# Indexing Encrypted XML Documents in the SemCrypt Database Management System

## Diplomarbeit

zur Erlangung des akademischen Grades eines Magisters
der Sozial- und Wirtschaftswissenschaften

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirtschaftsinformatik
Data & Knowledge Engineering

Eingereicht bei: o. Univ.-Prof. Dr. Michael Schrefl
Betreut von: Mag.[a] Katharina Grün, Mag. Michael Karlinger

Verfasst von: Peter Lasinger

Linz, Juli 2006

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die Diplomarbeit mit dem Titel "Indexing Encrypted XML Documents in the SemCrypt DBMS" selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und alle benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Juli 2006

.....................................................

Peter Lasinger

# Acknowledgement

I want to thank o.Univ. Prof. Dr. Michael Schrefl for providing me with the possibility to participate in an innovative and challenging research project and for the various ideas, suggestions and the strong support.

Many thanks to Mag.[a] Katharina Grün and Mag. Michael Karlinger, whose support, knowledge, feedback and reflection was extremely important and valuable for this work.

Thanks to my colleagues that have been working in different areas of the SemCrypt project and that provided me with challenging new ideas and feedback.

Thanks to my family, for all the possibilities they have created for me and for giving me the best possible support and personal reflection.

# Kurzfassung

Indexstrukturen sind eine der effizientesten und verbreitesten Methoden um Abfragen auf großen Datenmengen zu ermöglichen. Dadurch stehen sie stets in einem engen Zusammenhang mit Datenbanksystemen. Es existieren zahlreiche Indexstrukturen, die mit speziellen Daten verwendet werden können, beziehungsweise in gewissen Anwendungsfällen den gezielten und schnellen Datenzugriff ermöglichen.

Das SemCrypt Forschungsprojekt befasst sich mit der Verwaltung und Speicherung von XML Daten in einem Outsourcing Umfeld und mit verschlüsselten Daten. Diese Umgebung beeinträchtigt die Verfügbarkeit der Daten und erfordert Indexstrukturen, um einen effizienten Zugriff zu ermöglichen.

Bisher existieren keine Ansätze, um Indexsstrukturen in einem XML Datenbanksystem einheitlich zu verwalten und zu verwenden. Um dies, sowie das Erweitern und Hinzufügen von Indexstrukturen im SemCrypt Datenbanksystem zu ermöglichen, wird ein Indexverwaltungssystem vorgestellt und entwickelt. Durch die Abstraktion von konkreten Indexstrukturen erlaubt dieses eine einfache Verwaltung und eine einheitliche Verarbeitung von verschiedenen Indexstrukturen. Durch Abstraktion von der Speicherstruktur wird weiters Unabhängigkeit von der Art der Speicherung erreicht.

Diese Arbeit beschreibt, wie bekannte Indexstrukturen adaptiert und erweitert werden können, um sie im SemCrypt Datenbanksstem einzusetzen, ohne die Sicherheit des Gesamtsystems zu gefährden. Des weiteren wird ein neuer Index vorgestellt, der die Indizierung von dynamischen Hierarchien erlaubt. Um beliebige Indexstrukturen verschachteln zu können, werden Konzepten aus dem Umfeld der Indizierung von XML und objektorientierten Daten erweitert. Das ermöglicht die Kombination der Fähigkeiten vorgestellter Indexstukturen und eine Vereinigung der Suche nach Werten und struktureller Information.

Diese Arbeit stellt nicht nur Konzepte zur Indizierung von verschlüsselten XML Dokumenten vor, sondern erläutert auch ihre prototypische Implementierung. Abschließend werden die implementierten Indexstrukturen anhand von qualitativen Kriterien und quantitativ, mittels durchgeführter Performancetests, evaluiert.

# Abstract

Index structures are one of the most efficient and most common methods to gain access to large amounts of data. Consequently they are closely interconnected with database management systems. Several index structures exist, which are optimized for special data or specific applications and which enable a targeted and fast access to data.

The SemCrypt research project addresses the database oriented management of XML documents in an outsourcing environment with encrypted data. The reduced availability of this data requires index structures to speed up the access.

So far no frameworks exist which speeds up accesses to the encrypted data and enables the integrative management and use of different index structures in XML database management systems (DBMS). This thesis proposes an extensible and flexible indexing framework for the SemCrypt DBMS. By abstracting from specific index structures, it is possible to manage different index structures in a uniform way, as well as to extend existing or to implement new index structures. Independence from the kind of data storage is reached by abstracting from the storage structure, which is important to meet the security requirements of SemCrypt.

This thesis describes how well known index structures can be adapted and extended, so that they can be used in the SemCrypt DBMS environment, without threatening the system's overall security. Furthermore a new approach to index data in dynamic hierarchies is presented. By extending the concepts of XML and object oriented indexing it is possible to arbitrarily nest index structures. This facilitates the combination of different capabilities of the presented index structures and unites the search for values and structural information.

The remaining part of the thesis describes how the presented concepts and index structures are implemented. Finally the implemented index structures are evaluated using qualitative criteria and performance tests.

# Table of Contents

1 INTRODUCTION....................................................................... 1
1.1 Motivation........................................................................................ 1
1.2 SemCrypt.......................................................................................... 2
1.3 Problem Definition............................................................................ 3
1.4 Running Example............................................................................. 4
1.5 Objectives......................................................................................... 9
1.6 Outline............................................................................................. 10
2 RELATED WORK.................................................................... 11
2.1 Index Specification......................................................................... 12
2.1.1 Lookup Domains............................................................................ 12
2.1.2 Lookup Functions........................................................................... 13
2.2 Value Centric Indexing.................................................................... 14
2.2.1 Hash Index.................................................................................... 14
2.2.2 Balanced Search Trees................................................................... 15
2.2.3 Multidimensional Index Structures.................................................. 16
2.2.4 Generalized Search Trees............................................................... 17
2.3 Information Retrieval....................................................................... 17
2.3.1 Inverted Files................................................................................. 17
2.3.2 Prefix B-trees.................................................................................. 18
2.4 Indexing in OODB............................................................................. 18
2.4.1 Class Hierarchy Indices.................................................................. 19
2.4.2 Nested Indices................................................................................ 20
2.5 XML Indexing................................................................................... 20
2.5.1 Structure Centric XML Indexing...................................................... 21
2.5.2 Content Centric XML Indexing........................................................ 21
2.6 Secure Indexing............................................................................... 22
2.6.1 Encrypted Hash Tables.................................................................... 22
2.6.2 Secure Trees.................................................................................. 23
3 INDEX STRUCTURES.............................................................. 25
3.1 Requirements and Strategies........................................................... 26
3.1.1 Applicability.................................................................................... 26
3.1.2 Extensibility.................................................................................... 27
3.1.3 Performance.................................................................................... 27
3.1.4 Security.......................................................................................... 27
3.2 Index Structure Concepts................................................................. 28
3.2.1 Index Definition.............................................................................. 28
3.2.2 Search Configuration...................................................................... 29
3.2.3 Data Structure................................................................................ 29
3.2.4 Index Configuration........................................................................ 30
3.2.5 Algorithms...................................................................................... 30

3.3 Exact Match Index.......... 30
3.3.1 Definition.......... 30
3.3.2 Data Structure.......... 31
3.3.3 Algorithms.......... 31
3.3.4 Security.......... 33
3.4 Range Index.......... 33
3.4.1 Definition.......... 34
3.4.2 Data Structure.......... 34
3.4.3 Configuration.......... 35
3.4.4 Algorithms.......... 36
3.4.5 Security.......... 40
3.5 Text Index.......... 40
3.5.1 Definition.......... 41
3.5.2 Algorithms.......... 42
3.6 Hierarchic Index.......... 44
3.6.1 Definition.......... 44
3.6.2 Data Structure.......... 47
3.6.3 Configuration.......... 48
3.6.4 Algorithms.......... 49
3.7 Nesting Index Structures.......... 54
3.7.1 Definition.......... 54
3.7.2 Data Structure.......... 56
3.7.3 Algorithms.......... 57
4 INDEX PROCESSING ARCHITECTURE.......... 59
4.1 SemCrypt Architecture.......... 59
4.2 Logical Index.......... 61
4.2.1 Index Variables.......... 61
4.2.2 Index Definition.......... 62
4.2.3 Index Configuration.......... 63
4.2.4 Search Configuration.......... 63
4.3 Internal Index.......... 64
4.3.1 Internal Index Definition.......... 65
4.3.2 Internal Index Configuration.......... 66
4.4 Physical Index Representation.......... 66
4.5 Index Processing Components.......... 67
4.5.1 Index Manager.......... 68
4.5.2 Index Engine.......... 69
5 IMPLEMENTATION.......... 71
5.1 Logical Layer.......... 71
5.1.1 Index Variables.......... 72
5.1.2 Logical Index.......... 73
5.1.3 Index Manager.......... 74
5.2 Internal Layer.......... 76

5.2.1 Internal Index ... 77
5.2.2 Nestable Internal Index ... 79
5.2.3 Access to Persistent Data for Internal Indices ... 81
5.2.4 Index Engine ... 82
**5.3 Index Specific Details ... 86**
5.3.1 Sequential Access Structure ... 86
5.3.2 Exact Match Index ... 86
5.3.3 Range Index ... 87
5.3.4 Text Index ... 89
5.3.5 Hierarchic Index ... 90

6 EVALUATION, CONCLUSION AND OUTLOOK ... 93

**6.1 Evaluation ... 93**
6.1.1 Criteria ... 93
6.1.2 Applicability ... 94
6.1.3 Extensibility ... 95
6.1.4 Security ... 96
6.1.5 Storage and Memory Consumption ... 97
6.1.6 Index Creation Performance ... 99
6.1.7 Index Retrieval Performance ... 101
**6.2 Conclusion ... 104**
**6.3 Outlook ... 104**

TABLE OF FIGURES ... 106

LIST OF TABLES ... 108

BIBLIOGRAPHY ... 109

APPENDIX ... 113

**A)Utilized Software and Libraries ... 113**
Development Tools ... 113
Java Libraries ... 113
**B)Running Example ... 114**
XML Schema ... 114
Sample XML Data ... 115
**C)Logical Index Metadata ... 116**
XML Schema ... 116
Sample Logical Metadata ... 117
**D)Internal Index Metadata ... 119**
XML Schema ... 119
Sample Internal MetaData ... 120

# 1 Introduction

1.1 Motivation........................................................................................ 1
1.2 SemCrypt........................................................................................ 2
1.3 Problem Definition........................................................................................ 3
1.4 Running Example........................................................................................ 4
1.5 Objectives........................................................................................ 9
1.6 Outline........................................................................................ 10

This chapter introduces the topic of this thesis. First the motivation and the context are outlined, followed by a brief overview on the SemCrypt concepts and general architecture. Then a running example is presented, which is going to be used throughout the thesis to apply and demonstrate concepts and algorithms. Thereafter the objectives of the thesis are explained in more detail. Finally the further structure of the thesis is outlined.

## 1.1 Motivation

Indexing has been a key methodology to retrieve information effectively and efficiently, since huge amounts of data have been collected and organized. In the past, libraries employed complex, manually maintained indices, to sustain a structure and overview over the fast growing domain of written work.

With computers, data management and retrieval reached a new level of volume and complexity. The creation, management and traversal of indices was automated. Especially the first relational database applications required new ways of efficient, targeted data access, which lead to the creation of various index structures. New tasks, like the search in full text, in hierarchical data or in multidimensional data posed different requirements to indices and lead to further, often more complex index structures.

Today, distributed DBMS and applications again call for new approaches of indexing. Simultaneously the need for annotated, structured data lead to the creation of the Extensible Mark-up Language (XML) [XML06]. The resulting requirement to store XML documents persistently and to provide them to multiple users guided the development of XML DBMS.

While technology still becomes more complex, companies start to focus on their core business and outsource “non-core” tasks. This lead to the idea of providing databases as a service [HIM02], so that an outsourcing-provider can focus on administration, recovery and backup of the database and is able to profit from economies of scale. On the other hand this approach creates various new challenges, especially related to privacy, security and performance.

The SemCrypt research project [SemCrypt] aims at creating such an outsourced database service, enabling access to encrypted XML data, stored at an untrustworthy storage provider. Index structures are an important necessity to provide this access and to allow data to be retrieved and updated efficiently [SchGD05]. Therefore this work is motivated in finding, adapting and implementing index structures that can be used in the SemCrypt setting to enable the search for values and in full text, while regarding the hierarchical structure of the XML data. To allow the integration of these index structures in SemCrypt, an adequate index management architecture needs to be designed and implemented.

## 1.2 SemCrypt

Due to the need to represent semi structured data for purposes of data storage and exchange, XML is becoming increasingly popular. The requirement to store XML documents consistently and to query them effectively calls for the use and adoption of database technology. There have been different approaches, either by adapting relational or object-relational databases to accommodate XML data, or by creating native XML databases, which use XML as the logical data model [KlMe03].

The SemCrypt research project goes one step further by realizing the database as a service model proposed by [HIM02], extending it for XML and therefore allowing secure outsourcing of XML data. However, data encryption and the requirement not to reveal any structural information to the untrustworthy storage provider, makes it difficult to access data and to process queries in an efficient way. Therefore the use of the labelling scheme described in [GKSch05], combined with the use of specific index structures is essential to facilitate navigation and targeted access to data.

As depicted in Figure 1, SemCrypt is based on the client-server concept, where part of the data and application reside in a not trusted environment. The encrypted data at the untrustworthy storage provider is accessed by the SemCrypt DBMS, which provides XML database functionality to end user applications. A more detailed description of SemCrypt's architectural design can be found in the system specification [GrKa06a].

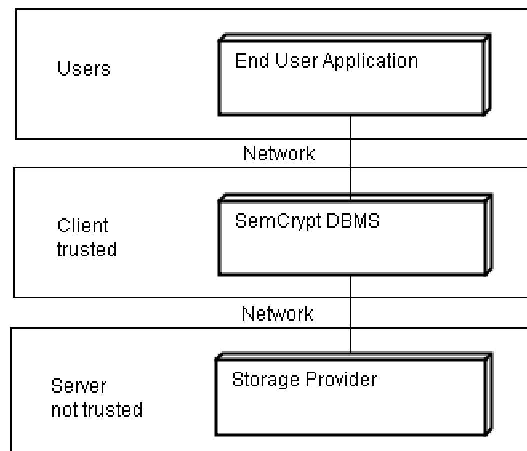



*Figure 1: SemCrypt System Setting*

SemCrypt aims at processing queries and updates on encrypted XML documents, by exploiting structural semantics of XML data, while using standard encryption techniques [SchGD05]. Whereas other approaches like [HILM02] and [Jamm04] fragment the encrypted data on the storage provider, SemCrypt does not expose any structural information to the storage provider. All data is represented as encrypted key-value pairs at the storage provider. Consequently only the trusted client knows the fragmentation structure and is able to perform queries and updates. However, this requires the client to perform tasks that are usually delegated to the server, like query processing or the management of index structures.

## 1.3 Problem Definition

The SemCrypt setting creates some challenges that need to be considered and overcome by the SemCrypt indexing framework and the embedded index structures:

1. **Distributed Environment:** Primary data and data required by index structures are persisted at an remote storage provider. Therefore the access to data is more costly and index structures need to be able to traverse their data remotely.

2. **Encryption:** Data residing at the untrustworthy storage provider are encrypted, and are only encrypted and decrypted at the client. Consequently the storage provider is only able to use primary indices on the encrypted data and cannot create and use secondary indices. Therefore the index management and traversal needs to be performed at the client. Due to encryption, access to data is slower and less efficient.

3. **Performance:** Being an XML DBMS, SemCrypt needs to provide adequate query

performance. Consequently index structures become even more important, as they can greatly influence the systems overall performance by enhancing expected and reoccurring queries.

4. **Extensibility:** The SemCrypt DBMS needs to be adaptable and must be easily extendible to fit the specific requirements of the user. This is also relevant to index structures, as existing index structures can be extended or new index structures can be implemented. Besides that it is desirable to abstract index structures from the storage structure, so that security mechanisms can be performed transparently.
5. **Security:** The major target of the SemCrypt project is to provide privacy and security of the stored XML data. Index structures need to consider security risks and to adopt adequate mechanisms to avoid any leakage of information.

The index framework and the index structures to be developed and presented in this thesis must regard these challenges. A balanced solution needs to be found, which enable the efficient access to data through indices in the SemCrypt DBMS.

## 1.4 Running Example

In order to explain and demonstrate the developed concepts and algorithms, a running example will be used, which is going to act as a line of thought throughout the whole thesis. In order to represent a use case scenario of SemCrypt, we chose the setting of an outsourced Email provider. The idea is similar to the services that are provided by companies like Google (http://mail.google.com), GMX (http://www.gmx.net) or Microsoft (http://www.hotmail.com).

However, the proposed approach is quite different, as the main objective is to provide a secure email store that can only be accessed by authorized users. Therefore users are able to read and answer their email independent from their location, while ensuring that nobody else is able to access and to read their emails. Furthermore users do not have to care about maintaining backups and gain the advantages of a sophisticated database for searches.

By using the SemCrypt DBMS, it is possible to provide this kind of service to a wide range of customers. In a technical perspective email processing is a suitable application for the use of XML, as emails do contain both structured information (header information like email addresses or time stamps) and unstructured information (the subject line, the text of the message or optional file attachments).

For reasons of clarity we focus on a simple data model (with reduced header information, lacking file attachments or additional meta-information), described by the XML-schema that is depicted in Figure 2 (see appendix B, page 114).

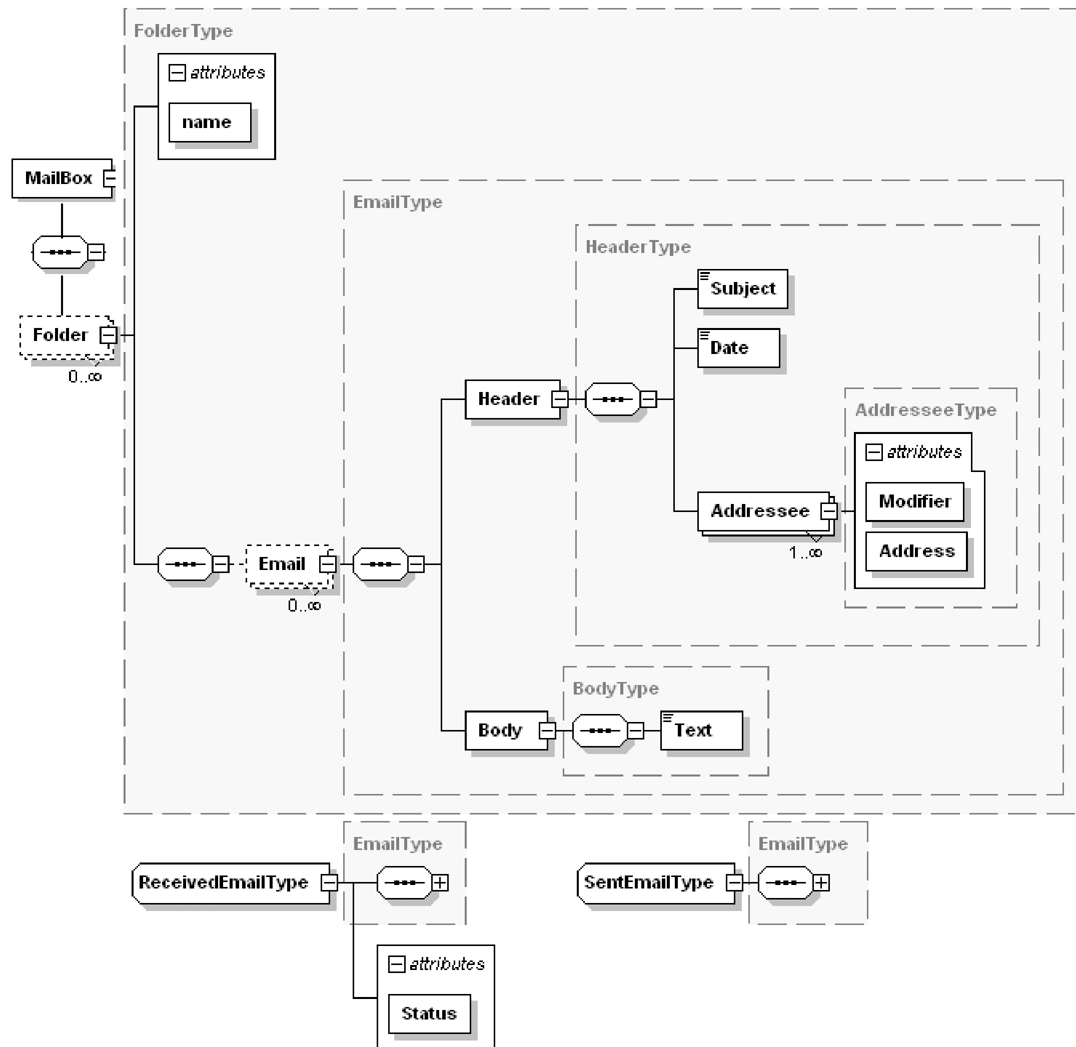



*Figure 2: Email Store - Schema*

The sample mailbox consists of different folders, which group the email messages according to the user's preferences.

An email message consists of header and body information. The header contains the subject of the email, the date when the email was sent or received, and a set of addressees. These addressees consist of an email address and a modifier telling whether the email was sent, sent as a copy or received from this address. The body contains the actual text transmitted with the email.

There are two types of emails, sent emails (SentEmailType) and received emails (ReceivedEmail Type). Received emails contain an additional status attribute, telling whether the email has been read, answered or forwarded. These two email types are generalized by an abstract email type that includes the information contained in both

sent and received emails.

For the running example three emails are used as exemplary data. The first email (A) was sent to three people in total, whereas the second email (B) is an answer to the first one. The third email (C) was sent to just one person. Their representation in XML can be found in the Appendix B, page 115, and a tree based representation is depicted in Figure 3.

For the running example the date information will be simplified by using a number that represents the time order of the three emails. So email A is assigned date 1, email B date 2 and email C date 3.

| **Email A** | **Test 1** | | **2006-05-08 \| 9:30 (1)** |
|---|---|---|---|
| From<br>To | michael@maier.de<br>peter@lasinger.at | Copy | franz@mitterer.de<br>julia@schnell.de |
| Text | *This is a little test message.* | | |

*Table 1: Running Example – Test Email A*

| **Email B** | **Re: Test 1** | | **2006-05-08 \| 17:00 (2)** |
|---|---|---|---|
| From<br>To | peter@lasinger.at<br>michael@maier.de | Copy | julia@schnell.de |
| Text | *Thanks for the email. This is my answer.* | | |

*Table 2: Running Example – Test Email B*

| **Email C** | **Test 2** | | **2006-05-09 \| 14:00 (3)** |
|---|---|---|---|
| From<br>To | julia@schnell.de<br>peter@lasinger.at | Copy | |
| Text | *This is a second test message.* | | |

*Table 3: Running Example – Test Email C*

In Figure 3 the XML nodes are depicted as rectangles, containing the name of the node (according to the schema) and attributes. XML text nodes are visualized as ellipses. The email identifier (A, B, C), which is not contained in the date itself, but is used for easier referencing the nodes is written to the right of the according email node.

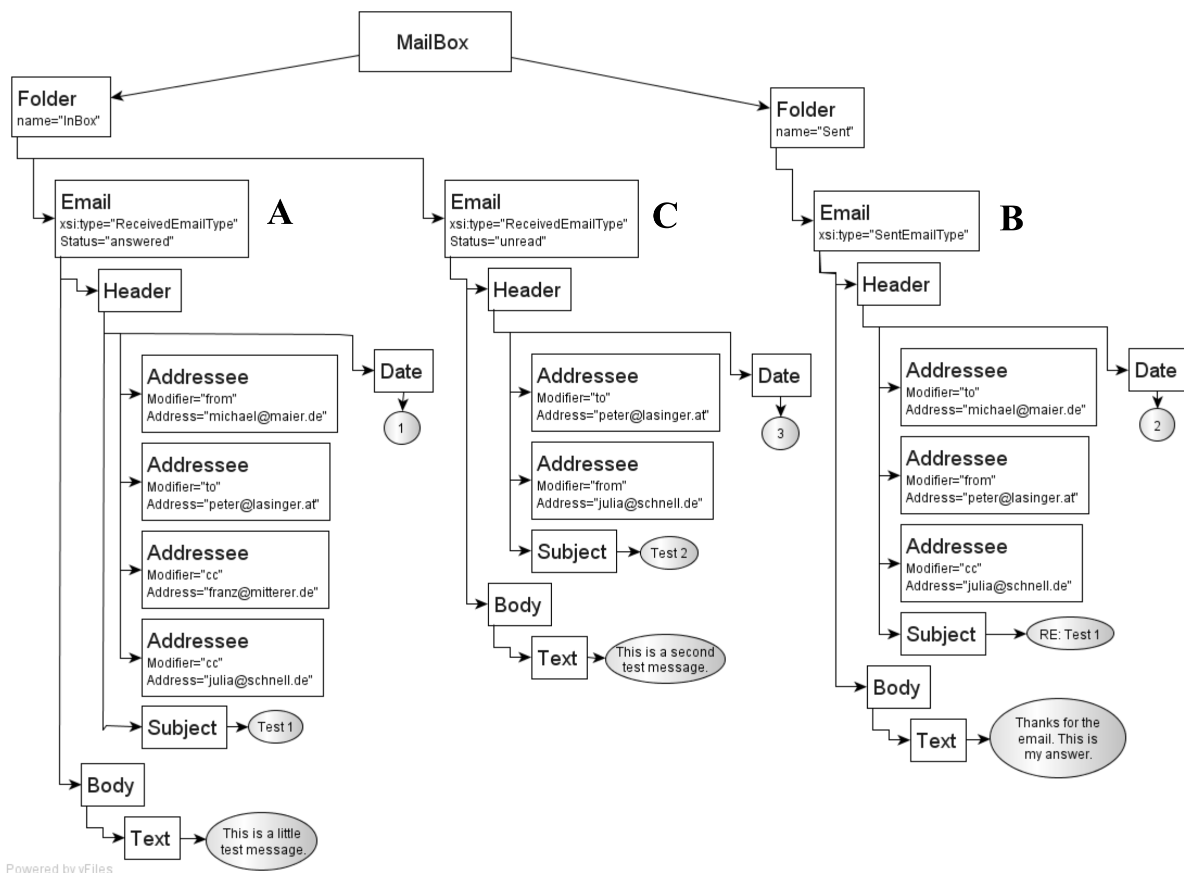



*Figure 3: Email Store Test Data*

We now define a set of typical, simple queries on this test data to show how different index structures can enable and speed up the retrieval of data. These queries are structured using five different categories, which specify common types of requests on XML data. The queries are outlined in XPath 2.0 syntax as specified by [XPath05]. For each query the expected result, based on the test data, is defined.

**Exact-Match Queries:**

Exact match queries look for data that satisfies (equals) a specific value constraint. We define two according queries on the Address attribute, which are looking for the according Emails containing this Address attribute.

1. *//Email[.//@Address="michael@maier.de"]*
   Retrieve all emails that contain the addressee with the email address "michael@maier.de", to determine the emails that where sent and received from this email address. The expected result is email A and B.

2. *//Email[.//@Address="franz@schnell.de"]*
   Retrieve all emails that contain the addressee with the email address "franz@schnell.de". The expected result is empty, as no email contains this addressee.

**Range Queries:**

Range queries extend the concept of exact match queries as they do not look for a single value but for a whole range of values. This implies that the queried data needs to follow a linear order. We define two example queries on the Date node.

3. *//Email[Header/Date<3]*
   Retrieve all emails that where sent or received before 2006-05-08 18:00 (the order 3 is used here). The expected result is email A and B.

4. *//Email[Header/Date>1 and Header/Date<3]*
   Retrieve all emails that have been sent or received on the 2006-05-08 (1) between 16:00 and 18:00 (3). The expected result is email B.

**Text Queries:**

Sometimes it is necessary to determine the occurrences of keywords or pattern in text. These kind of queries are called text queries. As an email contains textual information in its text field and subject line we define one query on the Text node and one on the Subject.

5. *//Email[contains(Body/Text,"message")]*
   Retrieve all emails that contain the word "message" in their text field. The expected result is email A and email C.

6. *//Email[.//Subject[starts-with(.,"RE:")]]*
   Retrieve all emails that were follow-ups to other emails and therefore have a subject line that starts with "RE:" (this prefix might be different in other languages). The expected return is email B.

**Hierarchic Queries:**

Until now we only considered queries restricting the value. However, as XML data contains structural information also queries regarding a structure can be posed. As the structures created by XML documents can be interpreted as hierarchies, we name these kind of queries hierarchic queries. In the following examples we query a type hierarchy

(kind of email) and a document hierarchy, created by the different Folders.

7. *//element(Email,SentEmailType)*
   Retrieve all emails that have been sent. The expected result is email B.
8. *//Folder[@name="InBox"]//Addressee/@Address*
   Retrieve all Addresses that are contained in the folder "InBox". The expected result is the four email addresses from email A and the two email addresses from Email C.

**Complex Query:**

Queries regarding structure and value restrictions can be combined in one query. We define these kind of queries as complex queries and demonstrate such a query by combining the hierarchic query for an Email type with an exact match query for the date.

9. *//element(Email,ReceivedEmailType)[Header/Date=1]*
   Retrieve all emails that have been received on the 2006-05-08 at 09:30 (1). The expected result is email A.

## 1.5 Objectives

The goal of this thesis is to provide efficient access to encrypted XML data in the context of the SemCrypt project with the use of index structures. Indices need to be created, managed and traversed efficiently at the client side, while no information about the data or the index structures must be disclosed at the storage provider.

While a framework for index management and index traversal in SemCrypt is developed, the areas of index update and index selection are beyond the focus of this thesis. Index update describes the task of keeping primary data and index structures consistent, through rebuilding or incrementally changing the index. Index selection denotes the task of selecting an appropriate index out of a set of indices, to optimally support a specific query. Though these tasks will not be particularly regarded, the indexing framework provides an abstraction that simplifies performing theses tasks and unifies various different index structures.

Index structures in SemCrypt must support a set of different query types [Grün06a]:

- Value centric queries, which search for a specific value (exact match query) or in a range of values (range query).
- Information retrieval centric queries, which search for keywords or parts of a

keyword in full text (text query).

- Queries regarding structural information that is embedded in XML documents, like type hierarchies or the document structure (hierarchic queries).
- Queries combining the search for values or keywords with the search regarding structural information (complex queries).

Hence relevant index structures need to be analysed and adopted to the specific requirements of the SemCrypt DBMS. The SemCrypt indexing framework and selected index structures providing these capabilities need to be implemented and integrated into the SemCrypt prototype.

## 1.6 Outline

The thesis is set up as follows. Chapter two presents related work on the topic of index structures and indexing methodologies and analyses relevant index structures. Chapter three describes the requirements of index structures in SemCrypt and develops specific index structures for different query types, while fitting into the encrypted environment. The general set-up of index structures and algorithms to be used are discussed as well. Thereafter chapter four introduces the general SemCrypt architecture and develops the architectural concepts used for index management and processing. Chapter five deals with implementation details and outlines how the concepts presented in chapter three and four have been implemented in the reference prototype. Finally chapter six evaluates the implemented index structures, concludes the thesis and provides an outlook on future work.

# 2 Related Work

**2.1 Index Specification........ 12**
2.1.1 Lookup Domains........ 12
2.1.2 Lookup Functions........ 13
**2.2 Value Centric Indexing........ 14**
2.2.1 Hash Index........ 14
2.2.2 Balanced Search Trees........ 15
2.2.3 Multidimensional Index Structures........ 16
2.2.4 Generalized Search Trees........ 17
**2.3 Information Retrieval........ 17**
2.3.1 Inverted Files........ 17
2.3.2 Prefix B-trees........ 18
**2.4 Indexing in OODB........ 18**
2.4.1 Class Hierarchy Indices........ 19
2.4.2 Nested Indices........ 20
**2.5 XML Indexing........ 20**
2.5.1 Structure Centric XML Indexing........ 21
2.5.2 Content Centric XML Indexing........ 21
**2.6 Secure Indexing........ 22**
2.6.1 Encrypted Hash Tables........ 22
2.6.2 Secure Trees........ 23

Chapter 2 describes work related to index structures. Relevant literature is presented, analysed and compared and the state of the art is outlined. The chapter is the foundation for further concepts and acts as a guidepost for algorithms and implementation considerations.

At first a general framework for index structures are presented. Thereafter specific index structures are described and categorized according to what type of queries they support [KlMe03]. An emphasis is laid upon indices that can be used in the setting of SemCrypt. As primary indices are defined and used only by the storage provider, only index structures that can be used as secondary indices are examined. This implies that the data associated with the secondary index need not to be sorted on the indexing key and may contain multiple values to be indexed [CoBe05].

