
3.2 Proposal 1 – Data Model ..................................................... 9
3.3 Proposal 2 – Data Model .................................................... 11
3.4 Proposal 3 – Data Model .................................................... 13

4.1 Proposal NLI – Data Model ................................................ 19
List of Tables

4.1 Characteristics of the datasets ............................................. 23
4.2 Query complexity for the data set addressbook sorted by word count .... 24
4.3 Query complexity for the data set material sorted by word count ...... 24
4.4 Average query times by business object, undistributed .................. 25
4.5 Slowest queries by business object, undistributed .......................... 25
4.6 Average query times by business object, distributed ........................ 26
4.7 Slowest queries by business object, distributed ............................ 26
4.8 Memory usage .................................................................... 26

A.1 Median of time per query for “addressbook” ................................. 31
A.2 Median of time per query for “material” ....................................... 32
A.3 Median of time per query for “material”, wildcard optimized .......... 32
A.4 Average query times by business object, on laptop ....................... 33
A.5 Slowest queries by business object, on laptop ............................... 33
Chapter 1

Introduction

1.1 Overview and Motivation

In many applications, design decisions have to be made on whether to focus on speed optimizations or rather use as little memory space as possible. With new generations of hardware, processing speed is increasing and the price per gigabyte of RAM is falling. Solutions that were considered impossible to realize a few years ago may now seem quite reasonable. Within the SAP NetWeaver development group, it was decided quite early that the operations of the search and data mining engine TREX should be carried out in installed memory. This decision allowed delivery of impressive query execution speed but also forced the choice of data representations that minimized consumption of memory space rather than CPU time. Since then, memory prices have continued to sink, and some of those early design decisions need to be reconsidered to see whether it is feasible to speed up the search engine even more by allowing the relevant data to occupy more space in memory.

1.2 Aim and Approach

The aim of this thesis is to explore possibilities to speed up the SAP NetWeaver search engine TREX and in doing so enable new kinds of search. To achieve this goal, we need to analyze the current situation, identify possible bottlenecks, and find solutions that avoid the bottlenecks. One of the more promising proposals can then be chosen for implementation as a prototype.
Chapter 2

TREX Architecture and Data Model

This chapter describes the current architecture and data model of TREX in what should be sufficient detail for the following chapters.

2.1 Architecture Overview

TREX is an integrated set of software components based on a client-server architecture. It consists of several services implemented as servers that can be used via two main clients. The five most important servers are as follows.

![Overview of the TREX architecture](image)

Figure 2.1: Overview of the TREX architecture
CHAPTER 2. TREX ARCHITECTURE AND DATA MODEL

- The **RFC server** handles communication between an attached SAP system and the TREX servers.
- The **queue server** coordinates the asynchronous indexing of text-based documents.
- The **preprocessor** performs the linguistic analysis and filtering of text-based documents.
- The **name server** manages the internal landscape of TREX, including service binding and load balancing.
- The **index server** includes engines for search, text mining, attributes (document metadata and structured mass data), joins, and natural language.

This thesis focuses exclusively on the engines in the TREX index server.

For application clients, TREX exposes a Java API and an ABAP API. Internally there are also development APIs in C++ and Python, but these are not visible to application users.

### 2.1.1 TREX Index Server: Main Engines

The TREX index server currently contains four main engines, which operate more or less independently of each other.

**Text Mining Engine**

The text mining engine implements the most commonly used text mining functionality for TREX. This includes searching for similar terms and for similar documents as well as document classification and clustering. Additionally, features can be extracted, generating a set of keywords which should accurately describe the content of the document.

**Search Engine**

The search engine implements indexing and search for unstructured or semi-structured data like PDFs or HTML/XML documents. It features all of the features commonly found in search engines, namely:

- **Exact Matching**: Documents that include exactly the entered search term are found.
- **Boolean Search**: Multiple queries can be concatenated by boolean operators such as *OR*, *AND*, and *NOT*. All documents for which the given boolean expression evaluates to *true* are returned.
- **Wildcard Search**: Parts of words in a wildcard search can be replaced by a wildcard character “?” representing one unknown character or “*” representing any number of characters.
• **Fuzzy Search:** Documents can be returned that only approximately match the given query. A tunable parameter determines how exact the match should be.

• **Linguistic Search:** In this case, the query is considered as natural language text and some basic operations are performed on it such as eliminating stop words and stemming inflected words before the search is executed.

In each case, result sets are ranked on the basis of the statistics of term frequencies in documents and indexes.

**Attribute Engine**

The attribute engine was originally designed as a volatile storage mechanism for document metadata but has grown to a full-featured storage and search engine for large volumes of structured data. It has the same search options as the search engine but can search on different types of data, such as strings, integers, fixed point numbers, and dates. On some of these types, range searches are also possible. The attribute engine performs all its operation in memory, which gives it excellent performance.

**Join Engine**

The join engine is coupled with the attribute engine and permits searches on more complex structures called *business objects* (see section 2.2). The business object data is stored in a normalized form in table indexes within the attribute engine and joins need to be calculated between the table indexes to reconstruct the original (denormalized) business objects. The join engine uses a sophisticated optimization algorithm to find a join order that minimizes the size of the intermediate result sets.

## 2.2 Data Representation

To store structured data, TREX uses physical indexes containing a normalized form of the data. The queries mostly do not address those indexes directly but instead are performed on so-called *business objects* (BOs). In fact, business objects are not only a data model but can also include methods, so they are at a higher abstraction level. However, in this work we also use the term “business object” to denote the underlying join index representing the data model. If ever a real business object is meant, we shall state this explicitly. Business objects form a view on the physical indexes given by a (static) join graph with its join conditions and the view attributes. The physical indexes are implemented as inverted indexes corresponding to individual attributes (we call these attributes the *base data*). A direct consequence of this design is that often many joins have to be performed to build up a query result.

**Example:** Figure 2.2 shows a simplified model of a *SalesOrder* business object. It is a join index with the base tables *SalesOrderRoot* (which serves as anchor table), *SalesItem*, *CustomerRoot* and *ProductRoot*. The SalesItem table contains identifiers linked
to entries in CustomerRoot and ProductRoot. The BO SalesOrder as seen by the user consist of the attributes “key” and “date-time” from the SalesOrderRoot table, the attribute “productname” from the ProductRoot table and the attribute “full name” from the CustomerRoot table. An instance of SalesOrder is uniquely identified by the key of the anchor table, although this instance may correspond to more than one row in the table representing SalesOrder. Note that it is quite possible (and even probable) that base tables like ProductRoot and CustomerRoot are also used for other business objects.

