Privacy-aware data management by means of data degradation

making private data less sensitive over time

Harold van Heerde

UNIVERSITY OF TWENTE.

CTIT
Centre for Telematics and Information Technology

UNIVERSITE DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

INRIA

Agreement of cotutelle de thèse
This thesis has been jointly supervised by the Unversity of Twente and the University of Versailles Saint-Quentin. Dr. N. Anciaux was supervisor on behalf of the University of Versailles Saint-Quentin.

Dissertation committee

Chairman/Secretary	prof. dr. ir. A. J. Mouthaan[1]
Promotor	prof. dr. P.M.G Apers[1]
Promotor	prof. dr. P. Pucheral[2,3]
Assistant Promotor	dr. M.M. Fokkinga[1]
Assistant Promotor	dr. N. Anciaux[3]
	prof. dr. I. Ray[4]
	dr. G. Miklau[5]
	prof. dr. W. Jonker[1]
	prof. dr. P.H. Hartel[1]

[1] University of Twente
[2] University of Versailles Saint-Quentin
[3] Institut national de recherche en informatique et automatique (INRIA)
[4] Colorado State University
[5] University of Massachusetts

SIKS Dissertation Series No. 2010-21

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

CTIT Ph.D. thesis Series No. 10-166

Centre for Telematics and Information Technology at the University of Twente.
P.O. Box 217 - 7500 AE Enschede
The Netherlands

The research in this thesis was partially supported by the NWO project Towards Context-Aware Data Management for Ambient Intelligence - From Imaginary Vision to Grounded Design & Implementation, Project number 639.022.403, and BRICKS, Project IS1: Organic Databases, and has been conducted in cooperation with the SMIS project of INRIA Rocqencourt, France.

Printed by Wöhrmann Print Service
ISBN 978-90-365-3002-6
ISSN 1381-3617
http://dx.doi.org/10.3990/1/9789036530026

© H.J.W. van Heerde, Enschede, 2010
All rights reserved. No part of this publication may be reproduced without the prior written permission of the author.

PRIVACY-AWARE DATA MANAGEMENT BY MEANS OF DATA DEGRADATION

MAKING PRIVATE DATA LESS SENSITIVE OVER TIME

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 4 juni 2010 om 15:00 uur

door

Harold Johann Wilhelm van Heerde
geboren op 8 december 1981
te Groenlo

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. P.G.M. Apers
prof. dr. P. Pucheral

en de assistent-promotoren:

dr. M.M. Fokkinga
dr. N. Anciaux

UNIVERSITE DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Ecole Doctorale Sciences et Technologies de Versailles - STV

Laboratoire PRiSM

THESE DE DOCTORAT

DE L'UNIVERSITE DE VERSAILLES SAINT–QUENTIN–EN YVELINES

PREPAREE EN CO-TUTELLE INTERNATIONALE DE THESE AVEC

L'UNIVERSITE DE TWENTE

Spécialité : Informatique

Présentée par :

Harold J.W. van Heerde

Pour obtenir le grade de Docteur de l'Université de Versailles Saint-Quentin-en-Yvelines et de l'Université de Twente en co-tutelle internationale franco-pays bas avec l'Université de Twente

Préservation de la Vie Privée par Dégradation Progressive des Données Personnelles

A soutenir le : 4 Juin 2010

Directeur de thèse :	Philippe Pucheral, Professeur, Université de Versailles Saint-Quentin-en-Yvelines
Co-Directeur de thèse :	Peter Apers, Professor, University of Twente
Co-encadrants :	Nicolas Anciaux, Chercheur, INRIA
	Maarten Fokkinga, Assistant Professor, University of Twente

Devant le jury composé de :

Rapporteurs :	Indrakshi Ray, Associate Professor, Colorado State University
	Gerome Miklau, Assistant Professor, University of Massachusetts
Examinateurs :	Willem Jonker, Professor, University of Twente
	Pieter Hartel, Professor, University of Twente

Numéro national d'enregistrement : 0grcoj02nk 8

The text depicted on the cover of this book is an arbitrary selection of queries taken from the query log disclosed by AOL *in 2006 [52]. As such, the content of the queries does not represent the personal view of the author.*

Acknowledgments

This thesis could not have been completed without the excellent supervision of both my French and Dutch supervisors, and the help from and discussions with many colleagues and friends. Therefore I want to acknowledge the following people.

First of all, I want to thank Nicolas for working together on data degradation since I was a master student at the University of Twente in 2005. Thanks to him I could experience working at INRIA in France, resulting in a cotutelle de thèse. We have had many excellent discussions, making travelling to Paris always worth the effort, and boosting the progress I made in my research. I also want to thank Philippe and Luc; I always looked forward to discuss the work with them, since the input they gave improved the work significantly.

Second, I want to thank Maarten, who was always available to listen to new ideas and gave those ideas a—formal—shape. His support and the discussions we had—which always took three times longer than planned—were of great value for me, as were the monthly discussions with Peter.

Beside the people I already mentioned, I want to thank all my colleagues of the database group who provided a great and pleasant working environment. We have had a lot of fun and many nice conversations, not only when we visited conferences and workshops together, but also during lunches and the 'groepsuitjes'. Special thanks go to Ida, who was always there to help and talk with, and Riham, who became a great friend with whom I had pleasant conversations and lunches, and made that I didn't loose confidence in my research.

Special thanks go to Berteun who helped me a lot with all the formatting of my thesis; Berteun's LaTeX skills are indeed unrivaled, which saved me a lot of time and frustration.

Finally I conclude with thanking my friends and family, especially my parents, for all their support during all those years.

Harold van Heerde
Enschede, May 2010

Contents

Acknowledgments **i**

Contents **iii**

1 Introduction **1**
1.1 Research questions 2
1.2 Organization of the thesis 4

2 Problem statement **5**
2.1 Motivation 5
2.2 Threat model 13
2.3 Related work 15
2.4 Conclusion 34

3 Limited retention and degradation model **37**
3.1 Limited retention 38
3.2 The concept of data degradation 41
3.3 Data hierarchies and generalization trees 49
3.4 Conclusion 54

4 Technical implications of data degradation **57**
4.1 The degradation model for relational data 58
4.2 Technical challenges 66
4.3 Impact of data degradation on core database system techniques 68
4.4 Revisiting the simplifications 90
4.5 Conclusion 93

5 Experiments and analysis **95**
5.1 Considerations 96
5.2 Prototype implementation 98
5.3 Degradation-friendly storage structures 105
5.4 Degradation-friendly indexes 124

5.5 Conclusion 130

6 Future research directions **133**
6.1 Service-oriented data degradation 133
6.2 Ability-oriented data degradation 139
6.3 Other models 143
6.4 Conclusion 146

7 Conclusions and Future work **147**
7.1 Revisiting the research questions 147
7.2 Future work 151
7.3 Concluding remarks 152

Bibliography **153**

Siks dissertations **165**

Summary **177**

Samenvatting **179**

Résumé **181**

Introduction

1

The rise of the Internet and the digital age triggered the collection of huge amounts of privacy-sensitive data. Enticed by free online services, people leave digital traces all around the Internet, managed by various online service providers. Those services enable people to maintain their social contacts, they open up access to vast amount of information through search engines, provide various tools to manage all kind of daily live necessities, such as online shopping lists, banking tools, insurance declarations, and much more. Various types of information associated with various activities of people, which used to take place in a private sphere, are now scattered around over databases all over the world, outside the control of the original owners of that data. Not only service providers collect privacy-sensitive data. Governments collect and store increasingly more information about their citizens. Examples such as automatic recognition of number plates [120] and retention of telecommunication data [73], show that more and more information which can be considered as personal and privacy-sensitive end up somewhere in a database, beyond the reach of the donors of that data.

It is hard to protect all these data. Various practical examples show that full security while keeping functionality is hardly possible [76, 2], [119, 117, 118, 102]. Negligence and human mistakes make that personal data will inevitably be disclosed and exposed to adversaries [52]. Weak policies can deceive people, giving them an inconsiderate feeling that their data is protected. Both victim and adversary can be anybody; out of curiosity, people might gain access to data of others, not only when the data has been disclosed by accident. Any kind of event can make somebody of interest to others. Moreover, personal information has become very valuable for marketeers, making it interesting for criminals to gain access to such data.

To limit the impact of the disclosure of information, one of the possible solutions is to *restrict* the collection and to *limit* the storage of data. The *limited retention* principle prescribes that data should no longer be stored than necessary to fulfill the purpose for which the data have been collected [5, 18]. Hence, data which cannot contribute to such a purpose

should not be collected at all. In this thesis, we embrace this principle and investigate how this principle can be exploited to limit the impact of data disclosure, while keeping it possible to offer users interesting and promising services. Moreover, to overcome the technical problems of irreversibly removing data, we will investigate the impact of limited retention on traditional database storage techniques.

However, our intuition is that data should not be removed at once, but gradually, comparable to fading footsteps in the sand. By progressively degrading the privacy-sensitive information, more and more details will be removed from the data, making the data less privacy-sensitive. This makes it possible to search for a better balance between data usability and privacy. For example, a location can be stored with precise coordinates, making it possible to follow the trace of a car. This location can be degraded firstly to *road number*—still possible to use it to predict traffic jams—and finally to *road type*—such that the driver can be charged for using specific roads.

The main contribution of this thesis is to investigate the limited retention principle, and to provide technical solutions to put limited retention into practice. We formulate the research questions in the following.

1.1 Research questions

The first problem of the limited retention principle is that due to a lack of transparency, the retention period is often overstated in advantage of the service provider. Moreover, users do not have the power and the knowledge they need to negotiate a reasonable retention period. To make limited retention common practice, we need to provide a framework in which it is possible to reason about retention periods. So, our first research question is:

Research question 1. How to model the interest of both service provider and user, to find the best retention period of privacy-sensitive data?

In Chapter 3 we conceptualize the interest of both parties to make it possible to reason about limited retention periods.

We relate the *worth* of personal data for the service provider and the *risk* for the user of storing data to the retention period. We combine both interest in what we name the *common interest*, such that we can find a retention period for which this common interest is optimal.

However, the all-or-nothing behavior of limited retention is too rigorous: after the retention period, the data will be completely destroyed, also destroying any possible use of the data. This makes it hard to balance data usability and privacy. This leads to the second research question:

Research question 2. How to refine the limited retention principle, to better balance the interests of service provider and user?

Also in Chapter 3 we introduce a refinement of the limited retention principle named data degradation.

By using well-known generalization techniques [51], we propose to *degrade* the precision of privacy-sensitive data after predefined retention periods, such that although the usability of that data will decrease, the privacy sensitivity will also decrease. We introduce the concept of *life-cycle policies*, which describe how and when the data should be degraded and finally removed. In some cases, data degradation can indeed be used to increase the common interest of both parties.

Policies alone are not enough. The difficulty will be how to implement such a policy, making sure that the data is indeed irreversibly degraded and finally removed from the system. Removing data from a database system is not a straightforward task [71]. Hence, our third objective is to investigate the technical difficulties associated with implementing and enforcing data degradation.

Research question 3. What is the impact of data degradation on traditional database systems, and is it feasible to implement the technique?

In Chapter 4 we will study this impact and propose new techniques required to support data degradation, followed with a performance analysis in Chapter 5.

We will see that many aspects of traditional database systems need to be revisited; we provide degradation friendly alternatives for storage structure, indexes, and transaction management. Using the results of our experiments and analysis we provide suggestions under which conditions which alternative is the best implementation choice.

To investigate and show the technical feasibility of data degradation, simplification have been introduced which put restrictions on the usability of data degradation itself. Releasing those simplifications can lead to different perspectives from which data degradation can be used.

Research question 4. How can the concept of data degradation be further exploited when the simplifications are released?

Chapter 6 will give an outlook to how data degradation can be extended into richer models describing the life-cycle of data.

Data degradation can be exploited in several ways, which opens up many future research directions. We make a first attempt to investigate a service-oriented approach, in which data degrades according to services' purpose specifications. We also investigate an ability-oriented approach, in which

not the precision of data is degraded, but the *ability* to support specific types of queries.

1.2 Organization of the thesis

The organization of this thesis is as follows:

- Chapter 2 elaborates the problem statement by sketching the context in which limited retentention and data degradation take place and defining the threat model. Furthermore, it discusses the underlying concepts of privacy and anonymity, and motivates why limited retention is necessary. We discuss related work in privacy-aware data management: anonymization, which shares techniques used by data degradation, access control, client-side protection schemes, and the concept of Hippocratic databases. Finally, we look to existing work on measuring data usability and privacy.
- Chapter 3 elaborates on finding a balance between data usability and privacy using limited retention, and in particular data degradation, using the concept of *common interest*. We will show that there are cases that data degradation can indeed lead to a higher common interest, validating the benefits of our approach. Furthermore, we present the data degradation concepts in more detail.
- Chapter 4 is about the impact of implementing data degradation on top of traditional database systems. We introduce properties of the data degradation model in the context of relational database systems, and investigate what should be done such that data can be irreversibly degraded taking performance issues into account. We propose new storage structures and indexes, discuss the transaction mechanisms, and investigate the impact on query semantics.
- Chapter 5 analyzes the performance costs introduced by data degradation using a prototype implementation. Using experiments we show under which conditions which storage structure is best suitable for particular loads on the database system. Furthermore, it presents an analysis of index structures, investigating how suitable they are in the context of data degradation.
- Chapter 6 looks at possible instantiations of the data degradation model. It is an outlook to how data degradation can be put in practice, and how the model can be extended to serve different types of scenarios.
- Chapter 7 concludes and proposes future work directions.

Problem statement

2

In the previous chapter we introduced the privacy problems triggered by the unlimited retention of privacy-sensitive information. In this chapter we first provide more background on the underlying difficulties and position data degradation—and limited retention in general—among other privacy-preserving techniques, and give an in-depth motivation why limited retention is an important and necessary component in privacy-aware database management.

Furthermore, we explain our threat model, and give an overview of related work in privacy-aware data management. We show that data degradation is orthogonal to most privacy-preserving techniques such as access control and privacy-preserving data publishing. We conclude with a short overview of metrics for privacy and data usability.

2.1 Motivation

Privacy has become a popular topic, triggered by the vast amount of web services with an apparently unsatisfiable desire for their users' personal data. Acquiring personal data is big business, a new gold mine for Internet companies, boosting all kind of new web services, increasing the threat to privacy even further [35]. It works; Google can reach over half a billion unique individuals each year, collecting—among many different types of personal data—their search queries, which to a high extent encapsulate their daily lives' habits [36]. Google made in 2008 a revenue of $22.1 billion [127]; given the fact that selling advertisements is Google's core business, this amount is a good indication of the value this personal data has for the company and its clients. Google is not alone; in their footsteps many other companies followed, and many will follow.

What do the users get in return for their personal data? Indeed, they profit from all the services which ease their lives. The web has been made accessible thanks to search engines, and communicating with friends and relatives has never been easier. However, until the Internet era, transactions between producer and consumer have been much more transparent for both

parties. The consumer pays the price which has been negotiated between producer and consumer, and the producer delivers the good or service. If the price is not satisfactory for both parties, the transaction will not take place. So, it pays off for the producer to be transparent. Today, business models are different. Services are offered for free—in terms of money—to the user, so that, at first glance, there is no reason to negotiate anymore. Personal information has become to be the currency of the Internet economy [95], although there is still a lack of an appropriate exchange rate to capture the privacy risks and the value of personal information. Hence, the market needs urgently to be regulated and, most importantly, to get *transparent*.

Here the privacy danger becomes apparent. Transparency is one of the key foundations of privacy [56]; it must be clear for the user how his or her data is being handled, stored, and to whom it will be disclosed. Asymmetry of power between users and service providers leads to privacy risks for the users, because service providers are in a better position to serve their interests [54]. Hence, more power and control should be granted to the user; if the service provider can argue that the data is needed to offer certain kind of services, the user may want to decide to allow the service provider to keep the data longer, paying a higher price, most probably benefiting from a better service. In other words: the price a user has to pay for a service should be expressed in terms of privacy risks, whereas it was expressed in terms of money in the old days.

So why is it a problem that companies store all these data about us? The fact is that, even if we put full trust in the service provider, these data can always be subject to data disclosure due to attacks, corrupt employees, governments demanding the data, *et cetera*. No access control mechanism has been proved to be both usable and fully secure; to give an example, even servers of the Pentagon [102] and FBI [118] have been hacked, credit card companies and mobile communication companies have lost personal data of their customers on several occasions [123]. Moreover, human mistakes are hardly preventable: politicians and policemen lose usb sticks or other media with sensitive information [105], obsolete personal computers sold secondhand are subject to forensic analysis with sometimes shocking results [90]. Governments play an important role too; various types of data can be subpoenaed by governments, even crossing boundaries. Initially, the US was granted unlimited access to all bank transactions of all EU citizens [104]. In such a situation, privacy-sensitive information is taken fully outside the control of the original owner of that data. Recently, the European parliament rejected the deal with the US, because the excessive storage of data was too invasive for its stated purpose [114].

Finally, personal data is often weakly protected by obscure and loose privacy policies which are unjustly presumed to be good and acceptable for a given service.

The privacy violation will only increase with the growth of data which

has been collected about us. All these data, even when "legally" obtained by the service providers themselves, foster ill-intended scrutiny and abusive usages justified by business interests, governmental pressures and inquisitiveness among people. Not only criminals and terrorists are threatened. Everyone may experience a particular event (e.g., accident, divorce, job or credit application) which suddenly makes her digital trail of interest for someone else. Moreover, identity fraud is nowadays becoming one of the most serious crimes, with huge consequences for the victims [44]. The retention problem has become so important and the civil pressure so high [121] that privacy practices start changing. For instance, Google and other search engine companies announced to shorten the retention period of their query logs.

Limiting the retention of personal data indeed reduces the privacy problems sketched above. *Limited retention* is a widely accepted privacy principle, complementary to techniques such as access control, and is included in various privacy regulations [40]. The principle prescribes that data should not be stored longer than necessary to fulfill the purpose for which the data has been collected [5]. By limiting the time that data is stored, the impact of disclosure of a store is less severe [36].

However, limited retention is difficult to put in practice, because of the difficulty of determining what the retention period should be. The principle prescribes that data should not be retained longer than strictly necessary to fulfill the purpose for which the data has been collected. This implies that those purposes should be *atomic*, that is, a purpose can be either fulfilled completely—within a foreseeable time period—or it will never be completed at all. For some services—such as the delivery of a book—it is clear when the purpose has been fulfilled completely, and which data was necessary to fulfill the purpose. For other purposes, this can hardly be determined. When, for example, is the purpose of a recommendation system completely fulfilled? How long does it need to store privacy-sensitive context data to make recommendations better?

Privacy-aware data management is required to overcome not only the dangers of an ever growing hunger for personal information, but especially the unlimited and unrestricted storage of this information. Without a counterbalance, on-line companies will continue collecting and retaining personal information, triggered by the enormous profits which lay ahead, disregarding the privacy issues they create. This is why we need to find a mechanism to balance privacy and usability. Otherwise, we either end up with a lot of highly valuable personal information for the data collector and zero privacy for the user, or zero value and full privacy [36].

2.1.1 Privacy in relation to anonymity

Privacy is an elusive concept which is hard to define and captures many aspects [80]. Although tempting, we will therefore never state that we have the solution for *the* privacy problem. There are many solutions in literature which claim to be privacy protecting, but in fact only cover a small subset of all privacy-*related* concepts. To make the concept 'privacy' workable and understandable in the scope of our research, we limit ourselves to the terminology and taxonomy of Halpern et al. [49] and later refined by Tsukada et al. [89]. They state that privacy is typically about *"hiding personal or private information from others"*, or more precisely, to *"hide what has been performed"*. Using this understanding of privacy, an attempt to *protect* privacy can be achieved from different angles. Limited retention, and as a refinement *data degradation*, typically *hide* data by removing or obscuring the privacy-sensitive data. Access control techniques typically try to hide the data from others by limiting the *access* to that data, whereas encryption-based techniques hide the true contents of data by cloaking it with a secret key.

Privacy and *anonymity* are therefore—especially in the scope of our research—orthogonal concepts. Privacy is about hiding *what* has been performed by, or is related to, a certain individual. Anonymity is about hiding *who* performed an action or who is related to—possibly—private sensitive data. Tsukada [89] identified the concepts of privacy, anonymity, onymity, and identity, and related them together as in Figure 2.1.