## 2.1 Index Specification

In an environment where direct access to data is costly, as in SemCrypt, index structures enable fast, targeted access to specific data. According to Ramakrishnan and Gehrke "..., an index is an auxiliary structure designed to speed up operations that are not efficiently supported by the basic organization of records..." [RaGe00]. Elmasri and Navathe state that indices are access structures, "..., which are used to speed up the retrieval of records in response to certain search conditions" [ElNa00]. Zobel et al. describe indices more specifically as "data structures that identify the locations at which indexed values occur" [ZMR95]. Therefore indices can be defined as access structures, which act as a kind of abbreviation to specific data, considering certain constraints.

Before analysing specific index structures in detail, it makes sense to extract some general concepts and attributes that every index structure has in common. This results in an index specification that can be used to generalize, compare, and categorize index structures.

### 2.1.1 Lookup Domains

An index structure is always linked with the task of searching and locating appropriate information, thus comparing certain keys for selecting the right data [Knuth73]. Consequently an index is a mapping from a specific domain $K$ (key) to a set of occurrences $R$ (return) [MeSt99]. These domains can have different types that are outlined in Table 4:

| *domain type* | *description* |
|---|---|
| Nodes | Elements, associated with a location in a certain structure and an optional value. |
| *+ Values* | Atomic values that can be linearly ordered. |
| + Keywords | Keywords (specific string values) |
| - Patterns | Character patterns (like prefixes) of keywords. |
| *+ Structures* | Composition of an element space with quantifiable locations. |
| - Paths | Locations of a set of elements in a hierarchy. |
| - Types | Common characteristics of elements, projected in a type hierarchy. |
| - Identifiers | Pointers to specific locations (nodes) in an element space. |

*Table 4: Lookup Domain Types (adopted to the SemCrypt setting)*

Lookup domain types describe what an index can take as input and what it returns (power set of R) $K \rightarrow P(R)$ , thus defining the index and expressing the supported mappings.

Lookup functions are used to determine if an element $a$ (which can consist of multiple attributes) is contained in a set $X$ (which is the indexed set in the domain of $K$ ). Thereby a comparison operator ○ is used $f_x(a \circ x | x \in X) \Rightarrow true \vee false$ . Together with the lookup domain types of $K$ and $R$ , the lookup functions define the index specification.

## 2.1.2 Lookup Functions

There are a variety of different functions that can be used to determine if an attribute $a$ satisfies any existing attribute in a set $X$ , while making use of certain operators ○ [MWA+98]. Knuth outlines three basic types of lookups, which can be supported by index structures [Knuth73]:

a. **simple lookups**, which check for equality of a certain attribute.
   $f_{simple}(a \circ x | x \in X), \ \circ \in \{=\}$
b. **range lookups**, which query an attribute in a certain range. This requires that the attribute's lookup domain is linearly ordered.
   $f_{range}(x_1 \circ a \circ x_2 | x_{1,} x_2 \in X), \ \circ \in \{<, \leq, >, \geq\}$
c. **boolean lookups**, which combine simple and range queries with boolean operators. The total number of occurring attributes to be checked is often called dimension.

Querying structured, semi structured and unstructured (text) data leads to further types of lookups:

d. **structural lookups** consider certain structural information, like paths or types (in object oriented or XML data). This structural information can also be represented as a hierarchy. The lookup function determines if a hierarchy $a$ is contained in $X$ . The hierarchy can be directly contained (=) or it can be part of another hierarchy contained in $X$ (*isA*). $f_{structural}(a \circ x | x \in X), \ \circ \in \{=, isA\}$
e. **Text and pattern lookups**, which query for strings or patterns (expressions ). Strings can be searched and compared regarding various comparators. Clarke et al. [CCB94] present an algebra for text search and also define a set of comparison operators, like *contains* or *startsWith*. These operators can be used in the lookup function definition: $f_{text}(a \circ x | x \in X), \ \circ \in \{=, contains, startsWith, ...\}$

It can be necessary to perform queries regarding multiple attributes of an element at the same time. Ahn et al. [AMW01] presents some of these cases, like queries for nearest neighbours, distance or contains queries. Therefore an additional look-up type is needed, which makes it possible to compare vectors of attributes and which extends the boolean look-up described before:

f. **Multidimensional look-ups**

$$f_{multidim}(\begin{pmatrix} a_1 \\ a_n \end{pmatrix} \circ \begin{pmatrix} x_1 \\ x_n \end{pmatrix} | x_1 \in X_1 .. x_n \in X_n), \quad \circ \in \{=, <, \leq, >, \geq\}$$

The different lookup types, their associated comparison operators and a short cut for the lookup type are depicted in Table 5. The complex look-ups (boolean and multi-dimensional) are not shown, as they can dissected in multiple simple lookup functions.

| *Short cut* | *Lookup type* | *Comparison Operators* |
|---|---|---|
| *simple* | Simple lookup | = |
| *range* | Range lookup | =, <, >, ≤, ≥ |
| *keyword* | Keyword lookup | contains, startsWith |
| *pattern* | Pattern lookup | matches |
| *structure* | Structural lookup | =, isA |

*Table 5: Lookup Function Types and Comparison Operators*

## 2.2 Value Centric Indexing

Value centric index structures map from atomic values to the place of their occurrences. This type of indices is most commonly used in RDBMS (relational database management systems) [ElNa00], [CoBe05], but also highly important for object oriented and XML database systems [KlMe03]. Representatives for one dimensional (one attribute set) value centric indices are balanced search trees and hash indices. Partitioned hash indices, grid files, bitmap indices and special multidimensional search trees represent multidimensional value centric indices.

### 2.2.1 Hash Index

In a hash index the location (address) of a page is calculated using a hash function, which is chosen in such a way that records are evenly distributed throughout the file [CoBe05]. Consequently retrieving a specific record is usually possible with one access. However, depending on the index size and the hash function, collisions may occur, leading to overflows and slowing down the index. An extensive analysis of hashing and

according algorithms is given by Knuth [Knuth73].

$$Values \rightarrow P(Identifiers)$$
$$f_{simple}, \quad \circ \in \{=\}$$

*Figure 4: Hash index interface*

As depicted in Figure 4 a hash index maps a specific value to a set of occurrences (identifiers). Although hash indices have excellent update and retrieval characteristics, there exist several limitations. For example, hash indices are only able to perform simple look-ups (checking for equality). Other restrictions can be overcome by extended hash techniques, like dynamic hashing (letting the hash index (hash function range) grow and shrink dynamically) or partitioned hashing (to allow hashing of multiple attributes, see [KlMe03]).

### 2.2.2 Balanced Search Trees

Balanced Search Trees (B-trees), first presented by Bayer and McCreight [BaMc72], have become a broadly used, standard access method in various database systems. Due to their good performance characteristics, their dynamic grow and shrink behaviour and the ability to be used in multi-user environments, most database systems use B-trees as a secondary access structure [Com79].

There exist many variants of B-trees, the most prominent being the B+tree and B*-tree [HeSt78]. These variants mainly differ in how compact they store entries, how they perform splits and merges and the structure of link- and leaf-nodes. Consequently these differences result in various retrieval, update and storage characteristics. For example the leaves in a B+tree are linked together forming a "sequence set" [Com79], which facilitates very efficient range queries.

A detailed explanation of several B-tree variants and according algorithms can be found in [Com79] and [RaGe00]. Variations of B-trees can also be used in information retrieval, for example the Prefix-B-tree, which will be discussed in more detail in Chapter 2.3.2.

The advantage of a B-tree structure lies in its balanced structure, which is achieved by using algorithms for insertion and deletion that keep the tree balanced. Therefore B-trees can be updated incrementally. Furthermore the retrieval costs stay low and can be predicted (logarithmic to the total node count and equalling the current height of the tree).

$$Values \rightarrow P(Identifiers)$$
$$f_{simple} \vee f_{range}, \quad \circ \in \{=, <, \leq, >, \geq\}$$

*Figure 5: B-Tree index interface*

As depicted in Figure 5 the look-up pattern of B-trees is a mapping from a set of values to a power set of attached locations. A B-tree supports simple and range look-up functions.

Due to their dynamic update behaviour and balanced structure, B-trees are a good candidate to enhance range and exact match queries in SemCrypt. However, as they cannot be used in their original form in encrypted environments, an approach to traverse tree-like structures at the client side, while preventing the leakage of information, is presented in Chapter 2.6.

### 2.2.3 Multidimensional Index Structures

Multidimensional index structured can be classified by how many dimensions they support, while providing adequate performance, the internal structure they use and what look-up functions they support. According to Böhm et al. [BBK+00] index structures like the Grid File [NHS84], the KdB-tree [Rob81] or the R*-tree [BKS+90] are only suitable for low dimensional data. For high dimensional data X-trees [BKK96], Pyramid-trees [BKK98] or UB-trees [Baye96] can be used. Another index structure suitable for multidimensional data, especially when the domains contain a low number of possible values, is the bitmap index [CoBe05].

Regarding the tree-structured index structures, one can distinguish two approaches, the first one tries to accommodate multidimensionality by relying on a special tree structure (KdB-tree, R*-tree, X-trees), whereas the second one maps multiple dimensions to one dimension and relies on standard B-trees (Pyramid-tree, UB-tree).

The supported look-up functions depend on the specific index structures. For example a bitmap index does only support exact match look-ups, while the tree-structured also supports range queries.

A further discussion of range queries in multidimensional data can be found in [BeFr79] and Gaede and Günter analyse the various multidimensional access methods extensively in [GaGü98]. Another classification of multidimensional access methods together with an overview on different index structures and their performance characteristics is given by Ahn et al. [AMW01].

### 2.2.4 Generalized Search Trees

An interesting approach to unify different tree structured indices in a general framework was made by Hellerstein et al. [HNP95]. They introduced a generalized search tree (GiST), which is extensible regarding query types (lookup functions) and data types (lookup pattern domain types). The authors outline that the essential nature of any database search tree is the explicit partitioning of a dataset.

Therefore the GiST provides an abstraction of the search key, which is defined as "any arbitrary predicate that holds for each datum below the key" [HNP95]. Also the comparison function (named key methods by the authors) on these keys and the split functionality is abstracted and exposed to the user. This ensures that various different tree structured indices can be handled in the same way, an approach that is very interesting for SemCrypt, to ensure extensibility and a meta-view on index structures.

## 2.3 Information Retrieval

Unlike in value centric indexing, in information retrieval text is accessed by using keyword search, pattern matching and ranking techniques. As information retrieval is a very complex area on itself and is not a primary focus of this thesis, only some basic underlying index structures will be discussed.

There are many different approaches mentioned in the literature, like Tries and Patricia Trees [Knuth73], Suffix Trees [Ukko95], Inverted Files or Signature Files [ZMR98]. Exemplary two index structures, which provide keyword matching (inverted files [KlMe03]) and prefix matching (prefix B-trees [BaUn77]) functionality, are presented.

### 2.3.1 Inverted Files

One straightforward and commonly used approach to index large amounts of full text data are inverted files [KlMe03]. These consist of a dictionary containing all indexed keywords and inverted lists, containing references to the occurrences of these keywords (including word or character positions).

The index interface of an inverted file resembles the one of a hash index, which is depicted in Figure 4. Therefore a hash index can be used for the implementation of an inverted file, by calculating hash addresses from the keywords and by storing the relevant location information. A detailed analyses of inverted files and its variations and a comparison with signature files can be found in [ZMR98].

### 2.3.2 Prefix B-trees

Another possibility to enable text search is by the use of B-trees that use keywords or patterns as index keys. Bayer and Unterauer present such a tree structure, called prefix B-tree [BaUn77]. They index keywords, however do not store the whole key, but compress it, maintaining the linear order between keywords. This is possible due to a prefix based compression.

The algorithms for insertion, deletion and retrieval equal the ones for regular B-trees, however for the creation of branch pages, not the full indexed string is used but the smallest prefix that can be used to separate two strings. For example if the two words "*toast*" and "*tree*" are indexed, the algorithm looks for a minimal prefix separating the two words, which in this case is "*tr*". In the best case one letter is sufficient to distinguish two words, in the worst case the whole second word needs to be taken.

While this key-compression saves memory space, the authors do not explicitly mention the positive side-effect regarding text-queries, namely to enable the search for prefixes. Therefore a prefix B-tree combines the advantages of a Patrica Tree [Knuth73] (prefix search, key compression) with the one of a B-tree [Com79] (balanced behaviour).

## 2.4 Indexing in OODB

Data can not only be queried for values, but also regarding an underlying structure. The problem of indexing data embedded in a hierarchic structure first occurred with the creation of object oriented databases (OODB). Hierarchic data follows an IS-A relationship that can be found in class- (object orientation) or type hierarchies (XML types). Bertino et al distinguish three approaches that can be taken to index hierarchic data [BCC98]:

1. Firstly, index structures can be adopted to accommodate values and hierarchic information, by segmenting parts of the index according to hierarchies. Most index structures that follow this approach (like the hierarchy class chain index (HCC-index) [SrSe94]) have been defined as extensions of the class hierarchy index (CH-index) proposed by Kim et al. [KDD89].

2. Secondly, index structures can be nested to represent hierarchic information. This approach has been chosen by Low et al. [LOL92], with the hierarchical tree (H-tree). They suggest maintaining a tree for every class that are then interlinked to capture the inheritance relationships.

3. The third approach is to use a multidimensional index structure, while regarding

hierarchic information as an additional dimension. Ramaswamy and Kanellakis [RaKa95] follow this approach by transforming the search in a class hierarchy and a value domain into a range search regarding two dimensions. Therefore they map classes to ranges and represent subclass relationships with contains dependencies.

$$Values \times Hierarchy \rightarrow P(Identifiers)$$
$$f_{simple}(a_v \circ_1 x_v \wedge a_h \circ_2 x_h \mid x_v \in Values \wedge x_h \in Hierarchy), \quad \circ_1 \in \{=\}, \quad \circ_2 \in \{isA, =\}$$
$$f_{range}(x_{v1} \circ_3 a_v \circ_3 x_{v2} \wedge a_h \circ_2 x_h \mid x_{v1}, x_{v2} \in Values \wedge x_h \in Hierarchy), \quad \circ_3 \in \{<, \leq, >, \geq\}$$

*Figure 6: Hierarchic index interface*

The index interface of a hierarchic index structures is depicted in Figure 6 and shows the combination of a value with a hierarchic domain. A hierarchic relationship can either be determined by an equal comparison (=) or by a sub-hierarchy comparison (isA).

In the following two chapters the first two approaches will be presented as approach three makes use of multidimensional index structures that have already been described in Chapter 2.2.3.

### 2.4.1 Class Hierarchy Indices

Kim et al. were the first to propose a single index on an attribute in a class hierarchy, called CH-index, replacing many indices maintained for each single class [KDD89]. The authors adopt a standard B-tree to accommodate the structural information contained in a class hierarchy. While the branch nodes are identical to usual B-trees, the leaf nodes are segmented into the according classes. This allows efficient queries on a single attribute over a class hierarchy, however is less efficient for queries on a single class.

The authors also present an overflow mechanism, which splits records regarding to which class they belong. This ensures that in case a single class is queried, a minimum amount of pages needs to be loaded. It has been shown that the efficiency of the CH-index increases with the number of classes, compared to many indices maintained for single classes. This is true for both exact match and range queries. However, as mentioned before, a single index outperforms the CH-index when single classes are queried.

A slightly different approach (HCC-index) has been taken by Sreenath. and Seshadri [SrSe94], who adopts a standard B-tree similar to the CH-index. However, the class structure is not represented in the leaf nodes, but in a separate, underlying layer called oid nodes. Also the branch pages are extended in a way that they store an additional bitmap vector, which allows determining on branch level if records exist for a certain class. The authors show that the HCC-index outperforms the HC-index in case a single

class, or few classes in a hierarchy are queried, while performing equally well on many classes.

### 2.4.2 Nested Indices

A straightforward approach to index class hierarchies is to maintain a separate index for every class and to search some or all of these index structures when evaluating a query. Although this approach is ideal to answer queries on a single class, the performance decreases when querying multiple classes or the whole hierarchy, as all indices need to be traversed.

To overcome this limitation, Low et al. [LOL92] presented an index structure called H-tree. The main idea is to maintain an H-tree for each class, "allowing efficient search on a single class. These H-trees are appropriately linked to capture the superclass-subclass relationships, thus allowing efficient retrievals of instances from a class hierarchy" [Ooi+96]. Due to the nesting of the single H-trees queries on multiple classes can be evaluated efficiently. While the performance of nested H-trees is roughly equal to HCC-indices, Sreenath and Seshadri outline that the maintenance of an H-tree is more complex and difficult [SrSe94].

## 2.5 XML Indexing

Semi-structured data, like XML data, contains additional structural information. However, this structure is often more flexible than the one found in class hierarchies and contain larger text junks, which need to be indexed using information retrieval techniques. An extensive survey on XML indexing and searching was done by Kuk et al. [LLD+02] and several XML index structures are described by [CMV05].

Regarding indexing techniques for XML, they can be segmented into two categories:

- Content Centric XML Indexing
- Structure Centric XML Indexing

Content centric approaches first segment the indexed data regarding the content of the XML file, like text or values, whereas structure centric approaches first model the content structure and then index the content. These approaches either combine a regular index on content with a specific index on structural information (as implemented by McHugh et al. [MWA+98] in the Lore DBMS), or use specific index structures suitable for this task.

A special case is regarding both content and structural information at the same time, which can by achieved with a multidimensional approach [Krat04]. An example of this proceeding has been presented by Kratky et al. [KPS02], who use the multidimensional UB-tree [Baye96] for indexing XML data.

### 2.5.1 Structure Centric XML Indexing

The structure of XML data often is not known, or it changes dynamically, so that it is not possible to define a static schema which encodes structural information. Therefore Goldman and Widom present a dynamic schema, which they call *data guide* [GoWi97]. A data guide can be created on any XML document and represents the current structure of this document in an unbalanced tree structure. Every document can have several valid data guides, which can be reduced to a minimal data guide. Besides their use for queries on structural information, data guides can be used for query formulation and -optimization.

Another structure centric approach is the T-index proposed by Milo and Suciu [MiSu99]. A T-index is a path index, which makes use of path templates defining a set of paths that can be indexed and evaluated. The advantage compared with the data guide is the capability of a T-index to evaluate multiple paths at the same time (as long as they fit to the specified path template). Like the data guide, the T-index only indexes the XML document structure.

Cooper et al. [CSF+01] follow a different strategy and suggest to interpret paths as character-chains and then use nested Patrica Trees [Knuth73] to index these paths. The created *Index Fabric* is a balanced structure capable of supporting complex and branching queries.

### 2.5.2 Content Centric XML Indexing

There exist several examples for content centric XML indexing. As mentioned before one approach is to used regular value based index structures (hash table, B-tree,...) and to combine them with a structural index [MWA+98]. This approach is often found in combination with information retrieval tasks, when keyword and pattern matching queries shall be supported on XML documents. Poola and Haritsa [PoHa01] propose an index structure (SphinX) that combines a tree, representing the document structure, with multiple balanced trees on values, one for each path.

Another possibility is the use of bitmap indices, mapping values to bitmaps. This is done by Yoon et al. [YRC01], who also encode structural information in bitmaps and create a

three-dimensional “bitcube” for indexing content and structure in XML documents.

An interesting approach has been presented by Weigel et al. [WMB+04] that extend the date guide to make it content aware and suitable for information retrieval, while preserving the structural information. They distinguish two approaches, a content- and a structure-centric one, depending on what information will be used first for segmentation. The authors prefer the second one, as the content-centric approach would replicate the document structure for every indexed keyword. A *content aware data guide* is enriched with content information to exclude non relevant sub-trees (that do not contain the desired content) early. One way to achieve this is adding inverted files to every node in the data guide, another one is to represent the content of sub-trees by signatures in the data guide.

## 2.6 Secure Indexing

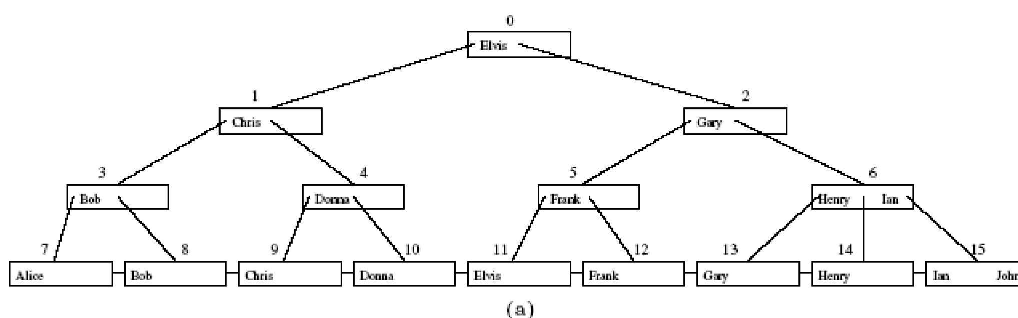
A consequence of the database as a service paradigm is the importance of data security and information hiding. Until recently there were no approaches to support queries over data, which cannot be read by the server with index structures. This setting requires the client to create, maintain and traverse indices. Additionally indices must not reveal their structure or the data they contain to the storage provider.

In the following chapters two approaches for secure indexing with hash based and tree based index structures are presented.

### 2.6.1 Encrypted Hash Tables

According to Damiani et al. [DVJ+03], Dang [Dang04] and Ceselli et al. [CDV+05] security and efficiency are always in a trade-off relationship. They propose to store data required by index structures (index pages) in an encrypted hash table. The hash index allows direct access for exact match queries. For B-trees the authors suggest to traverse the tree on the client, therefore accessing encrypted entries at the storage provider and decrypting them for further traversal.

A sample B-tree and the according hash tables in a decrypted and an encrypted version is depicted in Figure 7. Each row of the hash table represents one index page which is fetched from the storage provider.




B+-tree Table

| ID | Node |
|---|---|
| 0 | (1,Elvis,2,_,_) |
| 1 | (3,Chris,4,_,_) |
| 2 | (5,Gary,6,_,_) |
| 3 | (7,Bob,8,_,_) |
| 4 | (9,Donna,10,_,_) |
| 5 | (11,Frank,12,_,_) |
| 6 | (13,Henry,14,_,_) |
| 7 | (Alice,_,8) |
| 8 | (Bob,_,9) |
| 9 | (Chris,_,10) |
| 10 | (Donna,_,11) |
| 11 | (Elvis,_,12) |
| 12 | (Frank,_,13) |
| 13 | (Gary,_,14) |
| 14 | (Henry,_,15) |
| 15 | (Ian,John,_) |

Encrypted B+-tree Table

| ID | C |
|---|---|
| 0 | SeCS0U/7ZIY.A |
| 1 | /WKu5y8laqK82 |
| 2 | jzKzVi0D1as8E |
| 3 | AXYaqohgyVObU |
| 4 | IUf7R.PK5h5fU |
| 5 | rzaslxohWS2l2 |
| 6 | EXITGCUlYTVBc |
| 7 | uOtdm/HDXNSqU |
| 8 | GLDWRnBGIvYBA |
| 9 | a9yl36PA3LeLk |
| 10 | H6GwdJpXiU8MY |
| 11 | uOtdm/HDXNSqU |
| 12 | zj33kVaNvLFVk |
| 13 | V9rMw904cix3w |
| 14 | xTFcWtd6.IE.A |
| 15 | ji.gtDER6Hjis |

*Figure 7: Encrypted B-tree using hash tables, example from [DVJ+03]*

### 2.6.2 Secure Trees

Although the approach using encrypted hash tables looks secure at first, Lin and Candan [LiCa04] argue that the different frequency of accessing nodes may lead to a reconstruction of the tree structure, breaching the overall security. For example the root node is accessed most often and therefore can be easily identified.

The authors propose two techniques, access redundancy (depicted in Figure 8) and node swapping (depicted in Figure 9), which prevent the leakage of structural information when used in combination Access redundancy means that if certain data needs to be retrieved from the storage provider, additional random data is requested, disguising the request from the storage provider.

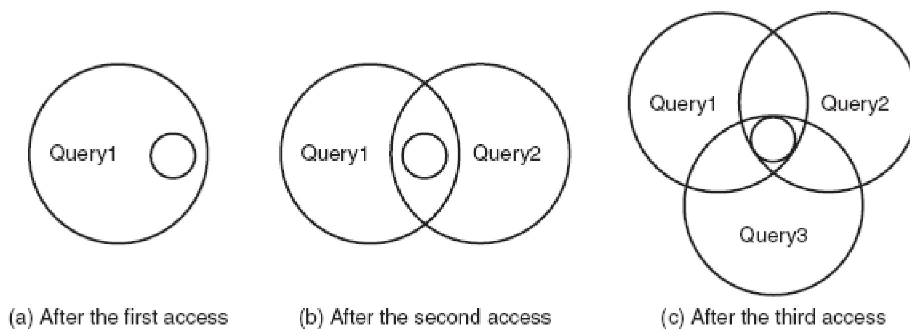



*Figure 8: Access redundancy for hiding tree structure, [LiCa04]*

While access redundancy hides which node of a tree is accessed, information can still be revealed in case multiple queries target the same node (as shown in Figure 8). Therefore the authors propose to move the accessed node after every access. The retrieved data needs to contain at least one empty that is used for the swapping.

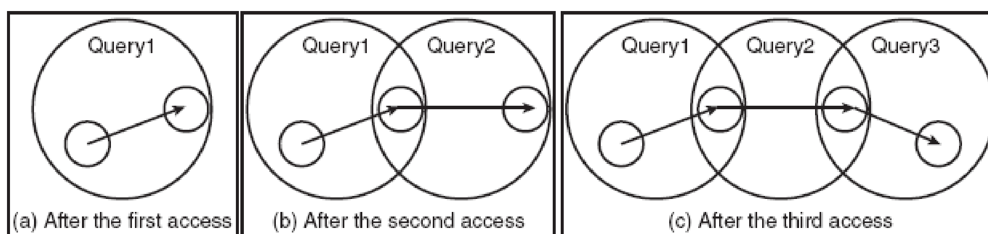



*Figure 9: Node swapping for hiding tree structure, [LiCa04]*

As depicted in Figure 9, when combining node swapping with access redundancy the accessed information can be concealed. However, these two techniques require a lot of additional processing and generate transaction overhead. Furthermore, due to the node swapping, every read access results in a write access which slows down performance.

# 3 Index Structures

**3.1 Requirements and Strategies ... 26**
3.1.1 Applicability ... 26
3.1.2 Extensibility ... 27
3.1.3 Performance ... 27
3.1.4 Security ... 27
**3.2 Index Structure Concepts ... 28**
3.2.1 Index Definition ... 28
3.2.2 Search Configuration ... 29
3.2.3 Data Structure ... 29
3.2.4 Index Configuration ... 30
3.2.5 Algorithms ... 30
**3.3 Exact Match Index ... 30**
3.3.1 Definition ... 30
3.3.2 Data Structure ... 31
3.3.3 Algorithms ... 31
3.3.4 Security ... 33
**3.4 Range Index ... 33**
3.4.1 Definition ... 34
3.4.2 Data Structure ... 34
3.4.3 Configuration ... 35
3.4.4 Algorithms ... 36
3.4.5 Security ... 40
**3.5 Text Index ... 40**
3.5.1 Definition ... 41
3.5.2 Algorithms ... 42
**3.6 Hierarchic Index ... 44**
3.6.1 Definition ... 44
3.6.2 Data Structure ... 47
3.6.3 Configuration ... 48
3.6.4 Algorithms ... 49
**3.7 Nesting Index Structures ... 54**
3.7.1 Definition ... 54
3.7.2 Data Structure ... 56
3.7.3 Algorithms ... 57

This chapter describes the index structures that are implemented in this thesis. At first the requirements posed at the specific indices are outlined. Then the concepts and algorithms of the index structures are presented and the required meta-data is explained. Finally the concept of nesting index structures in SemCrypt is outlined. No

implementation details will be given, as these are presented in Chapter 5.

## 3.1 Requirements and Strategies

Being an outsourced database service, SemCrypt has high requirements regarding performance and security. These requirements result in limitations and constraints for index structures that need to be regarded and which require new strategies. As mentioned before index structures help to speed up certain types of queries. However, in SemCrypt they must not hazard the system's overall security.

Several requirements need to be fulfilled by these index structures to fit into the SemCrypt architecture and to satisfy the SemCrypt requirements:

### 3.1.1 Applicability

The index structures need to support certain queries, that can be described as lookup functions as outlined in Chapter 2.1.2. The kind of typical queries that are executed on XML data have already been outlined in Chapter 1.4 and 1.5. By mapping these query types to lookup types, the index structures which are required to index XML documents can be deduced. The according mapping is shown in Table 6.

| *Query Type* | *Index Lookup Type* | *Proposed Index Structure* |
|---|---|---|
| Exact match queries | Simple lookup | Exact Match Index,<br>Range Index |
| Range queries | Range lookup | Range Index |
| Text queries | Keyword lookup | Text Index |
| | Pattern lookup | |
| Hierarchic Queries | Structural lookup | Hierarchic Index |
| Complex Queries | Boolean lookup | Combination of Indices,<br>Multidimensional Index |
| | Multidimensional lookup | |

*Table 6: Query Types, Lookup Types and Associated Index Structures*

The proposed index structures are selected and explained in [Grün06a]. Simple lookups are supported by an exact match index or a range index and range lookups are supported by the range index. Keyword lookups can be supported by a text index (pattern lookups will not be supported). The hierarchic index supports structural lookups. Regarding the boolean and multidimensional lookups, a combination of structural with simple, range or keyword lookup can be supported by nested indices. Multidimensional indices, which are able to support boolean and multidimensional lookups are not considered in this thesis.

### 3.1.2 Extensibility

Extensibility means the capability of adding new index structures at a later point of time, to adopt and extend existing indices (new data types) and to combine the capabilities of different index structures. Consequently extensibility results from the performed abstraction from the storage structure (independence from underlying concepts) and the index structures (abstraction provided to overlying concepts).

This quality attribute is provided by the proposed representation of indices on a logical and internal layer, which generalizes index structures and provides common interfaces for managing and traversing indices (details will be given in Chapter 4).

### 3.1.3 Performance

Performance of index structures can be measured regarding the amount of storage accesses and amount of data they transfer, the memory and processing time they consume and the amount of meta-data they require. Performance can be differentiated into the performance of retrieval and the performance of maintaining (updating and creating) an index.

The reason for these performance requirements lies in the expensive communication with the storage provider. Encryption and decryption create an additional overhead. Therefore the number of accesses and the total amount of transferred data needs to be reduced.

Primary target is to minimize the transferred data and to provide adequate retrieval performance. This can be achieved by the use of a dynamic bucketing mechanism that ensures that similar sized pages are transferred and that reduces the total amount of data transferred to answer a query. Memory and processing time consumption and maintenance performance is second important.

### 3.1.4 Security

Generally speaking indices must not reveal any information about the data they index or the structure of the index at the storage provider. Index structures potentially threaten the system's security, as they duplicate the data they index. Furthermore structural information that can be retrieved by analyzing indices may be used to reconstruct the structure of the primary data (this is because indices always segment similar data in some way).

Security issues arise when querying and updating an index. The redundancy created by index structures is an advantage when querying data, as not only the primary data are accessed. Consequently it becomes more difficult to map a certain query to a certain set

of data. On the other hand updates are more problematic, as a change of the primary data is always reflected by according changes in the index data.

Security is primarily ensured by making indexed data indistinguishable from primary data, using secure encryption techniques and caching frequent accesses.

## 3.2 Index Structure Concepts

As outlined in Chapter 2.1 index structures share some common characteristics. Concepts relevant to all presented index structures are presented in this subsection and the created structure is re-used in the chapters describing the specific index structures.

An index is an access structure that realizes a mapping between certain data and its occurrences. According to Chapter 2.1, we call every domain that is indexed and that can be queried a **key** and the domain of data that is returned the **return**. Consequently an index maps keys to a return. In the SemCrypt DBMS the return of index structures are always nodes, the core element of any XML document in SemCrypt.

### 3.2.1 Index Definition

The kind of mapping of keys to a return for a certain index is described by an index definition, in a way that is independent from a specific index structure. The index definition defines what the index is based on (keys), what is returned (return) and dictates the structure of the lookup function, which is used to query the index. When querying the index, the keys can be regarded as variables, which when set to a specific value will lead to an according return.

> For example if we want an index that returns every email that was sent at a specific date, the index needs to map Date --> Email(s). This can be expressed more formally by the modified XPath statement:
>
> *//Email[Header/Date=$var]*
>
> Hereby Date means the value of the Date node, which is the key of the index. Email is the return, in this case a set of Email nodes. If the key variable ($var) is set to a specific value the index returns the according emails.

We require additional information to determine what kind of queries we can pose at the index. In Chapter 2.1.2 according lookup types and comparison operators that can be used to indicate this information have been presented.