![Business Object "SalesOrder"](image)

<table>
<thead>
<tr>
<th>Order</th>
<th>Date</th>
<th>Product</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>04.03.2008</td>
<td>Pen</td>
<td>Jens K.</td>
</tr>
<tr>
<td>2</td>
<td>04.03.2008</td>
<td>Paper</td>
<td>Jens K.</td>
</tr>
<tr>
<td>3</td>
<td>22.03.2008</td>
<td>Pen</td>
<td>Marie K.</td>
</tr>
</tbody>
</table>

![SalesOrderRoot and SalesItem](image)

![ProductRoot](image)

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Paper</td>
</tr>
<tr>
<td>2</td>
<td>Pen</td>
</tr>
<tr>
<td>3</td>
<td>Ink</td>
</tr>
</tbody>
</table>

![CustomerRoot](image)

<table>
<thead>
<tr>
<th>Key</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Petra H.</td>
</tr>
<tr>
<td>2</td>
<td>Franz F.</td>
</tr>
<tr>
<td>3</td>
<td>Jens K.</td>
</tr>
<tr>
<td>4</td>
<td>Marie K.</td>
</tr>
</tbody>
</table>

Figure 2.2: Business object with two base tables

### 2.3 Data Distribution

TREX has been designed to be easily scalable, in order to deal with large amounts of data or a high frequency of user requests. Since a single index server can only handle a limited amount of data (all indexes have to fit into the available memory) and execute a finite number of requests, TREX distributes data over multiple index servers when the load is too much for one server. The servers may be hosted on scalable hardware, such as multiple blade servers in an enclosure with space for additional blades. The most straightforward way to scale TREX over multiple index servers is to assign separate indexes to the respective index servers. The TREX landscape always includes a single active master name server that handles all incoming requests transparently in the sense that the actual location of
the data in the host landscape may remain invisible to the user. However, joins between indexes are easier to carry out if the indexes are on the same host, so the distribution does matter for performance. If an individual index is very large, which is very likely in data warehouse scenarios where indexes tend to become huge, TREX can split the index horizontally into a number of parts, which can be stored in memory on different hosts. In this case, TREX builds a logical index over the split parts so that again the user does not have to care about the distribution. It is a serious development challenge to ensure scalable performance in distributed scenarios.
Chapter 3

Reducing Joins

At present, TREX needs to perform many joins to be able to find the keys in the anchor table for rows in the business object that satisfy the query conditions. The task is to minimize the number of required joins without exceeding the given space constraints. The current TREX setup allows us to make certain additional assumptions. First, the base data of an entry can always be determined directly by means of the view attributes. Second, the input set for a join can be restricted to certain tuples, and TREX can execute the join faster when this set is smaller.

We use a simplified model here to describe queries, data updates, and query execution. This model is intended to reflect the reality as well as necessary. We define a query as a set of keywords, optionally with attribute names, denoted as \( k_1(A_1), \ldots, k_n(A_n) \), plus a specification of the business object to be queried. We shall ignore the optional attribute names wherever naming the attributes used in a query is nothing more than a descriptive simplification.

3.1 Initial State – Status Quo

This section outlines how TREX currently avoids unnecessary joins. TREX performs a coarse-grained filtering using the inverted index of the base data as an oracle. Many false positives may remain.

3.1.1 Data Structures

The main data structure used is the inverted index on the attribute values. It supports searching for a specific value and returns the surrogate IDs (SIDs) in which the value is found. This lookup takes constant time per generated resulting SID.

3.1.2 Queries in Detail

Additional Input: \( I_1, \ldots, I_m \) denoting the inverted indexes on the base data, determined by the view attributes of the BO
// returns true iff k is found in I
bool probe(Keyword k, Index I);

for (i := 1; i <= n; i++)
{
    bool succ = false;
    for (j := 1; j <= m; j++)
    {
        if (probe(k_i, I_j))
        {
            succ = true;
            break;
        }
    }
    if (!succ)
    // at least one keyword could not be found
        return empty;
}

// do joins, find result

In all the pseudocode, we omit to state that probe is in fact used to reduce the input set of the joins. Since all the following proposals involve doing this in the same manner, this step is of no great interest here.

**Algorithm for probe()**

The probe function is just a wrapper for the lookup in the inverted index:

```cpp
bool probe(Keyword k, Index I){
    Array<SID> reducedInput = I.invertedIndex.lookup(k);
    return !reducedInput.empty();
}
```
CHAPTER 3. REDUCING JOINS

Costs for probe()

The cost for an inverted index lookup and therefore for probe is linear in the number of SIDs generated in reducedInput.

3.1.3 Updates in Detail

Updates are handled lazily, which means that they are not executed instantly but gathered in a so-called delta index. This is true for all the proposals considered here, but updates are of minor importance for our considerations. The inverted index is recomputed when the delta index is merged with its main index.

Algorithm for Updates

Updates can be done by just inserting the new value into the inverted index. With delta indexing, the update of the base data occurs when the inverted index is recomputed.

Costs for Updates

In general, the cost for updating is a constant.

3.2 Proposal 1: Finer Grained Filtering

This proposed solution extends the existing solution to reduce the number of false positives. It does so by storing the BOs in which an entry occurs together with the inverted index. This can be done in several ways, e.g., with an ordered list.

<table>
<thead>
<tr>
<th>CustomerRoot</th>
<th>Inverted Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Key</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 3.2: Proposal 1 – Data Model
3.2.1 Data Structures

The main data structure is an extended version of the inverted index on the attribute values. This supports direct lookup of values belonging to a specific BO. Inserting a value with a specific SID and BO ID needs $O(\log(\#BO))$ time when using an ordered list.