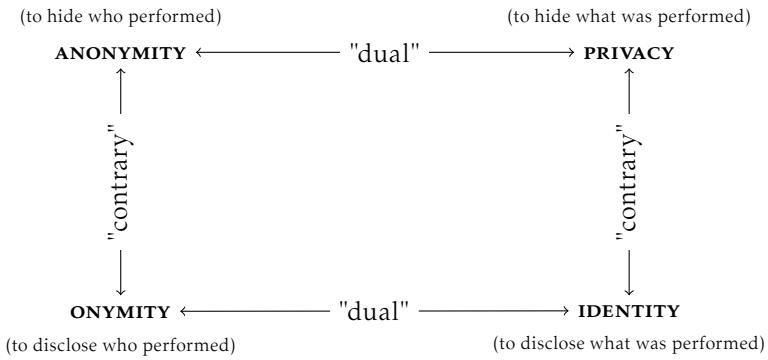


Figure 2.1 Privacy-related properties [89] showing that privacy is related, but also orthogonal to anonymity. The aim of data degradation is to make *onymity* possible while preserving *privacy*. The aim of (for example) k-Anonymity [85] is to provide *anonymity* while preserving the *identity* property.

Strictly following this reasoning about privacy, anonymity techniques do

not attempt to protect privacy itself. Indeed, making a data set anonymous *protects* individuals from being *related* to privacy-sensitive facts, but do not *hide* those facts. Without additional privacy protection, it will therefore happen that when the anonymization process fails—such as happened in the infamous AOL-case [52]—the victims will end up with no privacy at all. On the other hand, if privacy protection fails or privacy protection is not possible at all, in cases where the sensitive information is required for a given purpose, anonymization can be the solution to protect people from being linked to the privacy-sensitive information.

Data degradation in relation to anonymity

In many situations people do want to be able to share private information with others [53]; anonymization does not cover this kind of applications. The benefit of choosing to *hide what has been performed* compared to anonymization techniques is that we can keep the identifier intact, and therefore can support user-oriented services. Indeed, although anonymized data can support many (research) purposes, most value for commercial parties can be generated thanks to the personal and identifiable data they possess.

2.1.2 (Limited) Data Retention

In the past years, there has been much debate in politics about the retention of data. Triggered by the 9/11 attacks, there is a tendency within governments to demand the retention of telecommunication data to prevent terrorism, accumulated in the Data Retention Directive [18]. At the same time, Article 29 Working Party, set up by the European Parliament according to the Directive 95/46/EC [40], urges market companies such as Google and Microsoft to limit the retention period of data [121]. Hence, although governments clearly see the need for privacy by enforcing commercial parties to limit the retention periods of privacy-sensitive data, they are now also convinced they need huge amounts of their citizens' personal information to fight against crime. Still, the *Convention for the Protection of Individuals with regard to Automatic Processing of Personal Data*, article 5.c and 5.e, clearly states that personal data undergoing automatic processing shall be: [103]

5.c adequate, relevant and not excessive in relation to the purposes for which they are stored;

5.e preserved in a form which permits identification of the data subjects for no longer than is required for the purpose for which those data are stored.

Hence, whatever the purpose of data retention is—from either a governmental or business perspective—only data should be stored which serves that purpose, and only for the period it is required to fulfill that purpose.

As a result, even though the Data Retention Directive prescribes a *minimum* retention period, this retention period should at the same time be interpreted as the *maximum* retention period.

Although covered by law, the limited retention principle has often been overlooked in the privacy literature. Most research focuses on controlling the *access* to personal information, although Agrawal et al. [5]—inspired by the privacy principles described in the various laws—included the principle in their Hippocratic database framework. Blanchette et al. [21] argue that limited retention is necessary to maintain in what they call *social forgetfulness*; people should have the opportunity "to move on beyond one's past and start afresh". Mayer-Schöberger [69] suggests *"that we should revive our society's capacity to forget"*; humans have always been forced to carefully consider the trade-offs of retention and deletion. The price of remembering everything was simply too high. Although the cost in terms of resources has been drastically decreased thanks to the digital age, the new cost can be high if "the lack of forgetting may prompt us to speak less freely and openly". Mayer-Schöberger [69] proposes therefore to associate data with meta-data specifying the retention period of the data, enforcing the automatic deletion of it, making *forgetting* the default instead of *remembering*.

This is not the only reason why limited retention is so important. History shows that it is very hard to protect private information from being disclosed by any kind of (access control) technique. The 2008 CSI Computer Crime and Security Survey [76] shows that 49% of the 522 participating organizations were subject to virus incidents, and 27% of the organizations have detected a 'targeted' attack. In 17% of the cases the incident involved the theft or loss of customer data. Acquisti et al. [2] analyzed over a time window of eight years (from 1999 to 2006) more than 200 privacy breaches that have been reported by publicly traded firms, of which 80 where caused by hacks and exploits. They show that there is indeed an impact of privacy violations, not only for their customers, but also for the companies themselves. They conclude that the trust reputation of those visible companies (their stocks are traded at the New York Stock Exchange) can be significantly affected by negative reports about their privacy practices.

Moreover, the examples of successful attacks which make it to the news papers are plenty. Headlines such as *"Payment Processor Breach May Be Largest Ever"* [119] and *"Prime Minister's health records breached in database attack"* [117] are not uncommon. Of course, one might argue that those breaches could have been prevented with better security policies and implementations. But even when the security policies are strong, and even when the access control techniques themselves work perfectly, human mistakes, or even governmental pressure can lead to disclosure of data. A recent example at the T-Mobile company showed that employers sold millions of customer records to third parties [101], showing that personal information is indeed attractive to attackers, and disclosure hardly preventable.

Benefits of data degradation

Limiting the retention of data is necessary to limit the impact of the unavoidable disclosure of privacy-sensitive information. As a new interpretation of the limited retention principle, data degradation makes it possible to find a good balance between data usability and limiting the impact of disclosure. The main benefit is—orthogonal to other privacy protecting principles—increased privacy with respect to unintended data disclosure. The amount of accurate data which can be disclosed—whatever the cause is—is limited and therefore the impact on privacy will be less severe.

2.1.3 Use case scenarios

To introduce our core ideas, we sketch here two scenarios; the first is based on the retention of query logs by search engines, and the second is based on the proposals of a *congestion pricing system* in the Netherlands by the Dutch government [109]. For both scenarios we will give an example how *data degradation* can help solving the related privacy issues.

Use case scenario 1: degradation of query logs

Search engines record the queries of their users in a query log. For example, Google records the ip address, the user's browser type, language, date/time, and cookie_id together with each search query, and the url of the page the user visits after his search. Even when a user is not explicitly logged in, using these attributes, queries can be related to individuals so that personalized advertisements can be presented [52, 35]. Moreover, when using a search application on a mobile phone, a user can opt-in to provide exact location details which can be used to provide location-aware search results [107].

Query logs are a valuable assets for search engines, but also contain privacy-sensitive information. For example, even when the location is not explicitly provided, the ip address can be used to determine the location of the user. Although the relevance for the search engine to know the exact location of a user throughout his whole history decreases over time, the privacy sensitivity of those facts might not. Therefore limited retention should be applied to limit the privacy risk for the users [121]. However, to better balance data usability and privacy, query logs can also be progressively degraded.

The search engine personalized services will be less able to define the interest of a user when the query log is degraded, but can still operate. For example, during a short period, when precise location information, such as *street address*, is available, location-aware search results or advertisements can be provided. After the locations have been degraded, for example to the *city* the user lives in or has visited in the last period, the search

engine can still provide location-aware advertisements—but with a lower precision. This process continues, for example by degrading the location to *country*, and the precision of the location-awareness of the advertisements will decrease accordingly.

The benefit for the search engine to apply data degradation instead of limited retention is that the search engine can benefit longer from the information contained in the query log, without severely damage the privacy of their users. In chapter 3 we show that the *common interest*—the combination of the usability interest of the service provider and the privacy interest of the user—can be higher when data degradation is used compared to limited retention.

Use case scenario 2: congestion pricing system

Although the current plans for a congestion pricing system (see [125] for a definition) are still in an early stage, the system will be based on GPS devices installed in every car, monitoring the exact location of every car at any time. The purpose is to price every driven kilometer based on the time of the day and the road used. For example, driving during rush hour on a heavily used highway will be more expensive than driving during the night on a secondary road. The data which is needed to support this purpose might also be useful for other purposes, making the implementation of such a system additionally interesting. However, a governmental system monitoring the movements of citizens will raise privacy concerns. Data degradation can limit those privacy concerns. Still, the following scenario is only a hypothetical scenario.

The collection and retention of the exact locations of an individual can be valuable for both government and private organizations to support different kinds of services. For example, it enables to compute fine-grained traffic congestion information, and it can help to build short term traffic predictions. With the use of road usage information, roadwork can be planned. Commercial parties, such as insurance companies, might want to use the locational information to provide fine-grained insurance policies (e.g., you pay less if you don't drive during rush hours on busy roads). Finally, to fulfill the main purpose—to be able to bill the driver—information about the type of roads used per time period is needed.

However, to fulfill those purposes, it is not necessary to endlessly retain all information in a precise form. To make the stored information less privacy-sensitive, and to make possible misuse less likely, we can let the information be subject to timely degradation. A possible life-cycle of the location attribute is shown in Figure 2.2.

This example shows the typical use of data degradation where purposes can be matched on the required information, and the required precision of that information. We name this service-oriented data degradation, which

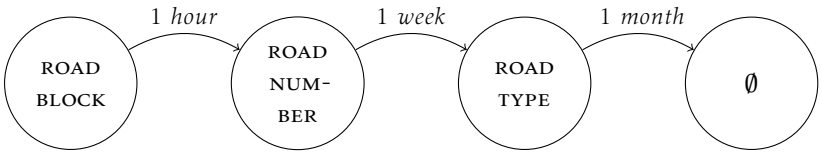


Figure 2.2 Example of the life-cycle of a typical privacy-sensitive piece of information useful for an automatic and fine-grained congestion pricing system. Drivers pay monthly for the use of specific roads, and therefore this information is required to be kept for a month. To provide real-time traffic information, it is useful to keep the actual position of a car on a road for an hour. To be able to estimate the weekly usage of roads and predict traffic jams, the road number should be sufficient.

will be further discussed in chapter 6. As we have seen in the first use-case scenario, it is also possible to define a single purpose which can use both precise and less precise data, although the extent to which this purpose can be fulfilled will decrease when data is less precise.

2.2 Threat model

A threat model describes which threats a particular technique takes into consideration, and against which threats it provides protection. This way, the threat model can be used to position a technique, and to make clear what can be expected from it. In the following related work section, we will refer back to the threat model, to indicate why a particular technique is or is not a solution for the same problems as data degradation.

First, we define a *trail* as all the information collected by a particular service provider which can be linked to an individual. *Trail disclosure* is the event that a trail is unauthorized exposed to an adversary. We want to limit the *impact* of a trail disclosure; the goal of data degradation is not to *prevent* trail disclosure.

As mentioned in the previous section, data degradation is a derivative of the limited retention principle, and therefore the threat model we consider is the same. It assumes that the server responsible for storing the data is what we call *honest*. This means that it implements the retention policies and does its best effort to enforce the timely removal (or degradation) of the stored data. Moreover, it implements any security policy needed to restrict unauthorized access to the data.

Honest is a weak form of *trustworthy*: a trustworthy server is assumed to be *not* vulnerable to trail disclosure [75]. In contrast, we cannot make this assumption for a honest server. Although it does its best effort to prevent, it is still vulnerable to trail disclosure. It cannot fully prevent all forms of

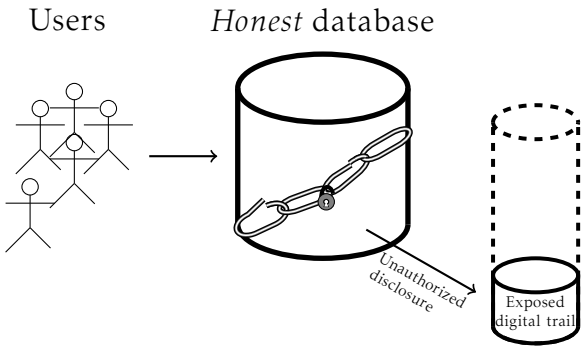


Figure 2.3 Users provide data which will be stored in a database by the service provider. The database system is *honest*: it implements security policies and removes or degrades data as specified in policies. When the database has been successfully attacked and the data is disclosed, only a subset of the original data can be scrutinized by an adversary.

attacks, negligence, or weakly defined policies resulting in the exposure of a digital trail of a victim to an adversary. We argued earlier in this chapter that the main class of today's servers are honest, and although sometimes wrongly assumed, not trustworthy.

The types of causes of trail disclosure we consider are the following:

- *Piracy attack:* an adversary breaks the security policies and bypasses the access control techniques. Even those of what we can assume to be the most secure servers have been shown to be vulnerable to this type of attack, such as those of the FBI [118] and the Pentagon [102].
- *Negligence:* due to mistakes or careless handling, privacy-sensitive data sets can be made public. For example, AOL released query logs which were assumed to be sufficiently anonymized, but nevertheless revealed trails which could be linked to individuals [52].
- *Weak policies:* due to a lack of transparency and openness, ignorant users might discover they have provided too much privacy information given their current situation. Merges of service providers might result in a join of the collected private information of both service providers, possibly leading to a larger privacy risk than was pre-assumed. Weak policies might also result in situations that malicious employers easily can access privacy-sensitive data, as happened recently at a mobile telephone company [101]. Moreover, changes in privacy policies might be overlooked by the ignorant user, weakening the protection of their privacy-sensitive information [113].

Everybody can become a victim and can become somehow of interest for an adversary. Any event can suddenly cause a person to become subject

to investigation; this can be a divorce, a conflict with your employer, an accident, *et cetera*. An adversary can be anybody who can either on purpose or accidentally get hands on your digital trail.

Limited retention does *not* protect against continuous spying on the database. However, to retrieve one's full digital trail, an attacker needs to repeatedly gain access to the database server, at least once per retention period. such a repetitive attacks are more likely to be detected by intrusion detections systems [32].

2.3 Related work

Related work in privacy-aware data management can be divided in two main classes, as shown in Figure 2.4. Traditionally, the first class deals with how to release privacy-sensitive information to third parties in a privacy-preserving way [17]. We extend this class with techniques which make information less privacy-sensitive already *before* disclosure, to limit the impact of *unauthorized* disclosure. Hence, although there is a clear difference in objective and context, limited retention and data degradation fall in the same class as anonymization techniques, since both try to decrease the privacy sensitivity of a data set. However, within this class, the techniques are orthogonal to each other; anonymization-based techniques try to unlink the privacy-sensitive part of the data from the identity of the users, while, for example, data degradation aims to decrease the privacy sensitivity of the sensitive attributes.

The second class deals with limiting the chance of the *disclosure* of a privacy-sensitive data set to unauthorized users or third parties. The most common technique to achieve this is by making use of access control techniques [77, 78], of which many derivatives exist today. Many implementations based on access control are designed to enforce policies; a standardized way of expressing privacy policies is P3P, which we will elaborate on, including some of its proposed extensions. Other techniques we will describe in the following are client-based: they are aimed at keeping the sensitive information only stored or accessible by the users, and run queries against the information which will be kept safe by the user. The user herself is then responsible for protecting her own data, and can control which data will be released to whom.

The organization of this section is as follows. We start with describing related work in access control and policy-based solutions for protection against disclosure of privacy-sensitive information, followed by techniques which make the disclosed information—authorized or non-authorized—less privacy-sensitive. Where applicable, we will indicate why a given technique cannot help solving our research questions, or why the technique cannot give protections against the threats described in our threat model. Where

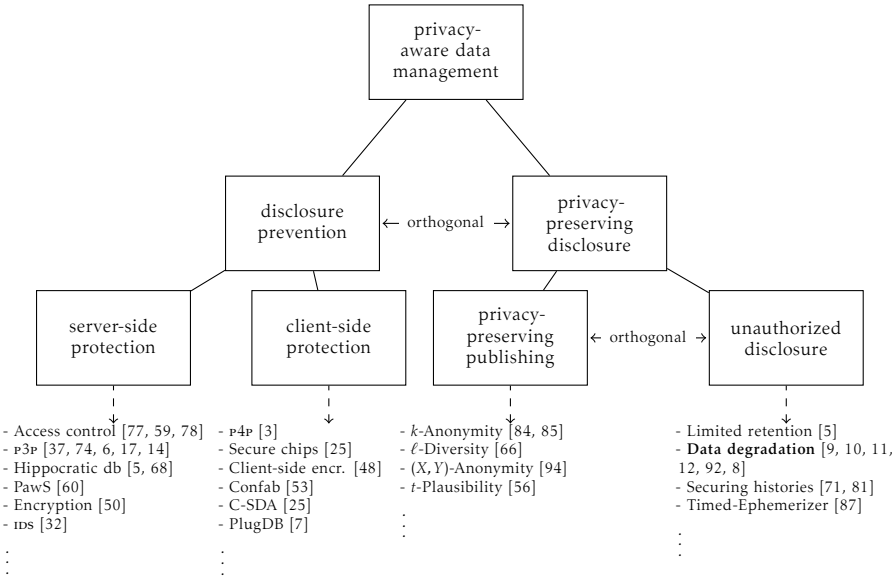


Figure 2.4 Rough classification of privacy-aware data management and pointers to related work. Data degradation can be found in the unauthorized disclosure group. This group of techniques deals with limiting the impact of unauthorized disclosure. Note that data degradation is orthogonal to privacy-preserving publishing techniques such as k-anonymity, and to disclosure prevention techniques such as access-control based techniques. Some work could have been placed in multiple groups, such as Hippocratic databases. This work is mainly based on disclosure prevention, but partly discusses limited retention to limit the impact of unauthorized disclosure.

possible, we indicate whether or not the technique is complementary to data degradation, and where the technique overlaps with data degradation. We finish with an overview of the literature on how to *measure* the amount of privacy protection, and possible loss of usability, which can be achieved by a certain technique.

2.3.1 Disclosure-preventing techniques

Access control

Access control basically constraints what a *legitimate* user can do with the stored privacy-sensitive information [77], and therefore helps to prevent security breaches and the unauthorized disclosure of data. The first form of access control used to limit access to stored data is *discretionary access control* (DAC) [59], and is the traditional file access restriction mechanism

in UNIX systems. *Mandatory access control* (MAC) [78], closely related to multi-level security systems, uses a partial ordering of security levels (e.g., top-secret, confidential, classified, *et cetera*). Every user and object in the system is labeled with a security level; a user is only allowed to read an object if his security level is equal or higher than that of the object, and should not write an object with a lower security level than his own. This type of access control is often used in military settings.

Role-based access control (RBAC), implemented in many SQL databases, defines which permissions belong to which *role* [78, 41]. Roles can then be assigned to individuals. This makes administration of access to information easier; a particular user is simply assigned an appropriate role, and this role defines the permissions.

Byun et al. [30] build further on RBAC and propose *purposed-based access control*. The main contribution is to associate purpose information to the data elements, and regulate access to those elements based on the purpose for which it needs to be accessed. By using the concept of *intended purposes*, it is possible to describe for which purposes data can be accessed, and which purposes cannot be used to access the data. It is the systems responsibility to determine the *access purpose*, and match this with the set of intended purposes to decide whether or not access will be granted. Access purposes can be associated with roles, which can be managed using regular RBAC techniques.

Finally, Oracle introduced the concept of *virtual private databases*, in which access to rows and/or columns can be regulated based on a context. Queries are rewritten based on context information, such that only authorized rows are returned [111].

Platform for Privacy Preferences (P3P)

A first attempt to express privacy policies based on regulations, has been conducted by the Platform for Privacy Preferences, known as P3P policies [116]. These policies let users know which data will be collected for what purpose, and how long the data will be retained. It is then up to the service provider to implement the technical means to enforce the policies.

Although P3P is useful to communicate the resulting policies, the technique does not contribute to solve our research questions. The actual content of the policies, such as the retention limit, is still only based on the services provider's requirements, and does not directly take users' privacy requirements into account. Weak policies can still easily be pushed on the ignorant user; although P3P is supported by several modern web browsers, and many web sites already specify policies, the policies are quite concealed and few users actually read them or are able to fully understand them [37]. To make P3P more accessible to the users, there are tools available nowadays, which make it possible to express user preferences which can be matched

against the collectors' policies [112]. Still, this makes it only possible to opt-out or reject policies, with as a result that the user cannot benefit from the offered services, making it unlikely that the user indeed will reject the policy.