In the example we define that the index supports match queries on the Date. We encode this information (the kind of comparisons that can be performed) in a meta-type of the variable. Therefore a *"simple"* indicator (short cut for the simple look up type in Table 5) is added to the variable:

*//Email[Header/Date=$\$var_{simple}$]*

### 3.2.2 Search Configuration

In order to query an index we create variables for every key of the index and set these variables to specific values, ranges, keywords or hierarchies. We call this set of variables passed to an index the search configuration, as it defines what an index is looking for. A search configuration can also be used to specify the keys, where to insert, update or delete nodes.

We set the variable to the value we want to search for: $\$var_{simple} := 1$. In this case the search configuration only consists of one variable. The index returns all emails that satisfy the specified condition (Date = 1).

### 3.2.3 Data Structure

Every index structures its data in a certain way, which ensures that a specific mapping and an according lookup function are best supported. However, the SemCrypt environment poses certain limitations to this structure, as the SemCrypt DBMS only supports storing of a mapping of identifiers to data. Consequently index structures need to access and to write structured packages of data, which we call pages.

Pages are the persistent representation of the mapping imposed by the index. The sum of pages belonging to an index contains all the indexed information of this index. As an index makes use of multiple pages, these pages contain both the indexed data and additional data that is required for the traversal of the index, like references to other pages.

A sample page might contain a date and all the email nodes belonging to this date: [ 1 --> A ]

Every index must make use of pages. Each page consists of an identifier that can be used to access the page and content. The structure of the content is not compulsory and can be defined by every index structure. Therefore pages might even contain other pages.

### 3.2.4 Index Configuration

Often an index requires additional meta-data that determine its structure and characteristics. The sum of these parameters is called the index configuration. A configuration contains information that is required for initializing an index, but also information that is needed for the operation of the index.

Every configuration needs to contain meta-data that expresses the relationship of the key indexed by the index structure and the variable defined in an index definition. This information enables the index to identify its affected keys in case of a query.

### 3.2.5 Algorithms

Every index has a certain way of handling indexed data, which we call the algorithms of an index. These algorithms are required to create, update, delete and query the index. They are independent from the indexed data. Then again there are algorithmic components that are data-specific and which are used for the comparison and manipulation of this data. We are going to call these data-specific algorithms operators.

A sample insertion algorithm is: load page --> check equals key --> insert value --> save page. In case of a page [ 1 --> A ] and the insertion of [ 1 --> B ] this leads to [ 1 --> A, B ].

The required operator is the equals (=), as the algorithm needs to compare the key of the existing page to the key of the data to be inserted. In this case the equals operator compares two numeric values (1 = 1).

## 3.3 Exact Match Index

The exact match index provides the value centric indexing capabilities that were discussed in Chapter 2.2, for simple (exact match) queries. Considering the encrypted SemCrypt storage approach and the performance characteristics we chose an inverted file approach, which is capable of emulating a hash based approach [SchGD05].

### 3.3.1 Definition

An exact match index supports simple lookup functions, which means it is able to retrieve nodes belonging to a specific key. This can be expressed by passing an index variable, which contains the specific key that shall be retrieved, or that defines the key of the node to be inserted or to be deleted.

**Index 1**

We define a sample exact match index on the Address that retrieves the according Email(s). $key1 indicates that the index can be used to search for Addresses, using a exact match comparison (simple lookup type). The return (Email) is determined by the XPath expression.

*//Email[.//Addressee/@Address=$var1$_{simple}$ ]*

The simple nature of an exact match index does not require any specific configuration parameters. Therefore an exact match index configuration only contains the identification of the exact match variable (compare with Chapter 3.2.4).

### 3.3.2 Data Structure

In an exact match index essentially identifier and value of the primary data are interchanged, which allows querying for certain values, discovering whether they exist and where (in an XML document) they can be found. This means that the primary data directly becomes the key of the exact match index, which resembles the structure of an inverted file. A page of the exact match index therefore contains the return belonging to a specific key. The key is used as a page identifier.

An example page for index 1 and the key *michael@maier.de* contains Email A and B. The whole index for the example data is depicted in Table 7.

| ***Page Identifier*** | ***Page Content*** |
|---|---|
| michael@maier.de | Email A, B |
| peter@lasinger.at | Email A, B, C |
| franz@mitterer.de | Email A |
| julia@schnell.de | Email A, B, C |

*Table 7: Index 1 – Exact Match Index Data Structure*

### 3.3.3 Algorithms

An exact match index uses three basic algorithms that provide the indexing functionality (insert, delete and retrieve) and a page identifier function to transfer the passed keys into page identifiers. The page identifier function takes a passed key as input and transforms it into a unique identifier for the associated page. In case a specific key is requested it can be transformed into a page identifier, which is then used to retrieve the required page.

The **insertion algorithm** (depicted in Figure 10) gets a specific key and nodes belonging to that key as input. At first the passed search configuration needs to be processed to determine the key, where to insert the passed nodes. Then the page identifier is calculates from the passed key and the page belonging to that id is retrieved from the storage. In case the page does not exist, a new page is created. The nodes are added to the page and finally the changed page is saved. This algorithm assumes that the page size is not limited, so all data belonging to a specific key is always stored in one page.

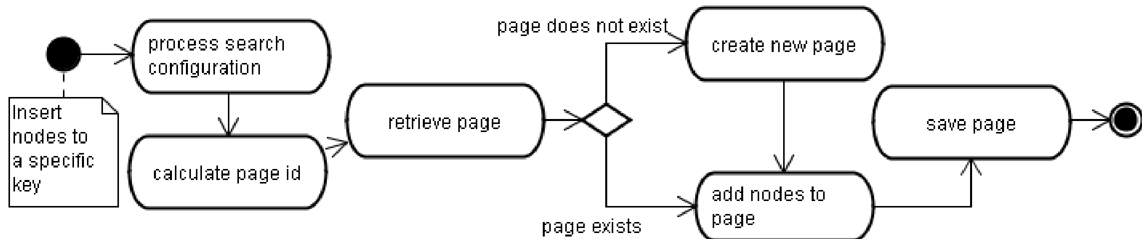



*Figure 10: Exact Match Index - Insertion Algorithm*

The **deletion algorithm** (visualized in Figure 11) removes nodes belonging to a certain key. After the search configuration is processed and the page id has been calculated, the relevant page is loaded. Then the nodes are removed from this page. In case all data contained in the page is removed and the page is empty, it is deleted. Otherwise the changed page is saved. Therefore when deleting an exact match index, it must be provided with all the values that are used as page identifiers. The exact match index itself does not know, which pages it contains. This implies that if a hash index needs to be removed, it cannot delete its associated pages by itself. It needs to execute page requests and to evaluate if the corresponding page exists or not.

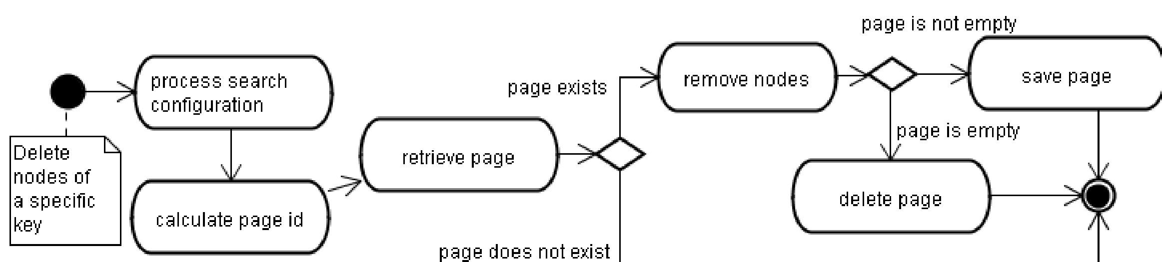



*Figure 11: Exact Match Index - Deletion Algorithm*

The **retrieval algorithm**, shown in Figure 12, is similar to the other two algorithms. At first the search configuration is processed and the page id of the requested key is calculated. Then the page is retrieved from the storage. In case no page exists an empty result set is returned, otherwise relevant nodes are selected from the page and the result is returned.

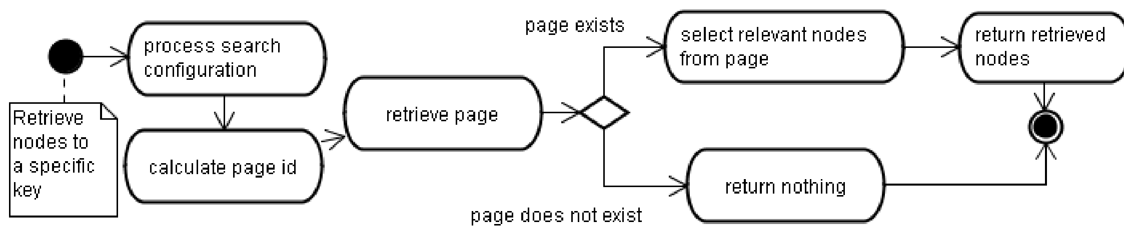



*Figure 12: Exact Match Index - Retrieval Algorithm*

> The exact match **index 1** as defined earlier is depicted in Table 7. To every existing email address (the key that is used as a page identifier) references to the according emails are stored.
>
> In case **query 1** is executed on that index, it is straightforward to see that email A and B are returned, as this is the page content stored to the key and page identifier *michael@maier.de*. **Query 2** leads to an empty result, as *franz@schnell.de* does not exist as a key.

### 3.3.4 Security

By definition an exact match index is a reverted view on the primary data. Therefore changes in the primary data automatically lead to changes of the affected index page (index update). Consequently it may be possible to associate different occurrences of the same value, as same values lead to a change of the same index page. If an intruder is monitoring the storage provider and is able to execute insert, update or delete statements they might gather structural information (like the occurrences of certain values). However, due to the encrypted data no content information is exposed.

## 3.4 Range Index

Like the exact match index, the range index provides value centric indexing, with the difference that it can also be used to execute range queries efficiently. We chose to adopt the B+Tree variant as described by Held and Stonebreaker [HeSt78], so that it can be used in the distributed SemCrypt environment. Therefore the range index uses the pages described in Chapter 3.3.2. The algorithms are adapted in a way to work with these pages and to abstract from the indexed data by using generic operators to manipulate and compare the indexed data.

### 3.4.1 Definition

A range index supports simple and range queries based on the index key. Therefore the search configuration of a range index contains a range match variable, fitting to the key defined in the according index definition. This range match variable needs to be able to express a range, which can be defined by a lower and upper bound, both times with a modifier expressing if these bounds are included. This is analogous to the mathematical syntax of a range definition: *[Lower Bound .. Upper Bound[*, where a bracket showing inward signalizes the inclusion of the bound, one showing outward the exclusion.

A simple query can be expressed by letting the lower bound equal the upper bound and by including both bounds. In case a bound is not set, the meaning is that this bound needs not be regarded. A special case is the setting of no bounds, which returns the whole content of the range index.

**Index 2**

The sample queries can be supported with a range index on the date-field of emails. A possible definition that indices the date-field and returns relevant emails is:

*//Email[ Header/Date =*$\$var1_{range}$ *]*

In case we want to query the index for a range, we need to pass a relevant range match variable. The variable for query 3, which is looking for all emails with a date smaller than three, is set to no lower bound and an upper bound of 3, which is excluded:

$\$var1_{range}$ *:= ] .. 3 [*

For query 4 the variable is set to:

$\$var1_{range}$ *:= ] 1 .. 3 [*

In case an exact match query for 2 is performed the bounds are [ 2 .. 2 ] and with [ .. ] everything contained in the range index is returned (if there is no bound set, it does not matter if the bound is included or not).

### 3.4.2 Data Structure

A range index separates pages that contain data (leaf pages) from pages, which are built on top of the leaf pages and that create the tree structure (branch pages). As the leaf pages are linked together in a sequence set, very efficient range queries are possible. Furthermore this minimizes the required amount of storage accesses, because as soon as

the first leaf page has been found, every following leaf page can be retrieved with one additional access.

Another advantage of separating leaf pages from branch pages is the possibility to provide more efficient splitting algorithms and a better update behaviour when deleting nodes. So branch pages only need to be updated when there are splits or merges on the leaf-level. The page concept for the range index is shown in Figure 13 by using an UML class diagram. As a leaf page is part of the sequence set, it provides a next reference to the following leaf page. A branch page holds the references to its children in the tree structure, which can either be other branch pages or leaf pages.

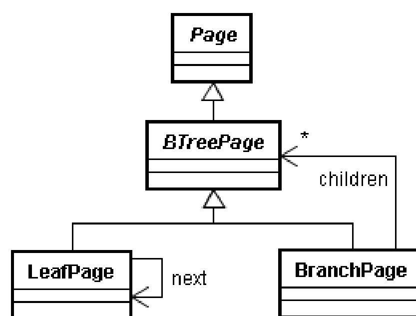



*Figure 13: Range Index Pages*

### 3.4.3 Configuration

The inner structure of a range index, its retrieval, update and storage characteristics (page sizes) are dependent on three core parameters. A fanout parameter that is relevant for the branch pages and that defines the maximum amount of references of a branch page to other BTreePages. The fanout determines the flatness of a range index, the higher the fanout the larger the branch pages and the flatter the tree. Therefore the fanout is a critical parameter for the expected retrieval time, as the height of a tree equals the amount of pages to be accessed to retain a certain leaf page. The fanout parameter also dictates when a branch page splits and merges.

Two other parameters define the size of leaf pages and their split behaviour. A maximum size defines the maximum possible size of a leaf page, while a minimum size defines the minimum size of a leaf page. In case a leaf page grows larger, it splits, in case it grows smaller it is merged, or data is redistributed with a neighbouring leaf page.

Consequently these three parameters can be used to specify the average and maximum page sizes and to optimize a tree for the data to be indexed and the retrieval and update characteristics in a specific setting.

In case a fanout of 2, a maximum leaf page size of 1 and a minimum leaf page size of 1 are chosen, the resulting tree resembles the one depicted in Figure 14. In a running environment these parameters will be a lot higher, but for the example these minimum parameters are chosen to create a tree structure, which only uses the data of the running example.

The range index consists of three interlinked leaf pages (sequence set), each of them is totally filled and contains one date and the reference to the according email. On top of the leaf pages are three branch pages, which link to two other pages at most (fanout) and which contain the minimum value of the part tree containing the larger values (in case of branch node V there is no such part tree, so there is no minimum value). Branch page I is the root of the tree and the starting point of any insert, delete or retrieve query.

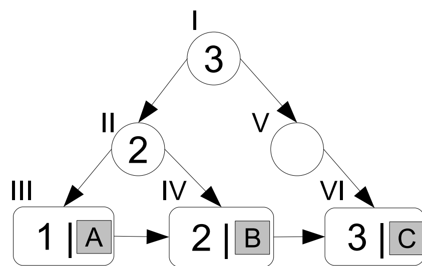



*Figure 14: Sample Range Index*

| *Page ID* | *Page Content* |
|---|---|
| I | II <3> V |
| II | III <2> IV |
| III | <1> [A] IV |
| IV | <2> [B] VI |
| V | VI |
| VI | <3> [C] |

*Table 8: Sample Range Index in a Table Representation*

This tree structure is mapped to a table, where every row represents a page of the tree and the first column expresses its identification and the second column its content. For the sample range index the according representation is shown in Table 8.

The page ids in the content column represent references to other pages, values in <> brackets are index keys (in this case dates) and values in [] brackets represent the indexed data (here references to the according email nodes).

### 3.4.4 Algorithms

A range index is a complex and dynamic data structure and therefore requires a set of algorithms to allow insert, delete, update and retrieve operations. First of all there are algorithms concerned with the distribution and balancing of data in pages, in case an overflow or underflow occurs. The merge algorithm takes two pages (of the same type)

as input and combines their content in one page. On the other hand, the split algorithm equally distributes the content of one page among two pages. The redistribute algorithm balances the content of two pages, which equals performing a merge first that is then followed by a split.

The **insertion algorithm**, visualized in Figure 15, is used to create and update the structure of the range index. At first the passed search configuration needs to be processed to determine the key, where to insert the passed nodes. Starting from the root page the tree is traversed to locate the relevant leaf page for insertion. This traversal equals the one performed by any regular B-Tree. In a next step the nodes and the key are inserted into the page, or the nodes are added in case the key already exists.

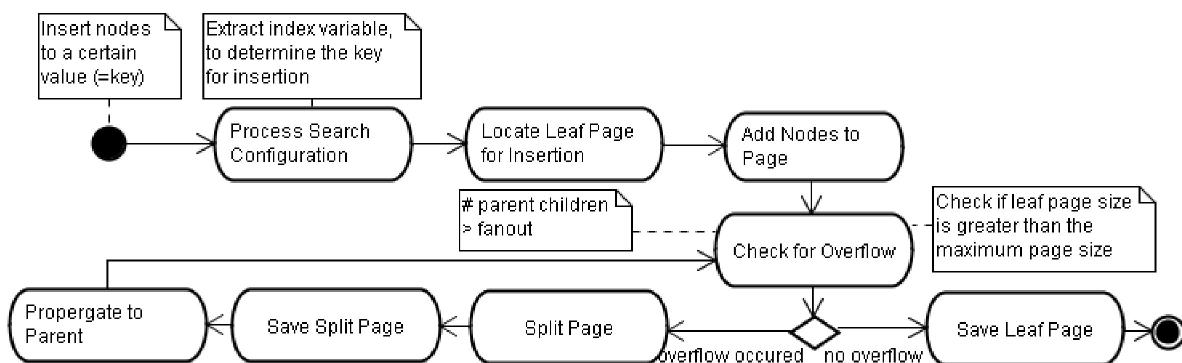



*Figure 15: Range Index - Insertion Algorithm*

If the updated page outgrows the maximum page size, a split needs to be performed and the parent page needs to add the reference to the newly created page. In case the parent page also overflows and splits, this procedure is repeated until there are no more overflows. An overflow of a branch page is checked against the fan-out parameter. In case the root page splits, a new root page is created, linking to the old root page and its split page and consequently increasing the height of the tree by one.

The reverse algorithm is the **deletion algorithm** (depicted in Figure 16) that can be used to update a range index in case nodes are deleted or updated and that also manipulates the tree structure. The first steps resemble the one of the insertion algorithm, however the located leaf page is used for deletion. In case the page belonging to the concerned key does not exist, the algorithm returns. Otherwise the nodes are deleted from the page. Then the leaf page is checked for an underflow, meaning that the total leaf size has become smaller than the minimum page size parameter.

In case an underflow occurs, a sibling page (sharing the same parent node) needs to be retrieved. If there are multiple candidates the left sibling (containing data belonging to smaller keys) is used. Three possible situations can occur:

1. No sibling page exists, which is the case when there is only the root page and one leaf page left. In case the page is empty, it is deleted, otherwise nothing happens.

2. The summed up sizes of the page and the sibling page is smaller than the maximum page size. Consequently the two pages can be combined (merged) into one and the sibling is deleted.
3. The summed up sizes of the page and the sibling is greater than the maximum page size. As the pages cannot be combined, their content is equally redistributed.

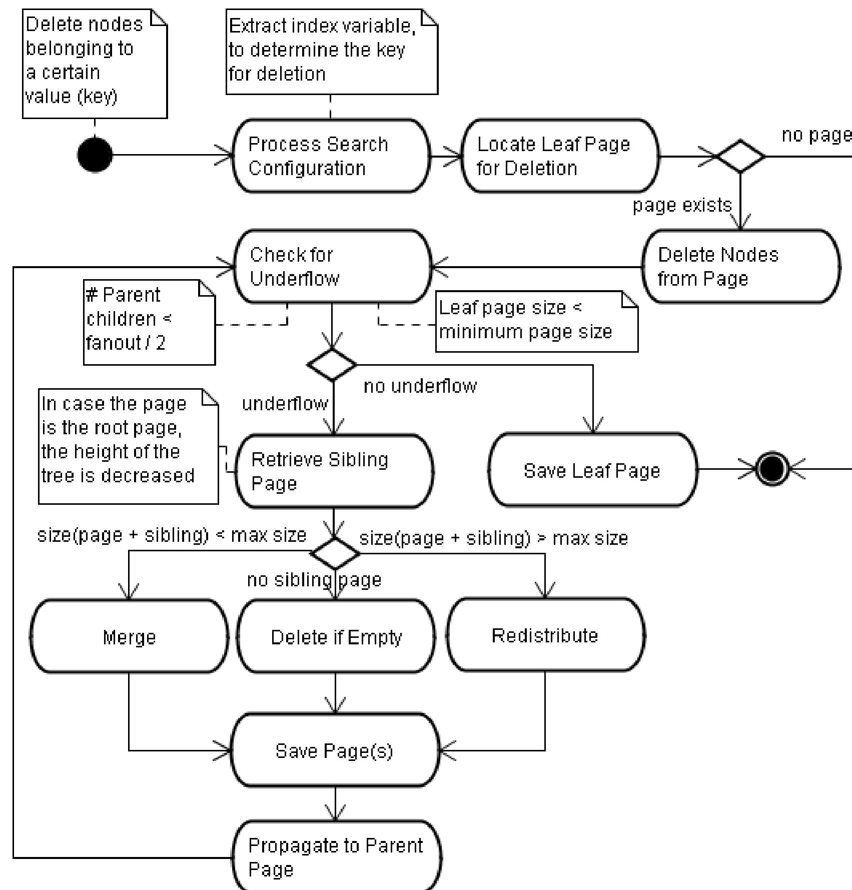



*Figure 16: Range Index - Deletion Algorithm*

In the next step modified pages need to be saved and in case a merge or redistribution occurred the parent page needs to be updated. A merge leads to the deletion of a reference in the parent page, which may lead to further underflows thus repeating the process of handling underflows. A redistribution changes the minimum value of the sub-trees, consequently reference information in parent pages needs to be updated to keep the tree valid. Due to multiple merges it is possible that the root page only holds one reference to a child page. In this case this child page becomes the new root page.

The **retrieval algorithm** (depicted in Figure 17) can be used to execute range queries on a range index. Again the first steps resemble the ones described in the deletion and insertion algorithms. Further information is extracted from the search configuration (upper and lower bounds and indicators if these bounds shall be included in the query).

As the leaf pages are linked together in a sequence set, every range query can be processed in the same way. At first the leaf fitting to the lower bound is retrieved and then the following leaf pages are retrieved as long as the upper bound is not violated.

When a leaf page has been located, all keys satisfying the bounds are selected. These results are then aggregated and combined with results from further leaf pages. When the first key violating the upper bound is found, or there are no further leaf pages, the aggregated result set is returned.

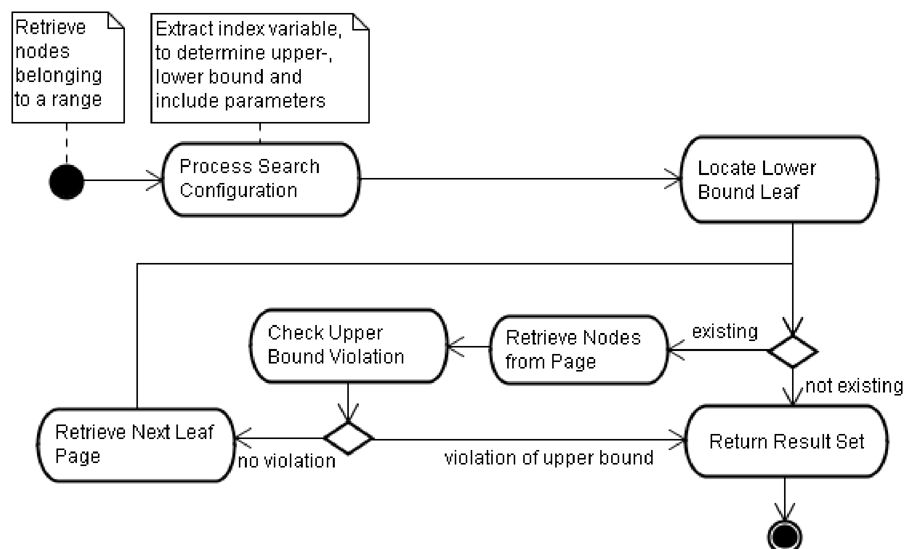



*Figure 17: Range Index - Retrieval Algorithm*

Due to the interlinked tree structure a range index can be deleted easily, without knowledge of the indexed data. Starting from the root node the whole tree can be traversed until every page has been deleted.

**Query 3** asks for all emails whose dates are smaller than 3. First the leaf page with the minimum key is searched. Starting with the root, page II is loaded next, that directs to leaf III. Leaf III contains the first entry within the bounds, but as the upper bound is still not matched, the next leaf (IV) is loaded. It is not necessary to traverse the tree more than once, as the required information is contained in the according leaf. Leaf IV contains the next match but the upper bound is still not violated, therefore the next leaf is loaded. However, leaf VI does not contain any more relevant data and its key violates the upper bound (there are no more results in following pages), therefore Emails A and B are returned.

**Query 4** is looking for all emails whose date is greater than 1, but smaller than 3. Again the retrieval starts at the root page I, with the aim to find the lower bound page first and then to select all relevant leaf pages till the

upper bound is violated. So at first branch page II is examined that directs to leaf page IV. Here the first valid element (Email B) is found and the next leaf page is loaded. Leaf VI does not contain any more relevant data and violates the upper bound, so only Email B is returned.

### 3.4.5 Security

In its concepts the range index in SemCrypt resembles the secure trees presented in Chapter 2.6. By the use of pages the tree structure is mapped to a table that can be stored as encrypted key-value pairs at the storage provider. When traversing the tree, the identification of the root node must be known, so that it can be accessed. The root page is then evaluated and additional pages are loaded as they are required. Therefore a range index can be accessed by multiple clients at the same time (if there is additional concurrency control in place) and the tree structure is hidden from the storage provider.

As discussed in Chapter 2.6 (and identified by Lin and Candan [LiCa04]), the frequent accesses of the root page and pages that are located near the root in general can be a security problem. In the SemCrypt setting this problem is less critical, because an index is stored together with the primary data (and other index structures) and cannot be distinguished from them. Furthermore SemCrypt caches frequent accesses, so that the performance can be increased, while frequently visited pages (like the root page) are accessed less often at the storage provider. Consequently the concepts of node swapping and access redundancy proposed by the authors, which would reduce performance drastically, need not be applied to attain a similar level of security.

## 3.5 Text Index

The text index is an extension of the range index and can be used for indexing textual information. It resembles the prefix B+Tree discussed in Chapter 2.3.2 and makes use of key-compression techniques. Compared to other structures used for information retrieval (like Patricia tries), a prefix B+Tree is balanced.

The text index enables simple keyword match and prefix match functionality required by SemCrypt. The internal structure equals the one used by the range index and also the algorithms are similar. The advantages, disadvantages and security concerns are identical to the ones described for the range index, therefore only the differences will be outlined.

As mentioned before the text index is more specific than the range index, as it indexes just keywords. The leaf pages are identical, however the branch pages are different, because they do not contain the full key for distinguishing sub-trees, but a minimal prefix. As described in Chapter 2.3.2 this enables prefix search (the retrieval of all keywords starting with a specific set of letters) and compresses the keys.

The text index cannot efficiently answer boolean queries on a set of keywords, does not provide any similarity measure (like ranking mechanisms) and does not support advanced pattern matching (like regular expressions). Boolean queries need to be executed by accessing the index several times and then merging the results.

### 3.5.1 Definition

The text index supports keyword (exact match) and prefix match queries on the indexed text data. Therefore the search configuration contains a text match variable on the index key, defined in the index definition. This text match variable consists of the keyword or the prefix that shall be looked for and an indicator telling whether prefix or exact match shall be performed.

The queries 5 and 6 can be supported by text indices based on the Text and the Subject of an Email. We define **index 3** with:

*//Email[.//Text/text()=$var1*$_{keyword}$*]*

and **index 4** with:

*//Email[.//Subject/text()=$var2*$_{keyword}$*]*

The text match variable to search for the keyword “message” in index 3 (query 5) is:

*$var1*$_{keyword}$ *:= “message”*

and the text match variable to search for every subject line starting with “RE” in index 4 (query 6) is:

*$var2*$_{keyword}$ *:= startsWith(“Re”)*

This information can be transformed into a range query that can be executed by the prefix B+Tree. In case of a keyword search, the keyword represents both the upper and lower bound of the search.

The search for "*message*" (query 5) leads to the range query:

*["message" .. "message"]*

In case of a prefix search, the prefix indicates the lower bound of the range, yet the upper bound needs to be calculated and is the first letter combination that does not satisfy the prefix.

Consequently the search for all keywords starting with *"Re"* (query 6) can be expressed with [*"Re"* .. *"Re"*+x[, where x is a virtual letter with the characteristic that it is larger than any other letter indexed $\forall\, l, l \in IndexedLetters \rightarrow l < x$ .

## 3.5.2 Algorithms

Before text can be indexed by a text index, the text needs to be prepared. This process is similar to the one used in general information retrieval and usually independent of the index structure. As depicted in Figure 18 information retrieval can be represented as a twofold process. The first one prepares the text and builds up the index, while the second process targets on the retrieval (returning a result to a certain query) [GoHa05].

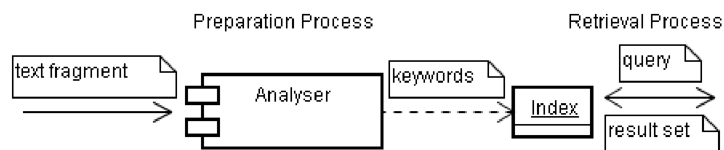



*Figure 18: Information Retrieval Processes*

The analyser prepares the text to be indexed. It performs the tasks of tokenizing the text fragment (splitting it into keywords), stop-word removal (removing articles, punctuation and other not expressive information), stemming (reducing words and verbs to their root form) and lower casing. The resulting keywords are then passed to the index for indexing. The text index then performs insertion or deletion of nodes for each individual keyword. Therefore the analyser acts as a kind of filter, preprocessing the input for the index. The analyser is independent of the retrieval process, as queries are directly posed at the index itself.

**Index 3**

In a first step the index needs to be created. For this purpose the texts of the emails are passed to the analyser, which creates keywords that are then indexed. The resulting tree is depicted in Figure 19, with a fanout of two

and a maximum page size of three.

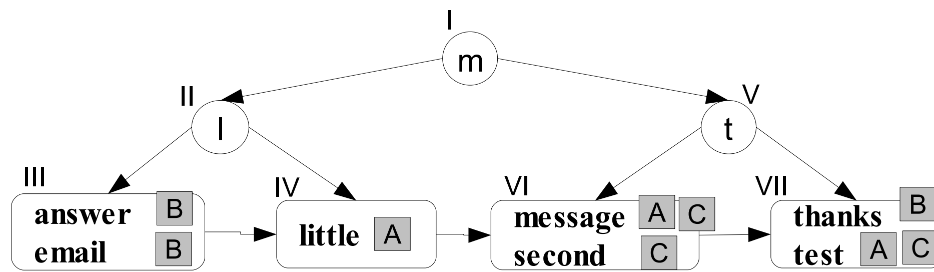


*Figure 19: Sample Text Index - Index 3*

Stop words like "this", "the", "is" or "a" have been removed by the analyser and are not included in the tree structure. The prefix compression is shown by the branch pages I, II and V, as one letter is sufficient to distinguish the sub-trees.

When looking for the keyword "message", the traversal starts at the root page. The comparison directs to page V, but as "message" is smaller than "t" page V directs to leaf page VI. The keyword is found there and the occurrences, Email A and C, are returned.

**Index 4**

When creating index 4, we first need to build up the index using the subject lines. We assume that the lines are indexed as a whole, so we neglect the analyser. The resulting prefix B+Tree resembles Figure 20, with a fanout of two and a maximum page size of one.

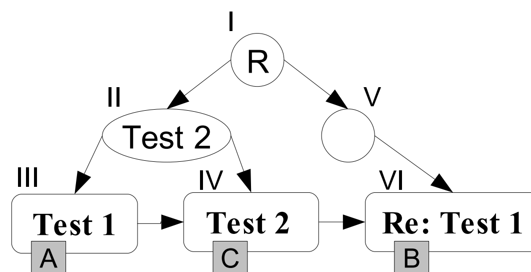


*Figure 20: Example Text Index - Index 4*

As can easily be seen, the structure is identical with the one of a regular range index, however the branch pages (their keys) are slightly different, as they always represent the minimal prefix required to distinguish two sub-trees. Two special cases are shown here. Branch page II requires a maximal prefix, as the two subject lines differ only in the last letter. In the root page

the letter *R* is sufficient to distinguish *"Test 2"* from *"Re: Test 1"*.

Query 6 looks for all emails containing the prefix "Re" in their subject line. Therefore the root page is examined first. The comparison of "R" with "Re" directs to branch page V which directs to leaf page VI. The contained subject line satisfies the prefix and as there are no further leaf nodes, Email B is returned.