3.2.2 Queries in Detail

The code is the same as in the initial state except for the signature of \texttt{probe}:

\begin{verbatim}
bool probe(Keyword k, Index I, BusinessObject bo);
\end{verbatim}

Algorithm for \texttt{probe}()

Again, \texttt{probe} is only a wrapper for the lookup feature of the (now extended) inverted index.

\begin{verbatim}
bool probe(Keyword k, Index I, BusinessObject bo){
    Array<SID> reducedInput =
        I.extendedInvertedIndex.lookup(k, bo);
    return !reducedInput.empty();
}
\end{verbatim}

Costs for \texttt{probe}()

The query costs depend on the implementation. If an ordered list is used, then finding out whether an entry occurs in a specific BO needs $O(\log(\#BO))$ time. This has to be done for every result that is generated, to give a running time of $O(\log(\#BO) \times \text{sizeAttribute})$.

3.2.3 Updates in Detail

Insertion of a new element is treated as if this new element would occur in all possible BOs. This results in a strict overapproximation. Deletion of an element does not lead to deletion of the entry in an inverted index until a complete recomputation is done (there is room for quite simple optimizations here). Changing an entry corresponds to deleting the old one and inserting the new one, and can be handled as follows.

Algorithm for Updates

\begin{verbatim}
void update(Index I, SID sid, Value newVal){
    for (BusinessObject bo : AllBOs)
        I.extendedInvertedIndex.insert(sid, newVal, bo);
}
\end{verbatim}
Example

Consider a physical index for an income table that includes an entry for the employee ‘Mayer’ with an income of 50 000 euros. Let us suppose he shows up in exactly one BO, namely Employees. Now he gets a big raise and his income is doubled. Since there may be other employees with an income of 50 000 euros, we cannot simply delete that value from the inverted index. However, we need to consider adding the new entry for Mayer to other BOs. For example, his new income may make him a candidate for the BO High-income-emps.

Costs for Updates

An update as written above would need $O(\#BO \ast \log(\#BO))$ time. However, it can easily be done in $O(\#BO)$ time if the BOs are given IDs generated in ascending order and sorted, so the proposed solution is not really needed. Note that if the update includes a deletion, the result set may contain false positives, which may slow down query execution by inflating the number of joins to be calculated.

3.3 Proposal 2: Distributed Materialization

This version is an extension of the first proposal in which not only the BOs are stored but also the keys of the BO view for every entry in the inverted index.

![Figure 3.3: Proposal 2 – Data Model]

3.3.1 Data Structures

This proposal uses another form of extended inverted index. As in proposal 1, the index can be queried directly for all values $v$ that occur in a specific BO. However, it returns as result a list of keys of the specified BO. The running time is again linear in the output size.
plus $\log(\#BO)$ when an ordered list is chosen for implementation. A special symbol key \textit{INFINITE} is provided to represent the infinite set of all possible keys.

### 3.3.2 Queries in Detail

// returns the set of keys of a BO whose tuple contains k
Set<key> probe(Keyword k, Index i, BusinessObject bo);
$\bigcap_{i=1..n} \left( \bigcup_{j=1..m} \text{probe}(k_i, I_j, BO) \right)$

// create complete result tuples by executing joins

Algorithm for \textit{probe}()

Again, the core of the algorithm is the lookup feature provided by the extended inverted index. When an \textit{INFINITE} element is encountered, it is returned as the (only) result.

List<Key> probe(Keyword k, Index I, BusinessObject bo){
    Array<Pair<SID, key>> resultPairs =
        I.extendedInvertedIndex.lookup(k, bo);
    Array<Sid> reducedInput = Array<Sid>();
    Set<Key> result = new Set<Key>();
    for (Pair<Sid, Key> aPair : result)
    {
        if (!resultPairs.has(aPair.first))
            result.push_back(aPair.first);
        result.push_back(aPair.second);
        if (aPair.second == INFINITE)
        {
            result.clear();
            result.push_back(INFINITE);
        }
    }
    return result;
}

Costs for \textit{probe}()

As in the first proposal, finding out whether an entry occurs in a specific BO needs $O(\log(\#BO))$ time. However, all keys corresponding to that value and BO have to be fetched, which in the worst case takes $O(\#ValuesInBO)$ time.

### 3.3.3 Updates in Detail

A newly inserted element needs to be treated as if it occurs in every possible BO with every possible key (which can be represented as an infinite set). This is problematic, since
every union with such an element results in an infinite set, and even an intersection can
give an infinite set. In this case, a full join has to be carried out. Deletion and update of
elements can be handled as in the first proposal.

Algorithm for Updates

void update(Index I, SID sid, Value newVal)
{
    for (BusinessObject bo : AllBOs)
        I.extendedInvertedIndex.insert(sid, newVal, bo, INFINITE);
}

Example

Consider the same example as above, where Mayer gets a raise from 50 000 to 100 000
euros. Not only could this entry occur in many BOs but also the entry could be associated
with virtually any key in a BO.

Costs for Updates

Regarding costs, the same considerations apply as in the first proposal. However, the query
costs increase dramatically in the presence of updates.

3.4 Proposal 3: Materialized View

A third possible solution is to create a completely new inverted index at the level of the
BOs.

![Data Model](image)

Figure 3.4: Proposal 3 – Data Model

3.4.1 Data Structures

The central data structure here is an inverted index of every BO. It stores not only the
value of each field but also the view attribute to which this value belongs.
3.4.2 Query in Detail

// returns the set of keys of a BO whose tuple contains k
Set<key> probe(Keyword k, BusinessObject bo);

\[ \bigcap_{i=1..n} probe(k_i, BO) \]

// create complete result by either finding all the entries
// belonging to each key or pushing down to physical indexes
// and execute join

This only works as long as we ignore the fact that our query model allows indication of the attributes in which to look for the value. To be able to answer these queries as well, we either have to change how we store the new inverted index (which results more or less in the second proposal) or change the algorithm to:

// returns the set of keys of a BO whose tuple contains k
Set<key> probe(Keyword k, BusinessObject bo, Attribute A);
\[ \bigcap_{i=1..n} probe(k_i, BO, A_i) \]

Algorithm for probe()

Set<key> probe(Keyword k, BusinessObject bo, Attribute A){
    return BOInvertedIndexes[bo].lookup(k, A);
}
\[ \bigcap_{i=1..n} probe(k_i, BO, A_i) \]

Costs for probe()

Finding a value in the inverted index needs constant time. However, checking whether the value belongs to the correct attribute and fetching all the results corresponding to the given attribute and value uses in the worst case \( O(#ValuesInBO) \) time.

3.4.3 Updates in Detail

Exactly the same problems arise as with the second proposal and we deal with them in the same way. There is a lot of research done regarding updating materialized views. For starters consider [1], [2] and [3] for an overview.