Preibusch [74] proposed an extension to P3P to overcome the limitation that policies cannot be negotiated. He identifies four dimensions on which privacy can possibly be negotiated: *recipient* of the data, *purpose* for which the data can be used, *retention period* of the data, and the *kind* of data which can possibly be collected. The model tries to capture the usability for the user when providing which data, given its privacy sensitivity, and to use this usability in the negotiation process with the service provider. However, although stated as one of the privacy dimensions, the model does not take retention periods into account as a determiner of the risk for a user to provide its data. Hence, although the amount of data and the type of data provided to the service provider might be limited due to the outcome of the negotiation, the provided data might still be stored unnecessarily long because of an overstated retention limit.

A characteristic of P3P is that it only *describes* policies and preferences and does not *enforce* them [6]. Client-side tools can check if the user preferences don't restrict data collection as stated in a policy, and warn the user or even prevent the data to be sent to the server. Server-centric architectures, such as described by Agrawal [6], make it possible to match the user preferences as part of the data storage itself, making it easier to also actually enforce the policies. Hippocratic databases, as we will describe in the following, are designed for that purpose. Moreover, Bertino et al. [17] give directions on how to design a privacy-preserving database which is able to enforce the policies, based on fine-grained access control techniques. Another attempt is called E-P3P [14], and offers an architecture to enforce P3P-like policies in enterprises, also based on access control techniques.

Above observations show that P3P is little more than a standardized complement to the privacy laws of most countries [45]. Especially when P3P is implemented based on access control, it does not provide protection against threats described in our threat model. The mentioned systems cannot prevent database administrators or malicious users to get access to the privacy-sensitive information in an unauthorized way [14]. Nevertheless, P3P has been a first step in making the handling of privacy-sensitive data more transparent, increasing the information symmetry, and helping to try to maintain the privacy-sensitive information in compliance with laws and privacy principles.

Hippocratic databases and other disclosure-preventing architectures

> "And about whatever I may see or hear in treatment, or even without treatment, in the life of human beings—things that

> should not ever be blurted out outside—I will remain silent, holding such things to be unutterable" - *Hippocratic Oath* [5]

Hippocratic databases, as proposed by Agrawal et al. in 2002 [5], are database systems which have the task to enforce privacy policies. In the same spirit as the usage of the 'Hippocratic Oath' by doctors—they swear to ethically practice medicine—a Hippocratic database is responsible for respecting privacy principles once privacy-sensitive information entered its system. The ten principles on which Hippocratic databases are built are directly derived from privacy laws [40]; they are *purpose specification*, *consent*, *limited collection*, *limited use*, *limited disclosure*, *limited retention*, *accuracy*, *safety*, *openness*, and *compliance*.

Some of those principles we already discussed before. For example, *purpose specification* requires that for all the data which has been stored in the database, the purpose for which the data has been collected is associated with that data. Massacci et al. [68] provide a way to reason about purposes and the data needed to fulfill those purposes. *Consent* requires that the donor has given its consent for those associated purposes. This is in the spirit of P3P, which makes it possible to communicate those purposes.

A Hippocratic database has to enforce the *limited disclosure* principle, which states that data should only be disclosed for purposes for which consent has been given. Because they are not capable of regulating access per data item, traditional access control mechanisms do not have the capabilities to enforce this limited disclosure principle. LeFevre et al. [61] provided a solution based on query rewrite rules which make it possible to limit access on a cell level, based on privacy meta-data stored in the database. However, although the technique is transparent for applications, and indeed regulates access to the stored data, it will not prevent malicious database administrators to bypass the access control mechanism. Moreover, any attacker who can bypass the access control mechanisms and grant himself access to the plain data files, will violate the limited disclosure principle.

Principles such as *safety* and *compliance* require that personal information shall be protected against theft and other abuse, and that the donor of the information should be able to verify compliance with the principles. Still, the donor can only expect that the system is *honest* and has no guarantees that the system will never be successfully attacked, even if it can held the database responsible. The *limited retention* principle helps to limit the impact of such an event. However, an open question remains how to effectively remove the data from the tables and log files [5, 71, 81]. Moreover, when multiple purposes are defined, the data has to stay in the system to serve the longest lasting purpose. A more fine-grained definition of limited retention is required to prevent that more data than needed is kept to fulfill those purposes. The limited retention principle as stated for Hippocratic

database only covers the restriction of the *quantity* of data needed for a purpose, but not the *quality*, whereas both quality and quantity determine the privacy sensitivity of the data [29, 53]. A more extended discussion on limited retention will follow later in this section.

Byun and Bertino [29] recognize that the decision to provide access to a particular data item should not be *binary*: access to a data item should not be either allowed or denied. For some applications, access to a less precise value of a particular data item can be sufficient to fulfill its purpose. By using generalization techniques, data can be disclosed through so-called *micro-views*. Those views provide different representations of the same data based on the associated privacy policy. Instead of not disclosing the data item at all, now at least a less precise representation of the data can be provided to an application, increasing the overall usability of the data. Whereas micro-views use this assumption to refine the limited *disclosure* principle, data degradation is based on the same assumption to refine the limited *retention* principle.

More privacy-preserving architectures have been proposed. For example PawS [60] (Privacy Awareness System) provides a sense of accountability for protecting privacy, but explicitly does not give any guarantee. The system aims to provide control to the users, and to provide ways to check whether the system indeed complies with the privacy policies, in the same spirit as Hippocratic databases. Again, although the system can be considered as being *honest*—it implements all reasonable access control techniques to prevent security breaches—it still requires limited retention to help overcome the fact that the system is not tamper-proof.

Server-side encryption

Although not completely preventable, the chance of a trail disclosure because of a piracy attack can be limited by applying various security measures. Encryption [50] can be used, but as long as the encryption keys are managed by the service provider, the technique is ineffective to prevent trail disclosure by negligence and weak policies, and it does not help to limit the impact of such a disclosure [48, 25]. Server-side encryption does not prevent trail disclosure by malicious data administrators, when the data is subpoenaed by court, or when the server cannot be fully and permanently trusted. Intrusion detection systems (IDS) [32] can be used to detect and prevent repetitive attacks, and is therefore very useful in combination with limited retention. With IDS it will be hard for an attacker to obtain a large set of consecutive history of data by spying a database.

Client-side protection

When the decryption keys are not stored at the service provider, and the user is needed to decrypt the privacy-sensitive information, encryption can still be a solution to prevent trail disclosure. Some queries can be executed partly on the encrypted data stored on the server, and for the remainder the queries will be executed at the client-side, where the data can be decrypted [48]. Bouganim et al. [25] propose a solution based on secure chips on (for example) smart cards required to execute queries on the server. Finally, information sharing across private databases [4] makes it possible to execute queries without disclosing the underlying privacy-sensitive information.

Other—visionary—techniques have been proposed to put the donor in control of his data, such as the P4P framework [3] (not to be confused with P3P). The framework aims at the 'paranoid' user who does not trust the service provider and wants full control over what is released to whom. Instead of having to trust *all* service providers to which sensitive information has been sent, the user only has to trust a single trusted agent. Confab [53], a toolkit for the construction of privacy-sensitive ubiquitous computing applications, is designed based on the idea that personal information should be processed as much as possible at the end-users' computer. Like P4P, the main purpose is to let users be in control over what is disclosed to service providers.

Although these solutions can prevent trail disclosure, they require that each query and update will first be communicated to the clients, putting a severe constraint on applications and service providers. In addition, such techniques require the end-user site to be trusted. This assumption is difficult to put into practise, since end-user sites are often infected by viruses and less protected than central servers. Introduction of secure hardware on the client side make this assumption valid, but requires adapting the processing techniques to strong hardware constraints [7]. Limited retention does not put these restrictions on the applications, since data can still be managed by the service provider in a centralized storage model, where only *honesty* is required.

2.3.2 Privacy-preserving data disclosure

In the previous section we described attempts to limit the disclosure of data, mainly with the use of access control techniques and various security measures. Another class of privacy-preserving techniques focuses on limiting the privacy sensitivity of a data set when the data will be disclosed anyway. In this section we make a distinction between *voluntarily publishing* a data set, and the *unauthorized disclosure* of a data set. Although the cause is different, both types of disclosures result in the exposure of a data set to

possible adversaries, which can use all possible available tools to scrutinize the data set to violate privacy.

Still, although the result of disclosure is the same—namely the exposure of data to possible adversaries—the underlying threat model is different. Privacy-preserving data publishing [84, 85, 62, 66, 88, 97, 1] assumes that as long as the data has not been published yet, the data is stored on a trusted server which thus is not vulnerable to attacks. After publishing, the data cannot be protected anymore, and therefore needs to be made less privacy-sensitive. The objective is therefore to publish the data set in such a way that no (or as little as possible) privacy-sensitive information can be linked to individuals, while keeping the overall data set as useful as possible. In the following we summarize the main contributions in this field; for an extensive survey we refer to Fung et al. [42].

Limited retention and data degradation aims to minimize the impact of *unauthorized disclosure*. As we have argued before, solutions based on limiting the disclosure of privacy-sensitive information can only give a sense of accountability; it can not give any guarantee. Hence, in the threat model considered by limited retention, as discussed in Section 2.2, it is assumed that data can be disclosed at *any time* and not at a predefined moment as is the case with data publishing. By limiting the retention of data, and thus keeping the data set small, the privacy sensitivity is reduced at any moment in time compared to the case when no privacy-preservation was applied. Hence, by applying limited retention the data is always *prepared* for being disclosed. As a result however, any usability of the data will be removed in the process, although the remaining set of data which *is* disclosed is still privacy-sensitive.

Why not continuously applying privacy-preserving data publishing techniques to the stored data set, instead of using limited retention techniques to limit the impact of unauthorized disclosure? Firstly, in many cases the service provider wants to be able to provide *personalized* services. After applying anonymization techniques, this is not possible anymore. Secondly, anonymization techniques can in general not be applied to a dynamic data set without keeping the original data, or at least a subset of it; the techniques are usually only applicable to make the data set less privacy-sensitive in one single run. If the original data has to be retained in the system, the technique does not fit into our threat model.

Still, as we will see in the following, *data degradation* borrows techniques used by privacy-preserving data publishing. Moreover, to fully understand the benefits and applicability of data generalization, and to be able to not confuse data degradation with anonymity, a good understanding of anonymity is helpful.

Anonymity research

The first category of threat models considered by privacy-preserving data publishing contains three types of possible privacy threats [42]. The privacy of an individual is breached if:

record linkage The attacker can link an individual to a record, or a group of records in the published data which are likely to belong to that individual. It is assumed that the attacker knows that the record of the victim is in the published data set.

attribute linkage The attacker can infer what the sensitive values belonging to a particular individual must be, or most likely are, given the published data set, and possible background knowledge of the attacker. Again, it is assumed that the attacker knows that the record of the victim is in the published data set.

table linkage The attacker can infer that the record of a particular victim is or is not in the published data set.

The second category of threat models deals with the prior and posterior beliefs of an adversary when the data is published. The privacy of an individual is breached when the attacker has more knowledge, or can know a privacy-sensitive fact with a higher probability than before he examined the published data.

When privacy-sensitive data has to be published without the possibility to link privacy-sensitive data to an individual (*record linkage*), the most obvious way to anonymize the data is to remove all unique identifiers, such as name or the social security number. However, Sweeney [83, 84, 85] showed that many data sets also contain a so-called *quasi-identifier* which, combined with external information, can be used to uniquely identify individuals. The example often used in literature is that 87% of the US population can be uniquely identified by the combination of their date of birth, zip-code and gender [85]. As a result, to make linkage impossible, not only those attributes which directly identify a user have to be removed, but also those attributes which form the quasi-identifier.

However, simply removing all (quasi)-identifying attributes leads to a maximal loss of usability. For many data mining purposes, it is interesting to know the relation between different types of users—for example grouped by geographic location—and other attributes. Although perfect for privacy, by removing the identifying attributes, such a linkage is not possible anymore. To achieve a better trade-off between usability and privacy, Sweeney introduced the concept of k-anonymity. A data set is said to be k-anonymous, when for each record at least $k-1$ other records share the same quasi-identifier. The equivalence class of a record r in a published table T, is the set of all records in T which contain the same quasi-identifier as r [72]. To form such an equivalence class, individual attributes values

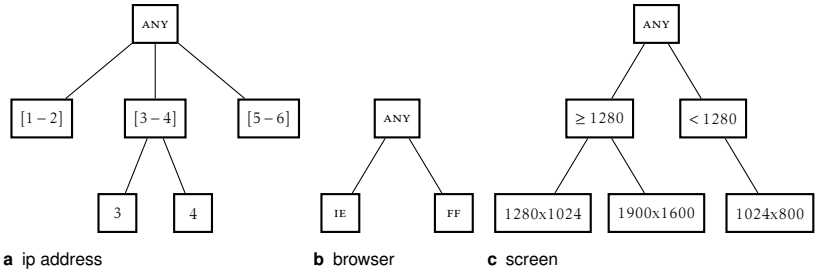


Figure 2.5 Example of generalization trees for *ip* (represented as a single number), *browser* and *screen*. Other generalization schemes are possible too. For example, an additional level might be added to generalize from a set of two ip address, to a set of three ip address. Hence, those generalization trees are arbitrary and can be adjusted to match application requirements.

of the quasi-identifier can be *generalized* so that more records share the same attribute values, and the same quasi-identifier. Three different *generalization trees*, in some literature named *taxonomy trees*, are pictured in Figure 2.5.

Example 1. To illustrate k-anonymity, we use an example based on a simple query log. The log contains the name of the user (for this enterprise search engine you have to sign-in), an ip address (for simplicity we represent the ip address with a single number), the browser (either internet explorer or firefox), the user's screen resolution, the query and the url of the suggested website. We say that the ip address is *not* a unique identifier, since the address can be shared by other users. In practice, additional characteristics of the users' platform, such as browser, screen resolution, browser settings, operation system, *et cetera*, can be used to uniquely identify a user. In this example we additionally say that the combination of ip, browser, and screen resolution can be used for that purpose and thus form a quasi-identifier, and that this information can be public knowledge used by an adversary. A naive anonymization method would be to remove only the name from the query log, as in Table 2.6b. However, by joining this table and the public information in Table 2.6a, the rows can be linked to individuals. Table 2.6c shows a 3-anonymized version of the data, containing three equivalence classes.

k-Anonymity has a number of shortcomings which make that correctly anonymizing a data set, which both ensures sufficient privacy *and* maintains enough usability, is not an easy task. We name some of those problems,

Name	Ip	Browser	Screen
Alice	1	IE	1900x1600
Bob	1	FF	1900x1600
Cathy	2	IE	1284x1024
Doug	3	IE	800x600
Emily	4	IE	1024x800
Fred	4	IE	1284x1024
Gladys	5	FF	1024x800
Henry	5	IE	1284x1024
Irene	6	IE	1284x1024

a External table

Ip	Browser	Screen	Query	Url
1	IE	1900x1600	breast cancer	cancer.com
1	FF	1900x1600	Mexican flu	influenza.org
2	IE	1284x1024	cervical cancer	cancer.com
3	IE	800x600	dogs	dogs.com
4	IE	1024x800	cats	cats.com
4	IE	1284x1024	children swimming	flickr.com
5	FF	1024x800	jobs	werk.nl
5	IE	1284x1024	britney spears	itunes.com
6	IE	1284x1024	Muslim church	religions.net

b Non-anonymized version of the data before publishing.

Ip	Browser	Screen	Query	Url
$[1-2]$	ANY	≥ 1284	breast cancer	cancer.com
$[1-2]$	ANY	≥ 1284	Mexican flu	influenza.org
$[1-2]$	ANY	≥ 1284	cervical cancer	cancer.com
$[3-4]$	IE	ANY	dogs	dogs.com
$[3-4]$	IE	ANY	cats	cats.com
$[3-4]$	IE	ANY	children swimming	flickr.com
$[5-6]$	ANY	ANY	jobs	werk.nl
$[5-6]$	ANY	ANY	britney spears	itunes.com
$[5-6]$	ANY	ANY	Muslim church	religions.net

c 3-anonymized version of the data after publishing

Figure 2.6 Simplified example of a query log containing a set of identifying attributes, and privacy-sensitive attributes. We assume that the ip address, and which browser and screen resolution of a particular user, can be public knowledge. *Name* is a unique identifier, <*ip*,*browser*,*screen*> form the quasi-identifier, and <*query*,*url*> are the sensitive attributes. Although the unique identifier has been removed from the data in table b., the privacy-sensitive information can still be linked to individual users. The data in table c. has been correctly anonymized such the probability of relating a particular query to the correct user is $\frac{1}{3}$. For demonstration purposes, and in line with the simplifying assumptions of most anonymization techniques, we assume that there is only one entry per user in the query log; in practice this will not be the case.

and refer where applicable to literature discussing those problems in more detail:

- First, the data publisher has to know which attributes an adversary can use to link the published data records to an external table. Hence, it has to choose a good quasi-identifier. Choosing a too small quasi-identifier leads to privacy risks, whereas a too large quasi-identifier leads to loss of usability. For example, if screen resolution would be left out from the quasi-identifier in our example, an adversary would still be able to perform a successful record linkage attack. A good choice of the quasi-identifier is still an open issue [42].
- Second, k-anonymity and many of its derivatives assume that each individual has only one record in the data set, as in our example. In practice, especially in query logs, this assumption will not hold. Without additional measures, users with many entries in the log will be less protected, since within an equivalence class of a certain record with respect to its quasi-identifier containing k records, a large subset might belong to one single individual. This violates the k-anonymity property that each record in an equivalence class can be linked to at least k distinct individuals.

  To solve this problem, Wang et al. [94] proposed (X, Y)-anonymity; each value in X (for example the quasi-identifier) must be linked to at least k distinct values in Y. This Y can for example be an incremental person identifier which after publishing cannot directly be linked to an individual, but can be used to group all records belonging to the same person identifier together. The number of distinct values in Y that co-occur with any value x in X, must be larger than k. Hence, there must be at least k different individuals which share the same quasi-identifier in a group after anonymizing the data set.
- Third, if most of the sensitive values in the same equivalence class are equal or similar, an attacker does not need to link a particular record in that class to the correct individual, to have a good guess on the sensitive value of that victim. For example, although Table 2.6c is 3-anonymous, two out of the three individuals have cancer. Hence, the probability that *Alice* has cancer is not $\frac{1}{3}$ but $\frac{2}{3}$. Moreover, given the background knowledge that Bob cannot have *breast* cancer nor *cervical* cancer, Bob must have Mexican flu—assuming that an entry in a query log reveals one's disease—and the probability that Alice has cancer is not $\frac{2}{3}$ but 1.

  This type of attack is called *attribute linkage*, and has been addressed by, among others, Machanavajjhala et al. [66]. They introduced ℓ-diversity, which requires that each equivalence class contains at least ℓ distinct sensitive values. Martin et al. [67] propose a language which can express background knowledge, and an algorithm which

can sanitize a data set given the worst case background knowledge scenario. Still, the more background knowledge is taken into account, and the more a data set needs to be sanitized to preserver privacy, the more usability will be lost. For an overview of all related work discussing this problem we refer to [42].

- Fourth, k-anonymity does not prevent *table linkage*, that is, the ability to know if a victim is or is not part of a particular published (sub) data set [72]. For example, if the publisher publishes an anonymized set of records containing the quasi-identifiers of individuals which have cancer, it should not be possible for an attacker to know if a victim is in that data set or not. If we would release the records of the individuals who have cancer, we could for almost all persons in the external table know whether or not they are in that released set. Only for Alice, Bob and Cathy the probability is $\frac{2}{3}$. This problem is related to the problem of diversity, as sketched above. The notion of δ-presence requires that the probability that a victims record r is present in an published data set is bounded by an interval $[\delta_{min}, \delta_{max}]$, where the published data set T is a subset of the public available knowledge P, and $r \in P$. Nergiz et al. [72] provide an algorithm which ensures δ-presence; however, the algorithm assumes that it is known what the external knowledge P of the attacker is, which might not be a practical assumption [42].
- Fifth, k-anonymization does not take into account that some records are more privacy-sensitive than others, and that some users require less privacy protection than others. Xiao et al. [97] provide *personal privacy*, by letting record owners specify how much privacy they require, and what level of sensitivity may be exposed to an attacker.
- Sixth, once a k-anonymous data set has been published, it cannot be updated anymore. However, it may be desirable, especially considering our threat model (see Section 2.2), to keep only an anonymous version of a data set, and insert new records directly in the anonymous set. The problem is that for each record, there must be already $k-1$ other records in the data set which share the same quasi-identifier. If not, already present records have to be generalized further to be able to insert the new record. This might result in a data set where all quasi-identifiers have been generalized to the most general level. Byun et al. [31] propose to buffer inserted records until there are enough records sharing the same quasi-identifier, possibly resulting in situations that the insertion will be delayed endlessly, and/or that large memory buffers are required.