## 3.6 Hierarchic Index

The hierarchic index is a very special index that satisfies the need to deal with hierarchic data. By using the hierarchic index it is possible to speed up structural queries that are based on a hierarchic structure contained in an XML document. Similar to the object-oriented indices it is sufficient to maintain one index for the whole hierarchy. This ensures that a set of indices can be replaced by one hierarchic index, which increases the search for part trees of a hierarchy (as only one index is queried). Furthermore the hierarchic index presented here is capable to deal with dynamic hierarchies and as shown later (Chapter 3.7) can be used with any other index to provide an additional segmentation according to the relevant hierarchy [Grün06a]. Regarding security the same considerations pointed out with the range index can be applied. Therefore the subsection security is omitted.

Kim et al. [KDD89] introduced the basic idea with their CH tree that segments the leaf pages of a B-Tree according to a class hierarchy. This concept is generalized, as the hierarchy not only segments according to a static class structure, but according to an arbitrary dynamic hierarchy. Furthermore it provides a dynamic overflow and split mechanism, which moves whole sub-trees of the hierarchy to sub-hierarchy pages.

If used in combination with another index, it resembles the content aware data guides proposed by Weigel et al. [WMB+04]. Likewise a hierarchic index can be used to either achieve content centric, or structure centric indexing (see chapter 2.5.2). However, the hierarchic index is not limited to the document structure, but can be used for any hierarchic structure.

### 3.6.1 Definition

The hierarchic index supports structural queries on hierarchic data, which in SemCrypt occurs in several ways:

- *Type hierarchies*: a family of types that are interlinked in a hierarchic relationship.

> An example is the Email Type and its subtypes *ReveivedEmailType* and *SentEmail* Type:
>
> *Email*
> *- ReceivedEmailType*
> *- SentEmailType*

- *Path hierarchies*: a hierarchy, which arises from the XML document structure.

> The running example defines folders that contain emails. Every folder can be seen as a sub-tree containing a set of emails. The super hierarchy is the Mailbox, whereas every folder creates a sub-hierarchy and so does every Email.
>
> *Mailbox*
> *- Folder "InBox"*
> *- Email A*
> *- Email C*
> *- Folder "Sent"*
> *- Email B*

- *Document hierarchies*: a hierarchy that is the result of bundling a set of similar XML documents to a collection. The collection is the super hierarchy, every XML document a sub-hierarchy. A complex hierarchy can be created by further nesting collections using sub-collections.

> In case there are several Mailboxes (documents), these could be structured using collections. For example:
>
> Collection "Peter's Mailboxes"
>
> - Document "Mailbox 1"
>
> - Document "Mailbox 2"

It is possible to abstract from these various representations, as there is always an underlying hierarchy, which contains certain hierarchic-relationships. We further specify that only leaves in a hierarchy (do not contain any sub-hierarchies) may contain values. With the generation of additional leaves, every hierarchy can be transformed into such a representation. Consequently the hierarchic index does not need to care about what hierarchy it is based on, as long as operators for dealing with the hierarchic data exist.

When a hierarchic index is queried, a specific hierarchy is passed and the index returns all data that satisfies this hierarchy, meaning that it either belongs to this hierarchy or one of its sub-hierarchies.

**Index 5**

In order to support queries 7 and 9 we define a hierarchic index on the type of Email. The used variable indicates that the type of the email is indexed using a index capable to deal with structural information.

*//element(Email, $\$var1_{structure}$)*

Query 7 is looking for all Emails that have been sent (*SentEmailType*). The according search variable is:

$\$var1_{structure}$ *:= SentEmailType*

**Index 6**

If we want to create a hierarchic index on path hierarchies, for example to support query 8, the index can be defined as:

$\$var2_{structure}$*//Addressee/@Address*

As one Address may be contained multiple times (in case it is contained in several emails) also the index is going to return duplicates.

The search variable can now be defined as any path selecting a specific node, which contains Addressees (this is the case for MailBox, Folder, Email and Header). The following search variable looks for all email addresses contained in the Folder named "InBox":

$\$var2_{structure}$ *:= //Folder[@name="InBox"]*

If we want to retrieve all email addresses, we can set the variable to MailBox or leave it empty:

$\$var2_{structure}$ *:= MailBox*

It is also possible to retrieve the email addresses for a specific Email (in this case the first email in the folder "InBox") with index 6:

$\$var2_{structure}$ *:= //Folder[@name="InBox"]/Email[1]*

### 3.6.2 Data Structure

A hierarchic index represents a certain element of the hierarchy with a hierarchy bucket. The hierarchic index only needs to store the hierarchies (buckets) that contain nodes, which is done by the leaf buckets depicted in Figure 21. They contain the hierarchy-key determining the hierarchy of the according nodes.

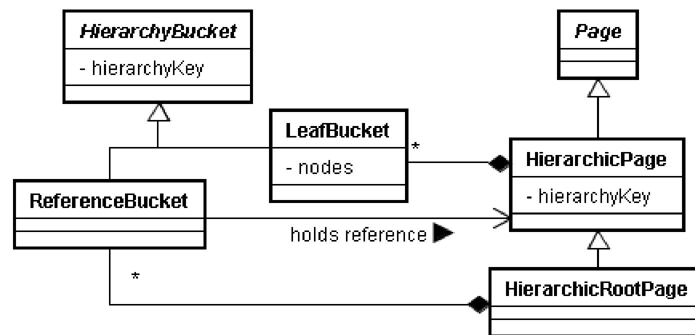



*Figure 21: Hierarchic Index - Pages and Buckets*

A hierarchic page that is retrieved from the storage provider may contain several of these leaf buckets. This would be sufficient in case there is no splitting behaviour, but as a page will split in case it exceeds a certain size, two additional constructs are necessary.

The traditional overflow concept is extended in a way that an overflow page contains structured data. We are going to call these kind of pages sub-hierarchy pages, as they contain a sub-tree of the primary hierarchy. This sub-hierarchy is extracted from the overflowing page and transferred to the sub-hierarchy page.

In case a page splits, the new sub-hierarchy page needs to be referenced. So a reference bucket links to the sub-hierarchy page. The hierarchy-key of these reference buckets needs to be a hierarchy, which contains all hierarchy-keys in the referenced page.

For example, in case a reference bucket links to a page that contains *sentEmailType* and *receivedEmailType*, the reference bucket hierarchy-key is *emailType*, which contains the two other types.

Hierarchic pages that contain both leaf buckets (data) and reference buckets (references to sub-hierarchy pages) are called hierarchic root pages. In addition to the functionality of a regular hierarchy page (splitting the page and retrieving all contained data to a specific hierarchy), they provide functionality to merge an sub-hierarchy page back into the hierarchic root page.

The sample hierarchic index (**index 5**) is depicted in Figure 22. It consists of one hierarchic root page, which contains two leaf buckets (SentEmailType and ReceivedEmailType). Each of these buckets contains the according references to the emails.

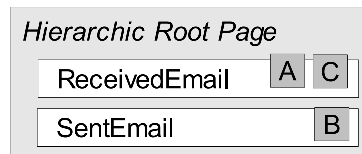



*Figure 22: Sample Hierarchic Index - Index 5*

**Index 6** is depicted in Figure 23. It also consists of one hierarchic root page and contains nine leaf buckets, one for every addressee. An Addressee represents the next hierarchy regarding a specific Address. In this special case an Addressee-hierarchy only contains one Address. Therefore each leaf bucket contains the reference to the according Address attribute (so A1 points to the node containing michael@maier.de, B1 to the node containing peter@lasinger.at, etc).

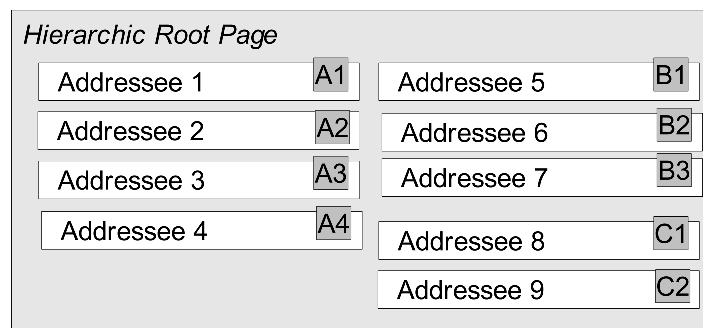



*Figure 23: Sample Hierarchic Index - Index 6*

The fine granularity of the hierarchies (only one entry per hierarchy) is chosen to demonstrate the dynamic split behaviour later on.

### 3.6.3 Configuration

A Hierarchic Index needs additional information to allow dynamic splitting behaviour, which can be expressed by two parameters:

- A **minimum page size** that defines the minimum size of a Hierarchic Page. In case a HierarchicPage becomes smaller it will be merged with the Hierarchic Root Page.
- A **maximum page size** that defines the maximum size of a Hierarchic Page. In case a HierarchicPage becomes larger it will be split.

### 3.6.4 Algorithms

Splits and merges can occur when inserting or deleting nodes from the index. In case a hierarchy page outgrows the defined maximum size, there will be a split. In case it becomes smaller than the minimum size, it will merge with the parent hierarchy page.

As splits are performed in a way that clusters according to hierarchies, efficient queries are assured. The hierarchy-sub-tree is determined and then moved to a new hierarchy page (sub-hierarchy page). The root page adds a reference to the newly created sub-hierarchy page and remembers the associated hierarchy. In case the hierarchic index is queried it is able to determine every required sub-hierarchy page by just evaluating the root page.

In order to create, maintain and query a hierarchic index, three operators are required, which ensure that the index is able to deal with the hierarchic information:

1. Equals Comparator [ = ] that is used to determine if two hierarchies equals are identical.
2. IsA Comparator [ ⊆ ], which is used to determine if a hierarchy equals the other hierarchy or if it is a sub-hierarchy of the other hierarchy.

sentEmailType ⊆ emailType --> TRUE

sentEmailType ⊆ sentEmailType --> TRUE

sentEmailType ⊆ receivedEmailType --> FALSE

3. GetSuper Operator [ ↑ ], which is used to determine the super hierarchy of a hierarchy.

↑ sentEmailType --> emailType

↑ emailType --> ALL (there is no parent for emailType)

The **insertion algorithm** is depicted in Figure 24. At first the hierarchy that is used for insertion is extracted from the search configuration passed to the index. Then the root page is processed to locate the according hierarchy bucket. If a reference bucket is found, this indicates that the requested leaf bucket resides at a sub-hierarchy page. Consequently the according sub-hierarchy page is loaded. When no bucket exists, it is created.

The nodes are then added to the bucket and the affected page is checked for an overflow. In case the page overflows, it is split and the root page is updated with the

reference to the created sub-hierarchy page. Finally the changed pages are saved.

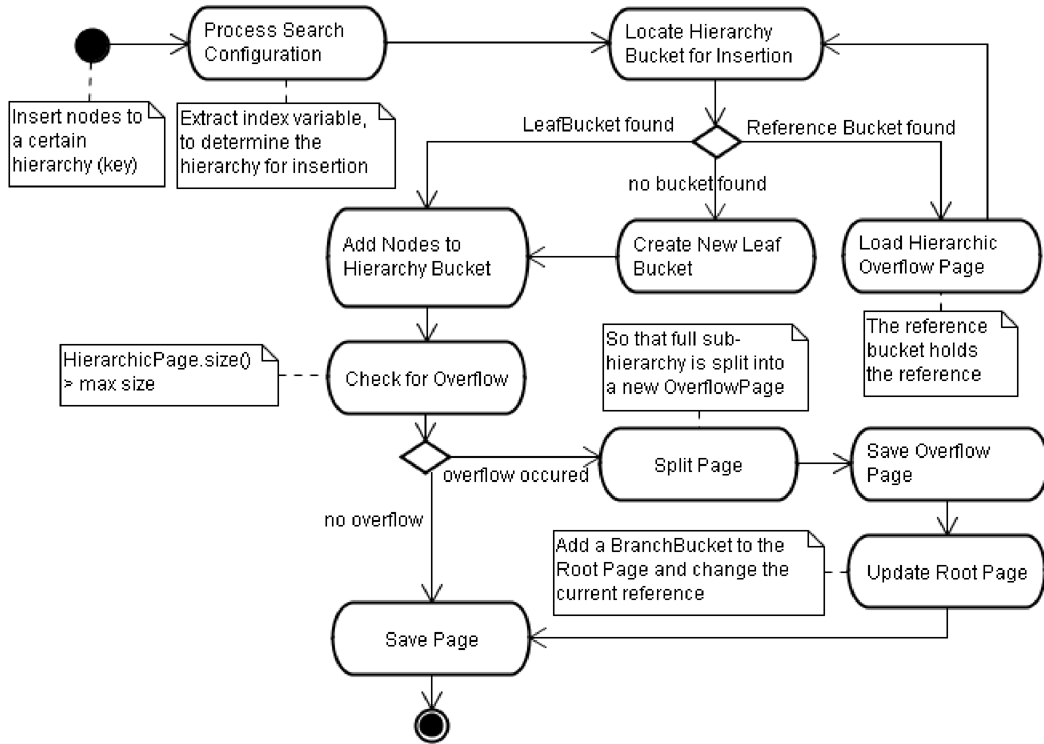



*Figure 24: Hierarchic Index - Insertion Algorithm*

The **deletion algorithm** removes nodes from the hierarchic index and is depicted in Figure 25. At first the hierarchy for deletion is determined and the page containing the according hierarchy bucket is located. If the bucket resides at a sub-hierarchy page, this page is loaded. If the bucket does not exist, the algorithm terminates.

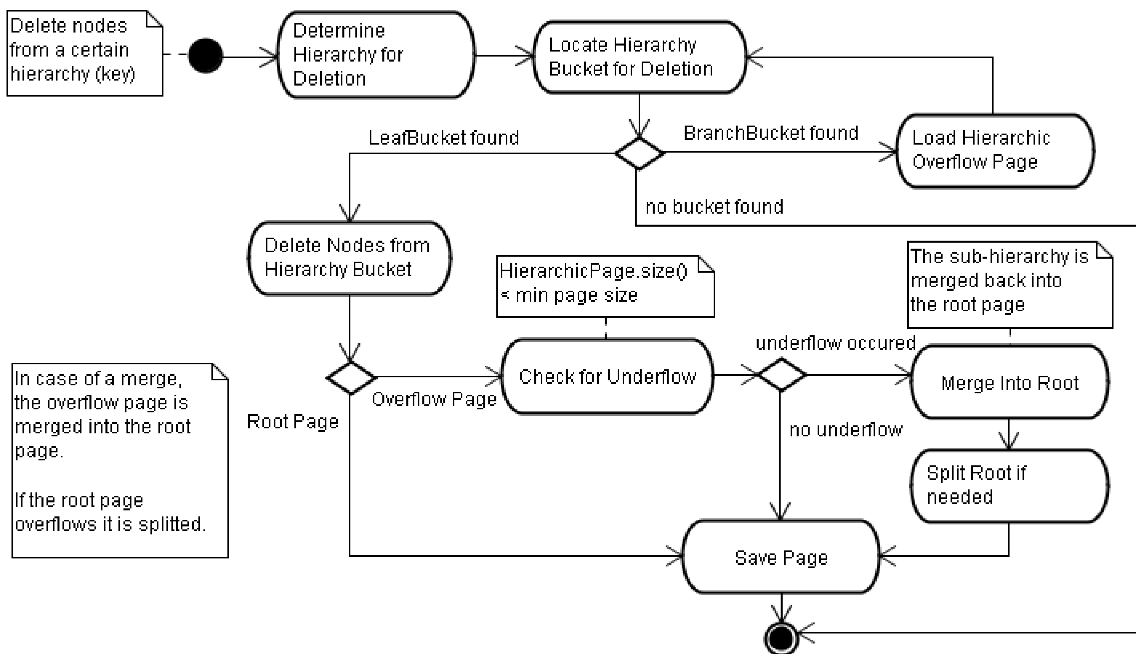



*Figure 25: Hierarchic Index - Deletion Algorithm*

The nodes are deleted from the hierarchy bucket and it is determined if the belonging page overflowed. If the affected page is the root page and it does not contain any more data, a sub-hierarchy page is merged back into the root page. If no such sub-hierarchy page exists, the root page can be deleted. In case the affected page is a sub-hierarchy page, it is merged into the root page (and deleted). As this may lead to an overflow of the root page, the root page needs to be examined. Finally the root page is saved.

Queries on a hierarchic index are answered with the **retrieval algorithm** depicted in Figure 26. After the hierarchy has been determined, all pages that contain hierarchy buckets, which satisfy the isA relationship ( $hierarchy_{bucket} \subseteq hierarchy_{searched}$ ) are retrieved. In case the buckets reside in sub-hierarchy pages, these pages are loaded. Nodes from buckets satisfying the isA relationship are aggregated and returned.

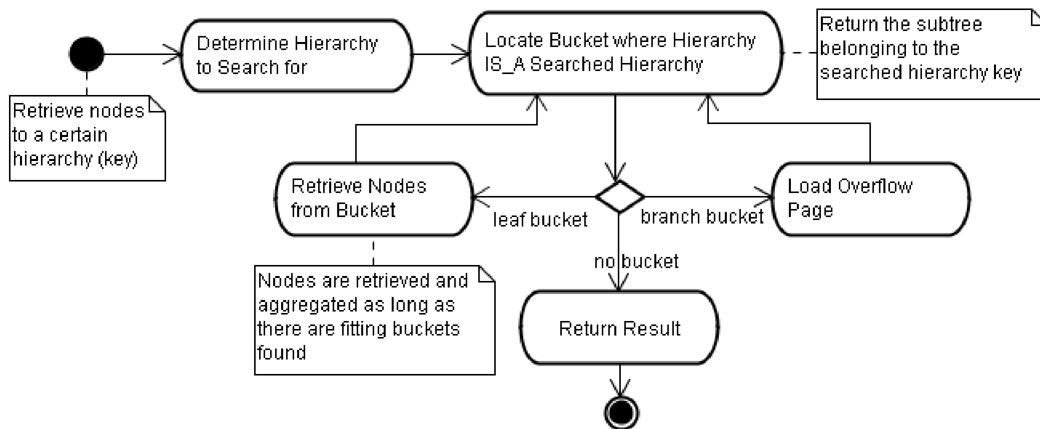



*Figure 26: Hierarchic Index - Retrieval Algorithm*

The uniqueness of the hierarchic index lies in its dynamic splitting behaviour, which ensures that sub-trees of the hierarchy are moved to sub-hierarchy pages. As the root page contains all information (references) to the sub-hierarchy pages, it is possible to access every leaf hierarchy with at most two page accesses (root page and one sub-hierarchy page). In case hierarchies containing several sub-hierarchies are queried, a minimal amount of pages needs to be loaded, as data belonging together (in a hierarchic relationship) is moved together to a sub-hierarchy page by the **page split algorithm** depicted in Figure 27.

To split the root page or a sub-hierarchy page, a sub-tree of the hierarchy is located, which contains more data than the defined minimum page size. An additional constraint is that also the remaining page needs to be larger than the defined minimum page size. A fitting sub-tree is located in the following way:

Until the sub-tree has been located, for every leaf bucket contained in a page the parent hierarchy is determined ( $\uparrow$ ) and all other leaf buckets belonging to this parent hierarchy are added. This can be seen as rebuilding the hierarchy. If there is no parent

hierarchy (which is the case for the root hierarchy), the next leaf bucket is processed.

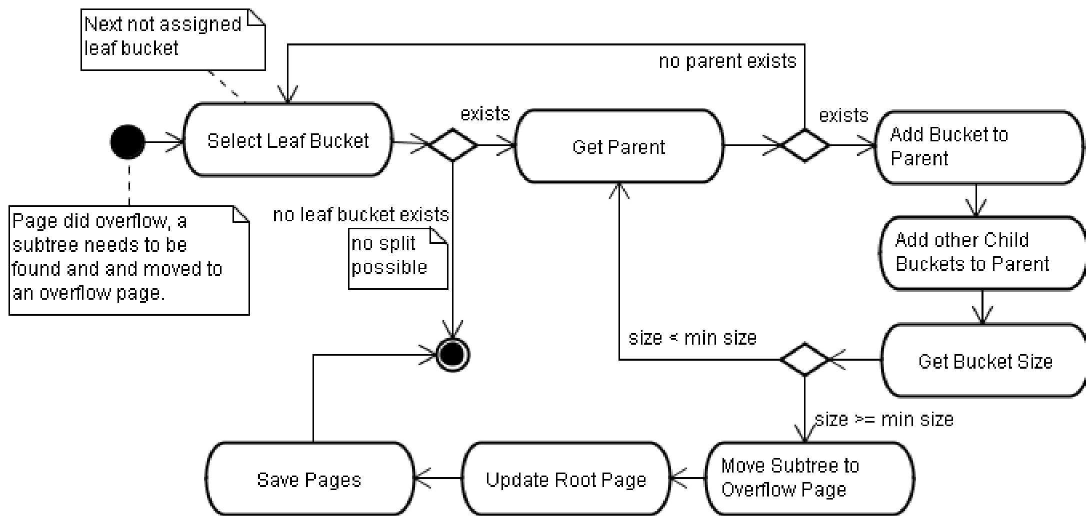



*Figure 27: Hierarchic Index - Page Split Algorithm*

When a hierarchy is found whose size is greater than the defined minimum page size (and the remaining page size is also greater than the minimum page size) the split is processed. Otherwise the split is not performed and another candidate is searched. If it is not possible to determine a fitting candidate, the page is not split and consequently becomes larger than the defined maximum page size.

After a candidate hierarchy has been located, this hierarchy and all belonging leaf buckets are moved to a sub-hierarchy page. The root page is updated and the changed pages are saved.

If we assume a maximum page size of 2 and a minimum page size of 1, **index 5** splits. The resulting index is depicted in Figure 28. The root page contains one reference bucket, which points to the sub-hierarchy page containing the relevant leaf bucket.

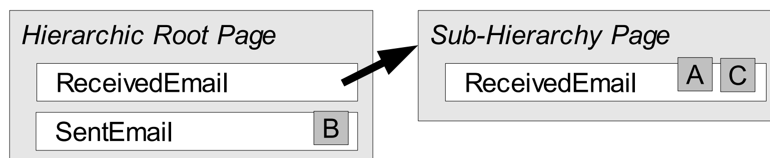



*Figure 28: Hierarchic Index 5 - After Split*

Query 7 is looking for all Emails of the type SentEmailType. As the root page already contains the required data, email B is returned. The sub-hierarchy page needs not be loaded, thus reducing the amount of data transferred.

In case we assume a maximum page size of 4 and a minimum page size of 2, the first split of **index 6** is depicted in Figure 29. The root page holds a reference to the sub-hierarchy page. Interesting is the hierarchy (Folder "InBox") of the reference bucket, which is the common parent hierarchy of all leaf buckets contained in the sub-hierarchy page.

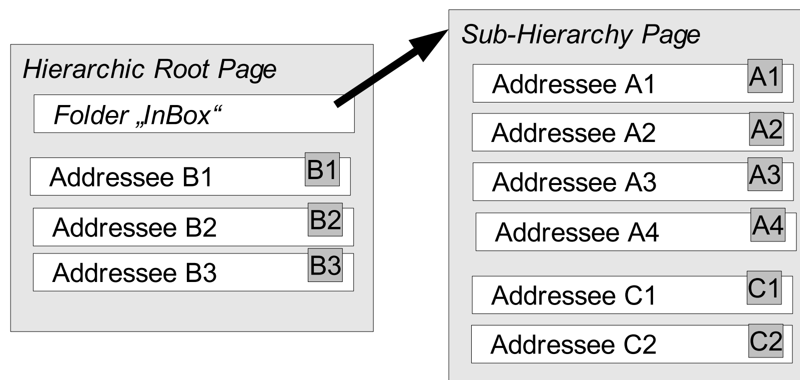



*Figure 29: Hierarchic Index 6 - After First Split*

The result of the second split (sub-hierarchy page) is depicted in Figure 30. The original sub-hierarchy page is split into two sub-hierarchy pages and the reference in the root page is updated. As the newly created sub-hierarchy page contains all addressees of one email, the reference buckets indicate the relevant email (this can be seen as a path-expression).

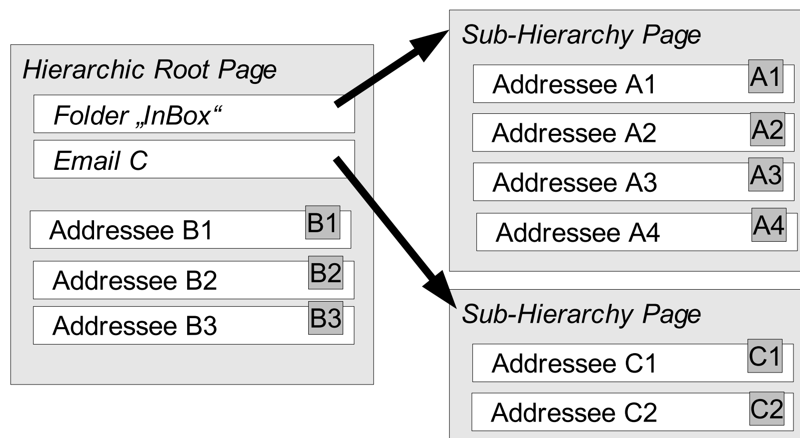



*Figure 30: Hierarchic Index 6 - After Second Split*

If we execute query 8 on this hierarchic index (looking for all email-addresses in the folder "InBox"), at first the root page is processed. The isA comparisons of the requested hierarchy with all buckets contained in the hierarchic root page return the reference bucket with the hierarchic key *Folder "InBox"* and the reference bucket with the hierarchic key *Email C*. The leaf buckets of the root page do not satisfy the isA comparison.

The two sub-hierarchy pages are loaded and processed using the isA

comparison, which returns all leaf buckets of these pages. Consequently the email addresses A1, A2, A3, A4 (from Email A) and C1, C2 (from Email C) are returned.

## 3.7 Nesting Index Structures

The SemCrypt DBMS manages XML documents, which additionally to value information contain structural information. Therefore index structures should be capable to support queries regarding this structural information. A commonly used approach in previous XML databases is to use an additional index for the structural information. However, this requires the management and traversal of two separate index structures. A better approach is to combine structural and value indices, which can be achieved by nesting according index structures. Therefore index nesting is a core element of SemCrypt's index processing approach [Grün06a].

Nested indices combine indexing capabilities from different index structures and can be used to simulate multidimensional index structures. The idea emerged from the need of an index capable of dealing with hierarchic data. The main concept is adapted from the CH Tree developed by Kim et al. [KDD89] as the authors structure the leaf-pages of a B-tree according to classes and consequently create a simple nested index (see Chapter 2.4).

This idea can be generalized, combining different kind of index structures while sharing a general access interface. One major requirement is that every index supporting nesting uses pages and that these pages can be nested. Consequently pages may contain pages of other (nested) index structures. Another requirement is that every index structures needs to adopt its algorithms, so that they are able to deal with nested pages and to perform forwarding of queries to the nested index.

### 3.7.1 Definition

The definition of a nested index resembles the definition of a multidimensional index as multiple keys are defined. The difference is that the order of the definitions defines the order of the nesting. This nesting order defines in which order the nested index structures segment the indexed data (in a multidimensional index the segmentation happens simultaneously). The advantage of this general definition is that the same index definition can be used for a nested or a multidimensional index. The definition is independent from the implementation. Consequently a nested index behaves like a single

index, which is a major benefit, as various indices can be handled in the same way.

An optimal nesting solution depends on the data and the queries that need to be supported. As a rule of thumb and based on experiences described by Kim et al. [KDD89] and Ooi et al. [Ooi+96] the more selective index should be put higher in the nesting hierarchy. This ensures that the possible result is narrowed down faster and the nested index structures become smaller in size.

Not every index structure can be nested. More precisely only index structures that can be initialized using a root page can be used, which is the case for all tree-like structures. Hash-based index structures are not nestable, as one key (which is also the identifier of the page) may occur several times, violating the segmentation that has been created by the superior index.

Query 9 queries for two keys, the date of an email and its type. One could use two index structures, index 2 for the date and index 5 for the type, to answer this query. But this requires a join of the results, which is inefficient for larger data. A solution is to nest the two indices together and to define a new nested **index 7** (**first approach**):

*//element(Email, $var1$_{structure}$)[Header/Date =$var2$_{range}$ ]*

As mentioned before, when nesting index structures there are always different possibilities on how to combine the indices. The **second approach** for **index 7** is:

*//element(Email, $var2$_{structure}$)[Header/Date =$var1$_{range}$ ]*

The two possibilities are shown in Figure 31. The first approach resembles a structure centric approach, as the structure is regarded first, followed by the value. This resembles a content aware data guide (see Chapter 2.5.2).

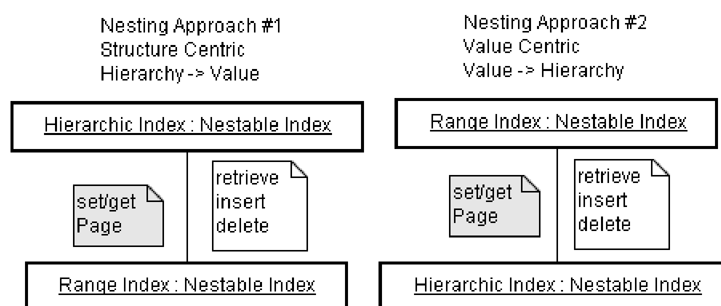



*Figure 31: Index Nesting Alternatives Example*

The second approach resembles a value centric approach, at first the value (Date) is regarded and then the type structure (this resembles the approach

taken in the CH-tree).

Query 9 can be expressed by setting two variables (first approach):

$\$var1_{hierarchy}$ *= ReceivedEmailType* $\$var2_{range}$ *= [1 .. 1 ]*

### 3.7.2 Data Structure

The data structure of the nested indices stays the same. However index pages need to be nested. A superior index needs to include initialization pages for the nested index and consequently stores these initialization pages in its own pages, replacing the indexed data. When a superior index is traversed it does not find the indexed data, but a nested initialization page. This page and the retrieval or manipulation task is passed to the nested index, which then retrieves or manipulates the required data.

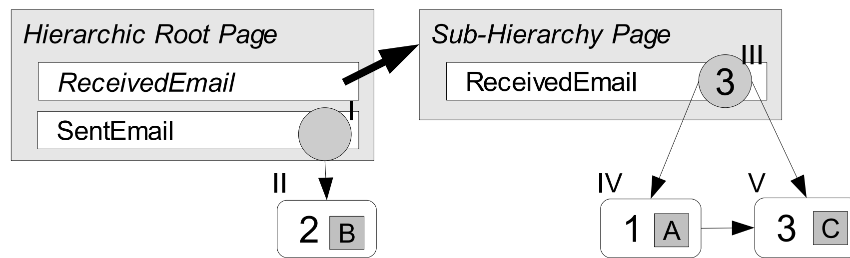



*Figure 32: Nested Index 7 - 1st Approach*

The **first approach for index 7** is depicted in Figure 32 (the fanout of the range index is 2, the minimum and maximum page size is 1. The minimum page size of the hierarchic index is 1 and the maximum page size is 2).

The leaf buckets of the hierarchic index do not contain data, but a root (initialization) page of the nested range index. This can be used to perform a further search with the range index.

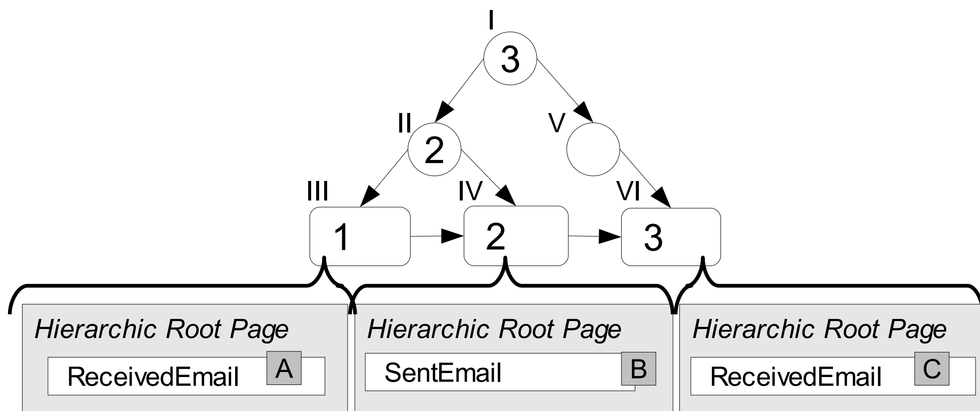



*Figure 33: Nested Index 7 - 2nd Approach*

The **second approach for index 7** is depicted in Figure 33 (same parameter

settings). Every leaf page of the range index contains a nested root page of the hierarchic index.