3.5 Implementation and Evaluation

Most of the options discussed here were already covered (without my previous knowledge) in [4], including an implementation of a slightly different version of proposal 2, with which a performance improvement factor of between 3 and 10 was demonstrated. The main reason
why none of this actually made it into the released version of TREX was the need for a much more complex and time-consuming update procedure. By contrast, the increased memory consumption of about 70% would have been quite tolerable.
Chapter 4

Improving NLI Performance

The natural language interface (NLI) is a relatively new (2007) addition to TREX. It allows users to phrase their queries in natural language. TREX then analyzes the phrase and computes a set of possible interpretations of the query. TREX can either execute the query under all these interpretations or select the most probable interpretation and use it for the subsequent search (which is the default behavior).

4.1 NLI Today

Extracting semantics from a natural language query is a very complex task, which no known computer program can do perfectly at present. TREX uses five steps to answer the query as accurately as possible. The business object or base table that is queried has to be stated explicitly. In this section, let us assume that a business object is queried. If this is not the case, the procedure may be simpler.

- **Tokenization:** The phrase is split up into words and built up into word chains of a length not longer than four. The resulting tokens have a start position, a length, and a content. Dates are recognized and added as an extra token in ISO format.

- **First estimation:** Every token is looked up in all the dictionaries of the base tables.

- **Semantic reduction:** An analysis of the semantics of the input phrase is performed (see below for details).

- **Second estimation:** If any token is left with no special meaning and is not found in the dictionaries, a wildcard search is started for a term that contains this token.

- **Range search:** Whenever a keyword indicating a range of values is found, an additional range search is performed. There is no need for such an additional search when only boolean operators are found.
4.1.1 Estimations in Detail

If a token is found, an estimation of how often it will occur in the business object is returned. This estimation is computed by considering the amount of base data and how often the given term occurs there. This estimation is often the most time-consuming action of the whole NLI search, since it has to be performed on the base data of every view attribute and carried out for every token, which sums up to about $4n$ times where $n$ is the number of words in the search phrase.

4.1.2 Semantic Reduction in Detail

The semantic analysis of the query includes recognition of attribute names and keywords (like boolean operators) as well as stemming of inflected forms and elimination of stop words. Aliases for attributes can be defined using more natural names. For example, the alias “family name” could be defined for the attribute “pers.fam.name”, to enable a user to search for a phrase like “family name of Petra in Walldorf”. These aliases need to be considered in the semantic analysis. Boolean operators (e.g., \textit{AND}, \textit{OR}, \textit{NOT}) and order relations with their various aliases also need to be recognized. For example, “later than” before a date is converted into a “$>$” relation.

4.1.3 Range Search in Detail

Whenever a keyword indicating a range is found (e.g., greater than, between, less than), an additional search is performed including the newly found meaning. This is only done if the argument of the range search has been identified as belonging to a type that allows range searches (e.g., dates and numbers) and the search is only performed on attributes with a matching type.

4.1.4 Example

Consider the example query “Franz with birthdate later than 1.1.1970”. Let us assume that birthdate is a view attribute (or an alias of a view attribute) and “Franz” occurs as a first name and as a company. First, the phrase is split into tokens, to give the result Franz, with, birthdate, later, than, 1.1.1970, Franz with, with birthdate, . . . , 1970-01-01. For each one of the resulting 19 tokens an estimation is built. Let us further assume that the only tokens with valid estimations are “Franz” and perhaps “1970-01-01”, depending on whether this date occurs in the business object. Performing the semantic analysis results in recognition of “birthdate” as an attribute, as well as conversion of “later than” into a “$>$” relation. The token “with” is discarded as a stop word. This still leaves the token “1.1.1970” as unrecognized, so a second estimation is done to search for every term containing this string. Assuming no results are returned, the token is overruled by the token representing the ISO form of the date. Since a relation operator was found, a range search is performed using the query “$> 1970-01-01$” on the attribute “birthdate” and an
estimation for that term is returned. As an intermediate result, we get the following two interpretations:

1. first_name EQ Franz AND birthdate > 1970-01-01
2. company EQ Franz AND birthdate > 1970-01-01

Their ranking depends on how often the respective terms appear. In general, TREX tests at least the top-ranking interpretation and returns the results to the user. If this interpretation returns no results, TREX tests the next-ranking interpretation.

### 4.1.5 Estimation Cost Analysis

If \( n_1, \ldots, n_v \) denotes the number of distinct values in the base data and \( v \) denotes the number of view attributes, the cost for the first estimation process can be approximated by

\[
O\left( \sum_{i=1}^{v} \log(n_i) \right) = O\left( \log\left( \prod_{i=1}^{v} n_i \right) \right)
\]

If \( n_{\text{max}} \) denotes the maximal number of distinct values of any base data, then the given term is clearly bounded by

\[
O\left( \log(n_{\text{max}}^v) \right) = O\left( v \log(n_{\text{max}}) \right)
\]

The second estimation process involves a wildcard search and has to look at every term once, so the cost can be approximated by \( O\left( \sum_{i=1}^{v} n_i \right) \).

### 4.1.6 Limitations

At present, an NLI search is only performed on a preselected number of view attributes. Although the user can change this setting, the default is just 20 view attributes, which is much less than the 300 or more that can occur in practice. This constraint was added because performance suffers greatly when many attributes with many distinct values are scanned. Another limitation is the token length, which is restricted to four. Again this is for performance reasons, but the user can easily work around the problem by setting quotation marks around word chains that should be treated as single tokens.

### 4.2 Proposal: Faster Estimation via Materialization

To support efficient search on large numbers of view attributes, the proposal is to build a table recording for each term which view attribute it occurs in. An additional flag validated records whether the term is certain to show up in the business object. At construction time, TREX computes an estimation using as statistical data the number of occurrences of the term in the business object and in the base data. The formula used is
The formula weights the two parts differently because for the user the only relevant number is how many hits in the BO the query finds. The hits in the base data are only considered because the estimations are not updated at insertion or deletion of elements. Taking account of the base data for the estimations allows for the fact that new join partners could make terms appear in the BOs that were not there at crawl time.