Data degradation uses the same generalization techniques as used with anonymization. However, as said earlier, k-anonymity and all above mentioned derivatives of the technique protect against different threats than

limited retention (and data degradation). Moreover, as we have seen, one of the main problems of anonymization is to provide the necessary guarantees that a particular individual cannot be linked to a sensitive value, even not with small probabilities, and especially not when data usability plays an important role. Although it is possible to *degrade* (quasi)-identifying attributes as well, data degradation does not aim at providing anonymity, but takes another approach to protect privacy by decreasing the privacy sensitivity of the sensitive attributes themselves. Hence, it does not prevent record linking, attribute linking, or table linking (until the point the sensitive values are fully removed).

The first difference in technique is that we do not need to generalize the (quasi)-identifier of a record, but only the sensitive attributes. The work on t-plausibility [55] adopts the same strategy in the context of sanitizing text documents. A text document d is t-plausible if at least t *base* texts can be generalized to d using a given ontology (comparable to generalization trees). Hence, given a t-plausible document, there are at least t possible 'original' texts. The result is that texts can still be related to individuals, but the sensitivity of the text itself has been decreased.

The second difference is that we do *not* need to generalize many records at the same time to ensure that at least k records share the same sensitive attribute. There is no foreseeable benefit—other than performance related—of such a method, which makes it easier to degrade dynamic, incremental data sets. Third, the assumption that only each individual has only one record in the data set is not needed, making data degradation better applicable for most applications in which several facts of an individual are recorded over time, such as in query logs.

In cases where data miners have much more interest in the sensitive values, and not (or much less) in the identity of the record owners, it is not desirable to degrade the sensitive values. In this case, anonymization will be a better choice. However, when the relation between identity and sensitive value is of more interest, a choice can be made between degrading sensitive values or hiding the identity. However, anonymization can always be used as a technique orthogonal to data degradation.

Data retention and data removal

The motivation behind limited retention is to limit the amount of privacy-sensitive data which can be disclosed, when the disclosure itself is not unauthorized. Indeed, when disclosure is not authorized, there is no time or opportunity to apply any privacy-preserving publishing technique to make the data anonymous. However, limited retention also has problems, of which some have been studied in literature, and some will be discussed in this thesis (see the research questions in Section 1.1). In this section we give a short overview of existing work.

One of the problems in data retention is the question of what to keep, and what to remove, and especially *how* to remove. Miklau et al. [71] argue that both retaining data to the benefit of system accountability and removing data to the benefit of privacy are legitimate goals. However, they argue that although it can be indeed useful to store a historical record of activities on the data to detect errors and malicious behavior, those benefits are small compared to the threat to privacy. The disclosure of a single privacy-sensitive item can cause a big privacy threat, while the single item alone is not important for accountability purposes. Moreover, data which is retained for the purpose of accountability (such as transaction logs) often falls outside the protection of access control mechanisms. Besides, the standard interface to the database system (such as SQL) shows a different view of the stored data than actually can be recovered by, for example, forensic analysis of the database and the underlying file system [81]. Therefore Miklau et al. propose that the system should be *transparent* in the sense that queries on the stored data faithfully represent what is retained in the system. This requires that data is indeed removed from transaction logs, indices, *et cetera* when the user issues a delete statement, or when retention periods expire. We adopt this principle in chapter 4.

Securely deleting data, that is, removing data such that it cannot be recovered is not an easy task [81]. The most basic technique to securely delete data is to overwrite the data. However, overwriting data on disks can be expensive, especially when the granularity is high. In theory, every delete operation would require one I/O operation, which is costly. Moreover, forensic analysis can reveal the original data after it has been overwritten. By analyzing the disk using magnetic force microscopy, it is possible to reveal the previous value of a particular bit, due to the fact that a disk head is never able to write an exact 'zero' or 'one', but only something what is close to it; 'how close' is related to the previous value of the to be written bit [47]. Hence, when an attacker is able to get hands on the physical disk, he might still be able to recover data which is assumed to be deleted.

Boneh et al. [24] suggested to 'delete' data using encryption techniques. The idea is simple: encrypt the data, and throw away the encryption key when the data should not be recoverable anymore. Assuming that the encryption is strong enough to be unbreakable in a reasonable amount of computing time, the effect is the same as physically destroying the data. Boneh et al. adopt this principle to be able to 'remove' data from old backups without needing to access those backups (which might be very expensive, especially when those backups are stored on tapes and off-site). However, destroying an encryption key means that all data encrypted with that key will be destroyed. Hence, to be able to perform fine-grained deletion, every data item needs to be encrypted with its own key, which requires fine-grained key management techniques. To enforce timely deletion of data using encryption, a third party can be used to manage those keys [87].

The third party is needed to gain access to the encrypted information, which will revoke the encryption key after the retention period has expired. Still, every access of the data requires decryption of the data, and requires, depending how the keys are management, costly key management overhead. Hence, although very efficient for deleting data, encryption has its shortcomings which make the technique less applicable in the context of service providers requiring fast and easy access to the stored data [71].

An important aspect of data retention is not only to decide what to remove, but also to decide what to retain. Although for privacy purposes the retention period should be bounded with a maximum, legislation might require that some records are stored during *at least* a certain period, requiring that those records should be protected against user deletes [15]. Protecting records against deletion can severely harm privacy when retention periods are overstated and not match the purpose; however, it can be indeed useful for service providers to be able to protect records against deletion to protect usability. In line of the limited retention principle, the records should be removed directly after the minimal retention period expires. We adopt this principle in chapter 6, where we will look at service-oriented data degradation. Ataullah et al. [15] provide a formal framework which enables users to specify retention limits, and makes it possible to verify that those retention limits do not harm protective retention policies. In a similar spirit, Schmidt et al. [79] describe an extension to relational algebra, to incorporate the notion of expiration time into relational database systems.

Data degradation, as a form of limited retention, adopts many of the techniques described in this section. First, we recognize the need for secure deletion, since it should not be possible to reverse a degradation step. In our threat model, this is a necessary requirement and differentiates data degradation from access control based protection. Without the requirement, access control techniques could be used to implement data degradation, providing the same level of protection, such as the work on micro-views [29]. In chapter 4, where we discuss the impact of data degradation on traditional database systems, we discuss this problem in more detail.

2.3.3 Metrics for privacy and usability

The idea behind privacy-preserving data publishing—and also behind data degradation—is to find the best trade-off between data usability and privacy. Such a trade-off can only be quantified if both usability and privacy can be measured. Most work concentrate on guaranteeing at least the privacy requirement—such as k, ℓ, (X,Y), *et cetera*—while maintaining as much usability as possible [65]. By increasing or lowering the privacy requirement, less or more usability can be maintained. However, as a result, Brickell et al. argue that even small increases in privacy caused by data anonymization result in a destructive loss of usability.

However, the difficulty is *how* to measure data usability, and the obtained privacy. It is hard to speak about a trade-off if the privacy guarantees themselves, as given by the different techniques, cannot be quantified. Some sort of basic ordering of privacy-preserving publishing techniques is possible. As we have seen in Section 2.3.2, we can say that k-anonymity provides less privacy than ℓ-diversity, but provides more usability since less generalization of the attributes is required. To *quantify* the privacy guarantees, and thus make statements as *how much* more privacy a technique provides, is not an easy task, since the concept 'privacy' is vague. Moreover, it can be the case that, for example, according to the *definition* a data set is k-anonymous, but that due to inference techniques the actual level of privacy is negligible. We have seen such an example in Table 2.6c, where although the table is said to be 3-anonymous, Bob has not gained any *additional* privacy compared to the non-anonymized data set.

In the following we give an overview on existing work on metrics for privacy and usability. We start with a summary of the privacy objectives of some of the anonymization techniques we discussed earlier. Although we do not provide a metric to measure the effect of *data degradation*, this related work can give some directions on how such a metric might look like.

Privacy and anonymity metrics

A strict interpretation of privacy in the context of privacy-preserving data disclosure is, that an individual's privacy has been preserved when an attacker does not know more facts about this individual after he has seen the published data set, compared to his knowledge before he has seen the published data set. This principle is named *differential privacy*, and is hard to achieve if background knowledge of the attacker has to be taken into account [39]. Most work on privacy-preserving data disclosure therefore assumes the presence of limited background knowledge, and privacy is only breached when an attacker can successfully link a record, attribute or table to an individual.

Hence, in general, the 'privacy' provided by most anonymization techniques can be expressed as the probability that a certain record or attribute value can be linked to an individual. With k-anonymization [85], the probability that a record can be linked to an individual should be at most $\frac{1}{k}$ for each record. With ℓ-diversity [66], and especially in the p-sensitive k-anonymous variant [88], each group of records sharing the same quasi-identifier should contain at least ℓ different sensitive values. Note that such a ℓ-diverse data set automatically is k-anonymous, since ℓ records must share the same quasi-identifier, and therefore the 'privacy-protection' is at least equal or better than that of k-anonymity.

However, those ℓ different sensitive values should be 'different enough' to really provide privacy. For example, if all really confronting and privacy-

sensitive diseases end up in the same group, the actual amount of privacy is still limited. Moreover, since ℓ is not based on a probability, such as k, it is hard to translate the meaning of the number to a human sense for privacy. As a result, it is hard to use such a number for making a trade-off between privacy and usability [42]. Another measure for privacy risk is the probability that a sensitive value can be inferred; it has been proposed by Want et al. [93]. For each quasi-identifier and sensitive value combination, the *confidence* of an attacker that he guesses the correct sensitive value for a particular quasi-identifier should be bounded to a percentage h. Still, although all measures say *something* about the privacy risks after a data set has been disclosed, it is hard to relate them to each other.

Data degradation can use the same approach to fix a privacy guarantee as used by anonymization techniques. When a particular data item x is generalized to x', it is *hidden* in a group with $n-1$ other possible values which also could be generalized to the same value x'. The larger this group is, the less privacy-sensitive a data item is. Note that this technique does *not* require that there are $n-1$ other records in the data set which have the same sensitive data item x'. Hence, there is no analogy with k-anonymity in the sense that n records have to share the same quasi-identifier, or that n records should share the same sensitive attributes.

However, the amount of privacy still depends on many factors. If *all* (or most) leaves of a generalization tree are equally privacy-intruding to an individual, generalizing the data item does not reduce the privacy sensitivity. For example, if *breast cancer*, *cervical cancer* and *lung cancer* are the only possible *specializations* of the generalized data item *cancer*, an attacker can only be 33% confident of the correct type of cancer a victim has. However, when the attacker has background knowledge, and because of that knows the victim is male, he can be 100% confident that the victim has *lung* cancer. Hence, generalization trees (taxonomies) should be diverse enough to indeed make data items less privacy-sensitive thanks to data degradation.

Finally, the limited risk for privacy because of the use of limited retention techniques is mostly caused by the fact that the *quantity* of the data which possibly can be disclosed is reduced. In chapter 3 we will take the retention period, which determines the size of a data set assuming constant insert rates, as the base measure for privacy risk.

Usability metrics

It is not only difficult to measure privacy, it is also difficult to capture the usability of data. Intuitively, it decreases when the precision of a data item decreases. However, if the precision is not required for the purpose for which a data item is being used, the usability will not decrease after generalization. On the other hand, when the purpose cannot be fulfilled since the purpose requires full precision, all usability will be gone after

generalization. Therefore, most metrics are based on the principle that a purpose can still partly be fulfilled with less precise data. For example, the *precision metric* used by Sweeney [84] for k-anonymity to capture usability uses the level of generalization (h) of a certain attribute value, relative to the total number of levels in the domain generalization hierarchy ($|dgh|$) for that attribute. More precisely: given an original table P and an anonymized version R with $|R| = |P|$ rows, quasi-identifier $QI = A_1, \ldots, A_n$, and the domain generalization hierarchy dgh_i for attribute A_i, the precision metric $Prec$ is defined as follows:

$$Prec(R) = 1 - \frac{\sum_{i=1}^{n} \sum_{j=1}^{|R|} \frac{h_{i,j}}{|dgh_i| - 1}}{|R| \times n} \tag{2.1}$$

A value of attribute i in row j is not generalized when $h_{i,j} = 0$, and is thus in the first level of its generalization hierarchy. Hence, when each $h_{i,j} = 0$, and thus no quasi-identifiers have been generalized, the precision is one. When each $h_{i,j} = |dgh_i| - 1$, and thus all quasi-identifiers have been fully generalized, the precision is zero.

Such a metric takes the distortion of the data due to each generalization step into account, but not the *amount* of distortion per generalization step. This amount of distortion can be measured using the number of values in the domain of a generalized value compared to the size of the domain of the not generalized value. This is used in a precision metric introduced by Xiao et al. [96, 97], where $count(h, dgh)$ gives the number of possible values at level h in the domain generalization hierarchy dgh:

$$Prec'(R) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{|R|} \frac{count(h_{i,j}, dgh_i)}{count(0, dgh_i)}}{|R| \times n} \tag{2.2}$$

When all attributes in the quasi-identifier of all tuples have been generalized to the highest level in the tree, the precision of the anonymized table will be close to zero, when the domain is large. When no attributes have been generalized, the precision is exactly one. Metric $Prec'$ (equation 2.2) can be interpreted as follows. Imagine that there are ten research groups, each group containing ten researchers. Hence, there are 100 different researchers, so that $count(0, dgh) = 100$ and $count(1, dgh) = 10$. When, after generalization, a researcher is represented by its research group, the usability of that piece of information has been decreased to $\frac{10}{100} = 0.1$. Hence, the metric is the sum of individual probabilities that the original value can be correctly guessed. Note that not all attributes might be equal in terms of the amount of usability it carries. Therefore, equation 2.2 can be extended with a weight factor assigned to each attribute (equation 2.1 can be extended in

a similar fashion):

$$Prec'_{weighted}(R) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{|R|} \frac{count(h_{i,j}, dgh_i)}{count(0, dgh_i)} \times w_i}{|R| \times \sum_{i}^{n} w_i}$$

Now, guessing the correct value of the ith attribute is rewarded with a weight w_i. Note that this metric does not take into account that intermediate values could also be guessed. For example, research group can be further generalized so that it reaches the root of the domain generalization hierarchy. After this generalization step, the precision of such a data item is $\frac{1}{100} \times w$ (with a total of 100 researchers). In earlier work, we proposed a refinement of this metric [91]. This refinement of the metric also assigns a weight $w' < w$ for guessing the correct research group, of which the probability is $\frac{1}{10}$. Then, the precision of this fully generalized data item would become $\frac{1}{10} \times w' + \frac{1}{100} \times w$.

Data degradation can use the same approach to measure the precision of the data set as used by anonymization techniques. Instead of measuring the distortion in the quasi-identifier, the distortion of the sensitive attribute will then be measured. Still, providing a suitable metric which is able to take all relevant aspects of data degradation into account is left for future work.

2.4 Conclusion

Recall the following starting points:

- Our objective is to limit the impact of unauthorized data disclosure.
- The threat model assumes that unauthorized data disclosure cannot be completely prevented.
- The failure of existing techniques to meet our objective make that limiting the retention period of sensitive data is necessary. Although access control and other security techniques can lower the probability of unauthorized disclosure, they cannot fully prevent it. Anonymization can be used to *publish* a data set in a privacy preserving way, but does not provide any protection when data is disclosed in an unauthorized way. Moreover, after anonymizing a data set, this data set cannot be used anymore for personalized services.
- Limited retention lowers the impact of unauthorized disclosure. However, the principle is too rigorous to find a good balance between privacy and data usability. To provide privacy, data will be fully destroyed when the retention period has ended. As a result, retention

periods are overstated to serve long lasting purposes, even if highly precise data is not required for those purposes. Data degradation can overcome this lack of balance between privacy and usability.

The objective of this thesis is to explore limited retention, and data degradation as a derivative of limited retention, as a new way to solve the problem sketched in above observations. This leads to the four research questions as described in Section 1.1, namely:

1. How to model the interest of both service provider and user, to find the best retention period of privacy-sensitive data?
2. How to refine the limited retention principle, to better balance the interests of service provider and user?
3. What is the impact of data degradation on traditional database systems, and is it feasible to implement the technique?
4. How can the concept of data degradation be further exploited when the simplifications are released?

The following chapters will answer those questions.

3 Limited retention and degradation model

The limited retention principle states that privacy-sensitive information should not longer be stored than necessary to fulfill the purpose for which the information has been collected. As we have argued in the previous chapter, some services do not have a purpose which is clearly bounded in time. When, for example, is the purpose 'improve search results' fulfilled, such that information should be removed from the query log? In such cases, service providers will store data as long as possible, which in practice can be infinitely long.

We argued before that storing data infinitely long leads to severe privacy risks. To limit the risks of data retention, the retention period should be limited. The problem is that service providers can easily *solely* determine the retention period, not taking the privacy risks, and thus the users, into account. This is partly due to the lack of a framework in which it is possible for users and service providers to *reason* about the retention periods, on equal footing.

In the following we present such a framework. The aim is to model the *interests* of *both* service provider and users, and bring those interests together in what we name the *common interest*. This makes it possible to reason about retention periods such that an optimal retention can be chosen.

Moreover, we introduce the concept of data degradation, which is a refinement of limited retention. Limited retention means that there is one single retention period, after which a collected data item should be removed; with data degradation, data is gradually removed. With each step, the precision of the data will be decreased, and be made less privacy-sensitive. In particular cases, under well-defined assumptions, it is possible to achieve a higher common interest for both service provider and users.

The organization of this chapter is as follows. First we introduce our framework in which it is possible to reason about limited retention. We model the interest of service provider and user, and show how to derive

an optimal retention period. Secondly, we refine the framework to incorporate the concept of data degradation, and show examples in which data degradation leads to a higher common interest, and gives rise to the research objectives of the remainder of this thesis. Finally, we give a more in-depth definition of data degradation and the underlying concepts. Those concepts will be used throughout this thesis. In chapter 6 we will propose extensions to this model, and provide an outlook to how the concept of data degradation can be used in practice.

3.1 Limited retention

3.1.1 Finding limited retention periods

To let the reader get familiar with our notations and reasoning, we introduce our concepts for the limited retention principle here. In the next section we will reuse and extend those concepts when we refine limited retention to data degradation. Our goal is a qualitative framework which makes it possible to reason about retention periods. We model the interest of a service provider and, separately, the interest of the user in a way that is as simple as possible; a more elaborate model is left for future work. Then we combine these to a *common interest*, from which an optimal retention period can be derived. We make no quantitative statements about *how* exactly these interests will be expressed in practice; again, the practical implementation is beyond the scope of this thesis.

3.1.2 Preliminaries

In the sequel, we consider only one user, which we refer to as 'the user'. Considering more users is no problem but would lead to the same result if we treat them on equal footing. Also, we shall not make a distinction between different kinds of data: we treat all data on equal footing.

The history of which data item is inserted into the store at which time, is called H; it is a set of pairs of a data item and its insertion time in the store. For example:

$$H \;=\; \{(d,b),\ldots,(d',b')\}$$

We take H as a constant, we let d range over the set of data and t over the set of time points, and use letter b (birth) for 'the insertion time of a data item into the store'. We use $\mathbb{N}$ to measure an *age*, i.e., length of a time interval, and let a,δ range over ages. A simplifying assumption is that the insert rate of data is constant over time; that is, there exists a constant c such that during each interval of length a the amount of data inserted into the store is $c \times a$:

$$\forall t,a \bullet \quad \#\{(d,b) : H \mid t \leq b < t+a\} \;=\; c \times a \tag{3.1}$$

By its definition, limited retention bounds the interval during which a data item is stored by a fixed *retention period* δ. Hence, the *store* at time t depends on δ and is expressed as follows:

$$store(\delta, t) \;=\; \{(d,b) : H \mid b \leq t < b + \delta\}$$

3.1.3 Service provider's interest

We use the term *worth* to indicate the *monetary worth*—although we will not concretely speak in terms of money—or *business value* of a data item for the service provider. We use worth as a technical term to prevent ambiguity of the term *value*, which in the context of this thesis means *the alphanumeric object denoted by an algebraic term*, for example, the value of d is *'Enschede'*.