### 3.7.3 Algorithms

The algorithms of index structures that support nesting need to be adopted. The general process when a nested index is queried is outlined in Figure 34. At first the super index selects and retrieves the relevant page (super index page) according to the passed search configuration. This page contains a set of keys and the according initialization pages for the nested index. The page is selected and set as the initialization page in the nested index. Thereafter the super index forwards the query to the nested index.

The nested index starts traversal from the set initialization page, selects a page fitting to the search configuration, manipulates that page or returns the results to the super index. During this process the initialization page may have been updated (or deleted), therefore the super index retrieves the changed page and updates its own page (or removes the initialization page from its own page).

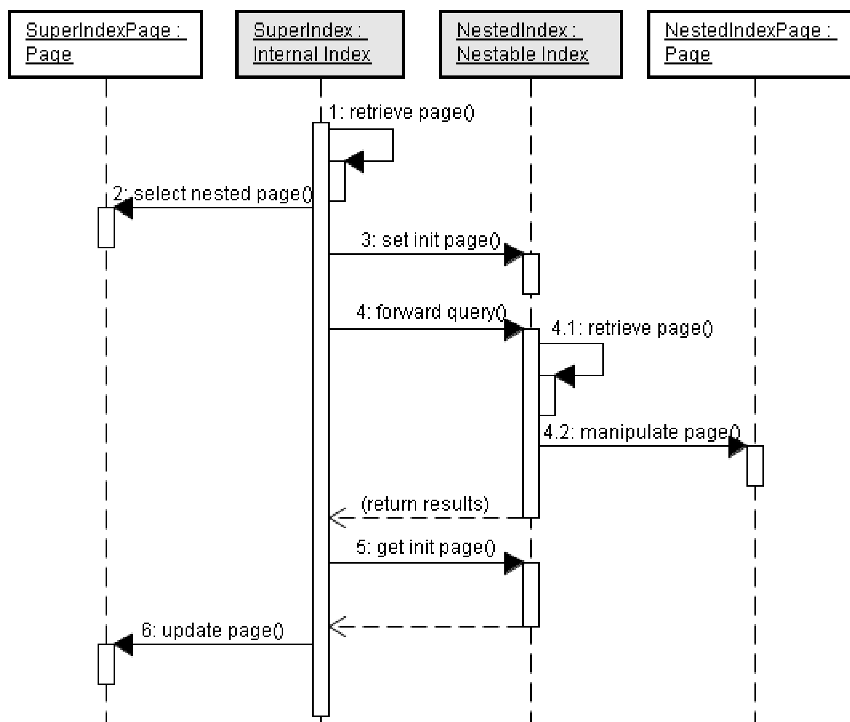



*Figure 34: Nested Index Processing Process*

In case more index structures are nested, this process is repeated. It is important to mentioned that the super index does not need to know the type of the nested index, as long as it is nestable and supports the setting and getting of initialization pages. Furthermore the super index does not need to care about the structure of the nested

initialization pages.

Consequently every nestable index can operate in two modes, one being an independent or super index and one being a nested index. In the first mode, the index needs to load and save the initialization page itself, in the second mode this is done by the super index.

We use **index 7 (first approach)**, which is depicted in Figure 32, to answer the sample query 9. At first the hierarchic root page is processed. The first search variable tells us to look for all received emails, consequently the sub-hierarchy page is loaded and the nested initialization page (III) is retrieved.

This initialization page is passed to the nested index, which looks for the date 1, as specified by the second search variable. The tree is traversed and email A is returned to the superior (hierarchic) index.

The hierarchic index aggregates the results (in this case this is not necessary, as only one initialization page has been retrieved by the hierarchic index) and returns email A.

# 4 Index Processing Architecture


**4.1 SemCrypt Architecture ..... 59**
**4.2 Logical Index ..... 61**
4.2.1 Index Variables ..... 61
4.2.2 Index Definition ..... 62
4.2.3 Index Configuration ..... 63
4.2.4 Search Configuration ..... 63
**4.3 Internal Index ..... 64**
4.3.1 Internal Index Definition ..... 65
4.3.2 Internal Index Configuration ..... 66
**4.4 Physical Index Representation ..... 66**
**4.5 Index Processing Components ..... 67**
4.5.1 Index Manager ..... 68
4.5.2 Index Engine ..... 69


Chapter four first introduces the SemCrypt architecture, which acts as a framework for the index processing architecture. The focus is on the components which manage and access index structures as well as on the interaction of index processing components with other components of SemCrypt. A concept for unifying index structures on an architectural level is presented, which regards index structures from a logical, internal and physical perspective. Then the logical and internal data models of index structures in the SemCrypt indexing framework are outlined. Also the physical representation of index structures is briefly discussed. This is followed by a structural explanation of the index processing components, their tasks and interactions with other SemCrypt DBMS components.

## 4.1 SemCrypt Architecture

The overall SemCrypt Architecture [GrKa06a] follows a layered approach, distinguishing four levels of abstraction (depicted in figure 35). The physical layer processes physical data and encapsulates storage, encryption and basic transaction capabilities [Dorn05]. The internal layer accesses and manipulates data with the help of the SemCrypt labelling scheme [GKSch05] and SemCrypt specific query and index processing techniques. The logical layer abstracts from internal representations and performs managing tasks. Finally the external layer provides a command interface and representation to the user.

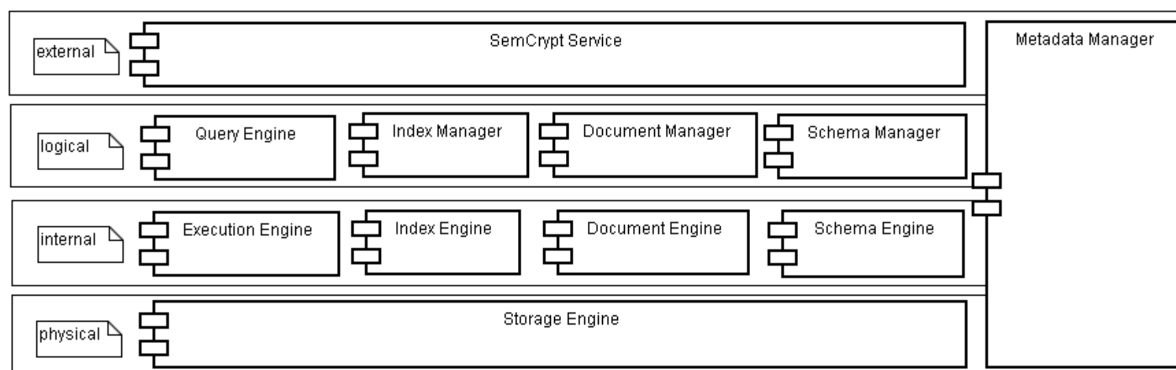



*figure 35: SemCrypt Architecture*

Each layer consists of multiple components that encapsulate SemCrypt functionality and interacts with other components of the same layer, use functionality of lower level components or provide functionality to upper level components. Components accept and return data of their level of abstraction. In case they require functionality of the underlying layer, the data first needs to be converted into the according representation. (Every layer has its own independent model, which abstracts from the model used in the underlying layer.)

The SemCrypt Service resides in the external layer and consists different of parsers, which transform external information transmitted by the user into the logical representation.

Components on the logical and internal layer are grouped in pairs according to their functional focus. The Query Engine (see [KaGr06b]) evaluates and optimizes logical queries and creates internal queries that are executed by the Execution Engine (see [KaGr06c]). The Index Manager deals with the creation, removal and management of index structures while the Index Engine creates, updates and traverses the index structures and provides an interface for accessing index structures. The Document Manager manipulates XML documents in their logical representation and transforms them into internal documents, which are used, stored, retrieved and manipulated by the Document Engine. The Schema Manager provides access to the logical schema and creates the internal schema that is used by the Schema Engine.

An exception is the Metadata Manager, which provides access to meta-data on all four layers. This is due to the fact that all components of the SemCrypt DBMS produce and require meta data (like available documents, schemas or index definitions). For more details on the Metadata Manager refer to Karlinger and Grün [KaGr06a].

The Storage Engine resides on the physical layer and handles database access, basic transaction management, serialization and encryption [Dorn05]. To enable parallel access to index structures in SemCrypt, advanced locking mechanisms and extended

transactions (like nested transactions, as analysed by Härder and Rothermel [HäRo93]) are required.

## 4.2 Logical Index

A logical index is independent from the type of the implemented index structure. This abstraction yields several benefits. A logical index provides a consistent data model for index structures that can be used independently of the implemented index structures. In case new index structures are implemented or existing index structures are changed, the logical index stays the same. This means that a logical index can be defined by telling what data to index and what data to return, without worrying which index (or set of indices) is going to be used internally to accomplish this task. Besides that a logical index also eases the selection of an appropriate index to support a specific query, as all logical indices are defined in a common way and can therefore be compared.

On the logical layer an index represents an access structure that can be used to retrieve certain data when provided with certain constraints (the keys of the index). This definition resembles the one presented in Chapter 2.1 and hides the internal structure and configuration of the associated index. A logical index also provides information about the retrieval costs associated with such an request. This cost model (which may be part of a later SemCrypt prototype) can be used by the Query Engine to select the fastest index if there are several suitable indices to choose from.

Additional meta-data defines the type of index to be used for indexing and the kind of nesting, when using multiple nested index structures. This meta data is specified in a logical index configuration, which is used to transform the logical index into an internal index (or a set of nested internal indices).

The following sub-sections explain the sub-elements of a logical index in more detail. At first index variables are introduced, which can be used to define an index. The chapter index definition describes how a logical index is defined. Then the logical index configuration used to contain additional information that is required for the transformation of a logical index into an internal index is explained. Finally search configurations, which can be derived from the index definition and which are used to query an index, are explained in more detail.

### 4.2.1 Index Variables

Index variables define the possible search parameters of an index and indicate the supported data and kind of look-up type (see Chapter 2.1.2) that is supported by the index. They are used to define an index by indicating what data is indexed by an index

(compare to Chapter 3.2.1). When index variables are assigned to a specific value, they can be used to query an index.

The XML documents contain value and structural information. Therefore we distinguish two main categories of index variables, value variables and hierarchic variables [Grün06b]. As value variables may be used in different kind of lookup-functions we refine them regarding the supported look-up type. This leads to the following five types of index variables:

- *Value Variables*, representing values belonging to nodes. These value variables can be further differentiated according to the supported comparison operations, which depend on the type of value that can be contained in the variable.
  - **Match Variables** that support an equals comparison on the contained value.
  - **Range Variables**, which support ordering and can be compared using smaller, greater and equals comparisons.
  - **Text Variables**, which contain keywords that can be compared using match and prefix comparisons.
- **Hierarchic Variables**, representing structural information. These can be further differentiated regarding the kind of hierarchic information expressed.
  - **Id Variables**, representing pointers to specific nodes in a document. These variables support hierarchic comparisons (isA and equals). They can be used to express the document structure of an XML document (path hierarchies), in the way it is demonstrated in Chapter 3.6.1.
  - **Type Variables**, indicating the type of a node. Like Id Variables, Type Variables are settled in a hierarchic structure and support isA and equals comparisons. Therefore they can be used to represent type hierarchies.

### 4.2.2 Index Definition

The index definition is independent of the implementation and type of the index structures and only describes the access interface of index structures. This is a big advantage, since the index definition is independent from the index structure and the Query Engine is able to compare and select indices comparing their index definitions. When new index structures are implemented the Query Engine does not need be changed, as the structure of an index definition stays the same.

The index definition defines which part of an XML document is indexed and what output is provided by an index. It determines the index interface explained in Chapter 2.1 and

specifies the lookup function of the associated index. The index definition is represented by logical operator graph that defines the index variables, which represent the key of an index. The return is determined by the output generated from the logical operator graph, which corresponds to the logical query graph (see [GrKa06b]).

### 4.2.3 Index Configuration

The index definition is not sufficient to define a logical index, as it misses information on which index structures to use when transforming a logical index into an internal one. In the case of nested index structures also the information on the nesting order is missing. Therefore a index configuration contains information on:

- The type of index to be used internally.
- The nesting order, when multiple internal indices are used to implement a logical index.

For example the index configuration for **index 1** looks like:

*[1]* $\$var1_{match}$ *--> Exact Match Index*

This means that the index, indexing $var1 is implemented with an exact match index. The assigned index must be able to support the relevant comparisons, in this case exact match capabilities.

The index configuration of **index 7** demonstrates the use of multiple keys and index nesting. The nesting order is $var1 --> $var2

*[1]* $\$var1_{structure}$ *--> Hierarchic Index*

*[2]* $\$var2_{range}$ *--> Range Index*

If the nesting order $var1 --> $var2 is chosen, the index configuration is:

*[1]* $\$var2_{range}$ *--> Range Index*

*[2]* $\$var1_{structure}$ *--> Hierarchic Index*

In case a multidimensional index (which may be implemented at a later point of time) is used the configuration looks like:

*[1]* $\$var1_{structure}$ *,* $\$var2_{range}$ *--> Multidimensional Index*

### 4.2.4 Search Configuration

To query an index it is necessary to restrict the variables, defined in the index definition, to certain values. This information is encapsulated in the search configuration, which

maps the variables defined in the index definition to certain values, ranges or hierarchies.

An advantage of search configurations in SemCrypt DBMS is that a search configuration corresponds to an index definition and therefore is independent from the realization of the index.

For example an exact match index variable ($\$var_{match}$) only encapsulates a value. However, a range match variable ($\$var_{range}$) defines an upper and lower bound and parameters that determine if the bounds are included.

A search configuration allows querying nested indices by passing a set of index variables. Every index picks the index variables that match its keys, processes the query and passes the search configuration to its nested index structure.

To demonstrate the concept, we transform three sample queries of the running example into search configurations for the index structures defined above.

Query 1 @ Index 1: $\$var1_{match}$ := *"michael@maier.de"*

The first search configuration just takes the values that are searched for.

Query 7 @ Index 7: $\$var1_{structure}$ := *SentEmailType*

$\$var2_{range}$ := *]..[*

Query 9 @ Index 7: $\$var1_{structure}$ := *ReceivedEmailType*

$\$var2_{range}$ = *[1 .. 1]*

Search configurations 7 and 9 are more complex, as they contain two index variables. The first one defines the hierarchy of the email type, the second defines the time frame of emails to be searched for (range on the date). Search configuration 7 defines the whole range for variable 2, while in search configuration 9 an exact match is expressed via the range variable.

## 4.3 Internal Index

The internal index is an abstraction of a variety of indices and their implementations and algorithms. It ensures that the Index Engine can access all internal index structures in a similar way, especially when updating, deleting or retrieving information. Furthermore it defines the basic capabilities indices must fulfil to allow index nesting.

The elements of an internal index are depicted in Figure 36. The internal index definition is the pendant to the logical index definition and can be used to determine what the index is indexing, which is essential to decide if an index is affected by updated data (index update). The index configuration holds relevant configuration parameters and meta-data required by the internal index. The data of the index is stored in pages that are written to and read from the Storage Engine at once. An internal index also provides internal costs that are calculated from relevant internal meta-data regarding the characteristics of the index and that are used to determine logical costs.

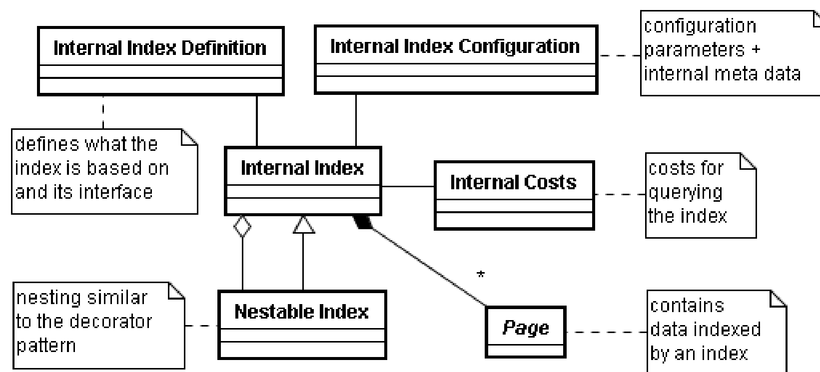



*Figure 36: Internal Index Elements*

As depicted in Figure 36 an internal index may have a nestable index underneath, thus realizing the index nesting concept. Every nestable index has the same elements as a regular internal index and may have an additional nestable index. This allows chains of nested index structures.

In the following subsections the elements of an internal index are explained in more detail. The search configuration and index variables are not considered, as they have already been discussed in detail in Chapter 4.2.

### 4.3.1 Internal Index Definition

The internal index definition is required to represent an index on the internal layer. Like the logical index definition it defines what parts of an XML document are indexed (the keys) and what is returned. This information is essential to determine which index structures are affected by changes of the primary data and to retrieve the necessary data to create or delete an index on an existing document (retrieve all the data to create or delete the index structure).

As the internal index definition contains the same information as the logical index definition, only the form of representation is different. In the SemCrypt DBMS an internal operator graph is going to be used for this purpose. This operator graph corresponds to an internal query graph (see [KaGr06c]). More information on the internal index

definition and its use for the index update problem is given by Grün [Grün06b].

### 4.3.2 Internal Index Configuration

Most index structures require certain meta-information or configuration parameters (like fan-out parameters or split sizes). This information is collected in the internal index configuration, that is stored as meta data. Consequently an internal index configuration is more specific than a logical index configuration, as it contains parameters relevant to a specific type of index structure.

An internal index needs to be able to retrieve pages that belong to it. Therefore every internal index configuration contains the internal index definition of the associated internal index. When internal indices are nested, every single index has its own index configuration, which contains the according identifier.

When an internal index is queried, a search configuration is passed to it (to the highest index in the nesting hierarchy). An index needs to determine, which index variables to regard (there may be multiple index variables in case indices are nested or a multidimensional index is used). Therefore the internal index configuration contains the identifiers of the index variables that can be processed by an index, so that an index can determine and extract the relevant index variables from a search configuration.

The amount of index variables and configuration parameters is dependent on the type of index structure. Consequently every internal index must provide an according internal index configuration.

## 4.4 Physical Index Representation

The physical representation of an index is the same as for primary data in SemCrypt. Every index is stored as encrypted id-value pairs on the not trusted storage provider. One id-value pair resembles a serialized (encrypted) index page and its page identifier. The content of this page is the value, the page identifier the key. The identifier of an index page only needs to be unique in the domain of index structures, as the Storage Engine ensures that these keys are extended in a way that they are unique in the overall domain.

The id is encrypted using a cryptographic hash function, while the value is enciphered using a encryption function. Consequently regarding the physical data it is not possible to tell, if the encrypted value contains primary data or index data, nor which index structure is retrieving which information. This is a core characteristic of SemCrypt's security

mechanism, as no information is leaked to the storage provider. More details on the physical representation of data is given by Dorninger [Dorn05].

## 4.5 Index Processing Components

According to the SemCrypt Architecture the processing and management of index structures occurs on four layers of abstraction that are represented by different components [Grün06a]. As the physical representation of information is uniformly managed by the Storage Engine, there exist two core components relevant for index processing (Index Manager and Index Engine). However, further components are involved, either accessing the index components or providing necessary functionality.

The whole index processing architecture is depicted in Figure 37. The components are depicted as rectangles and the arrows indicate a dependence relationship, meaning that one component uses functionality of another component. The SemCrypt Service depicted on the top is the interface to the user (external layer), which controls and initiates components of the logical layer. Regarding indices the SemCrypt Service will create, remove and alter indices via the Index Manager.

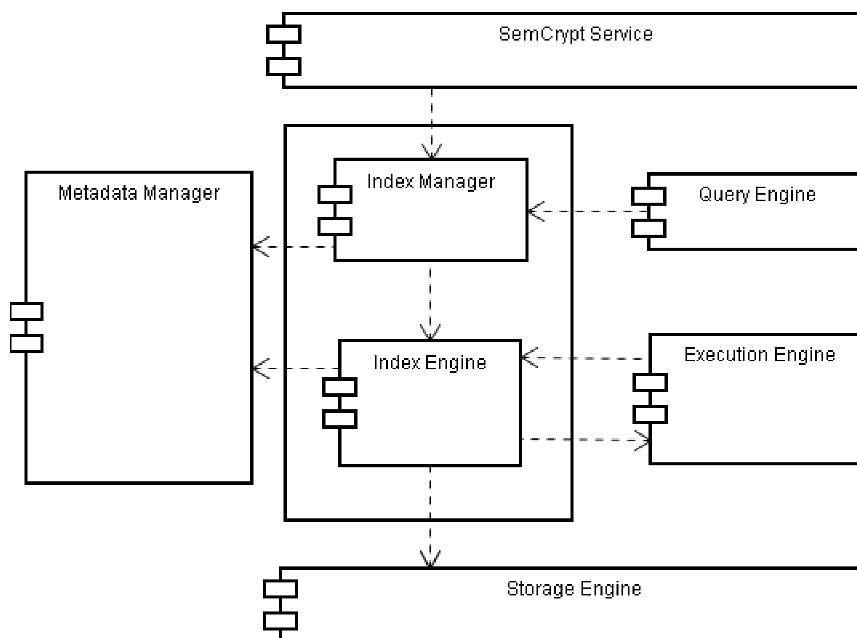



*Figure 37: Index Processing Architecture and Component Dependencies*

In the following subchapters the Index Manager and Index Engine components are explained in more detail, also regarding their interactions with other components.

### 4.5.1 Index Manager

The Index Manager abstracts from the internal representations of index structures and performs the tasks of index management and administration. It can be used by other components to gain access to index structures, determining which index structures currently exist and what they are based on. The Query Engine uses the Index Manager to retrieve meta-data, which is necessary to select appropriate index structures for query execution.

The Index Manager adds and removes indices, manages the logical meta data (like a cost model and index definitions) and reacts on changes in the document collection hierarchy, which may lead to changed index structures.

#### Index Creation and Deletion

The process of adding index structures is initiated by the SemCrypt Service, which transforms the external user input into the logical data model. It passes a logical index definition, a logical index configuration and an identifier to the Index Manager. The Index Manager stores this information in the meta-data and transforms the logical index definition into an internal index definition (via the Query Engine), which can be processed by the Index Engine. It then forwards the creation to the Index Engine and provides the Index Engine with the logical index identifier, the logical index configuration and the internal index definition. When removing an index it is sufficient to remove the according meta-data and to forward the logical index identifier, such that the Index Engine can delete the internal index.

#### Handling Indices on Multiple Documents

Sometimes it is desirable to define indices for a set of documents. Therefore the SemCrypt indexing framework supports the definition of indices on document collections (set of similar documents) and collection hierarchies (documents structured in a collection hierarchy). The Index Manager executes the transformation of these extended index definitions and forwards them to the Index Engine, so that the Index Engine is able to rebuild the affected indices and change its update routines.

In case a document is deleted and there are still indices existing for this document the Index Manager ensures that these indices are also deleted. If an index is defined on a collection of documents and a further document is added to this collection, the Index Manager directs the Index Engine to expand the existing index to this new document.

### 4.5.2 Index Engine

The Index Engine is a core component for index processing, as it updates and traverses the internal index structures described in Chapter 3. It accesses the Storage Engine and the Metadata Manager, so that index structures can permanently store index pages and access required meta-data. The Index Engine manages meta-data that is required to create and maintain internal index structures.

#### Index Creation and Deletion

One main task of the index engine is the addition and removal of internal index structures. When an index is added the internal meta-data of this index is written and the index is built. If an index is created on an existing document, the Index Engine retrieves the relevant data from the execution engine and creates the index structure. When an index is removed the Index Engine deletes the meta-data and ensures that all index pages belonging to that index are removed via the Storage Engine.

#### Index Update

The second important task of the Index Engine is to keep the index structures consistent with the primary data. Therefore index structures need to be incrementally updated. Index update is a twofold process. The first step is to determine the indices affected by changing data and to pass the appropriate data to these indices. The second step is the update of the index structure itself, through rebuilding or incrementally changing the index. The first step is not regarded in this thesis, while the second step is described for each index structure.

The Execution Engine notifies the Index Engine in case primary data is changing. The Index Engine then decides which indices are affected by the update. In case additional data is required for updating an index, the Index Engine retrieves this data via the Execution Engine. Afterwards the affected indices are updated. The detailed process of how to determine the affected indices is described by Grün [Grün06b].

#### Index Traversal

The third task of the Index Engine is the loading and traversal of indices to perform queries. When a specific index is queried or updated, the Index Engine loads the relevant index (including potentially nested indices) and forwards the task to the specific index. Thereafter the index (the according main memory object) is unloaded. This ensures that minimum resources are required at the SemCrypt client. If locking and concurrency controls are added at a later point of time, this allows the distributed access to index structures in case several SemCrypt clients share storage provider and meta-data.

The Index Engine is closely coupled with the Execution Engine. The Index Engine and its internal index structures make use of the operators provided by the Execution Engine to manipulate and compare data. This ensures that in case the Execution Engine is able to work with a new kind of data (by extending the operators), all index structures are capable to index this data as well. Index structures become independent of the type of data they index.

# 5 Implementation

**5.1 Logical Layer................................................ 71**
5.1.1 Index Variables................................................ 72
5.1.2 Logical Index................................................ 73
5.1.3 Index Manager................................................ 74
**5.2 Internal Layer................................................ 76**
5.2.1 Internal Index................................................ 77
5.2.2 Nestable Internal Index................................................ 79
5.2.3 Access to Persistent Data for Internal Indices................................................ 81
5.2.4 Index Engine................................................ 82
**5.3 Index Specific Details................................................ 86**
5.3.1 Sequential Access Structure................................................ 86
5.3.2 Exact Match Index................................................ 86
5.3.3 Range Index................................................ 87
5.3.4 Text Index................................................ 89
5.3.5 Hierarchic Index................................................ 90

Chapter five outlines implementation specific decisions and describes how the concepts described earlier have been implemented in the SemCrypt prototype. Details on utilized software and libraries can be found in Appendix A.

At first implementation details concerning the logical layer are outlined. This is followed by the discussion of the internal layer, focusing on the developed framework and its functionality. Finally the specific implementations of index structures are presented, describing implementation decisions and algorithmic details.

## 5.1 Logical Layer

The logical layer of the SemCrypt indexing framework consists of the Index Manager component and the logical index with all its associated meta-data. An according class diagram is depicted in Figure 38. The Index Manager manipulates and manages logical indices and saves the meta-data contained in the Index Configuration and Index Definition at the Metadata Manager. The logical index consists of an index identifier (a unique String), the Index Definition and the Index Configuration.

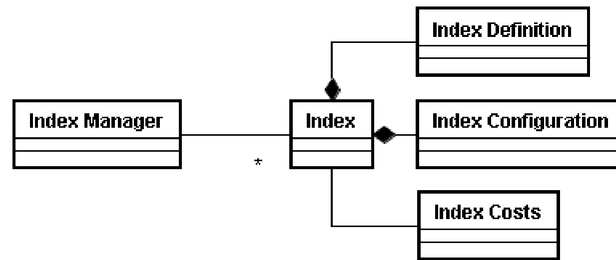



*Figure 38: Logical Layer Implementation*

In the following the implementation of the logical index and its elements (definition, configuration and costs) are described. Thereafter the implementation of the Index Manager component is outlined. We start with the implementation of the index variable concept, which is essential for the implementation of the index definition and configuration.

### 5.1.1 Index Variables

The concept of index variables has been introduced in Chapter 4.2.1. The implementation strategy is depicted as a class diagram in Figure 39.

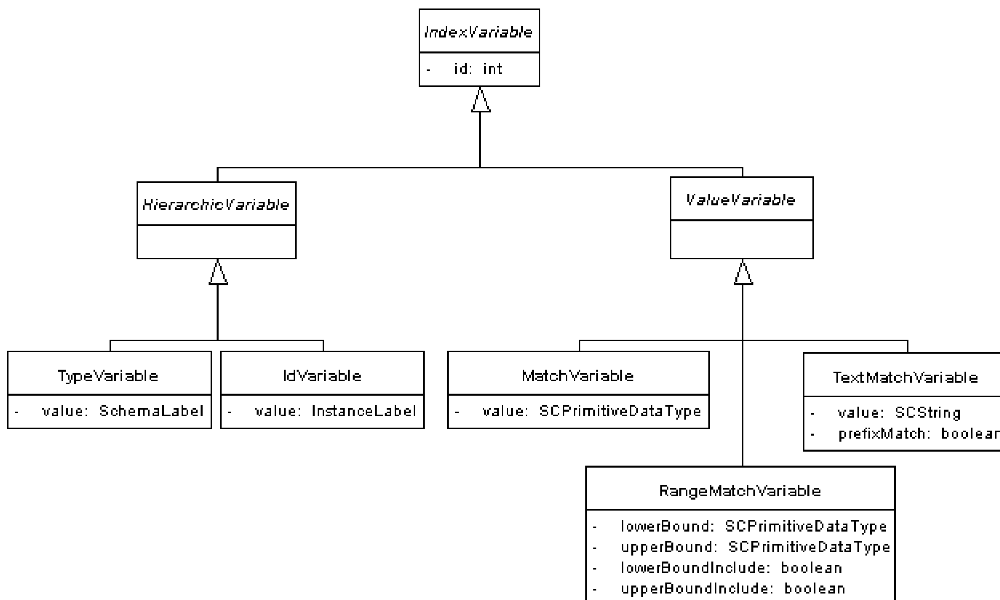



*Figure 39: Index Variables*

At the very top resides the abstract class Index Variable, which defines the basic capabilities of every index variable – a unique id. Basically, there are two types of index variables, hierarchic and value variables.

The hierarchic information can be further classified into type variables, which contain a type (expressed in a Schema Label) and the id variables, which contain a node identifier

(expressed in an Instance Label). Details about how structural information is encoded in labels in SemCrypt can be found in [GKSch05]

The Value Variable is further divided according to the supported comparison operations. The Match Variable just contains the value, which can be compared for equality. The Range Match Variable contains a range that is defined with an upper and lower bound and two indicators, telling if these bounds are included. Finally the Text Match Variable contains a value (keyword) and an indicator defining if the contained keyword is matched via a prefix comparison or equals comparison.

### 5.1.2 Logical Index

As depicted in Figure 40 a logical index consists of an Index Definition and an Index Configuration. The logical index itself is identified by a unique index identifier, which can be retrieved with the getIndexId method and which can be set with the setIndexId method.

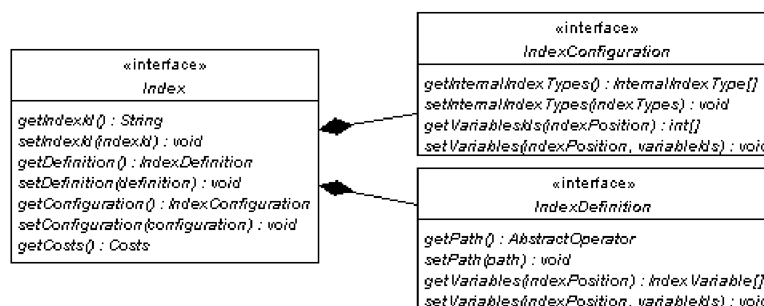



*Figure 40: Logical Index*

The logical index also provides a method getCosts, to retrieve a cost object – representing the query costs of the index. By now a cost model has not been implemented, however the logical cost model will make use of the internal costs provided by an internal index. In future implementations additional parameters may be necessary to retrieve the index costs.

#### Index Definition

The Index Definition contains the operator-graph defining the index and a set of Index Variables (see Chapter 4.2.1) that specify the type of query that can be posed at the index. The Index Variables are associated with an operator and specify the kind of data and type of query that can be performed. Due to the pending structure of the operator-graph the index definition has not been implemented yet.

#### Index Configuration

The Index Configuration defines the inner structure of a logical index. For this purpose it contains an array of Internal Index Types that describe the specific index to be used. In case the array contains more than one type, the order of the Internal Index Types in the array specifies the nesting order of the internal index structures. It also contains a two dimensional array of Index Variable Identifiers (so that multidimensional index structures, with more than one Index Variable, can be modelled). These provide the index variables for each internal index.

For example the Index Configuration of index 1 is:

Internal Index Type array: [Exact Match Index]

Index Variables array: { [1] }

Index 1 is implemented by an Exact Match Index, which is based on the index variable with id #1.

The more complex Index Configuration of the nested index 7 is:

Internal Index Type array: [Range Index, Hierarchic Index]

Index Variables array: { [1], [2] }

The super index is the Range Index and the nested index a Hierarchic Index. The Range Index indices variable is #1 and the Hierarchic Index variable is #2.

The same configuration for a multidimensional index would be:

Internal Index Type array: [Multidimensional Index]

Index Variables array: { [1, 2] }

### 5.1.3 Index Manager

The index manager extends the Component interface to provide basic initialization capabilities that are required by the SemCrypt prototype to deal with components.

Before explaining the methods in detail, some general concepts need to be introduced. As can be seen in Figure 41 every method has a transaction parameter, which is required to allow user defined transactions in the SemCrypt DBMS. This transaction object is passed to all methods that manipulate persistent meta data. Therefore it can be determined what data has been manipulated in a certain transaction.