In fact, since the data in TREX is typed, one new table per BO is not sufficient. TREX needs one new table for each combination of BO and type. A special case is the type \textit{FIXED} which represents fixed point numbers and takes two parameters, one for the number of digits before the decimal point and one for the digits thereafter. Since both exact and range searches have to be possible on this type, the only viable solution is to use a separate table for each combination of digits. This can lead to an excessive number of tables and lose at least part of the performance advantage gained by the new tables.

\subsection{Queries}

The steps taken to execute the query are the same as in the previous proposal. However, the estimation process is simplified to looking up in the new table whether a term exists, and if it does, returning an estimation for each view attribute.

\subsection{Estimation Cost Analysis}

If \( n_1, \ldots, n_v \) denotes the number of distinct values in the base data and \( v \) denotes the number of view attributes, the cost for the first estimation process can be approximated by \( O(\log(\sum_{i=1}^v n_i)) \). This is clearly better than the previous \( O(\log(\prod_{i=1}^v n_i)) \), but since \( \sum_{i=1}^v n_i > n_{\text{max}} \) the maximal performance gain is strictly bounded by \( v \). The second
estimation process still has to consider every term of all the base data once, giving the same asymptotic performance as before. However, since this full scan is now done on one table, connection costs might be reduced. On the other hand, it is harder to perform parallel processing on a single table.

Correctness

There is no precise definition of what it is for an answer to an NLI query to be correct. In general, the only constraint is to check whether the answer to a query is what the user wanted. In some cases, more than one answer can be meaningful and it makes little sense to check whether a new method always returns the same result as before.

4.2.3 Updates

Updates have two kinds of impact:

- New terms need to be found immediately
- Estimations might become incorrect over time

The first issue is rather straightforward to deal with. We can propagate every view attribute update immediately to the corresponding new table, inserting it with an estimation of \( \frac{1}{\text{size}(\text{viewAttribute})} \) and without setting the `validated` flag.

If we wish to ensure that the estimations always conform to the formula given in section 4.2, every update on the base data must trigger a complete recomputation of the estimations of any BOs with the updated attribute in their join path. This is utterly infeasible. Instead, we can take a heuristic approach and simulate an aging process. One possibility is to include search results to set and reset the `validated` flag and see the real estimation value as a combination of the value saved under `estimation` and the `validated` flag. We can assume that deletion happen very rarely, which means the `validated` flag is never reset due to aging. And we can count the number of new insertions into the different view attributes and periodically update the `estimation` values accordingly. However, an additional structure would be needed to save this data. It would consist of a hash table which maps from a value to an integer representing the number of new insertions for this value. In TREX, the periodical update could be executed during a delta merge, when the assembled updates, which are in a tree form until then, are merged into the flat base data. However, this choice is rather arbitrary and we need test results to show whether this proves to be a good idea.

4.2.4 Security Aspects

As a search engine operating on business data, TREX is required to perform some kind of user handling, since not every user is allowed to see all the available data. This also concerns the estimation process since an interpretation is only meaningful to a user when
the user is actually able to see the result. In the proposed solution we did not take account of this aspect of search. The obvious solution to this problem is to perform a separate estimation for each user. Generally, this is infeasible, since there may be a huge number of users. However, in TREX, as in many other systems, user have roles, that define their rights. The number of roles might still be fairly large so storing an estimation per role will in general still not be feasible. Nevertheless, it might be possible to store a validated flag for every role, which would already indicate whether results are visible to a specific user. As an alternative, it might be possible to define a small number of security levels such that only one estimation per level needs to be stored and a user gets the estimation for the level that best suits that user’s role.

4.3 Implementation

This proposal was implemented in TREX using normal tables (which in TREX are called indexes) as storage. These indexes features four attributes:

- **validated** of type INTEGER (since booleans are not available but integers are highly compressed)
- **value** of the same type as the underlying data
- **attribute** of type STRING representing the name of the view attribute in which value was found
- **estimation** of type FLOAT

4.3.1 Collecting Information on Base Data

Our first task was to construct meaningful estimations. We wrote a program that, for a given BO, iterates over all base data and dumps all its distinct values as well as a count of how often each value appears. Second, TREX joined the complete BO. This may take several hours since the BOs can be quite large and the join engine of TREX is not optimized for doing huge joins. Third, the values were validated by checking how often each value actually appeared in the BO. Finally, the indexes were created, one per type.

4.3.2 Integrating the New Search Method

The new search method was directly integrated into the core of TREX. Using extensions or scripting would not have led to comparable performance results. Since we were requested not to change the API in any way, we implemented the only change to an existing function with a simple naming convention. This was the check whether to use the new method: if there exists an index with a certain name (consisting of the name of the BO plus a suffix), use the new search method. To switch back to the old search method, it was only necessary to rename or delete this index.
4.3.3 Updates

We planned to implement updates through a mechanism supported by TREX called python extensions. We planned to write a python extension to do the following: (1) catch a pending update before it is committed, (2) check whether (a) this update concerns one of the view attributes covered by the new dictionary and (b) whether it is a new distinct value for that view attribute, and (3) if both (a) and (b) hold, insert the new value into the dictionary. However, due to time constraints, we did not implement updates, and therefore we have no empirical data on the correctness of the approach or whether the proposed heuristics are useful.

4.3.4 Security Aspects

None of the proposed solutions of 4.2.4 where actually implemented, again due to time constraints. There is however a mechanism implemented which should estimate values visible for many users higher than those with more limited visability. In TREX, roles are represented by a table which is joined to every base table and the table containing the users are joined to the role tables. When crawling the data to build the initial estimations the tables containing the users are disregarded since they blow up the resulting join index. We do regard the roles table though. This means that instead of just counting occurences of a value, the number of roles for which the value occurs is relevant too. We get a sum over all the roles as an estimation which therefore will be higher when a value is visible to more roles. However, it can still happen that an interpretation is found to be the most probable which is not visible to a specific user. These interpretation will not be disregarded until the very last step of the NLI search, when the query is actually carried out.