In practice, the *worth* of a data item for the service provider depends on multiple factors, such as the actual content of the data item, the context in which the data item has been acquired, the user from which the data item has been acquired, time of day, and possibly many other factors depending on the type of use. Our model does not limit the possibility to include those parameters; however, for the sake of simplicity we omit them. Thus, we assume that for the service provider the worth at time t of a data item with birth b depends only on the data item's age $t{-}b$:

$$worth((d,b),t) \;=\; wt(t{-}b)$$
$$\text{where } wt(a) \text{ is non-negative, monotonic descending in } a$$

The auxiliary function wt is monotonic descending since older data is assumed to be less valuable for the service provider. Figure 3.1 gives the typical shape of function $wt(a)$.

The assumption that older data is less valuable is necessary in the context of limited retention. If the worth of the data would *increase* over time, limited retention would severely impact the benefit of collecting data in the first place. Hence, either the data will be stored for a long period without limited retention restrictions (with all privacy risks), or not stored at all. To limit the privacy risk in such a scenario, access control techniques can block access to the data items, and release the data items after some time to restrain the 'actual' retention period. However, such a solution does not provide any protection against our threat model (see section 2.2).

The service provider wants, at each point t in time, to maximize the *total worth* of the data in the store:

$$\begin{aligned} & totworth(\delta, t) \\ = \; & \textbf{definition} \\ = \; & \textstyle\sum_{(d,b):store(\delta,t)} worth((d,b),t) \end{aligned}$$

$$
\begin{aligned}
& \textstyle\sum_{(d,b):store(\delta,t)} wt(t-b) \\
= & \begin{cases} \text{A stored data item of age } a \text{ contributes} \\ wt(a) \text{ to the sum. Thanks to the constant} \\ \text{insertion rate } c\text{, the number of stored data} \\ \text{items of age } a \text{ is the same (namely } c\text{) for} \\ \text{each } a = 0,\ldots,\delta. \end{cases} \\
= & c \times (wt(0) + wt(1) + \cdots + wt(\delta)) \\
& c \times \textstyle\sum_{a=0\ldots\delta} wt(a)
\end{aligned}
$$

It turns out that $totworth(\delta,t)$ does not depend on t, hence we omit parameter t and simply write $totworth(\delta)$. Figure 3.1 gives the typical shape of function $totworth(\delta)$. Without other constraints, the service provider would achieve his goal by setting δ to infinity.

3.1.4 User's interest

For the user it is risky to have data stored at the service provider: in some way or another (hackers' attacks, for instance) the data might be disclosed. We assume that the harm for the user of a disclosure of his data is proportional to the amount of data. Below, we take the risk *equal* to the amount of data since this simplifies the formulas and nevertheless gives the same results. Again, there are multiple other factors which influence the risk of storing a data item. For example, the fact that a user searched for HIV is more risky than a search for flowers. Also, we assume that disclosure of old data is as harmful as disclosure of recent data. Our model does not restrict the possibility to take those factors into account, but we leave this for future work. Thus we define and simplify *risk* as follows:

$$
\begin{aligned}
& risk(\delta,t) \\
= & \quad \textbf{definition} \\
& \#\{(d,b) : store(\delta,t)\} \\
= & \#\{(d,b) : H \mid b \leq t < b+\delta\} \\
= & \quad \text{constant insert rate (3.1)} \\
& c \times \delta
\end{aligned}
$$

So, $risk(\delta,t)$ doesn't depend on t and we simply write $risk(\delta)$.

The goal of the user is to minimize $risk(\delta)$. It is equivalent to *maximize* the inverse: $1/risk(\delta)$, which we call the *privacy* guarantee. It follows that the privacy guarantee is infinite when there is no data in the store, $\delta = 0$ (and goes down to zero when retention is unlimited, $\delta = \infty$). To escape mathematical problems (division by zero) and come to a slightly more realistic model of privacy, we apply a *smoothing* technique so that privacy cannot be infinite: add a constant to the denominator of $1/risk(\delta)$. The

smoothing constant s may have a reasonable interpretation; for example, the fact that there is no data in the store, might be interpreted as an indication that "the user has something to hide" and so his privacy guarantee is not infinite [39]. Thus our definition reads:

$$priv(\delta) = \frac{1}{s + risk(\delta)} = \frac{1}{s + c \times \delta}$$

Figure 3.1 gives the typical shape of function $priv(\delta)$. The user wants to maximize $priv(\delta)$, which without further constraints is achieved by taking δ as small as possible.

3.1.5 Common interest

Above we defined the interests of both service provider and user. Those interests are conflicting; whereas the service provider benefits most when δ is large, the user aims for a δ as small as possible. We want a concept of *common interest* which both parties can agree upon. We expect that the common interest leads to a retention period which is both *limited* ($\delta < \infty$) and *non-zero* ($\delta > 0$).

To define the common interest $CI(\delta)$, we require two things. First, $CI(\delta)$ is proportional to the service provider's goal function $totworth(\delta)$ when the user's interest is viewed as constant. Second, $CI(\delta)$ is proportional to the user's goal function $priv(\delta)$ when the service provider's interest is viewed as constant. Since both goal functions are non-negative, a suitable function $CI(\delta)$ is the *product* of these:

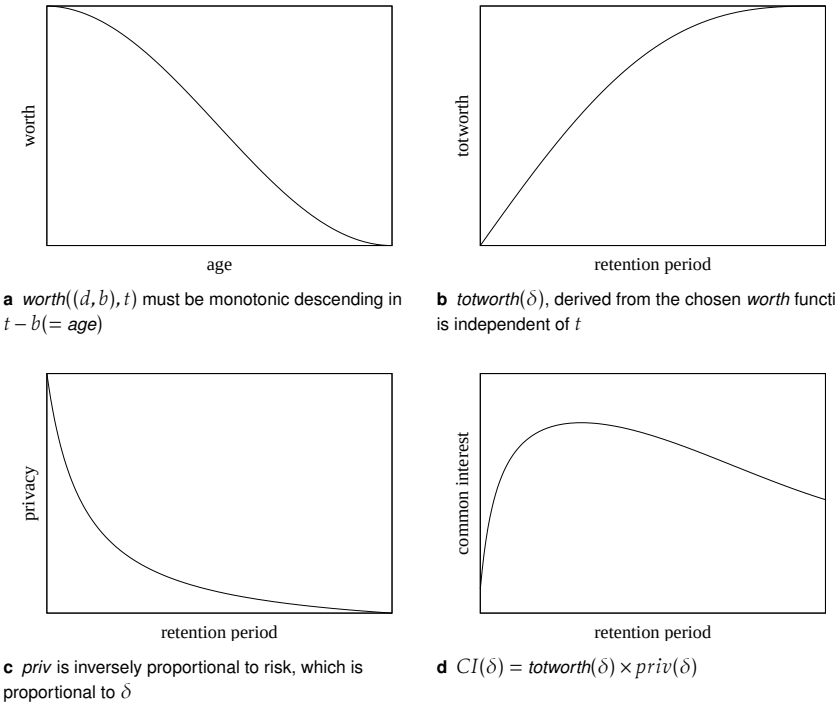
$$CI(\delta) = totworth(\delta) \times priv(\delta)$$

Figure 3.1 gives the typical shape of function $CI(\delta)$.

Since *worth* is monotonic descending and *risk* is (almost) proportional to δ, it follows that $CI(_)$ has a maximum, which it takes on argument δ_{opt}, say. The existence of a maximum can be interpreted in the following way. By setting the retention period smaller than δ_{opt} the user will gain more privacy, but the common interest will be lower because the decrease in stored data induces a greater loss of total worth for the service provider. Similarly, by setting the retention period larger than δ_{opt} the service provider will gain more total worth of the stored data, but the common interest will be lower because of a larger decrease of the user's privacy.

3.2 The concept of data degradation

Recall that the principle of limited retention tries to satisfy the service provider by allowing to store data for at least the retention period δ, and it tries



a *worth*$((d,b),t)$ must be monotonic descending in $t-b(=$ *age*$)$

b *totworth*(δ), derived from the chosen *worth* function, is independent of t

c *priv* is inversely proportional to risk, which is proportional to δ

d $CI(\delta) =$ *totworth*$(\delta) \times priv(\delta)$

Figure 3.1 The common interest function reaches its highest point at δ_{opt} meaning that a retention period of δ_{opt} gives the best balance between *totworth* for the service provider and *priv* for the user.

at the same time to satisfy the privacy concern of the user by ensuring that the data is stored for at most the retention period δ (and the previous section shows how to reason about the optimal retention period). The principle is a crude all-or-nothing approach: a data item either exists completely in the store or not at all. The principle of *data degradation* overcomes the all-or-nothing approach by storing data in progressively less precise forms, so as to make it less privacy-sensitive over time while still providing some worth to the service provider. A well accepted form of data degradation is data *generalization*. This technique is often used in k-anonymity research [85], and is also applied in data mining and warehousing [51]. More about the techniques we use to generalize data—based on domain hierarchies and generalization trees—can be found in section 3.3.

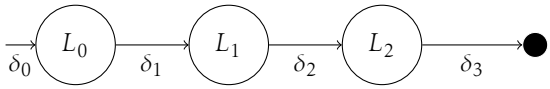


Figure 3.2 Graphical representation of a simple *life-cycle*. Edges denote transitions between *L*evels of precision after a *retention period* δ.

3.2.1 Life-cycle policies

To formalize the data degradation principle, we need some terminology and notation. First, we distinguish several *levels* of precision, say $L_0, L_1, \ldots, L_{n-1}$ in decreasing order of precision. Level L_0 denotes level of highest precision. Second, the degradation from L_{i-1} to L_i is denoted τ_i (τ is mnemonic for "transformation"). Third, the interval from the birth of a data item to its degradation to L_i is denoted δ_i; it follows that $\delta_0 = 0$ because a data item is supposed to enter the store with level of highest precision, and we let δ_n be the interval from birth to removal from the store. The notation $\vec{\delta}$ abbreviates the sequence $\delta_1, \ldots, \delta_n$. Almost all this information is captured in a so-called *life-cycle*, as illustrated in figure 3.2.

So, the store consists of data of age at most δ_n, and degraded to the appropriate levels:

$$
\begin{aligned}
& store(\vec{\delta}, t) = \\
& \quad \{(d,b){:}H \mid b{+}\delta_0 \leq t < b{+}\delta_1 \bullet (d, b, L_0)\} \\
& \cup \quad \{(d,b){:}H \mid b{+}\delta_1 \leq t < b{+}\delta_2 \bullet (\tau_1(d), b, L_1)\} \\
& \quad \vdots \\
& \cup \quad \{(d,b){:}H \mid b{+}\delta_{n-1} \leq t < b{+}\delta_n \bullet (\tau_{n-1}(d), b, L_{n-1})\}
\end{aligned}
$$

The product of our framework is a life-cycle *policy*, which captures how and when data needs to be degraded, and with which the service provider has to comply. As long as the service provider is *honest*, the life-cycle policy ensures that *if* the data store is attacked, the impact of disclosure will be less severe; only a small subset of the data will be stored in a precise form, the rest will be either degraded or destroyed. Some of the technical challenges related to the implementation and enforcement of such policies on traditional database systems are discussed in section 4.

3.2.2 Interests revised

We assume that degraded data is less worthwhile for the service provider but also less risky for the user to store, and we will revise the definitions of *worth* and *risk* accordingly. The definitions of the total worth, privacy, and

common interest in terms of *worth* and *risk* remain the same except for the replacement of δ by $\vec{\delta}$.

Worth and total worth

For the service provider, the worth of a data item depends not only on the age "$t-b$" but also on the level of precision l:

$$worth((d,b,l),t) \;=\; wt_l(t-b)$$

$wt_l(a)$ is non-negative, monotonic descending in a and l

The effect of degrading a data item to a level of lesser precision is that its worth for the service provider is decreased. Indeed, '$wt_l(a)$ is monotonic descending in l' means that for all a:

$$wt_0(a) \geq wt_1(a) \geq \ldots \geq wt_{n-1}(a)$$

As before, the total worth of the store at time t is the aggregation of the worth of all data in the store:

$$\begin{aligned}
& totworth(\vec{\delta},t) \\
= \quad & \textbf{definition} \\
= \quad & \textstyle\sum_{(d,b,l):store(\vec{\delta},t)} worth((d,b,l),t) \\
& c \times (\\
& wt_0(\delta_0) + wt_0(\delta_0{+}1) + \ldots + wt_0(\delta_1 - 1) + \\
& wt_1(\delta_1) + wt_1(\delta_1{+}1) + \ldots + wt_1(\delta_2 - 1) + \\
& \ldots \\
& wt_{n-1}(\delta_{n-1}) + wt_{n-1}(\delta_{n-1}{+}1) + \ldots + wt_{n-1}(\delta_n - 1)\ ) \\
= \quad & c \times \textstyle\sum_{l=0}^{n-1} \sum_{a=\delta_l}^{\delta_{l+1}-1} wt_l(a)
\end{aligned}$$

It follows that $totworth(\vec{\delta},t)$ is independent of t, and we can simply write $totworth(\vec{\delta})$. Note that the contribution to $totworth(\vec{\delta})$ of the most degraded level may be negligible in comparison to the less degraded levels: both the age and the level are higher.

Risk and privacy

The risk of having data in the store was proportional to the amount of data in the store. However, with data degradation, the assumption is that storing data in a more degraded level is less risky than for a more precise level. To express this, we weight the risk of storing a data item in level L_l with a factor r_l such that $1 = r_0 \geq r_1 \geq \ldots \geq r_{n-1}$. Hence, the definition reads:

$$\begin{aligned}
& risk(\vec{\delta},t) \\
= \quad & \textbf{definition}
\end{aligned}$$

$$= \frac{\sum_{l=0}^{n-1} r_l \times \#\{(d,b,l') : store(\vec{\delta},t) \mid l' = l\}}{c \times \sum_{l=0}^{n-1} r_l \times (\delta_{l+1} - \delta_l)}$$

Again, $risk(\vec{\delta},t)$ is independent of t and we simply write $risk(\vec{\delta})$. The definition of privacy doesn't change:

$$priv(\vec{\delta}) \;=\; 1 \;/\; (s + risk(\vec{\delta}))$$

Common interest

The common goal of both service provider and user remains the same: optimizing the common interest. Except for the replacement of δ by $\vec{\delta}$ there is no change in the formalization:

$$CI(\vec{\delta}) \;=\; totworth(\vec{\delta}) \times priv(\vec{\delta})$$

Note that if there is just one level of precision, $n = 1$, the newly defined notions coincide with the already existing notions (such as $CI(\vec{\delta})$ and $CI(\delta)$) provided we take $wt_0(a) = wt(a)$ and $\delta_1 = \delta_n = \delta$.

3.2.3 Benefits of data degradation

The aim of data degradation is not to provide more privacy while ensuring the same amount of worth for the service provider, nor providing more worth while ensuring the same amount of privacy as what can be achieved with limited retention. Instead, we want to show that we can achieve a higher *common interest* with data degradation than with limited retention.

Since the contributions of $\delta_2, \delta_3, \ldots$ to CI are non-negative, data degradation with $n > 1$ will outperform limited retention:

$$\forall \delta_1, \delta_2, \ldots \bullet \quad CI(\delta_1) \leq CI(\delta_1, \ldots, \delta_n)$$

In the following we will show, using experimental examples, that for *some* set of examples which all comply with our assumptions the common interest will be higher when we use data degradation. Hence, our target is to give insights in what we can gain by choosing data degradation in comparison to limited retention.

Analysis

We use Matlab as the platform for our analysis. We implemented a set of worth and privacy functions which simulate the functions which have a practical shape, complying with the assumptions in section 3.1.1. Given those functions, we let Matlab optimize the common interest considering $n = 1 \ldots 4$ possible degradation steps. Hence, choosing $n = 1$ means the same

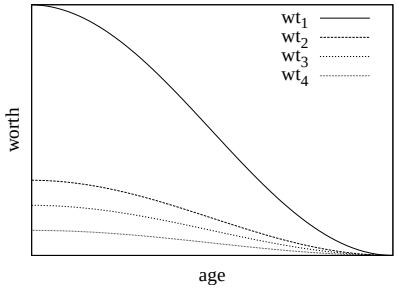


Figure 3.3 $wt_l(a) = w_l \times \left(1 + cos\left(\frac{a \times \pi}{180}\right)\right), 0 \leq a \leq 180$

as applying limited retention, whereas choosing $n > 1$ means we allow one or more degradation steps.

Although we can easily experiment with different types of worth functions to simulate a service provider's worth function—as long as they are monotonic descending—we choose the cosine function, as in figure 3.3.

In the worth functions we use weights $w_1 \geq w_2 \geq \ldots \geq w_n$. Recall that r_i are weight factors in the risk function. We choose $w_1 = r_1 = 1$ and $w_2 \ldots w_n$ vary over $0 \ldots 1$, similarly for r_i.

Increased common interest

To show that using data degradation indeed can result in a higher common interest, we let our script find the $\vec{\delta}$ for which the common interest is maximal, with at most $n = 4$ degradation steps. We choose the following parameters:

$i =$	1	2	3	4
w_i	1	0.3	0.2	0.1
r_i	1	0.2	0.1	0.05
$s =$	18			

With those parameters, we obtain the following results (also shown in figure 3.4).

$n =$	1	2	3	4
$\vec{\delta}$	[53]	[26,81]	[16,46,96]	[16,46,96,96]
$CI(\vec{\delta})$	0.7948	0.8358	0.8532	0.8532
$totworth(\vec{\delta})$	98.5582	76.8960	61.4329	61.4329
$priv(\vec{\delta})$	0.0081	0.0109	0.0139	0.0139

From this result we conclude for the chosen parameters:

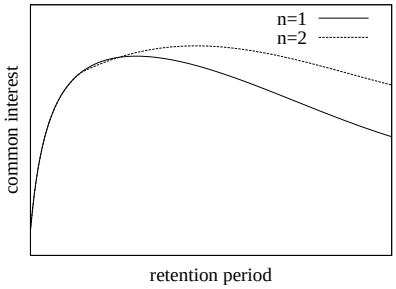


Figure 3.4 Common interest function for $n = 1$ (limited retention) and $n = 2$. For $n = 2$, we only vary over δ_2; the first retention period δ_1 is chosen such that $CI(\delta_1, \delta_2)$ is optimal for the δ_2 where this plot reaches its highest point. One can observe that δ_1 for $n = 2$ is shorter than the optimal δ_1 when $n = 1$, and that the common interest for $n = 2$ is higher than for $n = 1$.

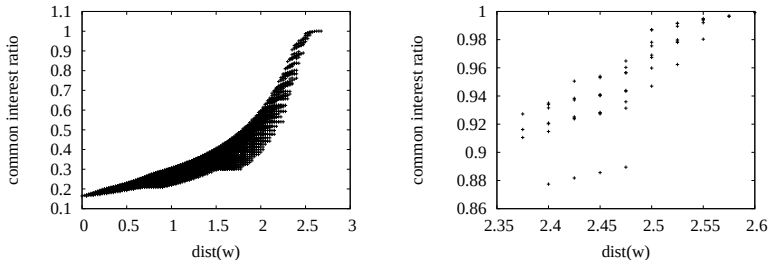
1. Common interests is higher when progressively degrading ($n > 1$) the data than with limited retention ($n = 1$) of precise data.
2. With $n > 1$, δ_1 is smaller than the single δ with $n = 1$. It thus makes sense to degrade the data earlier to achieve a higher common interest.
3. When $n = 4$, it turns out that $\delta_3 = \delta_4$, meaning that it is not possible to achieve a higher common interest with more than three degradation steps.

A closer look on weights and their effect on the common interest

We already have seen that data degradation can result in a higher common interest. An interesting question is how the ratios between weights $w_1 \dots w_n$ and $r_1 \dots r_n$ assigned to each level of precision L_i influence the gain in common interest which can be achieved with data degradation. Our expectation is the following: with $w_i > r_i$, degradation results in a higher common interest compared to limited retention.