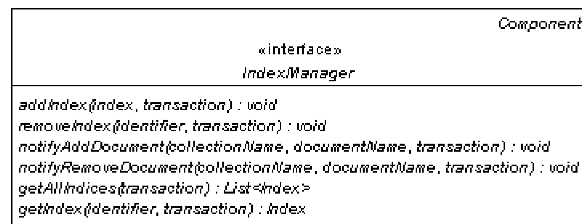



*Figure 41: Index Manager*

Also the handling of meta-data is similar to all components in the SemCrypt DBMS and therefore briefly presented here. Every component stores its meta-data via the Metadata Manager component, in a XML document defined by the component. The Metadata Manager operates like an independent XML database and can be queried with XPath statements (see [KaGr06a]). When a component is initialized and no meta-data document exists a new one is created.

The XML schema for meta-data stored by the Index Manager and example data related to the running example can be found in Appendix C. This schema contains an XML representation of the logical index identifier, the index definition and index configuration.

The following excerpt demonstrates the structure of the logical meta-data:

```
<Index ID="Index1">
  <Definition>
    <Operator OperatorId="1">
      <IndexVariable VariableType="SIMPLE" VariableId="1"/>
    </Operator>
  </Definition>
  <Configuration>
    <InternalIndex Type="EXACT_MATCH">
      <VariableId>1</VariableId>
    </InternalIndex>
  </Configuration>
</Index>
```

Every logical index has a unique logical identifier (ID), in this case Index1. It then contains a Definition and a Configuration. The definition contains an XML representation of the operator graph and associated Index Variables representing the keys of the index. The Index Variable is of the type SIMPLE (for simple queries) and has the id 1.

The configuration contains additional information, needed for translating a logical index into an internal index. It specifies the index-type (EXACT_MATCH index) and the ids of index variables associated with this internal index (id 1).

#### Add Index

A logical index and a transaction is passed to the Index Manager by the SemCrypt Service. The index manager validates that the requested index identifier does not exist yet and then saves the previously mentioned meta-data. The logical index definition is transformed into an internal one by the Query Engine.

The Index Manager then orders the Index Engine to create the internal index, by passing the logical id, the internal index definition and the logical index configuration.

#### Remove Index

After proofing that the requested index exists, the Index Manager tells the Index Engine to remove the according internal index (by passing the logical id). Then the logical meta-data is removed.

#### Notify Methods

These methods inform the Index Manager, when collections are changing. As index structures may be based on collections, the Index Manager needs to issue relevant commands at the Index Engine to update or rebuild the affected index structures. This functionality is not implemented in the prototype and can be added at a later point of time.

#### Getter Methods

The Index Manager also provides methods to retrieve a certain logical index (in case it exists) or to retrieve all logical indices currently existing. These methods are used by the Query Engine to retrieve the existing indices and to use the Index Definitions and Costs to select an index (the Index Definition is also used to create a search configuration for a specific index).

## 5.2 Internal Layer

The internal layer of the SemCrypt indexing framework consists of the Index Engine component (including some helper factory and adaptor classes) and the internal index, whose implementations contain the index specific algorithms. While the Index Engine performs management, creation and update tasks the internal indices perform the indexing, structuring and traversal of indexed data. The main classes of the internal layer are depicted in Figure 42.

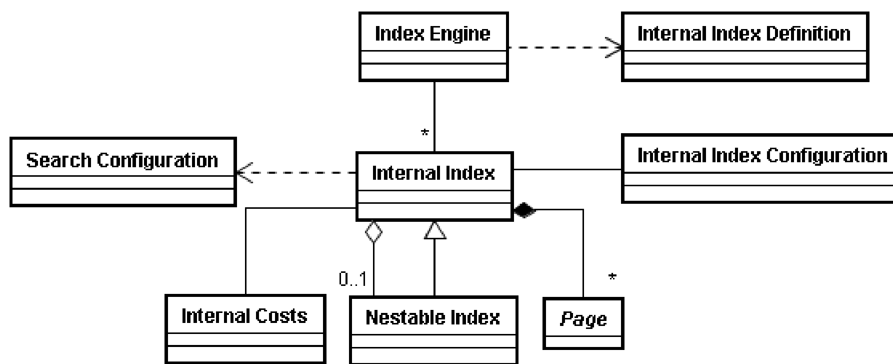



*Figure 42: Internal Layer Implementation*

The Index Engine stores internal meta data (Internal Index Definition and Internal Index Configuration) and accesses or manipulates the Internal Indices, by passing Search Configurations. The Internal Index contains pages and an optional nestable index. It also provides Internal Costs. Every Internal Index is configured by its Internal Index Configuration. In the following subsections the elements are explained in detail.

## 5.2.1 Internal Index

The internal index abstracts from specific index structures and defines the common interaction interface for processing index structures in SemCrypt. The internal index interface determines the functionality, which each index needs to support to be integrable in query and update processing. A class diagram of the internal index and the classes related to it are depicted in Figure 43.

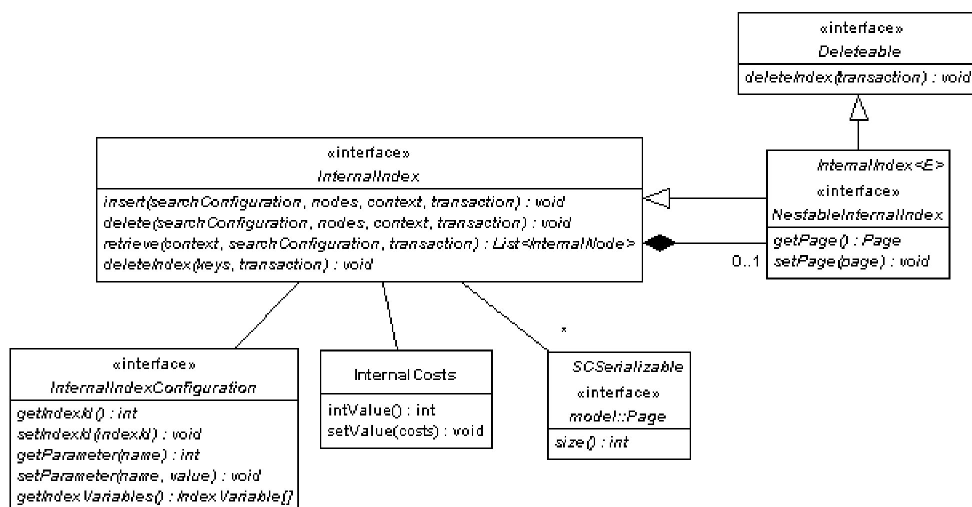



*Figure 43: Internal Index*

The internal index uses an index configuration to access index specific parameters and information. It provides internal costs, which can be used in a cost model to select index structures on the logical layer. Every internal index stores its data in pages, which must be serializable, so that they can be physically stored. Optionally an internal index

contains a nestable index.

The context information that needs to be passed to the internal index when manipulating or querying the index is required for being able to use the index operators provided by the Execution Engine (for example to compare values). This context specifies information that is needed by the operators (like schema information). Every internal index must provide the following four basic methods for interaction:

- **insert:** Inserts a set of nodes into the index. The keys for the insertion are defined by the passed search configuration. Also context information for the nodes and a transaction is passed. The implementing index needs to update its structure with the passed nodes.
- **delete**: The reverse action to the insert method. A set of nodes is removed. The passed information resembles the one of the insert method.
- **retrieve:** Queries the index with a certain search configuration. The index returns a set of nodes satisfying the query.
- **deleteIndex:** In case an index is deleted it is inefficient to remove all of its content via the delete method. To fasten this process an index structure provides a deleteIndex method. Using the passed set of keys, an index is able to fully delete itself. If an index does not require this set of keys (for example a tree-based index can delete itself, when it knows its root), an index can implement the Deleteable interface. The Index Engine will then choose this more efficient method in case the interface is implemented.

#### Index Configuration

The internal index is parametrized by its index configuration, which contains the internal id of the index, the identifiers of the associated index variables, configuration parameters and additional index specific meta-data that is required for the processing of an index. The kind of parameters and additional meta-data depends on the implemented index structures. The identifiers of the associated index variables are used to extract the relevant index variables from a search configuration passed to the index.

#### Pages

The physical data is stored by using implementations of the Page interface, which ensures that the pages can be serialized by the Storage Engine and that they provide a size method that is essential for index nesting and dynamic split behaviour.

### Internal Costs

Every internal index is able to provide an Internal Costs object, which contains information to determine the expected amount of index accesses for retrieval. These costs are calculated by the indices regarding their inner structure and current state (like the depth of an tree).

## 5.2.2 Nestable Internal Index

An internal index can contain a nested index, which needs to implement the Nestable Internal Index interface. This interface ensures that initialization pages of the nested index can be set and get and that it provides a deletion routine. The complexity of nestable indices can be reduced by using the Abstract Nestable Internal Index class (depicted in Figure 44), which hides a lot of the complexity related to nested index structures.

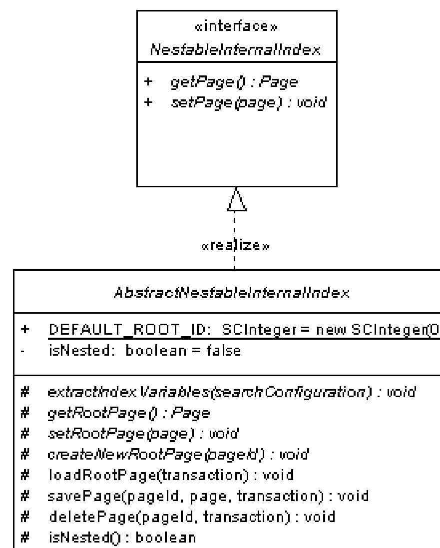



*Figure 44: Abstract Nestable Internal Index*

Abstract Nestable Internal Index provides the basic capability to use a nestable index in two settings, as an independent index or as a nested index.

The current state can be accessed using the isNested method. Abstract Nestable Internal Index implements the set and get page methods that are called by the super index to set and retrieve the initialization (root) pages and defines a set of abstract methods that must be implemented by nestable index structures to make use of the provided abstraction. These methods are:

- getRootPage, which retrieves the current root page of the nested index.
- setRootPage, which sets the root page of the nested index.
- createNewRootPage that tells the nested index to create a new root page using

the passed page identifier.

The Abstract Nestable Internal Index defines a default page id for the root page, in case the index needs to retrieve the initialization page on its own (independent mode).

> Index 7 consists of two internal index structures that are both nestable and therefore extend the Abstract Nestable Internal Index class. The range index is the super index, thus operating in an independent mode while the hierarchic index is nested and operates in a nested mode.
>
> If index 7 is loaded it needs to retrieve its initialization page (root page) on its own, and uses the default root page id. When the range index is traversed it locates nested pages in its leaves. These pages are initialization pages of the hierarchic index. The initialization page is set by the range index and the hierarchic index is traversed.

In the case of insert the problem occurs that an initialization page for the nested index may not exist. In this case the super index initializes the nested index with setPage(NULL). This notifies the nested index to create a new root page. After the insert operation the super index retrieves the new created root page and saves it as a nested initialization page in its own page.

A similar difficulty arises when deleting data from a nested index. After the deletion the affected part of the nested index may be empty and the root page can be deleted. This is indicated by the nested index by returning NULL in case the super index retrieves the initialization page after a delete operation. The super index then removes the initialization page from its own page.

Another problem is the interplay of different split algorithms used by the super and nested index. Therefore it is possible that the nested index splits when inserting data, which could require the super index to merge. Generally one thinks that an index can only grow when inserting data, but in this case the nested index performs a split, and the initialization page can shrink. The super index needs to regard this characteristic by testing the size of the nested page after every insert or delete operation.

If a nested index is deleted, the super index needs to delete all segments of the nested index. Therefore it passes all existing initialization pages to the nested index and calls its deleteIndex method.

### 5.2.3 Access to Persistent Data for Internal Indices

Internal indices need to persist two kinds of data, the indexed data using pages and meta-data that is accessed via the Internal Index Configuration. However, the Index Engine is the only component in the internal layer, which has access to this data. As internal index structures only require limited access and have specific requirements on the kind of data to save, this functionality is encapsulated by adaptor classes.

This allows the simplification of the communication interfaces, as transaction complexity is hidden from the index structures. Besides that the adaptor classes are able to provide buffering and caching functionality, specifically suited for indices (for example to build an index locally and to upload all data to the storage provider at once).

As depicted in Figure 45 there are two adaptor classes, the Storage Engine Adaptor that enables the internal index to get, set and remove pages (using the Storage Engine) and the Metadata Adaptor, which allows the Internal Index Configuration to get, set and remove meta-data (using the Metadata Manager).

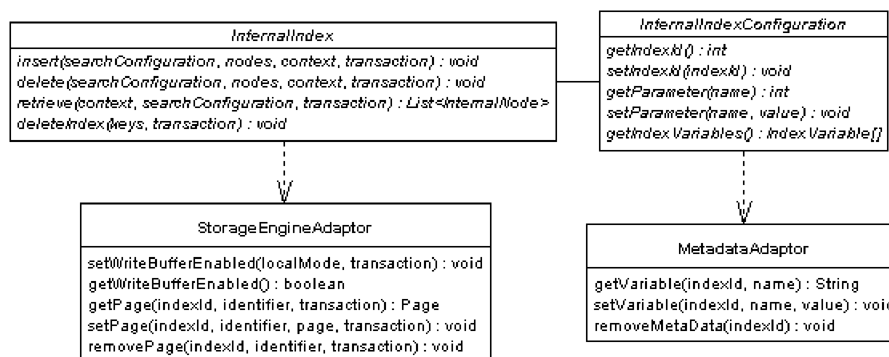



*Figure 45: Storage Engine and Metadata Adaptors*

**Storage Engine Adaptor**

The storage engine adaptor provides the following methods to an internal index:

- getPage: Retrieves a certain page identified by the index identifier and a page identifier.
- setPage: Stores a certain page belonging to the index identifier and a page identifier. In case a page belonging to the index id and page id already exists, it is overwritten.
- removePage: Removes the page belonging to an index id and page id.

In case a new index needs to be created it is extremely inefficient to incrementally build an index (transferring every single page on its own). Furthermore pages might be updated repetitively. This would require fetching and writing pages multiple times, creating a lot of overhead.

Therefore the storage engine adaptor provides a write buffer functionality, which enables the bulk creation of indices. This mode can be activated via the setWriteBufferEnabled method. From this moment, every set page is stored locally in main memory. In case a written page is retrieved, it is directly returned without the need of additional communication, encryption or serialization. After the index is fully built, the write buffer is disabled, which initializes the bulk transfer of all pages contained in the buffer.

The current implementation does not regard main memory constraints, which means that data is written to the buffer until it is disabled (the data is then transferred). Therefore the Index Engine needs to decide when to activate and when to commit the buffer. A future extension may automate this process.

For tree structures this strategy reduces the creation time 6x – 17x (range index with encryption). Even the creation time of an exact match index can be split 3x-5x.

#### Meta Data Adaptor

The internal index configuration contains all the meta-data that is required by an internal index. This meta-data can be accessed via the meta data adaptor, which provides the setting (setVariable) and getting (getVariable) of key-value pairs (especially suitable for setting and retrieving parameters).

Due to the missing implementation of the Metadata Manager at the time of implementation, the first version of the meta data adaptor saves its data in property files. However, it can be easily changed to save the data using any other data-source (the next version is going to use the Metadata Manager). Furthermore no index configuration or index structure needs to be changed, as the Meta Data Adaptor hides the data source and provides a consistent behaviour.

### 5.2.4 Index Engine

Like the Index Manager the Index Engine extends the Component interface, to allow for a common creation and interaction of the various components in the SemCrypt DBMS prototype. The Index Engine provides methods for creating and deleting internal indices, for updating indices and for posing queries at specific indices. It also provides functionality that can be used to retrieve a specific index (getIndex) or to determine which indices are currently existing (getAllIndices). A class diagram of the Index Engine and its helper classes is depicted in Figure 46.

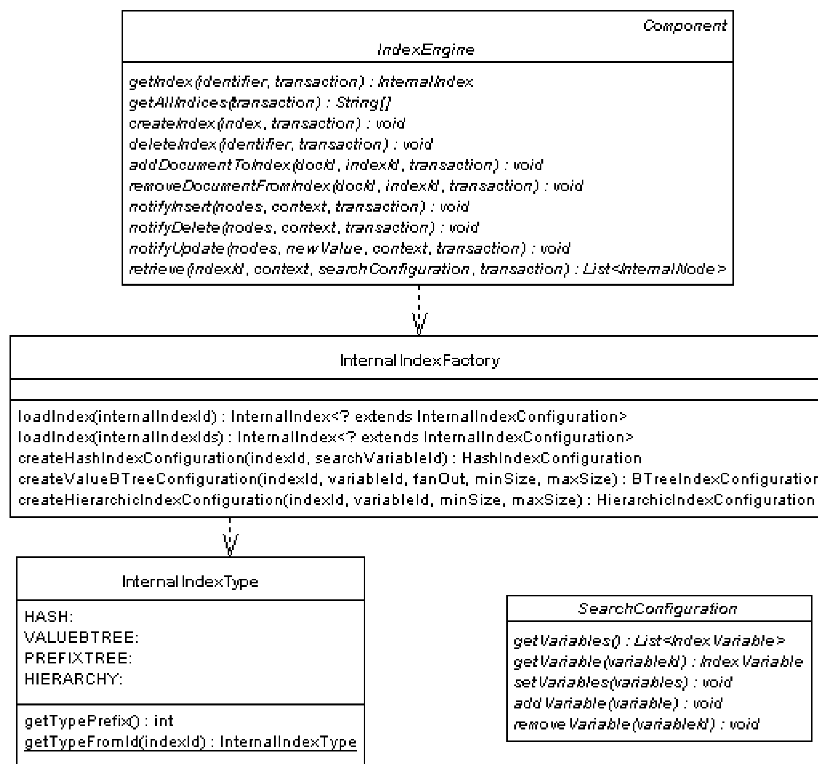



*Figure 46: Index Engine*

### Create Index

The Index Manager passes the logical index identifier, the logical index configuration and the internal index definition to the Index Engine. The Index Engine inspects the logical index configuration and creates according internal index configurations. Thereby the logical index identifier is mapped to a set of internal index identifiers (in case the logical index configuration defines a set of nested indices).

The Index Engine then saves the index identifier mapping, the internal index configurations and the internal index definition via the Metadata Manager. When the index update is implemented at a later point of time, also update patterns will be created using the internal index definition (see [Grün06b]).

In case the index is created on an existing document the index needs to be created. The relevant nodes can be retrieved by passing the internal index definition to the Execution Engine.

The creation of internal indices is a twofold process. Initially an internal index is created with the definition of its internal meta-data. However, at this point of time no index object is created. Index objects are created dynamically when operations need to be performed on a specific index. This process is performed by the Internal Index Factory, which provides methods to create internal indices from the meta data. The Internal Index Factory also supports creating internal index configurations that are required for the

definition of internal index structures.

Creating an internal index object from internal meta-data is a complex task, as internal index structures may be nested. By encoding additional type information into the internal index identifiers, the Internal Index Factory is able to create the internal indices from an array of internal index identifiers.

The Internal Index Type enumeration, which is also used in the (logical) Index Configuration, is used to define the type of index to be used. An internal index identifier (4 bytes integer) has the following conceptual structure:

- The first byte of the internal index identifier is the type id, expressing the Internal Index Type.
- The remaining three bytes can be used to create a unique identifier.

This allows the definition of $2^8$ (256) different index types, and $2^{24}$ (over 16 million) indices per index type. By using the type information, the Internal Index Factory is able to create the according index. The index identifier is also used to retrieve the relevant Internal Index Configuration, which is required by every index. (The mapping can be determined by using the getTypePrefix and getTypeFromId methods from the Internal Index Type enumeration.)

We assume that index 7 has been created and its logical identifier is "INDEX_7". This identifier is mapped to internal index identifiers (two, as there are two indices that are nested). The first byte of the index identifier contains the index type.

RANGE has the type id 1

HIERARCHIC INDEX has the type id 2

The other three bytes can be used to create a unique id. So exemplary the following two internal ids are created: [1]001 and [2]001. Every number represents one byte, while the first byte in brackets is the Internal Index Type id.

When the Internal Index Factory is passed these two ids, it is able to load the relevant Internal Index Configurations, to determine the type of index and the nesting order. The created index can then be accessed by the Index Engine.

### Delete Index

When an index is deleted, the index engine first delegates the deletion to the internal index. An internal index can either be able to delete itself without additional information, or requires the indexed keys (exact match index). In case an index implements the Deletable interface, the first, more efficient method is chosen. Otherwise the Index Engine determines the indexed keys and then passes these keys to the internal index.

After the internal index has been deleted, the Index Engine removes the internal meta data (Index Definition, Index Configurations, update pattern and the mapping of the logical index id to the internal index ids).

### Retrieve

The retrieve method is essential for the Execution Engine, as it allows posing a query at a specific index. This query is expressed using a Search Configuration, which contains all required index variables for the index (as defined in the index definition). Values are assigned to these index variables that are used to direct the search and return the requested information.

The Index Engine determines the affected index using the logical index identifier and uses the Internal Index Factory to create an instance of the required index (or set of nested indices). The Search Configuration is forwarded to the internal index and the results are returned. Afterwards the internal index is unloaded. This means that the main memory index object is destroyed. Therefore index structures only require memory and processing time, when there are actively used. Besides that all data regarding index structures is permanently saved.

### Notification Methods

The notify methods are used by the Execution Engine to inform the Index Engine when data is changing. The Index Engine determines the affected index structures and performs the required updates. This index update functionality is described by Grün [Grün06b] and its implementation was not part of this thesis.

The add- and remove document to index methods are relevant for index structures that are defined on collections. The Index Manager uses these methods to instruct the Index Engine to add or remove a document to or from an index. Due to the missing support for collections in the current prototype these methods have not been implemented.

## 5.3 Index Specific Details

After the general mechanisms have been outlined we now explain the index specific implementation details. The core algorithms are not considered as they have been presented in Chapter 3.

At first an additional access structure is presented, which is used to abstract from the indexed data and from the problem that one key in a secondary index might contain multiple values. It makes use of the index nesting capabilities and therefore can be used by every index structure. Then implementation details of the exact match index, the range index, the text index and the hierarchic index are explained.

### 5.3.1 Sequential Access Structure

The need to index different kind of data, together with the situation that there might by multiple nodes belonging to one key formed the idea of a simple access structure that is able to encapsulate this complexity from other index structures. The sequential access structure is the result of these considerations and has the positive side effect that in future it can implement additional functionality, like maximum page sizes, data compression or determining whether index structures are allowed to contain duplicated data.

The sequential access structure is nestable and can be used by every other index. In a nesting hierarchy it is always situated at the lowest level. The sequential access structure itself does not read or write any pages itself, but provides its sequential pages for other indices to nest.

Currently the sequential access structures only maps a key to a sequence of nodes. It does not regard page size restrictions nor an order of the contained nodes. This functionality may be added in later implementations. A sequential page contains a sequence of indexed data and resembles a record. This means that data belonging to one index key is aggregated in a sequential page, allowing a superior index to deal with this data in a way it would with one entry.

The sequential access structure does not have a configuration nor an internal id and is not able to store data by itself. As the insert, delete and retrieve algorithms are trivial they are not outlined (adding, removing and retrieving nodes in a linked list).

### 5.3.2 Exact Match Index

The exact match index is the implementation of a simple inverted file in the SemCrypt indexing framework. It cannot be used as a nested index, due to the kind of

segmentation (key = page identifier) and requires a list of index variables containing the indexed keys for deletion.

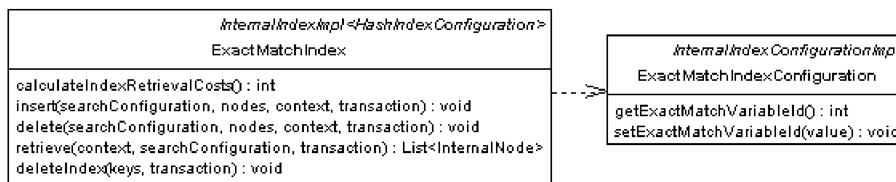



*Figure 47: Exact Match Index Implementation*

#### Configuration

The exact match index stores its meta-data in a Exact Match Index Configuration (depicted in Figure 47). The configuration only contains the identifier of the exact match variable, specifying the key of the index. No additional parameters are required.

#### Pages

The exact match index makes use of the sequential pages of the sequential access structure to store its data.

#### Methods

To determine the page id belonging to a certain key, the exact match index uses the serialized representation of the key, as this ensures that the equals condition is preserved.

The estimated costs (calculated by the calculateIndexRetrievalCosts method) are always one, as the exact match index only requires one page access to retrieve the desired information.

### 5.3.3 Range Index

The range index can be used as a nestable index and therefore extends the Abstract Nestable Internal Index class. Being a tree-based index structure the range index is able to delete itself and therefore implements the Deletable Interface.

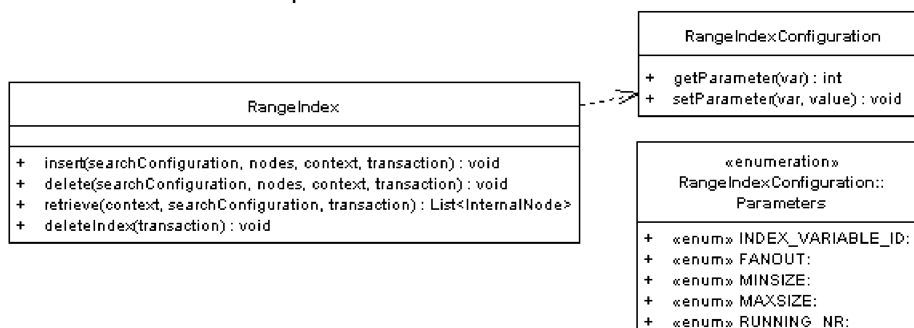



*Figure 48: Range Index Implementation*

### Configuration

The Range Index Configuration contains different parameters (specified in the Range Index Configuration Parameters enumeration) required by the range index (see Figure 48):

- INDEX_VARIABLE_ID, the identifier of the Range Match Variable supported by the range index.
- FANOUT, defining the branching of the tree and the maximum size of a branch page. Indirectly also the minimum branch page size is determined, as for the adopted B+tree it is half the fanout.
- MINSIZE, specifying the minimum size of a leaf-page.
- MAXSIZE, specifying the maximum size of a leaf page.
- RUNNING_NR a running number that is used to generate unique page identifiers in combination with the internal index identifier.

### Pages

The range index uses two types of pages the Leaf Page, containing the data and the Branch Page creating the tree structure. These are generalized by the Btree Page class, which contains a page identifier and an array of index keys (see Figure 49). It extends the Page interface and ensures serialization capabilities. Besides that the Btree Page ensures that Branch Pages and Leaf Pages implement split, merge and redistribute methods.

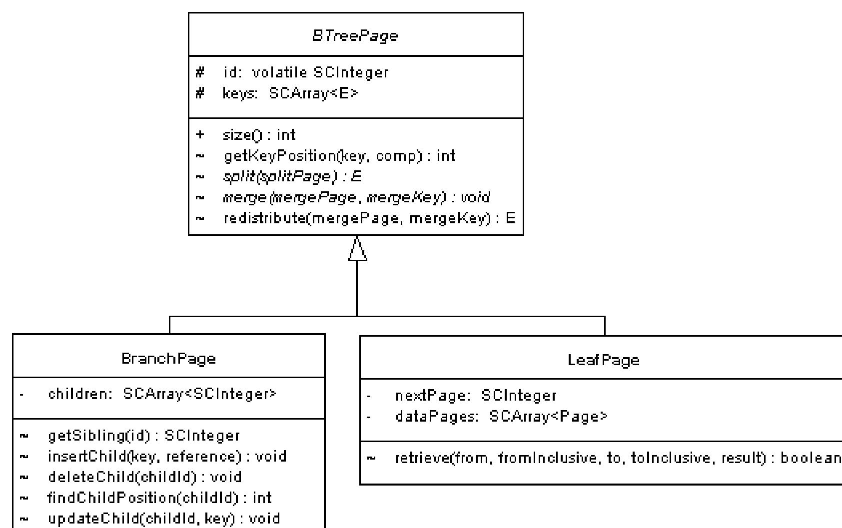



*Figure 49: Range Index Pages*

A Leaf Page contains an additional array of nested pages and the page identifier of the next Leaf Page (that is null in case there is no next leaf). The Branch Page contains an array of page identifiers indicating the children pages of this branch page.

#### Methods

The expected retrieval costs of the range index equal the height of the tree, which can be estimated using the current RUNNING_NR and the FANOUT value.

A core method of the range index is the findLowerBoundLeafPage method that traverses the index and returns the leaf page fitting to the lower bound. This helps locating the right leaf page for deletion, insertion and retrieval. The deleteIndex method traverses the whole index and removes every encountered page.

The split, merge and redistribute methods are used in case a Btee Page is split or merged and copy the relevant data. The split routine returns an index key that indicates the value which can be used to separate the two pages. In case of a Branch Page this is the minimum value of the part-tree, which contains the greater values.

The redistribute method is used in case one page underflows but cannot be merged with another page (as the new page would overflow). This operation can be expressed by a merge followed by a split

### 5.3.4 Text Index

As depicted in Figure 50 the text index is based on the range index and extends it. It overwrites the insert, delete and retrieve methods to transform the passed text match variable into a range representation and to pre-process text using the Text Analyzer (see Chapter 3.5). Algorithms and index configuration correspond to the range index.

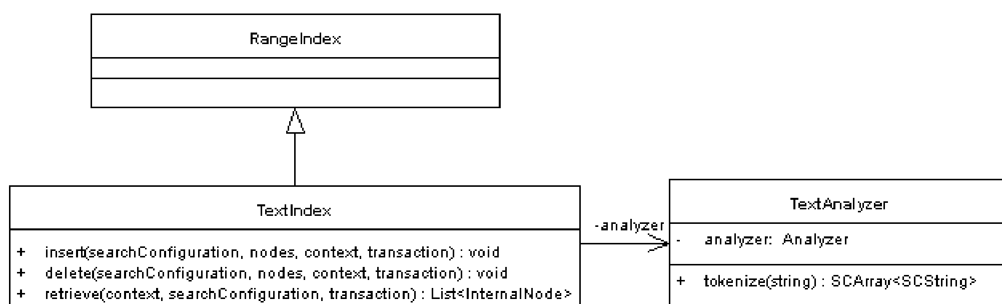



*Figure 50: Text Index Implementation*

#### Methods

The prefix operations needed for key compression are performed by a separate operator, which provides one method to determine the minimal prefix of two Strings and a second method to create the next higher prefix for a prefix.

In case the minimal prefix of “integrating” and “index” is calculated at first the two strings are sorted: “index” --> “integrating”. Then starting from the beginning a new string is created, till a difference is found: i --> n --> d/t The letter of the larger keyword is used and the prefix “int” is returned.

The next higher prefix for “int” is $\lim(\text{'int'}+x)=\text{'inu'}$ . This is implemented by adding the character '~' to the string (“int~”), which does not occur in keywords and which satisfies this condition.

#### Text Analyser

The Text Analyzer provides the tokenizing capabilities required by the text index to prepare text. For this purpose it uses the Lucene library. The Text Analyzer provides the method tokenize, which takes a string as input. This string is cleaned from punctuations, is tokenized into keywords and stemmed (executed by the Standard Analyzer class of Lucene). An example is provided in Chapter 3.5. The resulting keywords are then indexed in a range index manner.

In case other languages than English need to be regarded the underlying analyser can easily be changed.

### 5.3.5 Hierarchic Index

The hierarchic index can be used as a nested index and extends the Abstract Nestable Internal Index class. It provides the indexing of hierarchic data, while providing dynamic split behaviour. The hierarchic index and its index configuration is depicted in Figure 51.

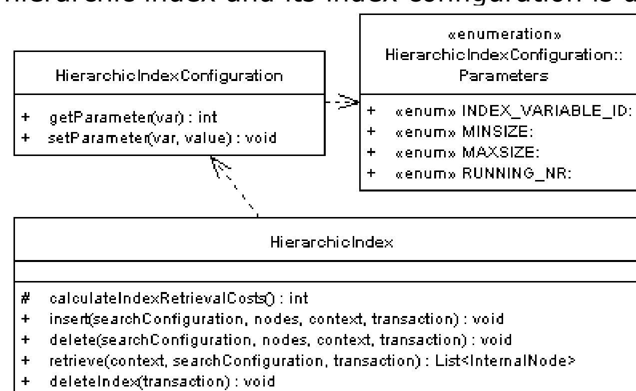


*Figure 51: Hierarchic Index Implementation*

#### Index Configuration

The hierarchic index makes use of an internal index configuration that contains the following meta-data:

- INDEX_VARIABLE_ID, the identifier of the Hierarchic Variable supported by the

hierarchic index.