4.4 Evaluation

4.4.1 Data Set

To evaluate the effectiveness of the changes, we created two sets of queries, one large and one small, for two BOS intended to represent typical datasets as found in practice. The larger set of queries was generated to use a large number of view attributes and was intended to test the theory that the performance gain would be more noticeable when more view attributes are present. The smaller set of queries was handcrafted because it is not trivial to generate queries similar to those a human would formulate, so we had to check the outcome of every query to see whether the interpretation found was meaningful and whether another result was expected. We did not require that the new estimation process should always favor the same interpretation as the old one. Table 4.1 gives the statistics for the two datasets.
4.4.2 Test Method

To measure the performance of our approach, we use the same test scripts as are used for other NLI tests in TREX. They measure the kernel time of the NLI search rather than the client time, which helps to give unbiased results where the communication overhead is irrelevant. The queries are executed using a round robin approach to reduce hardware related caching effects and the TREX cache is turned off. Each query is executed three times before the measurement begins to make sure that all relevant indexes are loaded correctly. After completing this initialisation step, 15 rounds of query execution are carried out. From these 15 rounds, only the median values are taken into account to correct for outliers.

4.4.3 Queries

The queries can be distinguished in multiple dimensions. The most interesting one might be query complexity. It is rather hard to define what kind of query is complex, but there are several indicators. The indicator with the most obvious performance impact is whether a query will force a wildcard search. Other indicators might be the number of words, the number of boolean operators, the presence of relational operators and whether a subselection has to be performed. Other characteristics of queries may be interesting yet not strictly related to complexity, e.g., the number of results a query produces.

Looking at Tables 4.2 and 4.3, this query may be an oddity:

who is the nl1manager of Petra

This is the only query requiring a subselection as answer. A meaningful interpretation would look something like this:

userid EQ (nl1manager of subselect first_name EQ Petra)

However, from the model we used it is nontrivial to see that it is the userid which should be matched against the result of the subselection. Both version get this wrong and return as their result:

nl1manager EQ (nl1manager of subselect first_name EQ Petra)

This interpretation selects all the people with the same nl1manager as any Petra, returning many more results than there are people called Petra in the data.

In Table 4.3, a “yes/no” entry in the wildcard column indicates that a wildcard search is done with the current implementation but not with the prototype. This difference occurs

<table>
<thead>
<tr>
<th></th>
<th>addressbook</th>
<th>material</th>
</tr>
</thead>
<tbody>
<tr>
<td>View attributes</td>
<td>74</td>
<td>351</td>
</tr>
<tr>
<td>Distinct values</td>
<td>1 024 025</td>
<td>1 115 685</td>
</tr>
<tr>
<td>Memory used (kB)</td>
<td>32 248</td>
<td>108 288</td>
</tr>
</tbody>
</table>

Table 4.1: Characteristics of the datasets
because the two systems behave differently when a term is found in the base data but does not occur in the BO.
4.4.4 Results

We ran our tests on various different TREX configurations to see whether this would lead to different results. The hardware for the tests reported below was a blade server with a 64-bit architecture mounting two dualcore Opteron™ CPUs and 8 GB RAM running under Linux. We also ran the tests on other hardware and/or operating systems. The OS had no significant influence on the test result. The results of the tests run on a 32-bit dualcore laptop can be found in the appendix.

Undistributed Queries

We assumed that due to the parallelization which can be done using the existing solution but not the new prototype, the outcome in favor of the new solution will be the best when the indexes are undistributed and therefore nothing is done in parallel.

The results shown in Table 4.4 distinguish between queries that force wildcard (wc) searches and those that do not, because the wildcard search makes up a large part of the overall time.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Time old</th>
<th>Time new</th>
<th>Lowest ratio</th>
<th>Highest ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>1331 ms</td>
<td>653 ms</td>
<td>0.82</td>
<td>8.88</td>
</tr>
<tr>
<td>material non-wc</td>
<td>443 ms</td>
<td>78 ms</td>
<td>2.1</td>
<td>10.8</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>120 ms</td>
<td>77 ms</td>
<td>0.47</td>
<td>2.64</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>105 ms</td>
<td>25 ms</td>
<td>2.68</td>
<td>6.04</td>
</tr>
</tbody>
</table>

Table 4.4: Average query times by business object, undistributed

As well as the average times, it is interesting to see how long the slowest queries need to be executed because this can decide whether a user tolerates the performance. Table 4.5 shows the results for the slowest queries.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Time old</th>
<th>Time new</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>1920 ms</td>
<td>1230 ms</td>
</tr>
<tr>
<td>material non-wc</td>
<td>994 ms</td>
<td>95 ms</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>174 ms</td>
<td>93 ms</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>183 ms</td>
<td>40 ms</td>
</tr>
</tbody>
</table>

Table 4.5: Slowest queries by business object, undistributed

Queries Distributed over Two Index Servers

For the next tests, the indexes were distributed over two index servers. It is quite a difficult problem to distribute indexes optimally, so we adopted the automatically optimized distribution suggested by TREX, although we might find a better solution for future tests.
The results are shown in Tables 4.6 and 4.7. Only a very few searches (namely those with no result that trigger a wildcard search) profited from the distribution. All other queries were slowed down due to the communication overhead.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Time old ms</th>
<th>Time new ms</th>
<th>Lowest ratio</th>
<th>Highest ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>1939</td>
<td>486</td>
<td>0.66</td>
<td>3.52</td>
</tr>
<tr>
<td>material non-wc</td>
<td>823</td>
<td>98</td>
<td>2.08</td>
<td>10.63</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>122</td>
<td>78</td>
<td>0.47</td>
<td>2.64</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>120</td>
<td>27</td>
<td>2.68</td>
<td>6.04</td>
</tr>
</tbody>
</table>

Table 4.6: Average query times by business object, distributed

<table>
<thead>
<tr>
<th>Business Object</th>
<th>Time Old ms</th>
<th>Time New ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>2120</td>
<td>1530</td>
</tr>
<tr>
<td>material non-wc</td>
<td>1294</td>
<td>295</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>180</td>
<td>93</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>210</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 4.7: Slowest queries by business object, distributed

Costs

Of course, this performance gain comes at a cost: the new indexes consume memory space, as Table 4.8 shows.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Memory old kB</th>
<th>Memory new kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>material</td>
<td>108 288</td>
<td>147 499</td>
</tr>
<tr>
<td>addressbook</td>
<td>32 248</td>
<td>52 871</td>
</tr>
</tbody>
</table>

Table 4.8: Memory usage

Decreasing Wildcard Search Time

The main performance issue with the proposed solution is clearly that wildcard searches are not getting significantly faster. These are the slowest kind of search anyway, and in some cases our solution makes them even slower than they are by the traditional search method (it is not clear to us why some searches are slower but we are investigating the matter). There are several ways to speed up wildcard search.
The straightforward way is to split the new index horizontally and distribute it over several index servers. The factor gained by this method approximately equals the number of parts into which the index was split. However, this is only possible if several index servers are available, and comes with an additional small overhead for all queries to assemble the results from the different servers.