For this experiment we choose arbitrarily r_1=1, r_2=0.2, r_3=0.1, r_4=0.05, and vary over w_i. We use a simple distance metric: $dist(\vec{w}) = \sum_{i=2}^{n} w_1 - w_i$ to express the drop in *worth* when degrading the data. Note that the smaller w_i is, the less *worth* is preserved on that precision level.

Figure 3.5 shows for various weights w_i the ratio between the common interest which can be achieved with only limited retention and the common interest which can be achieved by *at most* $n = 4$ degradation steps. Figure 3.5b shows only common interest points achieved with at least 3 degradation steps. The ratio indicates the fraction of common interest possible with limited retention compared to that of data degradation; it



a one-or-multi-step degradation **b** multi-step degradation

Figure 3.5 Ratio between the common interest which can be achieved with only limited retention and the common interest which can be achieved by allowing $n = 4$ degradation steps. A ratio equal to 1 means no increase in common interest, lower than 1 indicates that limited retention performs less than data degradation. This ratio can never be higher than 1; limited retention cannot perform better than data degradation.

cannot be higher than 1 since limited retention can never perform better than data degradation (see earlier this section).

We make the following observations:

1. When $dist(\vec{w}) \simeq dist(\vec{r}) \simeq 2.65$, the decrease in worth is similar to the increase in privacy, so that common interest hardly increases (CI ratio is close to 1).
2. When the decrease in worth becomes higher, data degradation hardly outperforms limited retention.
3. Most gain in common interest is achieved by applying *one-step degradation*: data is immediately degraded to a higher level, and fully degraded afterward. This can be concluded from figure 3.5b, in which only *multi-step degradation* is allowed. All combinations with multiple degradation steps lead to a ratio between 0.86 and 1, where all possible configurations (including one-step degradation) lead to a ratio between 0.15 and 1.

From the last observation, we conclude that for some (our current) parameter values it makes sense to generalize the data before *storing* it. In other words, a higher common interest can be achieved by sacrificing some precision to get much more privacy in return, especially when the privacy increase is much higher than the loss in worth. When the decrease in worth is more close to the increase in privacy, we showed again that multi-step degradation leads to an increase in common interest.

3.3 Data hierarchies and generalization trees

In the following we will go more into detail about *how* the data can be degraded from *level* to *level*. We introduce the concept of transformation functions, and give two particular realizations of those functions: generalization trees and generalization functions. We will discuss with which properties the transformation functions should comply.

3.3.1 Transformation functions

In section 3.2 we already introduced the concept of *Levels* of precision. From now on we consider a partially ordered set of levels of precision (*Level*,$\preceq$,*full*,*null*), and transformation functions between these levels of precision. This is defined as follows:

Level	A level of precision, denoted as α,β,γ,.... In figure 3.2, $\alpha = L_0$, $\beta = L_1$, and $\gamma = L_2$.
$\preceq$	A binary relation describing, for two levels in the set, the requirement that one of the levels must precede the other (if there is a relation between the two levels).
full	The most precise *Level*, so that $full \preceq \alpha$ for any α.
null	The least precise *Level*, so that $\alpha \preceq null$ for any α.

Hence, there is always a path from the most precise level *full* to the least precise *null*. In our particular realization, a level represents a set of attributes and its accuracies, and *null* represents final destruction of all attributes. In this representation, $\alpha \preceq \beta$ has the meaning that all attributes of α have a higher or equal precision than those of β.

For each $\alpha \preceq \beta$ we require a *transformation function*:

Val_α, Val_β	The sets of values at level α and β.
$\tau_{\alpha\to\beta} : Val_\alpha \to Val_\beta$	A function to transform an item from level α to β, where $\alpha \preceq \beta$.

(3.2)

To be able to comply with the limited retention property, we have to put restrictions on the levels which can be reached through transformation functions, and the transformation functions themselves. Hence, it must be possible to transform data to a level from any level *below* that level in the partial ordering. Moreover, if there is path from α to β, and from β to γ, it must be possible to *directly* transform data from α to γ:

$$\tau_{\alpha\to\gamma} = \tau_{\beta\to\gamma} \circ \tau_{\alpha\to\beta} \qquad \text{For each } \alpha \preceq \beta \preceq \gamma \tag{3.3}$$

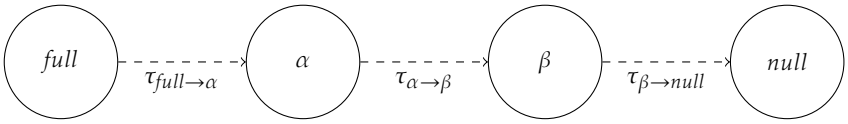


Figure 3.6 Simplified network of *Level* of precision, whit only a single path from *full* to *null*.

Restrictions 3.2 and 3.3 are only required to make sure the model can be implemented using limited retention; to be able to implement a transformation function, it is not needed to keep a copy of the original, precise data.

Transformations only triggered by time

Above, we implicitly made the simplification that degradation is triggered by time. This is a reasonable simplification, since the expiration of a time period is unavoidable, and therefore gives users the guarantee that their personal information indeed will be degraded. Especially when purposes are not clearly circumscribed, and thus do not have clear end-time, time-triggered data degradation is desirable.

Thanks to this simplification, the life-cycle of any data item is fixed; the path from *null* to *full* does not have branches. An example is given in figure 3.6. Requirement 3.3 is automatically fulfilled: $\tau_{full \to \beta} = \tau_{full \to \alpha} \circ \tau_{\alpha \to \beta}$; if it is possible to degrade from *full* to α, and from α to β, it is also possible to degrade from *full* to β.

We use this simplification in chapter 4, where we discuss the technical impacts of the degradation model on relation database systems.

Transformations triggered by any type of event

The data degradation model itself does not put a restriction on the type of events which trigger a transformation function and a transition between levels of precision. Hence, an event can be anything; apart from events such as 'the end of the retention period', it can also be 'the purpose of using the information has been fulfilled'. The latter cannot, in general, be expressed in the *age* of data.

Different events can trigger different transformation functions on the same data item. This makes that the life-cycle of data item is not fixed, as it is in the simplified version of the model, where transitions are only triggered by time. Different paths from *full* to *null* are possible, which is pictured in figure 3.7.

For two arbitrary levels, we require that there also exists a level which represents the so-called *meet* ($\sqcap$) between the two:

$$\alpha \preceq \beta \sqcap \gamma \quad \equiv \quad \alpha \preceq \beta \wedge \alpha \preceq \gamma \tag{3.4}$$

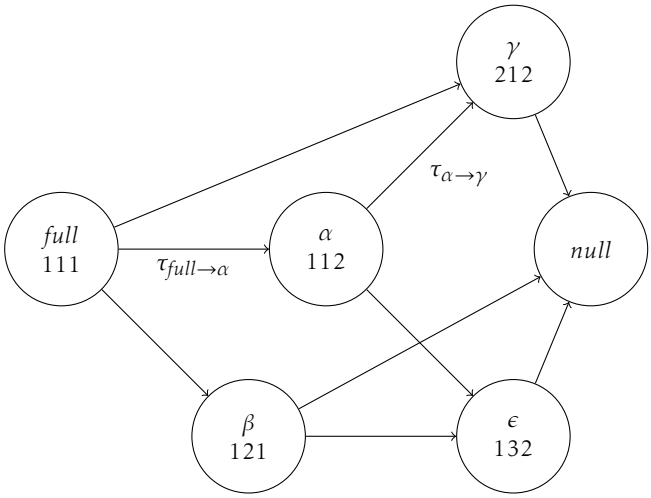


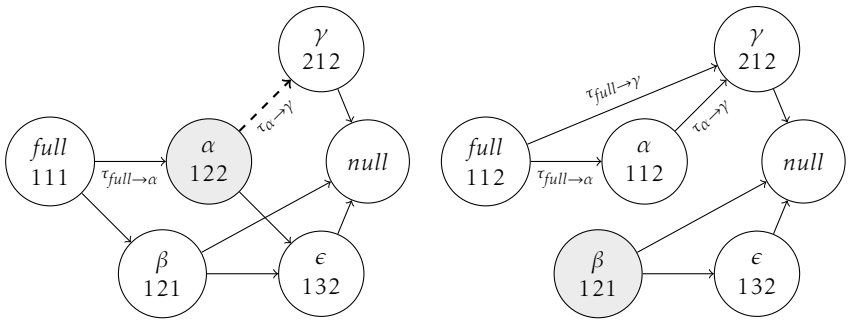
Figure 3.7 Particular example of a valid network of levels without the simplifying assumption that transitions are triggered by time. The edges represent the existence of a transformation function. A level is represented by a set of digits, each digit representing a level of precision of an attribute. Here, 111 denotes a tuple with three attributes all in the most precise state. State 132 means that the first attribute is still precise (1), the second attribute has been transformed to state 3, and the third attribute is in state 2. In this example, $\alpha = \gamma \sqcap \epsilon = 212 \sqcap 132 = 112$.

In our particular implementation, the *meet* of two levels can be calculated by taking for each pair of attributes of two levels, the most precise attribute of the pair. Hence, given two arbitrary levels, we require there is a path leading to both levels from a shared root. Note that this requirement is always fulfilled if we require the existence of a level *full*, from which all other levels can be reached.

The sets of levels and transformation functions as in figure 3.8 are *not* possible. If figure 3.8a would be allowed, it would be possible to *upgrade* the precision of an attribute. such a operation is only possible when the original value is stored, which is against the limited retention principle. The situation in figure 3.8b can occur when the collected, most precise data has a precision from which there is no transformation to some of the levels.

3.3.2 Generalization functions and trees

Transformation functions have the task to degrade data from one level of precision to the other. A particular implementation of transformation functions can be *generalization*, a well-known technique also used in anonymization techniques, as we have seen in section 2.3.2. We shortly discuss



a Invalid, because a transformation function $\tau_{\alpha \to \gamma}$ is not possible, since α does not precede γ. To be a valid *Level*, $\alpha \preceq \gamma \sqcap \epsilon$, which is not the case (requirement 3.4).

b Invalid, because level β is not reachable (requirement 3.2) through a transformation function. A transformation between *full* and β is not possible (requirement 3.4).

Figure 3.8 Two invalid sets of levels and transformation functions, where the level representation of figure 3.7 is used again. Figure a) invalidates the limited retention principle. It would be possible to provide this functionality using access control techniques. Figure b) contains levels which are unreachable, because the most precise data is incompatible with level β.

two generalization techniques; generalization trees—also known as generalization taxonomies—and generalization functions.

For simplicity we choose here to use a *crisp* generalization tree, although techniques for *fuzzy* generalization hierarchies exist and could be applicable to our degradation model [13]. In this context, *crisp* means that each data item can only be generalized in one single way. Moreover, we assume that generalization functions and trees match service provider's requirements. Hence, the levels of precision are meaningful for the service provider in such a way that it can still fulfill some, or all of its purposes on degraded data.

Some transformation functions can degrade a value by applying an operation on that value without requiring any auxiliary information. For example, a zip-code can be easily degraded by removing a digit, such as in figure 3.9. A time stamp can be degraded by removing, for example, the day. Using a function, a time stamp can be degraded to 'the day of the week'. Numerical values, such as salary, can be degraded by splitting the domain in intervals of arbitrary size, or rounding the number to thousands.

When the degradation of a piece of information (typically an attribute) cannot be captured by a generalization function, a *generalization tree* can be used. Given the *domain generalization hierarchy* for an attribute, a generalization tree for that attribute gives, at various levels of precision, the values that the attribute can take during its lifetime. For each domain in the do-

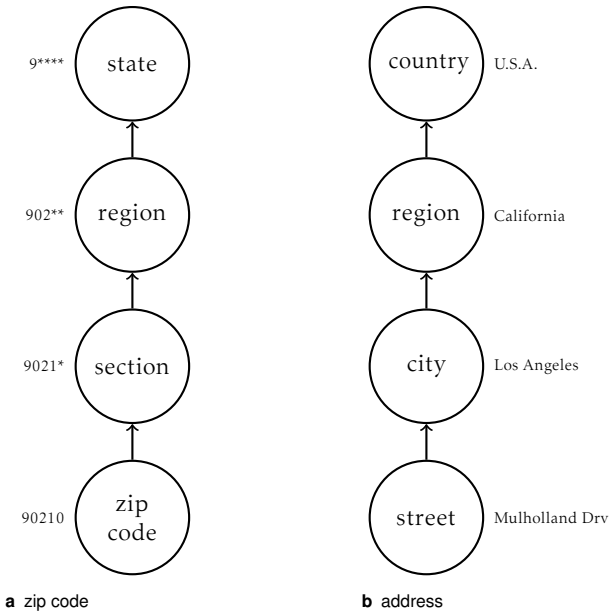


Figure 3.9 Two arbitrary domain generalization hierarchies for *zip-code* and *address*. Values of the first domain can be generalized using a generalization function, which simply removes digits from the code. This is not possible for address; for this domain, a generalization tree as in figure 3.10 can be used.

main generalization hierarchy, the corresponding level in the tree contains all possible values of that domain. Hence, a path from a particular node to the root of the tree expresses all degraded forms the value of that node can take in its domain. An example is given in figure 3.10; here a possible generalization tree for the domain generalization hierarchy of figure 3.9b is shown.

The disadvantage of a generalization tree over a generalization function is that such a generalization tree might not always be easily available. Moreover, for large domains, especially the most precise level of the generalization tree can be large. For example, there are roughly 200 million internet domain names; the number of individual web pages is even much larger. To implement and maintain a generalization tree capturing all those domain names and their categories might not be an easy task, although so-called web directories already exist today, such as Google Directory (containing 1.5 million websites) and Yahoo Directory [106, 126].

The advantage of a generalization tree is that it is flexible; a generalization tree can reflect exactly application needs. Every level in the tree

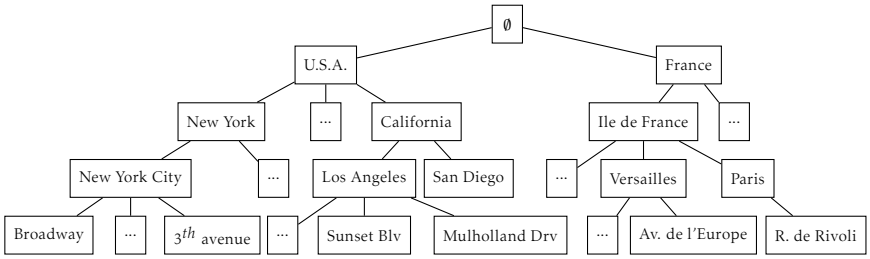


Figure 3.10 Example of a generalization tree for the location attribute. The leafs of the tree denote the most precise values (addresses).

can be defined such that the precision makes sense for the application. For example, if a service wants to provide suggestions based on a city, it does not make sense to degrade a location to 'road type'.
For some domains, a hybrid solution using both generalization functions and trees can be used. For example, an address containing a zip-code can first be generalized using a function which removes digit(s) to generalize the location information. For the least precise location, or for specific applications requiring a different generalization scheme than already contained in the zip-code hierarchy itself, a generalization tree can be used.

3.4 Conclusion

In this chapter, we presented a new framework to be able to reason about limited retention, and more specifically data degradation. We showed that using a function capturing the *worth* of storing data for the service provider, and a function for the *risk* of storing this data, there exists an optimal retention period such that the common interest for both service provider and user is maximized.

The increase of common interest which can be achieved depends on multiple factors. When the decrease of worth is much higher than the increase in privacy, it is good to apply data degradation, even if this means that the data will never be stored in a precise form. In our analysis, we showed cases where it is useful to progressively degrade the data from precise states to generalized states until final destruction of the data. Still, there are open questions, which fall beyond the scope of this thesis, but have to be considered in future work.

- Firstly, as mentioned earlier in section 2.3.3, it is hard to measure the actual privacy guarantee of data degradation and its effect on the risk functions. How much 'privacy' can be provided by generalizing the data is an important question in order to correctly define the

risk functions. Correctly defining privacy will always be subject of discussion, mainly because of its subjective nature. In that light, defining privacy as a function based on risk, and to relate this risk to the amount of stored data is a promising first step.

- We introduced a model in which we only take retention periods of data into account. Indeed, both privacy and worth functions can depend on various parameters other than retention periods. *Users* can choose different 'risk profiles' depending on the nature of the data items, and *service providers* can attach a lower worth to specific data based on the user, location, time of the day, *et cetera*.
- For practical reasons, an important question is *if* and *how* service providers will be able to express their worth functions. To put our framework into practice, it is necessary to provide tools which enable service providers to give transparency about their need to collect personal data.

Our framework shows it is indeed possible to reason about retention periods such that not only one but both parties will be satisfied. However, the resulting life-cycle policies have to be enforced such that data indeed is subject to data degradation, which is a task for the service provider. This leads to the important question if it is *feasible* to implement data degradation. In the following two chapters, we will discuss the *technical* problems relating to implementing life-cycle policies in a relational database system.

Technical implications of data degradation

4

In section 3.2 we introduced the concept of data degradation. We omitted most details about the context in which such a model should run. In this chapter we will place our model explicitly in the context of a traditional database management systems; we introduce concepts and definitions that particularly apply to relational data management. In this context, the data items will be attribute values contained in tuples, which are stored in relational tables. When we refer to traditional database systems, we refer to mainstream database systems which do not implement data degradation.

Traditional database systems are developed to efficiently and durably insert data, make updates to this data and to query them. The ACID properties ensure the correct execution of queries and updates keep the database in a consistent state. Although the ACID properties—which stands for *atomicity*, *consistency*, *isolation* and *durability*—ensure that the database will *internally* be in a correct state, operations on the data can be performed beyond the control of the DBMS, and then violate the ACID properties. For example, it is possible to corrupt the data files or destroy transaction logs, violating the durability property.

The consequences of such a violation will be highly dependent on the application of the DBMS; in our context of data degradation, the DBMS will be responsible to make data degradation irreversible. Thus, to apply the degradation model, we have to ensure that even operations performed outside the control of the DBMS cannot reverse the degradation steps; a delete statement should not only be made visible on the application level, but also externally irreversible. In current database system implementations this is not the case, as pointed out by Miklau et al. [71]. They showed that although data has been deleted from a *logical* point of view, for example by setting a deletion flag, the deleted data can still be recovered from the file system.

Hence, an important question will be how and to which extent the kernel of a traditional DBMS has to be adapted to get our degradation model

running. The question to be answered in this and the following chapter will be how *feasible* it is to implement the data degradation model, possibly on top of an existing DBMS, and to analyze the impact of data degradation on the performance of the DBMS. In this chapter we propose various strategies to efficiently store and query degradable data. We will describe those strategies in detail in the following sections. Those strategies have been implemented in a prototype. An extensive analysis of the performance will be presented in chapter 5.

4.1 The degradation model for relational data

In section 3.2 we introduced *life-cycles* as a sequence of levels of precision in decreasing order of precision, and the retention period for each level. In this section we give more details about what this means for tuples containing individual attributes which have to be degraded. What we previously called a *data item* will now be referred to as an *attribute value*; hence, attributes are the entities which will be degraded and undergo transformations to new levels of precision.

4.1.1 Simplifications

Although the model presented in chapter 3 does not put the following simplification, we introduce them here to simplify the discussion on the technical feasibility of data degradation. The techniques presented further on in this chapter benefit from the simplicities. We will, where applicable, indicate to which extent the techniques rely on the simplifications introduced here.

TIME TRIGGERED Transformation functions are triggered by time, as proposed in section 3.2. An alternative would be that transformation functions are triggered by any type of events, which will be discussed in more detail in chapter 6.

UNIFORMITY Life-cycles apply to all tuples of the same table uniformly, rather than being user dependent. Hence, for each type of attribute there can be only one life-cycle.

4.1.2 The life-cycle of an attribute

A life-cycle for an attribute is a sequence of *attribute states*; an attribute state represents the level of precision of that attribute. The life-cycle consists of transformation functions triggered after pre-defined retention periods. More precisely:

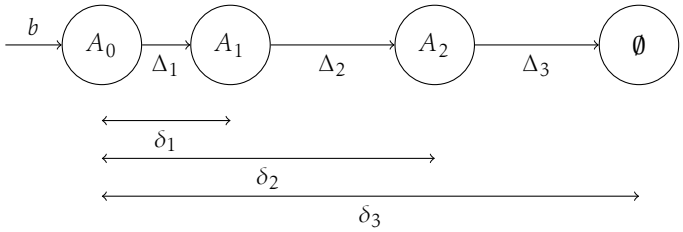


Figure 4.1 Example of the life-cycle of an attribute with 4 attribute states. Duration Δ_1 is the duration of state A_0, and should be read as "after being Δ_1 in state A_0, attribute A will be degraded to state A_1".