- MINSIZE, specifying the minimum size of a hierarchic page.
- MAXSIZE, specifying the maximum size of a hierarchic page.
- RUNNING_NR a running number that is used to generate unique page identifiers in combination with the internal index identifier.

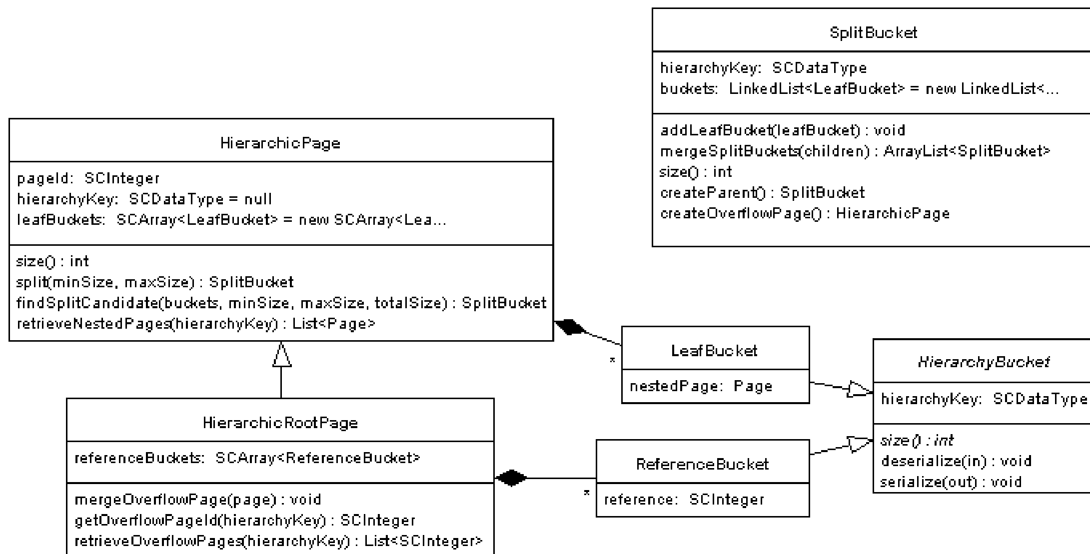



*Figure 52: Hierarchic Index Pages*

#### Pages

The hierarchic index consists of Hierarchic Pages and Hierarchic Root Pages, which contain the data segmented into hierarchies using Hierarchy Buckets. As depicted in Figure 52, a Hierarchic Page only contains Leaf Buckets, which consist of a hierarchy key (representing the hierarchy) and a nested page. The Hierarchic Root Page additionally contains Reference Buckets that contain the page identifier of another Hierarchic Page that is used as a sub-hierarchy page.

The class Hierarchic Page provides methods for splitting the page (split) and for retrieving all nested pages that belong to a certain hierarchy key. The Hierarchic Root Page provides an additional method to merge with a sub-hierarchy page (Hierarchic Page) in case of an underflow.

#### Methods

The split algorithm (visualized in Figure 27 on page 52) is the most complex part of the hierarchic index and makes use of the Split Bucket class, which is used as a helper class. The findSplitCandidate method determines the Split Bucket that can be used to split a Hierarchic Page. A Spit Bucket represents a part-tree of the hierarchic structure and

contains all Leaf Buckets that belong to this part-tree.

Recursively Split Buckets are created (by rebuilding the hierarchic tree from bottom-up), until one is found that satisfies the split conditions (greater than the minimum page size, while the remaining page is also greater than the minimum page size). The Split Bucket then provides methods that allow the creation of a sub-hierarchy page (createOverflowPage).

If a sub-hierarchy page splits, reference buckets that are in a sub-hierarchy relationship with existing reference buckets are created. This leads to a decision problem in case of a future insert, as multiple reference buckets (and the according sub-hierarchy pages) may contain the data. This problem is solved by always choosing the reference bucket, which is closer (in the hierarchic tree) to the hierarchy of the data to be inserted.

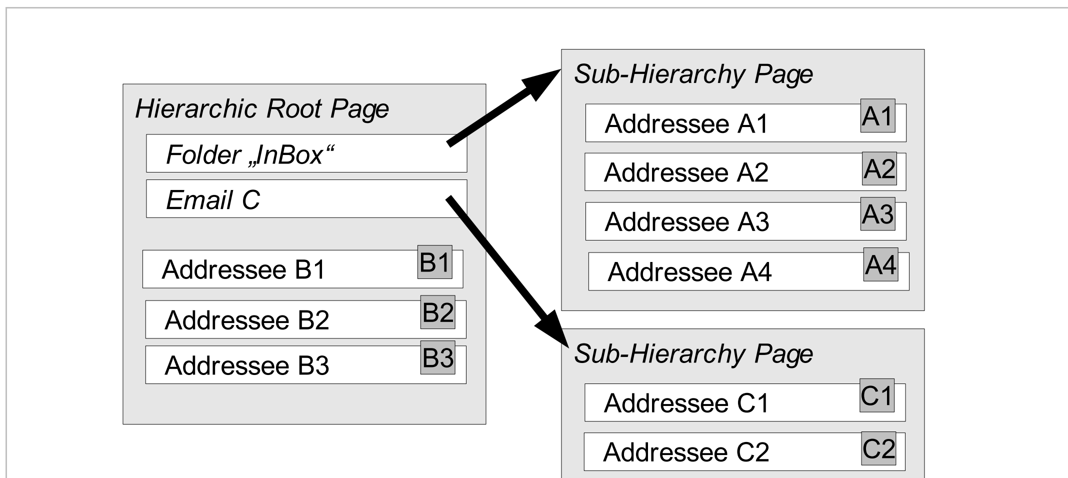



*Figure 53: Hierarchic Index 6 - After Second Split*

Using the example that has been discussed earlier (depicted in Figure 30) and that is shown again in Figure 53, we can see that in case we insert a new email address belonging to the hierarchy *Email C*, the Hierarchic Root Page provides us with two possibilities: the hierarchy *Folder "InBox"* (which contains the *Email C* hierarchy) and the hierarchy *Email C*. The algorithm chooses the hierarchy closer to the affected hierarchy, in this case *Email C*. Consequently the new email address is correctly added to the second sub-hierarchy page.

# 6 Evaluation, Conclusion and Outlook

**6.1 Evaluation........................................................................................ 93**
6.1.1 Criteria........................................................................................ 93
6.1.2 Applicability........................................................................................ 94
6.1.3 Extensibility........................................................................................ 95
6.1.4 Security........................................................................................ 96
6.1.5 Storage and Memory Consumption........................................................ 97
6.1.6 Index Creation Performance........................................................ 99
6.1.7 Index Retrieval Performance........................................................ 101
**6.2 Conclusion........................................................................................ 104**
**6.3 Outlook........................................................................................ 104**

This chapter evaluates the implemented index structures using qualitative comparison and quantitative measures. Thereafter the thesis is concluded and an outlook on future research work and possible extensions of the developed concepts and the implementation is given.

## 6.1 Evaluation

It is important to evaluate the implemented index structures to determine if the requirements have been met and to analyse the core characteristics of the index structures and the developed framework. Therefore at first criteria are outlined that are then applied to the index structures.

### 6.1.1 Criteria

Zobel et al. [ZMR95] provide a set of guidelines for evaluating and comparing index structures and describe a set of criteria, which are modified and extended for SemCrypt:

1. **Applicability**, the class of queries supported by an index.
2. **Extensibility**, the ease to modify and extend an existing index to operate on different data or to support additional queries.
3. **Security**, the level of security that is provided by an index and potential weaknesses (this criterion is missing in [ZMR95], but highly relevant for the SemCrypt DBMS).

4. **Storage and Memory Consumption**, the disk space consumed by an index and the required memory during manipulation and retrievals (combining the criteria outlined in [ZMR95]).
5. **Index Creation Performance**, the ability of an index to update itself in case data is inserted, modified or deleted.
6. **Index Retrieval Performance**, the ability of an index to identify answers to queries in a reasonable time.

Transaction and communication costs to the storage provider and the overhead created by encryption are not considered for index structures, as they are dependent on the Storage Engine and the used encryption algorithms and transmission protocols. These parameters highly influence the performance of index structures in SemCrypt, but they are not specific to indices and also apply to primary data. The implications of the index structures for concurrency, transactions and recoverability are not considered, as the current SemCrypt prototype interacts in a stand alone environment.

In the following chapters we apply the outlined criteria to the implemented index structures. The scalability of index structures is regarded together with query evaluation speed and index update speed. Also the implications of different index configuration parameters are considered.

### 6.1.2 Applicability

The kind of queries to be supported by index structures were outlined in the objectives (Chapter 1.5) and further detailed in Chapter 3.1.1. We now determine which index structures are able to support these queries and how well they support them.

Table 9 shows the correlation of query classes to index structures. The first line shows the different queries while the first column issues the index structures. The cells in between rate how well an index supports a certain query, or using a different perspective, which index structure can be chosen to support a certain type of query.

Rating:

- \+ The support is strong and the index is an ideal candidate for supporting the corresponding query type.
- o The query class can be supported using the index, but there are better choices.
- \- The query cannot be supported using the index structure.

As shown in Table 9 every required query class is efficiently supported. The range and text index also support simple lookups, but are less efficient than the exact match index.

| *Query Type* / *Index* | *Simple Query* | *Range Query* | *Text Query* | *Structural Query* |
|---|---|---|---|---|
| Exact Match Index | + | - | - | - |
| Range Index | o | + | - | - |
| Text Index | o | - | + | - |
| Hierarchic Index | - | - | - | + |

*Table 9: Applicability of Implemented Index Structures*

By the use of nested index structures more complex queries can be supported. It makes sense to combine the exact match, the range and the text index with the hierarchic index to add support for structural queries. When a multidimensional index is not available it makes sense to combine an exact match index with a range or text index to emulate a multidimensional index. However, nesting two range or text indices is hardly useful, as the combined tree structures segment the two-dimensional data in an inefficient way.

| Index Class | Query Type | *Simple* | | *Range* | | *Text* | | *Structural* | | *C* |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 |
| *Exact Match* | Index 1 | √ | √ | - | - | - | - | - | - | - |
| *Range* | Index 2 | - | - | √ | √ | - | - | - | - | * |
| *Text* | Index 3 | - | - | - | - | √ | - | - | - | - |
| | Index 4 | - | - | - | - | - | √ | - | - | - |
| *Hierarchic* | Index 5 | - | - | - | - | - | - | √ | - | - |
| | Index 6 | - | - | - | - | - | - | - | √ | * |
| *Nested* | Index 7 | - | - | √ | √ | - | - | - | √ | √ |

*Table 10: Running Example Queries and Index Support*

Table 10 shows the nine sample queries defined in the running example (Chapter 1.4) and visualizes, which of the sample indices (Chapter 3) can be used to support a query (√), which indices partly support a query (*) and which indices cannot be used to support a query (-). On the left side the according index class is stated and the top indicates the query class (C stands for complex query).

### 6.1.3 Extensibility

Extensibility is defined as the ability of index structures to support different types of data and the ability to support new index structures or the combination of existing ones.

Table 4 (Chapter 2.1.1) defined the different domains that can be indexed in SemCrypt. We focus on the three core domains in SemCrypt, which are shown in the first row of Table 11. The three core domains are values and hierarchies, expressed through

identifiers (nodes) and types. The first column names the different index structures and the cells in between show if a domain is supported (+) or not (-).

| *Domain (key)* / *Index* | *Values* | *Identifiers* | *Types* |
|---|---|---|---|
| Exact Match Index | + | + | + |
| Range Index | +[1] | +[3] | +[3] |
| Text Index | +[2] | - | - |
| Hierarchic Index | - | + | + |

[1] if values can be ranked (linear order)
[2] if values are Strings (chain of characters)
[3] in case the hierarchies follow a linear order (in-order tree traversal)

*Table 11: Extensibility of Implemented Index Structures*

The exact match index can index all three domains, however only supports simple queries. The range index supports values as long as values of the domain can be ranked in a linear order and the text index supports values that consist of character chains. The hierarchic index supports identifiers and types, regarding their hierarchic structure.

While the data indexed in SemCrypt are nodes of a document, the implemented index structures are independent of this data and only limited by the domains that can be used as index keys.

The presented index framework is very generic and due to the introduced abstraction can be easily expanded. As soon as the implementation classes and required meta-data are added to the framework, the new index structures can be used for indexing. To add an additional index, the internal index and internal configuration interface need to be implemented and the internal index factory needs to be extended. If new data types need to be indexed, it is sufficient to extend the relevant operators of the Execution Engine. The indexing of new domains requires more adoptions, as new index variables need to be declared and the components dealing with index variables (Query Engine, Execution Engine) need to be changed to accommodate the new type of index variable.

### 6.1.4 Security

Security is a very important requirement for SemCrypt. The use of index structures creates the following two major risks:

- Detecting Index Structure: An index structure has a certain structure. This risk describes the possibility that an intruder is able to identify an index structure (separate it from other index structures and the primary data) and can rebuild the structure of the index. This may be done by using frequency analyses to determine the root nodes of a tree-based structure.

- Detecting Content: Index structures duplicate data. This risk describes the possibility that an attacker is able to relate the content of index structures to the primary data. For example by monitoring changes in the primary data and the index structures and by using this information to group potential similar values. The worst case scenario is that an attacker can extract probabilities of occurrences of certain values.

As these two risks seem to be related to the kind of operation executed on the index structure, we divide them into risks when updating the index and risks when accessing the index structure. Potential risks are symbolized by +, no risk by – in Table 12.

| *Security Issue* / *Index* | *Detecting Index Structure* | | *Detecting Content* | |
|---|---|---|---|---|
| | *U* | *R* | *U* | *R* |
| Exact Match Index | - | - | + | - |
| Range Index | - | + | - | - |
| Text Index | - | + | - | - |
| Hierarchic Index | + | + | - | - |

**U** .. Issue when **updating** the index structure.
**R** .. Issue when **retrieving** via the index structure.

*Table 12: Index-Security Risk Matrix*

The exact match index creates a risk regarding the detection of content, when updating (as described in Chapter 3.3.4). The tree based structures have very similar risk profiles (which is an indication that the risk is depending on the structure of an index) that is related to the detection of the tree structure using frequency analyses (see Chapter 3.4.5). While there is little risk for performing frequency analyses when updating the range and text index (due to splits the root page changes), the root of the hierarchic index stays the same.

The risk of detecting the index structure can be overcome by using the algorithms described in chapter 2.6 (node swapping and access redundancy) or with the use of caching to buffer frequent accesses. The SemCrypt DBMS uses the latter approach.

The risk of detecting content cannot be entirely overcome, but is highly dependent on the encryption used and how updates are performed via the Storage Engine. On the other hand index structures reduce the chance to reveal content during retrieval, as the primary data is not accessed and information regarding occurrences of values is hidden.

More details on encryption and security in SemCrypt are given by Scharinger [Scha06].

### 6.1.5 Storage and Memory Consumption

It makes little sense to compare the storage requirements of index structures to the

primary data, as this ratio is highly dependent on the structure of the data and the definition of the index (which part of the data to index). A far better understanding of the storage overhead created by an index structure can be gained by comparing the data passed to an index for indexing to the total amount of data consumed by an index.

A sample set of data is used to create the index structures (10,000 keys and every key is assigned 81 bytes of data). The key domain used for value based index structures are integers (4 bytes), for hierarchic index node identifiers (labels) with an average size of 41 bytes size. The total storage consumption is compared to the indexed data, whereby determining the overhead created by index structures regarding storage consumption. As the overhead is primarily sensitive to the leaf page size, only this parameter and the occurring overheads are depicted in Table 13.

| ***Leaf Page Size (#keys)*** ***Index*** | ***5-10*** | ***10-25*** | ***25-50*** | ***50-100*** |
|---|---|---|---|---|
| Exact Match Index | 5% | 5% | 5% | 5% |
| Range Index | 32% | 24% | 22% | 21% |
| Hierarchic Index | 7% | 7% | 6,5% | 6% |
| Nested: Range-Hierarchy | 25% | 24% | 23% | 23% |
| Nested: Hierarchy-Range | 14% | 11% | 10% | 9% |

*Table 13: Storage Overhead of Index Structures regarding Leaf Page Sizes*

The two nested indices are a combination of the range and the hierarchic index. Depending on the nesting order the created overhead varies (the range index creates more overhead, however is more selective). In general the overhead of a nested index is in between the overhead of the indices it is consisting of, whereby the overhead is highly dependent on the kind of indexed data. The text index is not explicitly shown, as the overhead resembles the one of the Range Index. The additional storage required for storing strings as keys can be compensated by the prefix key-compression.

A second important measure is the main memory requirement during retrievals and updates. Index structures are not kept in main memory, only when accessing index structures a part of the structure is rebuilt. Therefore the required main memory is the product of an average page size with the required pages (one for the exact match index, two for the hierarchic index and the tree-height for the range index): $\text{Memory}_{required} \approx \text{Size}_{average\ page} \cdot (\text{required pages})$ Storage footprint of tree-based structures can be kept small by reducing the fanout parameter. However, performance is negatively correlated.

In case an index structure is built from scratch, it is preferable to create the whole structure locally and then to transfer it to the storage provider. The resulting memory requirement can be estimated by multiplying the amount of data to be indexed with the relevant overhead from Table 12: $\text{Size}_{Index} = \text{Size}_{Data} \cdot (1 + \text{Overhead}_{Index})$

### 6.1.6 Index Creation Performance

We conducted some experiments on the index structures to determine their update performance and to identify the influence of index parameters. We use different amount of data to test the scalability and also compare the results achieved in main memory with results using the Storage Engine. The Storage Engine has been configured to communicate with a local storage provider and with disabled cache (however, preserving the communication overhead required for remote storage providers). For the test using encryption DES is used. The tests are executed on a Mobile Intel Pentium with 1,4 GHz and 512 MB RAM, running Windows XP Service Pack 2.

The results for the **exact match index** are depicted in Figure 54. The creation time per indexed key stays constant with increasing amounts of data, therefore the creation of an exact match index is linear to the amount of data to be indexed.

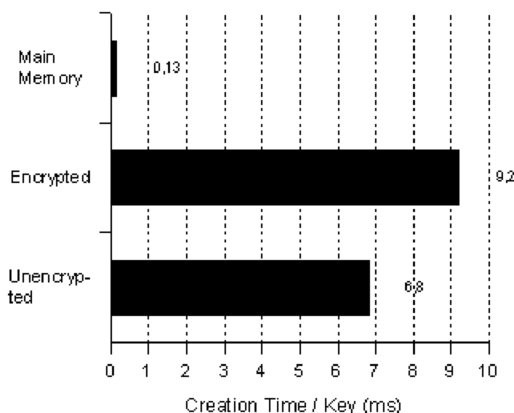



*Figure 54: Creation Performance - Exact Match Index*

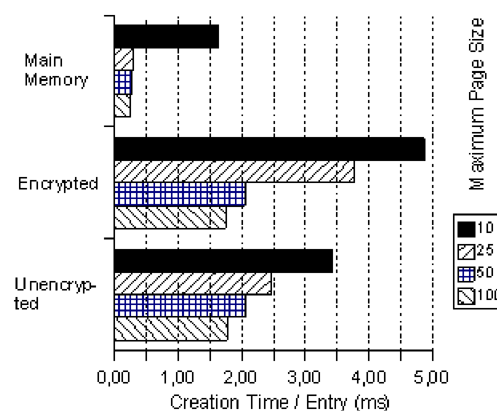



*Figure 55: Creation Performance - Hierarchic Index*

The performance results for the **hierarchic index** are depicted in Figure 55, which like the exact match index scales linearly to the indexed data. Index creation performance can be increased by choosing larger page sizes, in our tests the best performance could be achieved with a page size of 100 nodes that allowed the creation of one key in 1.8 ms. However, little changes when choosing with maximum page sizes greater than 50 nodes. The creation times of the exact match index cannot be directly compared to the creation times of the hierarchic index. Both test data contained 10,000 nodes, but the test data for the hierarchic index comprised 3 keys (hierarchies), while the exact match index was built on 10,000 different keys (values).

Regarding the range index we experimented, choosing various fanout and maximum leaf size values. As shown in Figure 56 the index scales well. While using the Storage Engine the speed (per indexed key) did even increase, which can be explained by the bundled and consequently more efficient upload. The main memory curve shows the expected result of a logarithmic increase with rising amounts of data (more branch pages).

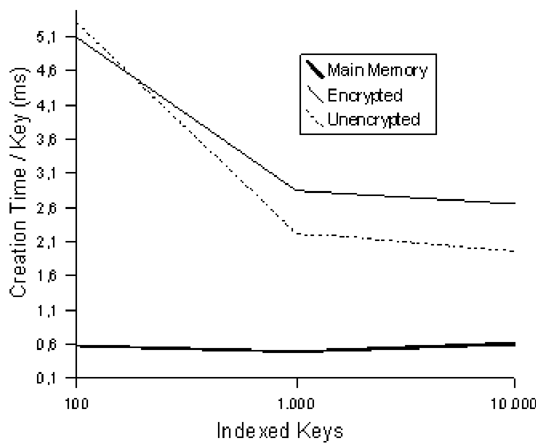



*Figure 56: Creation Scalability - Range Index*

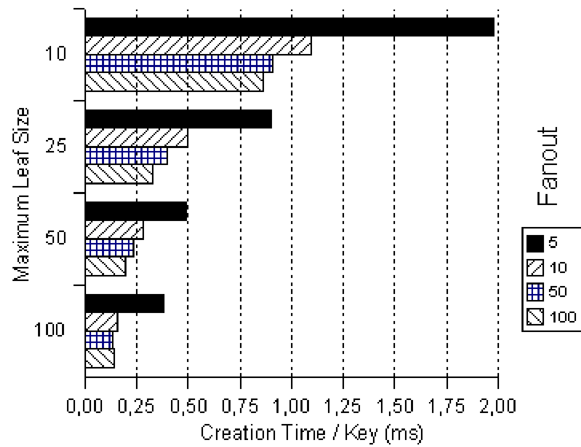



*Figure 57: Creation Performance – Main Memory - Range Index*

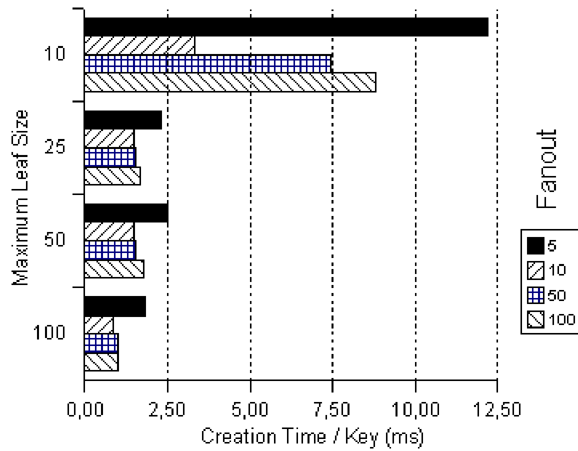



*Figure 58: Creation Performance - Unencrypted - Range Index*

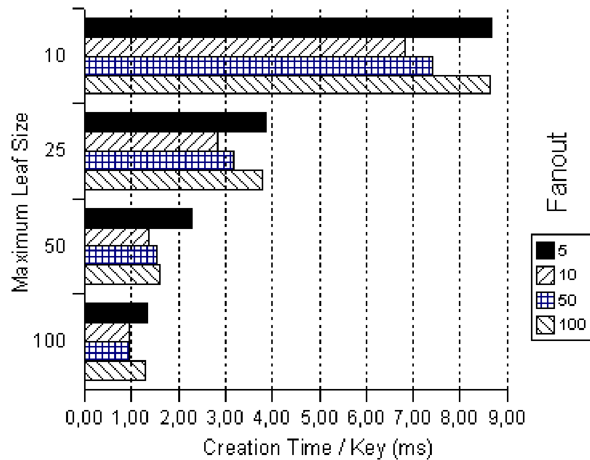



*Figure 59: Creation Performance – Encrypted - Range Index*

The optimal parameter settings and the implications for creation time can be extracted from Figure 57, Figure 58 and Figure 59. A maximum leaf size of 100 and a fanout of 10 provides the best creation performance. Especially in the unencrypted environment and when choosing a maximum leaf page size of 10, the fanout value has a huge impact on the overall creation performance. This may be caused by an optimal page size, which reduces the transmission overhead.

All parameters heavily influence retrieval performance, so the creation performance needs to be considered regarding the results of the next chapter.

The evaluation of index nesting is performed by combining a range and a hierarchic index. For the range index we defined a fanout of 50 and for the hierarchic index a minimum page size of 5 and a maximum page size of 10. The maximum page size of the range index was varied. The comparison of the two nesting possibilities is depicted in Figure 60 (range index as a super index, with a nested hierarchic index) and Figure 61

(hierarchic index with a nested range index). We added data for 500 different value keys and 2 different hierarchies. For every value key 20 entries were created.

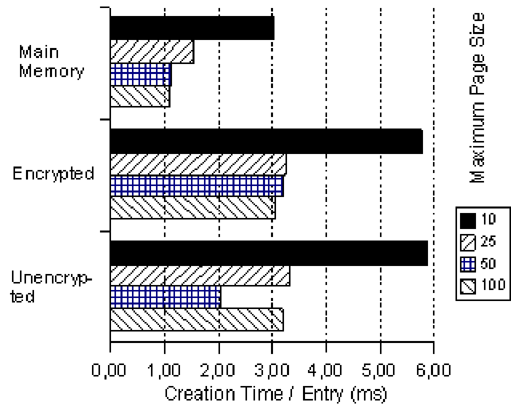



*Figure 60: Creation Performance – Nesting Range-->Hierarchy*

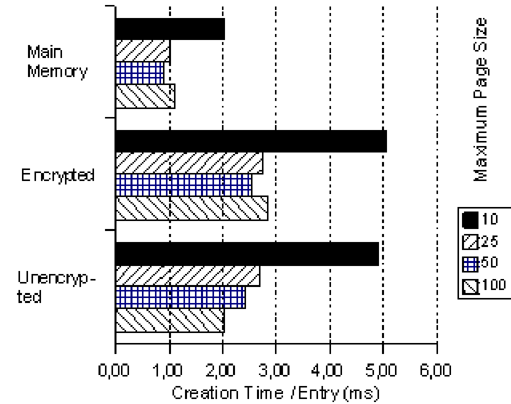



*Figure 61: Creation Performance – Nesting Hierarchy --> Range*

The best results, especially in the encrypted environment, were achieved using a leaf page size of 50 for the range index.

### 6.1.7 Index Retrieval Performance

Similar experiments to the one conducted in Chapter 6.1.6 for index updates are executed to identify the retrieval performance of index structures. Likewise scalability and different parameters are tested, using main memory storage and the Storage Engine. The testing system and used data is the same as before.

The experiments with the exact match index proved the independence of retrieval time and amount of indexed data. The average retrieval times in the different environments is depicted in Figure 62 and did not change with increasing amounts of indexed keys. Consequently the exact match index is by far the fastest index and very scalable as well.

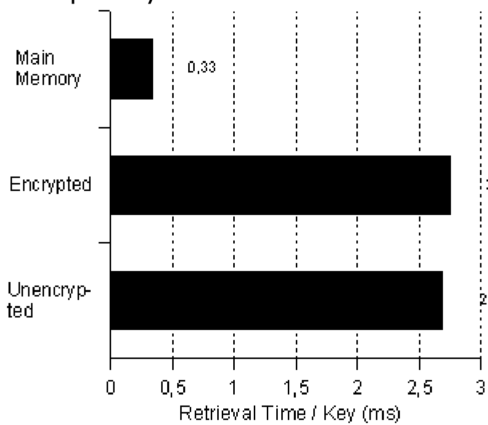



*Figure 62: Retrieval Performance - Exact Match Index*

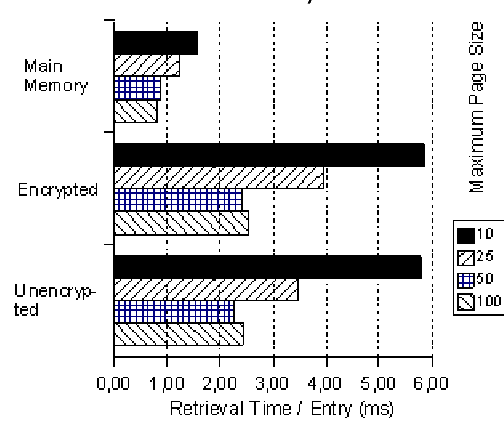



*Figure 63: Retrieval Performance - Hierarchic Index*

The results for the hierarchic index are shown in Figure 63. Retrieval time is depending on the maximum page size and has an optimum of 50 nodes in both the unencrypted and

encrypted environment.

As expected the retrieval performance of the range index is logarithmic to the indexed data (see Figure 64). Concerning the setting of the index parameters, when used in main memory (Figure 65), a maximum leaf page size of 25 combined with a fanout of 10 proved fastest.

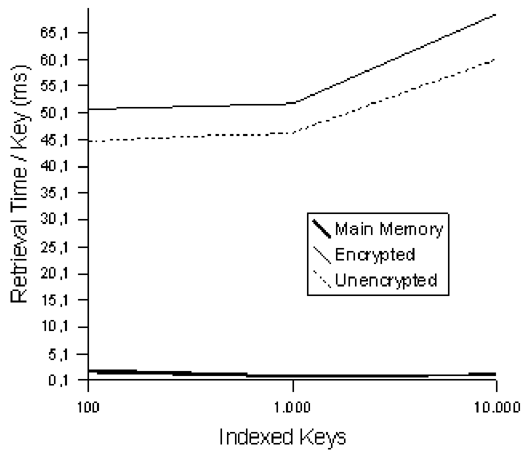



*Figure 64: Retrieval Scalability - Range Index*

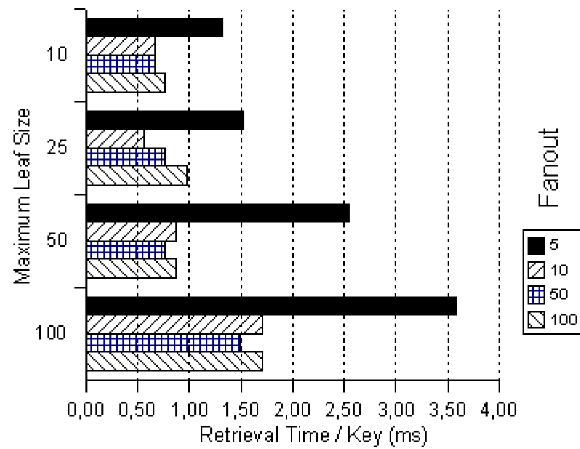



*Figure 65: Retrieval Performance – Main Memory - Range Index*

When using the range index with the Storage Engine, larger parameters for the fanout and page size resulted in better performance, as the encryption and transmission overhead can be reduced. The results for different parameter settings in an unencrypted environment are depicted in Figure 66. A maximum page size of 100 combined with a fanout of 50 enabled the fastest retrieval results. The results for the encrypted environment are shown in Figure 67, where the optimum was reached using a maximum page size of 100 and a fanout of 10. The retrieval in an unencrypted setting are an average 5-10% faster.

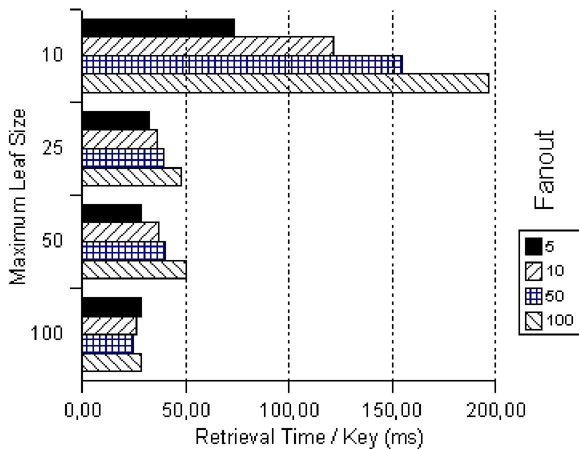



*Figure 66: Retrieval Performance - Unencrypted - Range Index*

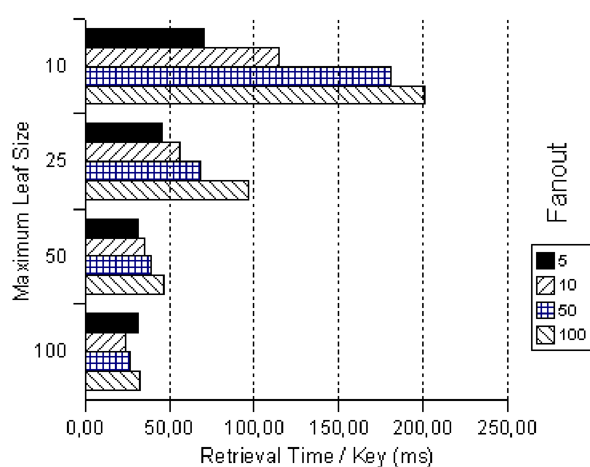



*Figure 67: Retrieval Performance - Encrypted - Range Index*

Interestingly the performance of the range index (when using a storage provider) decreased with growing fanout values. A possible interpretation is that due to the increasing size of the branch pages, the amount of transferred unnecessary data increases. Due to the expensive transfer (and encryption) it is favourable to transfer more smaller branch pages and to accept a higher tree size.