A second possibility is to use a so-called triple index. This index is essentially a hash-map that maps character triples onto values in which those triples occur. For example, given a query for *wild*, TREX can look up “wil” in the triple index, and if there are multiple matches it can even build an intersection with the lookup for “ild”. Since the lookup only costs constant time, this makes the query time of a wildcard search essentially dependent on the length of the query term instead of the size of the new index. In practice, this approach reduced the response time the NLI query on *material* for “wildewutz” from 339 ms to 23 ms. However, the costs are correspondingly high: The hash-table dictionary, as TREX calls it, consumes about 363 MB of memory, compared to 39 MB for the new index.

### 4.5 Future Work

The next thing to do is to check the consistency of our approach under updates. The performance figures look promising, but until we know whether the solution really works under practical conditions, we cannot take the risk of deploying it. It would also be worth testing whether the solution performs well enough to support global search, meaning a search which is not constrained to one business object.
Chapter 5

Summary and Future Outlook

It was shown that there are many points in TREX where there is potential for performance gains by permitting increased memory consumption. However, most of the potential gains also have other drawbacks, most often increased update costs. Our implementation of a prototype of an improved NLI search showed promising results regarding speed. Also, it seems to be easy to maintain, although this is as yet unproven, since we did not implement updates and it is unsure whether our estimations will be accurate in the presence of many updates.

5.1 Global Dictionary

During this research, an idea emerged that is not included in our analysis because it appeared to require a project that would greatly exceed the scope of this master thesis. However, it is worth giving a sketch of the idea here as a glance into the future.

5.1.1 Idea Sketch

A relatively obvious generalization of the implemented proposal would be a similar index on a global basis. This would mean that every value in the entire system would be stored twice, namely in the new index and in the dictionary on the level of the physical tables that translate values into IDs. It seems a natural step to store the values only once, in a global directory.

This directory would assign consecutive ID numbers, since renumbering the complete dictionary below a newly inserted value would be infeasible considering the huge size of the dictionary. However, the dictionary would have to be sorted to allow searches in logarithmic time and to perform range searches. This would require that a newly inserted value be inserted at the correct place, which is a rather expensive operation.

The next problem that needs to be solved is how to look up a value given its ID. To translate efficiently in this direction, one could introduce a hash table that maps the value IDs to row numbers of the dictionary. This would give a constant time reverse lookup,
but updates would be more complicated. Whenever the row number changed, the hash
table would have to be recomputed. If the dictionary used a delta index, this could still
be feasible since the row numbers would only change at a delta index merge.

5.1.2 Positive Aspects

Such a global dictionary would have some important consequences. First, joins would be
much cheaper, since they would always be performed on integers. Second, the additional
index for NLI performance would no longer need to store values but could use the IDs. A
global version of this index could be implemented at relatively low cost.

5.1.3 Issues

The outstanding issue here is certainly performance. A global dictionary would create an
extreme bottleneck. It must be possible to distribute it over several servers and it must be
accessible parallel in a very fast manner (locking the whole dictionary is not an option).
These are certainly difficult aims to achieve and it is not clear whether this is possible
at all. A second big issue is that every value ID needs about $\log_2(maxValId)$ bits space,
which can become a problem in the presence of very many values.
Bibliography


Appendix A

Evaluation Results in Detail

This appendix contains the detailed measurements performed for the evaluation. An overview of the key figures extracted from these measurements can be found in section 4.4.4. The results for the smaller business object *addressbook* are shown in table A.1.

<table>
<thead>
<tr>
<th>Query</th>
<th>Time o(^1)</th>
<th>Time od</th>
<th>Time n</th>
<th>Time nd</th>
</tr>
</thead>
<tbody>
<tr>
<td>email of Peter Janssen</td>
<td>67</td>
<td>72</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>jobid of General Manager</td>
<td>67</td>
<td>82</td>
<td>18</td>
<td>21</td>
</tr>
<tr>
<td>first name of Beenk</td>
<td>66</td>
<td>79</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>family name of Petra</td>
<td>61</td>
<td>68</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>user id of Beenk in Walldorf</td>
<td>145</td>
<td>186</td>
<td>24</td>
<td>26</td>
</tr>
<tr>
<td>who is the m1manager of Petra</td>
<td>91</td>
<td>95</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>Developer in Walldorf with name Janssen</td>
<td>111</td>
<td>127</td>
<td>35</td>
<td>44</td>
</tr>
<tr>
<td>job title of Janssen in Walldorf</td>
<td>107</td>
<td>124</td>
<td>23</td>
<td>26</td>
</tr>
<tr>
<td>email of Petra with entry date later than 1.1.1990</td>
<td>174</td>
<td>180</td>
<td>66</td>
<td>68</td>
</tr>
<tr>
<td>user id of Petra with entry date 1.2.2000</td>
<td>155</td>
<td>173</td>
<td>61</td>
<td>63</td>
</tr>
<tr>
<td>family name of all Developer in Walldorf</td>
<td>117</td>
<td>129</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>city and job title of Janssen or Du Plessis</td>
<td>178</td>
<td>200</td>
<td>33</td>
<td>35</td>
</tr>
<tr>
<td>email and first name of all General Manager</td>
<td>142</td>
<td>165</td>
<td>26</td>
<td>29</td>
</tr>
<tr>
<td>user id with entry date later than 2005</td>
<td>183</td>
<td>210</td>
<td>35</td>
<td>40</td>
</tr>
<tr>
<td>all Janssen not in Walldorf</td>
<td>107</td>
<td>114</td>
<td>40</td>
<td>34</td>
</tr>
<tr>
<td>all cities of any Janssen</td>
<td>68</td>
<td>69</td>
<td>21</td>
<td>24</td>
</tr>
<tr>
<td>all Developer in Walldorf</td>
<td>67</td>
<td>80</td>
<td>19</td>
<td>22</td>
</tr>
<tr>
<td>wildewutz</td>
<td>44</td>
<td>39</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>all masterstudents in Schaffhausen</td>
<td>107</td>
<td>94</td>
<td>87</td>
<td>88</td>
</tr>
</tbody>
</table>

Table A.1: Median of time per query for “addressbook”

Since all the queries are answered within 100 ms with the new estimation method we decided to omit tests with wildcard-specific optimizations for the business object *addressbook*.