- In the initial state of a degradable attribute A, denoted by A_0, the attribute has its original value as acquired at its *time of birth* (insertion time).
- A transformation τ_i from A_{i-1} to A_i takes place after a retention period δ_i since time of birth b, with $\delta_0 = 0$, meaning that an attribute immediately enters its first state after insertion. Note: τ_i is a shorthand for $\tau_{A_{i-1} \rightarrow A_i}$.
- The final state of an attribute's life-cycle containing n states, A_{n-1}, will be referred to with the symbol $\emptyset$, indicating the final destruction of that attribute.
- The period between τ_i and τ_{i-1}, denoted by Δ_i is named the *attribute state duration*, with $\Delta_0 = 0$. Hence, $\delta_i = \sum_{j=0}^{i} \Delta_j$. Note that Δ_i is the duration of the state *preceding* attribute state i.

A life-cycle is defined per degradable attribute. An example of a life-cycle is given in figure 4.1. In this figure, the life-cycle consists of 3 attribute states before the attribute is finally destroyed. The total *life time* of the attribute is therefore δ_3 (assuming the absence of explicit delete statements by users). If there doesn't exist a life-cycle for an attribute, this attribute is considered as a *stable* attribute.

4.1.3 Tuples

Tuples, as in traditional relational database, consist of a set of attributes $\{A, B, C, \ldots\}$. In our model, a tuple is the composition of *degradable* attributes and *stable* attributes. Stable attributes do not participate in the degradation process. Each degradable attribute however has its own life-cycle; when a transformation takes place in one of its degradable attributes, the tuple as a whole enters a new state. We name such a state a *tuple state*. More precisely:

- The initial state of a tuple t, denoted by t_0, is defined as the state in which all degradable attributes are in their initial state 0.

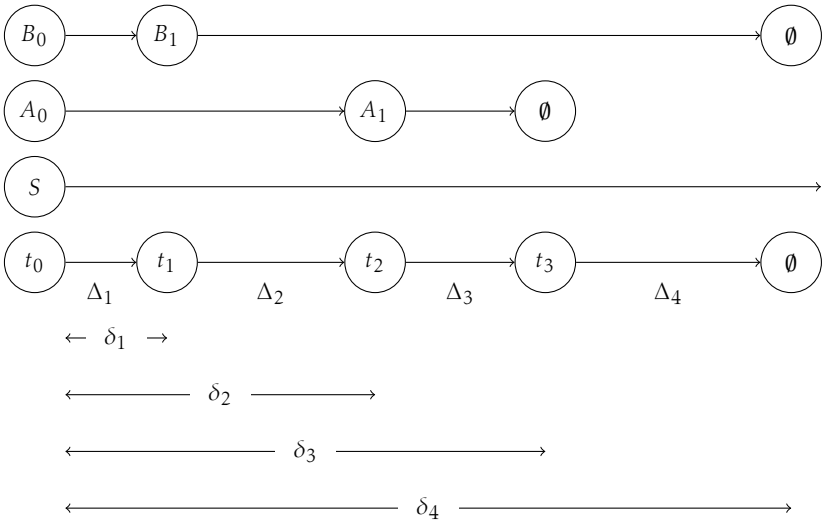


Figure 4.2 Representation of the life-cycle of a tuple with stable attribute S and degradable attributes A and B. The initial state is t_0. After Δ_0, attribute B will undergo a transformation to attribute state B_1, such that the new tuple state will be t_1. The life-cycle ends when both attributes are in their final state.

- The tuple state t_i is the composition of attribute states; given the attribute states $A_a, B_b, C_c, \ldots$, the tuple state number i is defined as $i = a + b + c$. Hence, an attribute transition from A_a to A_{a+1} results in a tuple state transition from t_i to t_{i+1}.
- The duration of a tuple state before the transformation to t_i takes place is denoted with Δ_i. The retention period δ_i at which the transition from t_{i-1} to t_i takes places is determined as follows: if $A_a, B_b, C_c, \ldots$ are the current attribute states for the attributes of tuple t, with retention periods $\delta_a, \delta_b, \delta_c, \ldots$, then $\delta_i = min\{\delta_{a+1}, \delta_{b+1}, \delta_{c+1}, \ldots\}$. Now, Δ_i is defined as $\delta_i - \delta_{i-1}$.
- When the retention period of two attribute states A_a and B_b are equal ($\delta_a = \delta_b$), the tuple will first undergo a transition from t_i to t_{i+1}, immediately followed by a transition to t_{i+2} ($\Delta_{i+1} = 0$).

The concept of tuple states and how the life-cycle of a tuple is constructed from its attribute life-cycles is pictured in figure 4.2.

4.1.4 Tuple state sets

During the life-cycle of a tuple, the tuple will be in different states of precision, indicated by its tuple state. We name the set of tuples which, at a

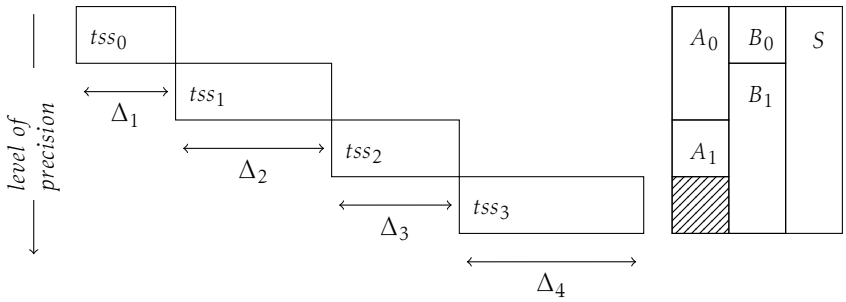


Figure 4.3 Visualization of tuple state sets, and the composition of the tuples belonging to those sets from its different attribute states. The most recently inserted tuples will be contained in tss_0 while the eldest tuples will be contained in tss_3. Note that *attributes* with the same *attribute state* can be member of different tuple state sets.

given point in time, share the same tuple state a *tuple state set*, abbreviated with *tss*. Hence, the database consists of a number of tuple state sets, equal to the number of tuple states. The following properties hold for a tuple state set:

- A tuple state set tss_i contains only, and all tuples which are in tuple state t_i.
- At a tuple's birth, the tuple will be inserted in tss_0.
- A tuple will be removed from a tuple state set tss_i at the age of δ_i, and will become member of the set tss_{i+1}, unless it reaches the end of its life-cycle.
- Assuming a constant insert rate, the *size* of the tuple state sets remain constant, since the tuple state durations are fixed. The size of a tuple state set tss_i is therefore determined by the tuple state duration Δ_i and the insertion rate.

Figure 4.3 visualizes the concept of tuple state sets. Note that the content of the set is determined by the *tuple state*, not by the *attribute state*, and therefore two different tuple state sets can contain tuples with an attribute sharing the same attribute state.

4.1.5 Query semantics

Since each piece of data is no longer stored with full precision, but instead with less precision after each degradation step, the semantics of querying the database need revision. To this end, it seems natural to look for results from fuzzy or probabilistic database theory, because less precise representations can be related to fuzzy or probabilistic representations. However, in

terms of the fuzzy database landscape defined by Buckles and Petry [28], we take the position that

i. the data inserted into to system is precise,

ii. the degraded representations are *less* precise but still *certain*, and

iii. the queries are *crisp*, meaning that results are supposed to be a correct and complete response to the query [28].

Positions (i) and (iii) need no discussion; of course, other positions are possible but will lead to a different research intention and direction. The reason behind (iii) is that we want to change the SQL query syntax (and semantics) only as little as possible. However, although the results are crisp, one can say that the *probability* with which a degraded result value can be traced back to the *originally* inserted value, has been decreased.

To motivate our position (ii) we argue as follows. Fuzzy (uncertain) data is, in terms of the fuzzy database model [70, 86], data to which a 'truth value' in $0 \ldots 1$ is attached, indicating the degree to which the data can be considered *possible*. Since the original value is not fuzzy, the degraded value cannot be considered fuzzy in a meaningful way: all more precise values have a degree 0 or 1 (and nothing in between) of possibility of being 'the' original one. Only the *precision* has decreased.

To provide appropriate query semantics, we have to define to which extent data at different levels of precision can be used to answer queries. We identify two possible usages of degraded data: either a service can have a single purpose, which can use data with several levels of precision, or a service has multiple purposes, each purpose requiring its own level of precision. Moreover, we recall the statement made in section 3.3.2, that in the degradation model, the levels of precision match predefined service requirements, captured by the levels in the generalization trees. Thus, services know the levels of precisions and forms a data item can take, as expressed by the generalization tree. Hence, a service with multiple purposes, each requiring its own level of precision, wants its predicates being evaluated only on the data with at least the precision they need to answer a specific query. This leads to the following query semantics for, specifically, the selection and projection.

For the sake of simplicity, we consider queries expressed over a single relational table R. Due to degradation, the dataset R is divided into sets of tuple states tss_i of tuples within the same tuple state t_i, having a strong impact on the selection and projection operators of queries. These operators have to take precision into account, and have to return a coherent and well-defined result. To achieve this objective, data subject to a predicate P expressed on a demanded precision level i, will be degraded before evaluating P, using a transformation function $\tau_{j \rightarrow i}$. Given τ, P and i, we

define the select and project operators $\sigma_{P,i}$ and $\pi_{*,i}$ as:

$$\sigma_{P,i} = \sigma_P\left(\bigcup_{j=0}^{i} \tau^*_{j\to i}\left(tss_j\right)\right) \qquad \pi_{*,i} = \pi_*\left(\bigcup_{j=0}^{i} \tau^*_{j\to i}\left(tss_j\right)\right)$$

Here, $\tau^*_{j\to i}(tss_i)$ has the following meaning: transformation function $\tau_{j\to i}$ will be applied on every tuple in tuple state set tss_i, so that every tuple will be represented by a degraded version. This does not mean that the tuple will be *physically* degraded.

The level of precision i is chosen such that it reflects the declared purpose for querying the data. Then, queries can be expressed with no change on the SQL syntax in the example below.

Example 2. Consider the table *Person* with two degradable attributes, *Location* and *Salary*. The purpose of a certain application is to gather some statistics about the current salary distribution over countries, for which it needs the location with precision *Country*, and the *Salary* in intervals of size 1000. It declares its purpose *Stat* as:

```
DECLARE PURPOSE Stat
SET PRECISION LEVEL Country for location, Range1000 for salary
```

Now the application can query the database using regular SQL statements:

```
SELECT * FROM Person
WHERE location like 'France' and salary = '2000-3000'
```

The predicate of this query will be evaluated on all data with a higher or equal precision as stated in the purpose *Stat*, hence it will contain all tuples from the *Person* table for which both attributes *Location* and *Salary* have at least the precision *Country* and *Range1000*. All other tuples are discarded. Before evaluating the predicate and the projection, the tuples are first degraded up to the requested level of precision (thanks to the transition functions) if required.

The semantics of update queries is as follows. First, the delete query semantics is unchanged compared to a traditional database system, except for the selection predicates which are evaluated as explained above. The delete semantics is similar to the deletion through SQL views. When a tuple must be deleted, both stable and degradable attributes will be deleted.

Second, insertions of new elements are granted only in the most accurate state. Finally, we make the simplification that updates of degradable attributes are not granted after the tuple creation has been committed. On the other hand, updates of stable attributes are managed as in a traditional database system. Hence, the semantics of update queries is as follows:

Delete The semantics remains unchanged compared to a traditional database system, except that the where statement, if present, is evaluated over the chosen subset based on the purpose specification rather than over the complete table (as with queries). Thus, the delete semantics is similar to the deletion through SQL views. When a tuple is deleted, both stable and degradable attributes are deleted.

Insert We adopt the simplification that insertions of new elements are granted only in tss_0.

Update We adopt the simplification that updates of degradable attributes are not granted after the tuple creation has been committed in tss_0. On the other hand, updates of stable attributes are managed as in a traditional database systems.

Updating degradable data is not considered, but this is only to simplify the discussion on transaction semantics and recovery procedures. In section 4.4.1 we will briefly discuss how updates on degradable data can be granted. Moreover, we will discuss alternative query semantics in general.

4.1.6 Consequences of data degradation for ACID-properties

In traditional database systems, the well-known ACID-properties make sure that transactions are processed in a well-defined and correct way. The properties are (in short):

A *Atomicity*. Transactions are either fully executed or not executed at all.

C *Consistency*. After (and before) a transaction ends, the database will be in a consistent state.

I *Isolation*. A transaction runs in isolation, meaning that transactions cannot see or use the effect of other transactions.

D *Durability*. Once a transaction has completed successfully, the effect of the transaction will be permanent.

Without data degradation, a transaction inserting a data item will be processed in the following way: once the insert transaction has been fully completed, without intervention of other transactions, the database system checks data consistency, and makes sure that the data can be recovered after a system failure. *With* data degradation, however, we can make the following observations regarding the ACID-properties:

ad **A** An insert statement for a tuple containing degradable attributes triggers a degradation process, which has to make sure that the lifecycle of the attribute is respected. This means that a transaction will cause multiple follow-up transactions which we name *degradation side-transactions*, all having the task to insert the tuple into (and remove from) its corresponding *tuple state set* at the correct time. Hence,

when a data item has been inserted successfully, the database system has to make sure all degradation side-transactions will be fully executed too.

ad **C** After an insert statement has been checked on consistency, e.g., the insert does not violate any integrity constraints, the inserted item will undergo updates triggered by the degradation transactions. Those updates are not allowed to violate any integrity constraints, since the above atomicity property requires that the degradation transaction will be fully executed.

ad **I** Degradation side-transactions have to be protected against deletes of conflicting, uncommitted regular user transactions.

ad **D** Although an insert statement should be durable in any case, the durability requirement ends at the time a degradation transaction takes place. Moreover, it should be *impossible* for the database system to redo an insert statement, and undo each degradation transactions from the moment the next degradation transaction takes place. More precisely: a tuple t stored in a tuple state set tss_i is stored durably only for the duration Δ_{i+1} of the corresponding tuple state. After δ_{i+1}, the content of t as it was in tss_i should be unrecoverable.

Above observations lead to revisions of the atomicity and durability property. First we give a definition of transactions in the context of data degradation for inserting a tuple with n tuple states. Instead of a single transaction T, we split T into a set containing a *main* transaction and side-transactions, $[T_0, T_1, T_2, \ldots, T_n, T_f]$, such that:

- T_0 is the *main* transaction inserting a tuple t into tuple state set tss_0. When T_0 finishes with a commit, the result of the transaction will be visible.
- For each i $(0 < i \leq n)$, T_i is a degradation side-transaction. The degradation side-transaction T_i has the following result: if any attribute A of a tuple t, with attribute state A_a, has a retention period $\delta_a = \delta_i$, then transformation function τ_a has been applied on attribute A. Hence, degradation side-transaction T_i on a tuple member of tuple state set tss_i results in a transition of t from tss_i to tss_{i+1}.
- T_f (with $f = n + 1$) is the *destruction* side-transaction, removing t from the last tuple state set tss_n.

If a tuple contains only stable attributes, there will be no degradation and destruction side-transactions, and the traditional ACID-properties will simply apply to T_0. For tuples which do contain degradable attributes, we define two extensions to the ACID properties, named Δ-*atomicity* and Δ-*durability*:

Definition 4.1.1. Δ-*atomicity*. A main transaction T_0 inserting a tuple t with n tuple states will be followed by a set of degradation side-transactions

T_i, $0 < i \leq n$, and a final destruction transaction. Any degradation side-transaction T_i must be run at time $b + \delta_i$, cannot be interrupted and has to be fully executed. To a main transaction T_0 the traditional atomicity property applies, and it can therefore be interrupted with a rollback, in which case also the side-transactions will be withdrawn.

Definition 4.1.2. Δ-*durability*. A transaction T_i working on a tuple t is *only* durable and thus recoverable during the period between T_i and T_{i+1}. This period is equal to Δ_{i+1}. After δ_i, T_i should be explicitly *not* recoverable anymore.

4.1.7 ρ-timeliness

To enforce Δ-*durability*, a degradation side-transaction T_i on tuple t has to start exactly at time δ_i after the birth of t, or to speak in database terms, after the commit of the insert transaction T_0. Respecting this time delay strictly to the second would incur severe performance penalties, for which we cannot foresee a benefit in practice. Therefore we propose a weaker property which we name ρ-timeliness:

Definition 4.1.3. ρ-*timeliness* A degradation side-transaction T_i on a tuple t in tuple state set tss_i, originally inserted into tss_0 at time b, must run (and finish) within the time window $[(b + \delta_i) - \frac{1}{2}\rho, (b + \delta_i) + \frac{1}{2}\rho]$.

We can also choose time window $[(b+\delta_i), (b+\delta_i)+\rho]$ or $[(b+\delta_i)-\rho, (b+\delta_i)]$. We believe that this choice has no foreseeable influence on the main benefits of ρ-timeliness. Moreover, although we believe that the value of ρ should be application dependent, our intuition suggests a direct relationship between the retention limit Δ and ρ; the smaller Δ, the shorter ρ is supposed to be.

The sole purpose of the ρ-timeliness property is performance, and it enables a feasible technical implementation of our degradation model. In the following sections we will discuss the technical challenges for implementing our model, and indicate where we have to redesign core database system techniques.

4.2 Technical challenges

Since we want data degradation to take place within a relational database system, the first and legitimate question which comes to mind is how complex will the technology be to support it. Identifying the impact of making a database system data-degradation aware leads to several important questions.

4.2.1 How to enforce Δ-atomicity and Δ-durability?

As stated in definition 4.1.2, a transaction should only be recoverable during a limited period. However, as pointed out in [71], traditional database systems cannot guarantee the non-recoverability of deleted data due to different forms of unintended retention in the data space, the indexes and the logs.

Two existing candidate techniques [71] tackle the recoverability issues and have to be properly adapted to the context of data degradation. The first one is physically overwriting the data with its degraded value, and a dummy value when the tuple reaches its final state. The second option is to pre-compute all values, encrypt them, and destroy the encryption key when the value should not be recoverable anymore, making the value undecryptable, assuming that it is computationally not feasible to break the used encryption method.

These two techniques exhibit opposite properties in terms of degradation and access efficiency. Overwriting tuples is highly I/O inefficient, therefore the number of I/O's should be limited as much as possible. The problem is particularly acute considering that each data item inserted in the database undergoes multiple degradation steps. And although removing encryption keys might be very efficient, encrypting data is highly inefficient when it comes to accessing this data to answer queries. In this light, the storage structure for degradable attributes, indexes and logs have to be revisited.

4.2.2 How to speed up queries involving degradable attributes?

Database systems have been designed to speed up queries. Some workloads induce the need of few indexes on the most selective attributes to get the best trade-off between selection performance and insertion/update/deletion cost. For other application, insertions are done off-line, queries are complex and the data set is very large. This leads to multiple indexes to speed up even low selectivity queries. Data degradation can be useful in both contexts. However, data degradation changes the workload characteristics in the sense that queries become less selective when applied to degradable attributes. This introduces the need for indexing techniques supporting efficient degradation.

4.2.3 How to guarantee ρ-timeliness?

The ρ-timeliness relaxation (see definition 4.1.3) has been introduced for the sole purpose of performance. Nevertheless it still requires that data should be degraded on time. However, implementing degradation side-transactions in a traditional DBMS by means of normal transaction management may lead to conflicts, with deadline misses and deadlocks as result.

Still, degradation side-transactions need some sort of isolation control; hence, we need to redesign a synchronization protocol for degradation (and destruction) side-transactions.

4.3 Impact of data degradation on core database system techniques

Now we have seen the challenges brought forward by data degradation, we will discuss technical solutions for it. Some of the techniques discussed here have been implemented into a functional prototype, which will be discussed in section 5.2. With this prototype, a performance study has been carried out. The results of this study will also be presented in chapter 5.

4.3.1 Storage structure

A storage structure is a representation of how data is stored and organized on an hard disk. We consider here only disk based storage and keep any main memory based database system out of consideration.