The retrieval times per key depend very much on the kind of query. The depicted results were retrieved with a set of average queries. In case larger range queries are executed, the retrieval time decreases with larger leaf sizes (as more data can be retrieved at once). However, exact match queries (or small ranges) become slower, as unnecessary data is fetched from the storage provider.

Index nesting enables the retrieval regarding multiple keys. We tested the combination of a range (value key) with a hierarchic (structure key) index performing queries on a data base of 500 value keys and two hierarchic keys. The average performance of the two nested indices depicted in Figure 68 and Figure 69 are pretty much the same. But the range-->hierarchy combination performs better on queries restricting the value (and not the hierarchy) and the hierarchy-->range combination is faster when not restricting the value. Consequently the kind of supported query influences the nesting order decision.

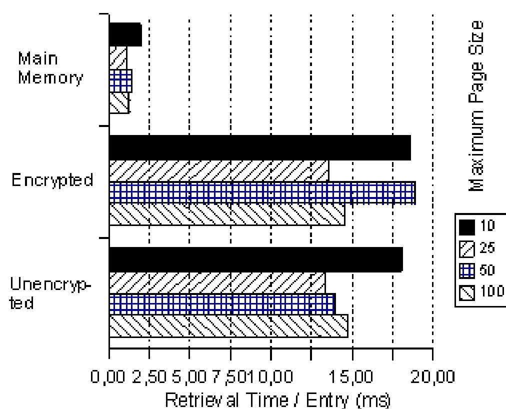



*Figure 68: Retrieval Performance – Nesting Range-->Hierarchy*

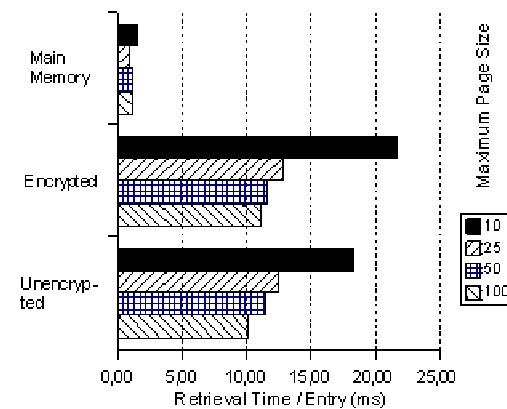



*Figure 69: Retrieval Performance – Nesting Hierarchy --> Range*

A comparison to queries without the support of indices could not be performed with the current prototype, as this functionality is not implemented yet. However, we expect that the presented index structures can substantially increase the overall performance of the SemCrypt DBMS.

## 6.2 Conclusion

Storing encrypted XML data remotely on an untrusted storage provider in SemCrypt created the necessity to enable faster access to outsourced data. This problem has been solved by the use of index structures that have been adopted and implemented in this thesis.

The requirement of flexible and extensible index management lead to a concept that generalizes various index structures by considering them as access structures. Consequently index structures can be defined, managed and accessed in a similar way. The concept has been implemented by the described index processing architecture.

As no index structure is able to support all kind of queries, different index structures have been adopted and implemented. While indices in general DBMS focus on the indexing of values, XML data contains additional structural information and is more likely to contain larger amount of textual information. The implemented index structures support value based queries (for equality, ranges and text) and structural queries.

A B-Tree variant has been adopted to support text queries. Compared to the inverted files, which are used regularly in information retrieval, the prefix B-Tree is balanced and supports prefix searches. A new index structure capable of dynamically indexing hierarchic data has been presented and implemented that is able to split regarding the inner structure of the indexed data.

Combined with the concept of nesting various index structures, value and structural queries can be supported by a combined index and multidimensional index structures can be emulated. The physical representation of index structures as id-value pairs makes data belonging to index structures indistinguishable from primary data. In combination with caching strategies this ensures and increases the overall security of the system.

The implementation of an indexing framework and several index structures for the SemCrypt DBMS has been described and its characteristics, strengths and weaknesses have been outlined in an extensive evaluation.

## 6.3 Outlook

During the development of the presented concepts and the implementation of the architecture and index structures, various new challenging problems and areas for improvement have been determined. Due to the focus of the thesis on index structures, interesting related areas, like information retrieval and the index update and selection problem have only been briefly touched.

Concerning the implementation, several optimizations and additional functionality can be added. While a general bulk loading mechanism for index structures, based on a write buffer has been implemented, pipelining (for index updates and retrievals) is not supported. Future extensions may also add compression of indexed data, as data that is part of the index definition needs not be saved. The index update mechanism and support for indices based on collections has been regarded in the architecture, but is not implemented.

While index nesting supports queries on multiple keys and is suitable to combine values with hierarchic information, it is inefficient for multidimensional data. Consequently a multidimensional index may be added in future versions and due to the general index framework these new index structures can be integrated easily.

Regarding information retrieval, the implemented text index supports keyword and prefix search. Future work can expand these capabilities to support boolean queries (regarding multiple keywords at once) and advanced pattern search.

Security and privacy regarding index structures is a very new area of research and there are few concepts (presented in Chapter 2.6) that can be used and analysed. Future research work may focus on this problem, analysing the index structures described and implemented in this thesis and developing new approaches to enable the secure access to data using index structures. Also an extended security analysis based on the implemented index structures may be performed.

The challenges of authorization and concurrency have not been regarded in this thesis. Authorization is of major importance for the SemCrypt project, however ensuring authorization when using index structures, which index data across authorization domains, is a complex and unexplored problem. For example, a user is only authorized to view a specific fragment of an XML document, but may use index structures defined on the whole document. Consequently a transmitted (and decrypted) index page might contain data, which must not be accessible to the user. As an index page is the finest granularity of transferred (encrypted) index data, the data cannot be filtered preliminarily. Possible solutions to this problem are:

- Use multiple encryption keys and a hierarchic encryption concept according to the authorization domains.
- Only define index structures within the authorization domains.

Finally the developed concepts may also be relevant to other contexts, for example the generalization of index structures can be used in other database management systems and the presented methods for storing and traversing index structures may be used in distributed environments.

# Table of Figures

Figure 1: SemCrypt System Setting........................................................................ 3
Figure 2: Email Store - Schema........................................................................ 5
Figure 3: Email Store Test Data........................................................................ 7
Figure 4: Hash index interface........................................................................ 15
Figure 5: B-Tree index interface........................................................................ 16
Figure 6: Hierarchic index interface........................................................................ 19
Figure 7: Encrypted B-tree using hash tables, example from [DVJ+03]........................ 23
Figure 8: Access redundancy for hiding tree structure, [LiCa04]................................ 24
Figure 9: Node swapping for hiding tree structure, [LiCa04]..................................... 24
Figure 10: Exact Match Index - Insertion Algorithm................................................ 32
Figure 11: Exact Match Index - Deletion Algorithm................................................. 32
Figure 12: Exact Match Index - Retrieval Algorithm................................................ 33
Figure 13: Range Index Pages........................................................................ 35
Figure 14: Sample Range Index........................................................................ 36
Figure 15: Range Index - Insertion Algorithm........................................................ 37
Figure 16: Range Index - Deletion Algorithm......................................................... 38
Figure 17: Range Index - Retrieval Algorithm........................................................ 39
Figure 18: Information Retrieval Processes.......................................................... 42
Figure 19: Sample Text Index - Index 3............................................................... 43
Figure 20: Example Text Index - Index 4.............................................................. 43
Figure 21: Hierarchic Index - Pages and Buckets.................................................... 47
Figure 22: Sample Hierarchic Index - Index 5........................................................ 48
Figure 23: Sample Hierarchic Index - Index 6........................................................ 48
Figure 24: Hierarchic Index - Insertion Algorithm................................................... 50
Figure 25: Hierarchic Index - Deletion Algorithm.................................................... 50
Figure 26: Hierarchic Index - Retrieval Algorithm................................................... 51
Figure 27: Hierarchic Index - Page Split Algorithm................................................. 52
Figure 28: Hierarchic Index 5 - After Split............................................................. 52
Figure 29: Hierarchic Index 6 - After First Split...................................................... 53
Figure 30: Hierarchic Index 6 - After Second Split.................................................. 53
Figure 31: Index Nesting Alternatives Example....................................................... 55
Figure 32: Nested Index 7 - 1st Approach............................................................. 56
Figure 33: Nested Index 7 - 2nd Approach............................................................ 56
Figure 34: Nested Index Processing Process.......................................................... 57

figure 35: SemCrypt Architecture........ 60
Figure 36: Internal Index Elements........ 65
Figure 37: Index Processing Architecture and Component Dependencies........ 67
Figure 38: Logical Layer Implementation........ 72
Figure 39: Index Variables........ 72
Figure 40: Logical Index........ 73
Figure 41: Index Manager........ 75
Figure 42: Internal Layer Implementation........ 77
Figure 43: Internal Index........ 77
Figure 44: Abstract Nestable Internal Index........ 79
Figure 45: Storage Engine and Metadata Adaptors........ 81
Figure 46: Index Engine........ 83
Figure 47: Exact Match Index Implementation........ 87
Figure 48: Range Index Implementation........ 87
Figure 49: Range Index Pages........ 88
Figure 50: Text Index Implementation........ 89
Figure 51: Hierarchic Index Implementation........ 90
Figure 52: Hierarchic Index Pages........ 91
Figure 53: Hierarchic Index 6 - After Second Split........ 92
Figure 54: Creation Performance - Exact Match Index........ 99
Figure 55: Creation Performance - Hierarchic Index........ 99
Figure 56: Creation Scalability - Range Index........ 100
Figure 57: Creation Performance – Main Memory - Range Index........ 100
Figure 58: Creation Performance - Unencrypted - Range Index........ 100
Figure 59: Creation Performance – Encrypted - Range Index........ 100
Figure 60: Creation Performance – Nesting Range-->Hierarchy........ 101
Figure 61: Creation Performance – Nesting Hierarchy --> Range........ 101
Figure 62: Retrieval Performance - Exact Match Index........ 101
Figure 63: Retrieval Performance - Hierarchic Index........ 101
Figure 64: Retrieval Scalability - Range Index........ 102
Figure 65: Retrieval Performance – Main Memory - Range Index........ 102
Figure 66: Retrieval Performance - Unencrypted - Range Index........ 102
Figure 67: Retrieval Performance - Encrypted - Range Index........ 102
Figure 68: Retrieval Performance – Nesting Range-->Hierarchy........ 103
Figure 69: Retrieval Performance – Nesting Hierarchy --> Range........ 103

# List of Tables

Table 1: Running Example – Test Email A.............................................................. 6
Table 2: Running Example – Test Email B.............................................................. 6
Table 3: Running Example – Test Email C.............................................................. 6
Table 4: Lookup Domain Types (adopted to the SemCrypt setting)............................ 12
Table 5: Lookup Function Types and Comparison Operators..................................... 14
Table 6: Query Types, Lookup Types and Associated Index Structures....................... 26
Table 7: Index 1 – Exact Match Index Data Structure.............................................. 31
Table 8: Sample Range Index in a Table Representation.......................................... 36
Table 9: Applicability of Implemented Index Structures........................................... 95
Table 10: Running Example Queries and Index Support........................................... 95
Table 11: Extensibility of Implemented Index Structures.......................................... 96
Table 12: Index-Security Risk Matrix..................................................................... 97
Table 13: Storage Overhead of Index Structures regarding Leaf Page Sizes................ 98

# Bibliography


[AMW01] H.-K. Ahn, N. Mamoulis and H. M. Wong: A Survey on Multidimensional Access Methods. Institute of Information and Computing Sciences, Utrecht University, 2001

[BaMc72] R. Bayer and E. McCreight: Organization and Maintenance of Large Ordered Indices. Proc 1970 ACM-SIGFIDET Workshop on Data Description and Access. 1972

[BaUn77] R. Bayer and K. Unterauer: Prefix B-Trees. ACM Transactions on Database Systems, Vol. 2, No. 1. 1977

[Baye96] R. Bayer: The Universal B-Tree for multidimensional Indexing. 1996

[BBK+00] C. Böhm et al: Multidimensional Index Structures in Relational Databases. Journal of Intelligent Information Systems. 2000

[BCC98] E. Bertino, B. Catania and L. Chiesa: Definition and Analyses of Index Organisations for Object-Oriented Database Systems. Information Systems. 1998

[BeFr79] J. L. Bentley, J. H. Friedman: Data Structures for Range Searching. ACM Computing Surveys. 1979

[BKK96] S. Berchtold, D. Keim, H.-P. Kriegel: The X-tree: An Index Structure for High-Dimensional Data. Proc. 22ndInt. Conf. on Very Large Data Bases. 1996

[BKK98] S. Berchtold, C. Böhm, H.-P. Kriegel: The Pyramid-Technique: Towards indexing beyond the Curse of Dimensionality. Proc. ACM SIGMOD Int. Conf. on Management of Data. 1998

[BKS+90] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. Proc. ACM SIGMOD Int. Conf. on Management of Data. 1990

[CCB94] C. L. A. Clarke, G. V. Cormack and F. J. Burkowski: An Algebra for Structured Text Search and a Framework for its Implementation. Dept. of Computer Science, University of Waterloo, Waterloo, Canada, 1994

[CDV+05] A. Ceselli et al.: Modeling and Assessing Inference Exposure in Encrypted Databases. ACM Transactions on Information and System Security. 2005

[CMV05] B. Catania, A. Maddalena and A. Vakali: XML Document Indexes: A Classification. 2005

[CoBe05] T. Connolly and C. Begg: Database Systems, A Practical Approach to Design, Implementation, and Management. Addison-Wesley, 2005

[Com79] D. Comer: The Ubiquitous B-Tree. 1979

[CSF+01] B. Cooper et al.: A Fast Index for Semistructured Data. Proc. 27th Int. Conf. on Very Large Data Bases (VLDB). 2001

[Dang04] T. K. Dang: Extreme Security Protocols for Outsourcing Database Services. Middlesex University, London, UK, 2004

[Dorn05] W. Dorninger: Securing Remote Data Stores. thesis: University of Linz, 2005

[DVJ+03] E. Damiani et al.: Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. Proc. ACM Conf. On Computer and Communication Security. 2003

[ElNa00] R. Elmasri and S. B. Navathe: Fundamentals of Database Systems. Addison-Wesley, 2000

[GaGü98] V. Gaede, O. Günther: Multidimensional Access Methods. ACM Computing Surveys, 1998

[GKSch05] K. Grün, M. Karlinger, M. Schrefl: Schema-aware Labelling of XML Documents for Efficient Query and Update Processing in SemCrypt. University of Linz, 2005

[GoHa05] O. Gospodnetic and E. Hatcher: Lucene in Action. Manning Publications Co., 2005

[GoWi97] R. Goldman and J. Widom: DataGuides: enabling query formulation and optimization in semistructured databases. In Proc. 23rd VLDB. 1997

[GrKa06a] M. Karlinger, K. Grün: Design Specification, SemCrypt Internal Project Deliverable d2.1. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[GrKa06b] K. Grün, M. Karlinger: Index Selection, SemCrypt Internal Project Deliverable d7.1b. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[Grün06a] K. Grün: Index Structures, SemCrypt Internal Project Deliverable d7.1a. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[Grün06b] K. Grün: Index Update. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[HäRo93] T. Härder, K. Rothermel: Concurrency Control Issues in Nested Transactions. The VLDB Journal. 1993

[HeSt78] G. Held, M. Stonebreaker: B-Trees Re-examined. ACM. 1978

[HILM02] H. Hacigumus, B. R. Iyer, C. Li and S. Mehrotra: Executing SQL over Encrypted Data in a Database-Service-Provider Model. In Proc. ACM SIGMOD Int. Conf. On Management of Data. 2002

[HIM02] H. Hacigumus, B. R. Iyer and S. Mehrotra: Providing Databases as a Service. 2002

[HNP95] J. M. Hellerstein, J. F. Naughton and A. Pfeffer: Generalized Search Trees for Database Systems. Proc. of the 21st VLDB ConferenceZurich, Switzerland. 1995

[Jamm04] R. Jammalamadaka: Querying Encrypted XML Documents. thesis: University of California, Irvine, 2004

[KaGr06a] Karlinger M., Grün K.: Metadata Manager, SemCrypt Internal Project Deliverable d6.1h. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[KaGr06b] Karlinger M. and Grün K.: Query Engine, SemCrypt Internal Project Deliverable d6.1d. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[KaGr06c] Karlinger M. and Grün K.: Execution Engine, SemCrypt Internal Project Deliverable d6.1g. Dep. of Data & Knowledge Engineering, University of Linz, 2006

[KDD89] W. Kim, K.-C. Kim and A. Dale: Indexing techniques for object-oriented databases. 1989

[KlMe03] M. Klettke, H. Meyer: XML & Datenbanken. dpunkt Verlag, 2003

[Knuth73] D. E. Knuth: The Art of Computer Programming. Addison-Wesley, 1973

[KPS02] M. Kratky, J. Pokorny and V. Snasel: Indexing XML Data with UB-trees. Proc.the 6th ADBIS. 2002

[Krat04] M. Kratky: Multi-dimensional Approach to Indexing XML Data. thesis: Technical University of Ostrava, 2004

[LiCa04] P. Lin and K. S. Candan: Hiding Tree-Structured Data and Queries from Untrusted Data Stores. Informations Systems Security. 2004

[LLD+02] R.W.B. Luk et al.: A Survey in Indexing and Searching XML Documents. Journal of the American Society for Information Science and Technology. 2002

[LOL92] C. C. Low, B. C. Ooi, H. Lu.: H-trees: A Dynamic Associative Search Index for OODB. In Proc. of the 1992 ACM SIGMOD Conference on the Management of Data. 1992

[MeSt99] H. Meuss, C. M. Strohmaier: Improving Index Structures for Structured Document Retrieval. 1999

[MiSu99] T. Milo and D. Suciu: Index Structures for Path Expressions. Tel Aviv University, 1999

[MWA+98] J. McHugh et al.: Indexing Semi-structured Data. Computer Science Dept., Stanford University, 1998

[NHS84] J. Nievergelt, H. Hinterberger, K. C. Sevcik: The Grid File: An Adaptable, Symmetric Multikey File Structure. Proc. ACM Trans. on Database Systems. 1984

[Ooi+96] B. C. Ooi et al.: Index nesting – an efficient approach to indexing object-oriented databases. The VLDB Journal. 1996

[PoHa01] L. K. Poola, J. R. Haritsa: SphinX: Schema-conscious XML Indexing. Indian Institute of Science, 2001

[RaGe00] R. Ramakrishnan, J. Gehrke: Database Management Systems, 2nd Edition. McGraw-Hill Companies, 2000

[RaKa95] S. Ramaswamy and P. C. Kanellakis: OODB Indexing by Class-Division. in. Proc. of the ACM SIGMOD Int. Conf. on Management of Data. 1995

[Rob81] J. T. Robinson: The K-D-B-Tree: A Search Structure for Large Multidimensibnal Dynamic Indexes. 1981

[Scha06] J. Scharinger: Security Report, SemCrypt Internal Project Deliverable d4.2b. Institute for Computational Perception, University of Linz, 2006

[SchGD05] M. Schrefl, K. Grün and J. Dorn: SemCrypt - Ensuring Privacy of Electronic Documents Through Semantic-Based Encrypted Query Processing. University of Linz, 2005

[SemCrypt] SemCrypt Website. http://semcrypt.ec3.at (last accessed December 2005)

[SrSe94] B. Sreenath and S. Seshadri: The hcC-tree: An Efficient Index Structure For Object Oriented Databases. In Proc. the 20th VLDB Conference. 1994

[Ukko95] E. Ukkonen: On-line construction of suffix trees. Springer New York. 1995

[WMB+04] F. Weigel et al.: Content-Aware DataGuides: Interleaving IR and DB Indexing Techniques for Efficient Retrieval of Textual XML Data. 2004

[XML06] Extensible Markup Language XML. http://www.w3.org (last accessed June 2006)

[XPath05] XML Path Language (XPath) 2.0, W3C Candidate Recommendation 3 November. http://www.w3.org/TR/xpath20/ (last accessed December 2005)

[YRC01] J. P. Yoon, V. Raghavan, V. Chakilarn: Bitmap Indexing-based Clustering and Retrieval of XML Documents. University of Louisiana, 2001

[ZMR95] J. Zobel, A. Moffat and K. Ramamohanarao: . Collaborative Information Technology Research Institute, Melbourne, Australia, 1995

[ZMR98] J. Zobel, A. Moffat and K. Ramamohanarao: Inverted files versus signature files for text indexing. ACM Press. 1998

# Appendix

## A) Utilized Software and Libraries

When developing software it makes sense to build on the experiences of other software developers and to reuse proven solutions and program code (libraries) and to make use of development frameworks. Although no suitable libraries exist for the task of index processing that could have been adopted for SemCrypt, some Java open source libraries have been used to ease and enhance development. Also one library for information retrieval (Lucene) has been used to provide basic information retrieval functionality for the text index.

### Development Tools

Eclipse is an open source IDE (integrated development environment), which is especially suitable for developing Java software. For the development of the index structures and index management framework **Eclipse version 3.1.2** and **Java version 1.5** have been used. To ease development two plug-ins were used:

- **Subclipse version 1.0.1,** to enable version control functionality via Subversion and to create a shared development environment.
- **TPTP framework version 4.1.0**, a test and profiling framework to gather detailed runtime information on storage utilization and processing times.

During the design process and for the UML figures depicted in this thesis, the freely available community edition of the UML tool **JUDE version 2.5.1** has been used.

### Java Libraries

JUnit is an open source library to write and execute unit tests for Java classes. Unit tests are written and tested using **JUnit version 3.8.1**.

In order to gather detailed information at runtime, to determine errors and to easier locate bugs a logging mechanism is required. For this purpose the open source library **Log4J version 1.2.13** has been used.

In order to make use of basic information retrieval techniques, required and described in Chapter 2.3.2 (text index) the analysis package of the open source **Lucene library version 1.9.1** has been used. This package provides all needed functionality to process texts and to transform them into a representation that can be used with the text index. More

details on the integration of Lucene in the text index is given in Chapter 5.3.4 and the Lucene project and functionality is described in detail by Gospodnetic and Hatcher [GoHa05].

# B) Running Example

Appendix B contains the XML schema used for the running example. The schema is followed by the sample XML data used.

## XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="qualified" attributeFormDefault="unqualified">
   <xs:simpleType name="emailAddress">
      <xs:restriction base="xs:string">
         <xs:pattern value="(.*)@(.*)\.(.*)"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="statusReceived">
      <xs:restriction base="xs:string">
         <xs:enumeration value="unread"/>
         <xs:enumeration value="read"/>
         <xs:enumeration value="answered"/>
         <xs:enumeration value="forwarded"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:simpleType name="addresseeModifier">
      <xs:restriction base="xs:string">
         <xs:enumeration value="from"/>
         <xs:enumeration value="to"/>
         <xs:enumeration value="cc"/>
         <xs:enumeration value="bcc"/>
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="FolderType">
      <xs:sequence>
         <xs:element name="Email" type="EmailType" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="EmailType" abstract="true">
      <xs:sequence>
         <xs:element name="Header" type="HeaderType"/>
         <xs:element name="Body" type="BodyType"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="SentEmailType">
      <xs:complexContent>
         <xs:extension base="EmailType"/>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="ReceivedEmailType">
```

```
    <xs:complexContent>
      <xs:extension base="EmailType">
        <xs:attribute name="Status" type="statusReceived" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="BodyType">
    <xs:sequence>
      <xs:element name="Text" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="HeaderType">
    <xs:sequence>
      <xs:element name="Subject" type="xs:string"/>
      <xs:element name="Date" type="xs:integer"/>
      <xs:element name="Addressee" type="AddresseeType"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="AddresseeType">
    <xs:attribute name="Modifier" type="addresseeModifier"
      use="required"/>
    <xs:attribute name="Address" type="emailAddress" use="required"/>
  </xs:complexType>
  <xs:element name="MailBox">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Folder" type="FolderType" minOccurs="0"
          maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

## Sample XML Data

```
<MailBox>
  <Folder name="InBox">
    <Email xsi:type="ReceivedEmailType" Status="answered">
      <Header>
        <Subject>Test 1</Subject>
        <Date>200605080930</Date>
        <Addressee Modifier="from" Address="michael@maier.de"/>
        <Addressee Modifier="to" Address="peter@lasinger.at"/>
        <Addressee Modifier="cc" Address="franz@mitterer.de"/>
        <Addressee Modifier="cc" Address="julia@schnell.de"/>
      </Header>
      <Body>
        <Text>This is a little test message.</Text>
      </Body>
    </Email>
        <Email xsi:type="ReceivedEmailType" Status="unread">
      <Header>
        <Subject>Test 2</Subject>
        <Date>200605091400</Date>
        <Addressee Modifier="from" Address="julia@schnell.de"/>
        <Addressee Modifier="to" Address="peter@lasinger.at"/>
      </Header>
      <Body>
```

```
        <Text>This is a second test message.</Text>
      </Body>
    </Email>
  </Folder>
  <Folder name="Sent">
    <Email xsi:type="SentEmailType">
      <Header>
        <Subject>RE: Test 1</Subject>
        <Date>200605081700</Date>
        <Addressee Modifier="from" Address="peter@lasinger.at"/>
        <Addressee Modifier="to" Address="michael@maier.de"/>
        <Addressee Modifier="cc" Address="julia@schnell.de"/>
      </Header>
      <Body>
        <Text>Thanks for the email. This is my answer.</Text>
      </Body>
    </Email>
  </Folder>
</MailBox>
```

# C) Logical Index Metadata

Appendix C contains the XML schema for logical meta data. After the schema definition, sample XML data used in the running example is outlined.

## XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="LogicalIndexMetaData">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Index" type="IndexType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="OperatorType">
    <xs:sequence minOccurs="0">
      <xs:element name="IndexVariable">
        <xs:complexType>
          <xs:attribute name="VariableId" type="xs:int" use="required"/>
          <xs:attribute name="VariableType" type="xs:string"
                use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="OperatorId" type="xs:int" use="required"/>
  </xs:complexType>
  <xs:complexType name="ConfigurationType">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="InternalIndex">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
          <xs:element name="VariableId" type="xs:int"/>
          </xs:sequence>
          <xs:attribute name="Type" type="xs:string" use="required"/>
        </xs:complexType>
```

```
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="DefinitionType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Operator" type="OperatorType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="IndexType">
    <xs:all>
      <xs:element name="Definition" type="DefinitionType"/>
      <xs:element name="Configuration" type="ConfigurationType"/>
    </xs:all>
    <xs:attribute name="ID" type="xs:ID" use="required"/>
  </xs:complexType>
</xs:schema>
```

## Sample Logical Metadata

```
<LogicalIndexMetaData>
  <Index ID="Index1">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="SIMPLE" VariableId="1"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="EXACT_MATCH">
        <VariableId>1</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
    <Index ID="Index2">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="RANGE" VariableId="1"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="RANGE">
        <VariableId>1</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
    <Index ID="Index3">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="KEYWORD" VariableId="1"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="TEXT">
        <VariableId>1</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
      <Index ID="Index4">
    <Definition>
      <Operator OperatorId="1">
```

```
        <IndexVariable VariableType="KEYWORD" VariableId="2"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="TEXT">
        <VariableId>2</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
      <Index ID="Index5">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="TYPE" VariableId="1"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="HIERARCHIC">
        <VariableId>1</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
      <Index ID="Index6">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="ID" VariableId="2"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="HIERARCHIC">
        <VariableId>2</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
  <Index ID="Index7">
    <Definition>
      <Operator OperatorId="1">
        <IndexVariable VariableType="RANGE" VariableId="1"/>
      </Operator>
      <Operator OperatorId="1">
        <IndexVariable VariableType="ID" VariableId="1"/>
      </Operator>
    </Definition>
    <Configuration>
      <InternalIndex Type="VALUEBTREE">
        <VariableId>0</VariableId>
      </InternalIndex>
      <InternalIndex Type="HIERARCHIC">
        <VariableId>1</VariableId>
      </InternalIndex>
    </Configuration>
  </Index>
</LogicalIndexMetaData>
```

# D) Internal Index Metadata

Appendix C contains the XML schema for internal meta data. After the schema definition, sample XML data used in the running example is outlined.

## XML Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="InternalIndexMetaData">
    <xs:complexType>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="Index" type="IndexType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="IndexVariableType">
    <xs:attribute name="VariableId" type="xs:int" use="required"/>
    <xs:attribute name="VariableType" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="ParameterType">
    <xs:attribute name="Name" type="xs:string" use="required"/>
    <xs:attribute name="Value" type="xs:int" use="required"/>
  </xs:complexType>
  <xs:complexType name="InternalIndexType">
    <xs:all>
      <xs:element name="IndexVariables">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
          <xs:element name="VariableId" type="xs:int"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Parameters">
        <xs:complexType>
          <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="Parameter" type="ParameterType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="InternalId" type="xs:int" use="required"/>
    <xs:attribute name="InternalType" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="DefinitionType">
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="Operator" type="OperatorType"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="IndexType">
    <xs:all>
      <xs:element name="InternalIndices">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
          <xs:element name="InternalIndex" type="InternalIndexType"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
```

```
        <xs:element name="Definition" type="DefinitionType"/>
      </xs:all>
      <xs:attribute name="ID" type="xs:ID" use="required"/>
    </xs:complexType>
    <xs:complexType name="OperatorType">
      <xs:sequence minOccurs="0">
        <xs:element name="IndexVariable">
          <xs:complexType>
            <xs:attribute name="VariableId" type="xs:int" use="required"/>
            <xs:attribute name="VariableType" type="xs:string"
                  use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="OperatorId" type="xs:int" use="required"/>
    </xs:complexType>
</xs:schema>
```

## Sample Internal MetaData

```
<InternalIndexMetaData>
  <Index ID="Index1">
    <InternalIndices>
      <InternalIndex InternalId="1" InternalType="EXACT_MATCH">
        <IndexVariables>
          <VariableId>1</VariableId>
        </IndexVariables>
        <Parameters/>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index2">
    <InternalIndices>
      <InternalIndex InternalId="2" InternalType="RANGE">
        <IndexVariables>
          <VariableId>1</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="Fanout" Value="2"/>
          <Parameter Name="MinSize" Value="1"/>
          <Parameter Name="MaxSize" Value="1"/>
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index3">
    <InternalIndices>
      <InternalIndex InternalId="3" InternalType="TEXT">
        <IndexVariables>
          <VariableId>1</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="Fanout" Value="2"/>
          <Parameter Name="MinSize" Value="1"/>
          <Parameter Name="MaxSize" Value="3"/>
          <Parameter Name="RunningNr" Value="1"/>
```

```
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index4">
    <InternalIndices>
      <InternalIndex InternalId="4" InternalType="TEXT">
        <IndexVariables>
          <VariableId>2</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="Fanout" Value="2"/>
          <Parameter Name="MinSize" Value="1"/>
          <Parameter Name="MaxSize" Value="1"/>
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index5">
    <InternalIndices>
      <InternalIndex InternalId="5" InternalType="HIERARCHIC">
        <IndexVariables>
          <VariableId>1</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="MinSize" Value="1"/>
          <Parameter Name="MaxSize" Value="2"/>
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index6">
    <InternalIndices>
      <InternalIndex InternalId="6" InternalType="HIERARCHIC">
        <IndexVariables>
          <VariableId>2</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="MinSize" Value="2"/>
          <Parameter Name="MaxSize" Value="4"/>
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
  <Index ID="Index7">
    <InternalIndices>
      <InternalIndex InternalId="7" InternalType="RANGE">
        <IndexVariables>
          <VariableId>0</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="Fanout" Value="2"/>
          <Parameter Name="MinSize" Value="1"/>
          <Parameter Name="MaxSize" Value="1"/>
```

```
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
      <InternalIndex InternalId="8" InternalType="HIERARCHIC">
        <IndexVariables>
          <VariableId>1</VariableId>
        </IndexVariables>
        <Parameters>
          <Parameter Name="MinSize" Value="2"/>
          <Parameter Name="MaxSize" Value="1"/>
          <Parameter Name="RunningNr" Value="1"/>
        </Parameters>
      </InternalIndex>
    </InternalIndices>
    <Definition/>
  </Index>
</InternalIndexMetaData>
```