\(^1\)o = old undistributed, od = old distributed, n = new undistributed, nd = new distributed
APPENDIX A. EVALUATION RESULTS IN DETAIL

The results for the bigger business object material are shown in table A.2

<table>
<thead>
<tr>
<th>Query</th>
<th>Time $\sigma^2$ ms</th>
<th>Time od ms</th>
<th>Time n ms</th>
<th>Time nd ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispo of Wellendichtringe</td>
<td>276</td>
<td>312</td>
<td>72</td>
<td>104</td>
</tr>
<tr>
<td>all matnr of Arbeitsanzuege</td>
<td>274</td>
<td>328</td>
<td>73</td>
<td>108</td>
</tr>
<tr>
<td>Kabel with creationdate later than 1.5.2000</td>
<td>1784</td>
<td>3534</td>
<td>201</td>
<td>1003</td>
</tr>
<tr>
<td>Trading goods with pstat KEL</td>
<td>467</td>
<td>508</td>
<td>78</td>
<td>99</td>
</tr>
<tr>
<td>Trading things with pstat KEL</td>
<td>1163</td>
<td>1295</td>
<td>740</td>
<td>760</td>
</tr>
<tr>
<td>txtmd of Trading things with mtref HAWA</td>
<td>1513</td>
<td>1664</td>
<td>756</td>
<td>776</td>
</tr>
<tr>
<td>aenam of valpos less than 2</td>
<td>994</td>
<td>1063</td>
<td>92</td>
<td>100</td>
</tr>
<tr>
<td>matnr of aenam not MUECKLICH</td>
<td>505</td>
<td>571</td>
<td>91</td>
<td>92</td>
</tr>
<tr>
<td>matnr of MUECKLICH</td>
<td>362</td>
<td>489</td>
<td>95</td>
<td>143</td>
</tr>
<tr>
<td>matnr of everything before 1.1.2000</td>
<td>1920</td>
<td>2975</td>
<td>1230</td>
<td>1232</td>
</tr>
<tr>
<td>WIESE</td>
<td>88</td>
<td>100</td>
<td>42</td>
<td>48</td>
</tr>
<tr>
<td>matnr of Additionals</td>
<td>624</td>
<td>646</td>
<td>82</td>
<td>101</td>
</tr>
<tr>
<td>Frequency</td>
<td>394</td>
<td>358</td>
<td>80</td>
<td>86</td>
</tr>
<tr>
<td>wildewutz</td>
<td>277</td>
<td>227</td>
<td>339</td>
<td>342</td>
</tr>
</tbody>
</table>

Table A.2: Median of time per query for “material”

Besides those normal runs, tests were also made with implementation that addressed specifically the issue that wildcard searches were too slow. The improvements are shown in table A.3

<table>
<thead>
<tr>
<th>Query</th>
<th>Time $ns^3$ ms</th>
<th>Time nh ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispo of Wellendichtringe</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>all matnr of Arbeitsanzuege</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>Kabel with creationdate later than 1.5.2000</td>
<td>170</td>
<td>160</td>
</tr>
<tr>
<td>Trading goods with pstat KEL</td>
<td>89</td>
<td>88</td>
</tr>
<tr>
<td>Trading things with pstat KEL</td>
<td>428</td>
<td>100</td>
</tr>
<tr>
<td>txtmd of Trading things with mtref HAWA</td>
<td>444</td>
<td>117</td>
</tr>
<tr>
<td>aenam of valpos less than 2</td>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>matnr of aenam not MUECKLICH</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>matnr of MUECKLICH</td>
<td>101</td>
<td>102</td>
</tr>
<tr>
<td>matnr of everything before 1.1.2000</td>
<td>1229</td>
<td>1241</td>
</tr>
<tr>
<td>WIESE</td>
<td>48</td>
<td>47</td>
</tr>
<tr>
<td>matnr of Additionals</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>Frequency</td>
<td>87</td>
<td>86</td>
</tr>
<tr>
<td>wildewutz</td>
<td>185</td>
<td>23</td>
</tr>
</tbody>
</table>

Table A.3: Median of time per query for “material”, wildcard optimized

---

$^2$see footnote for A.1
$^3$ns = new horizontally splitted, nh = new with triple index
APPENDIX A. EVALUATION RESULTS IN DETAIL

It may seem strange that the query *matnr of everything before 1.1.2000* does not become faster with the different optimization techniques that focus on wildcard search since a wildcard search is in fact performed for that query. The reason for this is that the query returns ten interpretations with exactly the same estimation (at least when using the new estimation method). Therefore, ten searches need to be performed on the join index to test which one is the most relevant interpretation. These searches constitute by far the most time-consuming action regarding this query. As mentioned in 4.4.4 we also carried out the tests on other hardware than the standard blades. The test results for the runs on an Intel Core Duo™ laptop are shown in table A.4.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Time old ms</th>
<th>Time new ms</th>
<th>Lowest ratio</th>
<th>Highest ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>15184</td>
<td>5623</td>
<td>1.08</td>
<td>21.13</td>
</tr>
<tr>
<td>material non-wc</td>
<td>3114</td>
<td>410</td>
<td>1.76</td>
<td>9.2</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>1071</td>
<td>770</td>
<td>0.58</td>
<td>3.42</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>763</td>
<td>174</td>
<td>2.48</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Table A.4: Average query times by business object, on laptop

The slowest queries using the traditional query method consume an unacceptable amount of time (see table A.5). The response times become slightly more acceptable when the new estimation method is activated, however, for some queries the user still has to wait more than ten seconds, which in our opinion is still too long.

<table>
<thead>
<tr>
<th>Business object</th>
<th>Time old ms</th>
<th>Time new ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>material wc</td>
<td>23389</td>
<td>10648</td>
</tr>
<tr>
<td>material non-wc</td>
<td>5307</td>
<td>577</td>
</tr>
<tr>
<td>addressbook wc</td>
<td>1384</td>
<td>1261</td>
</tr>
<tr>
<td>addressbook non-wc</td>
<td>1444</td>
<td>351</td>
</tr>
</tbody>
</table>

Table A.5: Slowest queries by business object, on laptop