A storage structure defines how rows of a table are organized in files [64]. The choice of storage structure for a particular application is based on finding the best balance between query and insert efficiency. There are many types of storage structures in use in traditional database systems, such as the *heap* structure, the *sequential* (or sorted) structure, the *hash* structure and the *B+tree* structure [43]. Our degradation model changes the characteristics of the traditional workload of a database system, and thus also the requirements for the storage structure. Apart from the heap structure—which we will discuss later—most storage structures are degradation unfriendly.

The most basic, but also most insert efficient storage structure to insert data, is an *heap* file, in which data is inserted in the end of a file, in chronological order. However, this leads to poor query performance when queries are selective; without an additional secondary index a sequential scan on the whole structure is required. To speed up queries, records can be kept ordered on the primary key of the relation. A *sequential file* is such a structure. Unless this primary key is an auto-increasing record number, this is not the insert order, and insertion costs will therefore be higher because of the maintenance of the ordering in the file.

Traditionally, one of the most commonly used structures to store data is the B+tree structure [64]. As a result, lookups of any particular tuple selected on its primary key takes $O(log(n))$ time, and range queries on the primary key can be answered efficiently. However, inserting any *individual* tuple is costly and, which is more problematic in the context of data de-

gradation, also updating an individual tuple takes $O(log(n))$ time. Hence, for such storage structures the cost of data degradation will be very high.

In the following we propose a new storage structure based on the heap structure, in which, as we will we see, a set of degraded data can be degraded in $O(log(1))$ time. Our objective is to find the best variant of this storage structure which takes beside inserts and queries, also data degradation into account. Along the way we take the following properties and limitations into account:

- Generating a generalized value comes for free. We only consider costs of *replacing* (or removing) a more accurate value for a generalized value.
- Some memory buffers or caches are available, but in the end we need hard disks to store the data.
- We consider two types of I/O operations:
  - random I/O: the disk arm has to move to find a particular page on disk to read or write it (typically 10 ms per page).
  - sequential I/O: the page can be found physically next to the previous page on disk and be written or read from there (typically 0.05 ms per page).
- To read or update a tuple, always the full page in which a tuple is contained will be read and/or written. Hence, I/O operations are performed on full pages.

A degradation friendly storage structure

A key factor in our investigations of a degradation friendly storage structure will be the ρ-timeliness property; this property allows us to be flexible in the time when a tuple has to be degraded, making it possible to *group* those tuples together. This way we are able to limit the amount of I/O operations drastically, because one I/O operation can be *shared* by multiple updates on tuples. To enable this feature, we have to order the data files on insertion time because then tuples which can be degraded together will also be physically stored close together. Hence, we are looking for a storage structure based on the traditional *heap* structure, with the main difference that we will *enforce* the ordering on insert time.

Note that the ordering on degradation time only holds because of the simplifications we made (see 4.1.1), stating that degradation is only triggered by time (TIME TRIGGERED) and that all retention periods are the same for each tuple (UNIFORMITY). Hence, regardless which derivative of our storage structure we choose, all tuples in the data files we use will be sorted on insert time. Some derivatives will consist of multiple files, each containing degradable data.

To each data file we assign at least one *insert buffer*, and, depending on the storage strategy, a *degradation* buffer. Those buffers are stored in RAM and make it possible to flush several tuples together to disk, and share the I/O's required to insert and degrade the data. Degradation of x tuples from a data set containing n tuples can therefore in theory be performed in $O(log(1))$ time, compared to the $O(x\ log\ n)$ time needed when using a storage structure in which the data is not ordered on insertion time. See figure 4.4 for a graphical representation.

A *degradation schedule* is attached to each data file and manages at what time which data has to be degraded. This can be implemented by maintaining a table in which pointers to the pages to be degraded are stored with the corresponding degradation time. Since tuples are grouped by intervals of size ρ, time can be measured in units of ρ, so that $t = n$ gives the time of the nth degradation step since startup time of the database system. The schedule itself can be stored in RAM; the size of the schedule is bounded by the tuple state duration and ρ. This is also pictured in figure 4.4.

As an optimization, we can choose to degrade only *full* pages, meaning that tuples will be degraded which, given the insertion time and the tolerance ρ should not be degraded yet. Although this will not harm privacy, it harms data usability since effectively the retention period of tuples will be shortened. However, the advantage in performance is that pages never have to be written more than once to disk.

Baseline approach

In a traditional database system, a tuple will be stored as a single entity: all attributes—both stable and degradable attributes—will be stored together in one single block on disk. Degrading a tuple can be enforced irreversibly by fetching the block containing the right tuple, and updating the tuple by overwriting the relevant attribute value(s) with the new value(s).

If tuples are large, less tuples will fit in such a block. Hence, for larger tuples, more blocks need to be read for the same amount of tuples. This is particularly important to know in the context of data degradation, in which many tuples need to be read, updated, and written back to disk.

For this reason we foresee that the performance of data degradation using this strategy will be poor, especially when the number of attributes is large. This effect will be amplified when a high number of degradation steps have to be performed; e.g., the number of tuple state sets is large. This approach, in which all attributes of a tuple are stored together will in the following be referred to as the *baseline approach*. To give an indication of how data will be organized internally we give an example in figure 4.5, where degradable attribute A stands for zip-code, degradable attribute B for a time-stamp, and *id* is a stable attribute. Every tuple state is represented by its own table, and each tuple state set is stored in its own data file (see

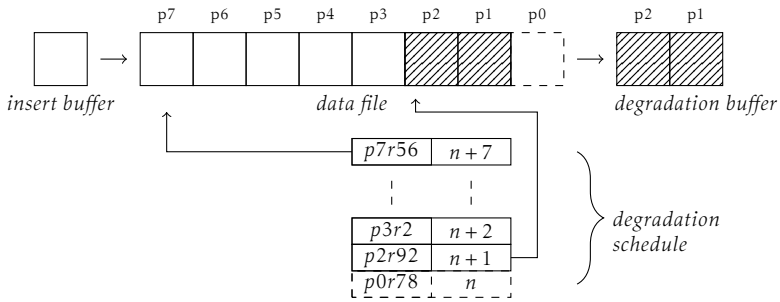


Figure 4.4 Degradation schedule and a heap file structure. A p stands for page number, r for record offset within the page. The schedule contains pointers to the youngest tuple which has to be degraded at a certain time, calculated as number of intervals of size ρ passed since database system startup time. The blocks represent full pages, the shaded area represents tuples which can be degraded together. The page with offset $p0$ has already been removed during the previous degradation step n. For degradation step $n+1$, pages $p1$ and $p2$ will be transferred to the buffer. Offsets in the buffer correspond to addresses of the pages when they were still on the disk. As an optimization choice (and for simplifying this figure), full pages can be degraded, even if the page contains tuples which should not be degraded yet.

figure 4.6).

In the following we consider a number of variations of the storage structure described above, which basically try to overcome the discussed disadvantages. The best storage structure will be that in which the amount of I/O needed to degrade data is minimal, while also keeping the amount of I/O needed for querying as low as possible. Basically, we will consider whether or not we should *pre-compute* and store all attribute and/or tuple states, and whether or not we should *fragment* or *cluster* the attributes over multiple data files. This leads to four combinations and storage models which we describe in more detail in the following sub-sections. The two dimensions are:

Clustered vs Fragmented *Clustered* means that all attributes of a tuple state are stored together in one data file (like in most traditional database systems). while *fragmented* means that all degradable attributes are vertically fragmented into separate data files. Some database systems, such as MonetDB, already use such a form of fragmentation [23].

Eager vs Lazy With an *eager* strategy, all attribute states are pre-computed and all generalized values an attribute will take are stored at insert time. Hence, the transformation functions are applied at *insertion time*. With a *lazy* strategy the new form of an attribute is computed at

$\underline{S}$	A_0	B_0
HJW	7522AH	09-06-29 14:19
GH	7543WR	09-06-29 14:18
ETH	7422NK	09-06-29 13:50
HVH	8325GG	09-06-29 14:19
⋮	⋮	⋮

$\underline{S}$	A_0	B_1
THH	8325AT	09-06-29
HJW	7522AH	09-06-28
BSI	1983AK	09-06-28
⋮	⋮	⋮

tss_0 tss_1

$R(\underline{S}, A, B)$

tss_2 tss_3

$\underline{S}$	A_1	B_1
GH	1050	09-06-26
⋮	⋮	⋮

$\underline{S}$	$A_\emptyset$	B_1
HvH	*null*	09-06-23
⋮	⋮	⋮

a $t = t_0$

$\underline{S}$	A_0	B_1
GH	7543WR	09-06-29
ETH	7422NK	09-06-29
HVH	8325GG	09-06-29
THH	8325AT	09-06-29
⋮	⋮	⋮

$\underline{S}$	A_0	B_0
HJW	7522AH	09-06-29 14:19
⋮	⋮	⋮

tss_0 tss_1

$R(\underline{S}, A, B)$

tss_2 tss_3

$\underline{S}$	A_1	B_1
HJW	7522	09-06-28
BSI	1983	09-06-28
⋮	⋮	⋮

$\underline{S}$	$A_\emptyset$	B_1
GH	*null*	09-06-26
HvH	*null*	09-06-23
⋮	⋮	⋮

b $t = t_1 > t_0$

Figure 4.5 A snapshot of a database containing tuples organized using the baseline approach taken at time t_0 and at time $t_1 > t_0$. Tuples in higher tuple state sets will contain less accurate versions of the degradable attributes. The tuple belonging to identifier HvH which was in tss_3 at time t_0 has been fully removed from the database at a time $t_0 < t < t_1$, and is not present anymore at time t_1.

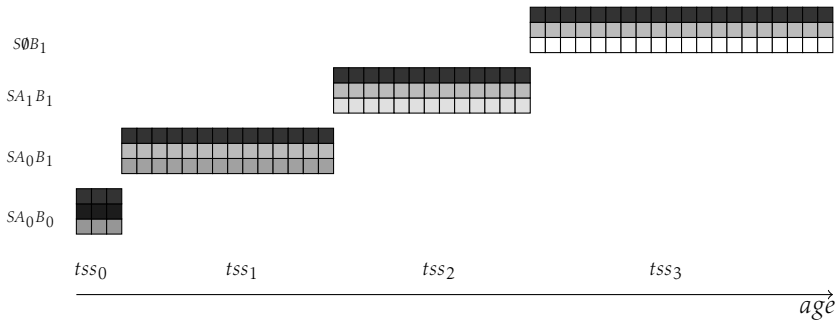


Figure 4.6 An abstract representation of the baseline storage structure. All tuples belonging to the same *tss* are stored in one single data file. Square blocks represent a bunch of attributes. The fading colors represent data degradation, where the black blocks are stable attributes, and white blocks are empty (deleted) spaces. This same representation will be used to represent the other storage strategies.

degradation time.

For each storage model, we will give the advantages and disadvantages of the chosen strategy combination.

Clustered eager

This strategy is schematically represented in figure 4.7a at page 76, in a similar way as figure 4.6. For the explanation of this and the following strategies, we will use the life-cycle as shown in figure 4.2, and add to that one stable attribute (as used in the baseline-approach example). A tuple with its original schema $\{A, B\}$ will be expanded into a set of tuples with schema's $\{A_0, B_0\}, \{A_0, B_1\}, \{A_1, B_1\}, \{B_1\}$. Each tuple will be inserted into its own dedicated data file, which will remain sorted on insertion time. Hence, for each tuple state there will be one data file.

The main advantage of this approach is that degradation can be implemented by means of *only* delete operations; it is not needed to read the attribute to delete it, and therefore no read I/O operations have to take place, reducing overall I/O cost. Because the files are ordered on insert time and therefore degradation time, the tuples to be deleted can be found at the end of the file, making it possible to efficiently manage the degradation process using a degradation schedule. Moreover, all attributes of a tuple are stored within the same data file, so reconstruction of tuples to produce query results is not needed.

The main disadvantage of this approach is that the data is stored redundantly, requiring more storage capacity and additional I/O costs at insertion time. For example, a copy of attribute A in attribute state A_0 is stored in

two data files, B_1 is even stored in three data files. The severeness of this disadvantage depends on the amount of degradable attributes and states per attribute.

Fragmented eager

In this strategy, for each *attribute state* there exists a data file in a similar fashion as the clustered strategy. To compare: in the clustered strategy there is a data file for each *tuple* state. With the fragmented strategy, a tuple with schema $\{A, B\}$ will be fragmented into a set of tuples consisting of only one attribute: $\{A_0\}, \{A_1\}, \{B_0\}, \{B_1\}$. This is pictured in figure 4.7b.

The main advantage of this strategy is that, like the clustered eager strategy, degradation can be performed without read operations. Moreover, unlike the clustered strategy, each attribute state is only stored once, reducing redundancy. This means that when data is deleted from a file, only data is touched which indeed had to be removed at that point, reducing the amount of I/O needed and fully benefiting from the I/O operation.

The disadvantage of this strategy is that, if n is the number of attributes, a *tuple state set* is in this strategy a join between n data files, unless one or more of the attributes are in their final state and thus removed; then a smaller number of data files need to be joined. To reconstruct a tuple in a particular tuple state set, additional I/O operations are needed, making such a strategy less efficient for querying. However, for conditional queries with a low selectivity, the query executor can benefit from this fragmented storage structure. Only the data files with attributes which participate in the select predicate have to be scanned. Since those data files are smaller in size—they contain less attributes—the sequential scan requires less I/Os. For this purpose, database systems using fragmentation exist, such as, among others, MonetDB [23].

Clustered lazy

As described earlier in this section, with the lazy strategies the generalized values of the attributes are computed at degradation time, and not pre-computed as with the eager strategies. For the clustered lazy strategy this means that the original tuple will remain intact, and will be inserted in one single data file representing tuple state set tss_0. At degradation time, a set of tuples which is allowed to degrade at the same time—thanks to the ρ-timeliness property—will be removed from the data file, degraded, and inserted into a new data file representing the next tuple state set in line. See figure 4.7c.

As with the clustered eager strategy, for each tuple state set there exists one data file. However, the difference is that each data file now only contains

tuples which only belong to one tuple state set. Hence, with the lazy strategy there is no redundancy.

At degradation time, pages have to be read from disk, to generalize them, and to insert them into the new data file representing the next tuple state set. This increases the amount of I/O needed for degradation and is therefore a disadvantage of this strategy.

Fragmented lazy

In this strategy, a tuple with schema $\{A, B\}$ will be fragmented into two tuples $\{A_0\}$ and $\{B_0\}$, representing the first attribute states of both attributes, and inserted into their designated data files. There exists a data file for each attribute state; an attribute is inserted in its new data file at degradation time (instead of at insert time with the eager strategy). See figure 4.7d.

With this strategy, as with the clustered lazy strategy, there is no data redundancy. Still, to answer queries, tuples have to be reconstructed since they are fragmented over multiple data files, making querying less efficient. Again, since the strategy is lazy, for each degradation step a read operation is required.

Comparison of storage structures

To give an indication of the strong and weak points of the variations, we present here a simple scenario, and compare the amount of disk pages needed to store a certain amount of tuples, and the I/Os needed to insert, degrade and query the data. A more elaborate performance study will follow in chapter 5.

To simplify the discussion, we consider a constant insert rate of 100 tuples per second, which makes that with a ρ of 10 seconds, 1000 tuples can be written and degraded at once. A single page has space for 100 *attributes*; hence, if a tuple contains 4 attributes, the page can contain 25 tuples. To store 1000 tuples of 4 attributes, we need 40 pages.

In figures 4.6 and 4.7, each block stands for 1000 *attributes*. Hence, in total 50×1000 tuples have been inserted. The total amount of pages needed to store this data basically depends on the amount of redundancy used by the chosen strategy. Two types of redundancy can occur; first, the clustered strategies store multiple copies of an attribute and secondly, the eager strategies store a degraded version of an attribute together with the non-degraded one. Hence, most pages are used by the *clustered eager* strategy. The number of pages needed are listed in figure 4.8.

At first glance, sticking to the baseline technique is the best choice for both inserting and reading data. Indeed, the baseline technique is the common storage structure used in traditional database systems for storing and querying data. However, degradation costs are much higher for this

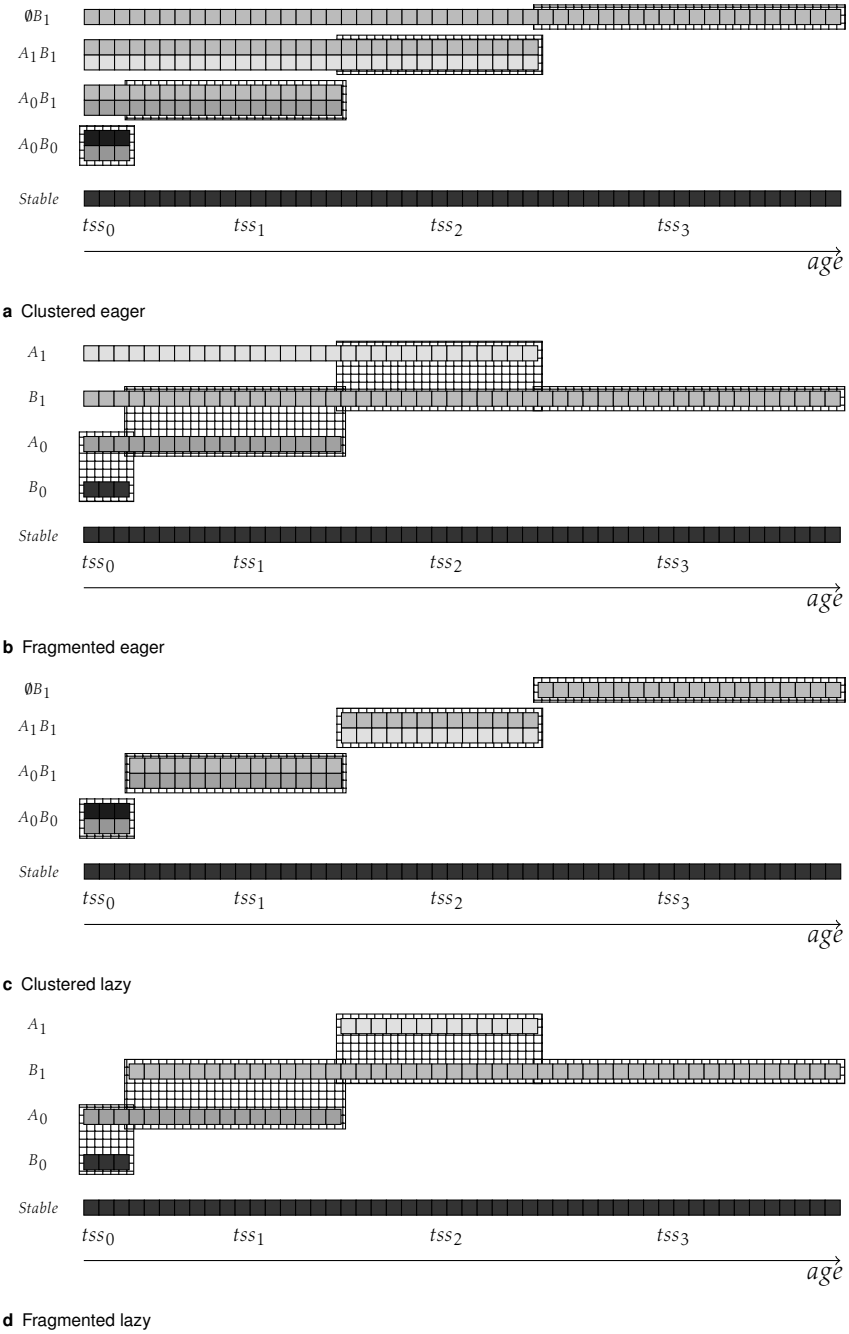


Figure 4.7 Degradation friendly strategies, where all blocks within a box belong to the same tuple state set.

	pages	*insert I/O*	*degradation I/O*	*read I/O*
BL	1650	1 × (1|32)w	3 × (1|32)r + 4 × (1|32)w	1 × (1|32)r
CE	2000	2 × (1|9)w + 2 × (1|19)w	4 × (1|19)w	(1|19)r + (1|9)r
FE	1400	5 × (1|9)w	4 × (1|9)w	3 × (1|9)r
CL	1300	(1|9)w + (1|19)w	3 × (1|19)r + 3 × (1|19)w + (1|9)w	(1|19)r + (1|9)r
FL	1300	3 × (1|9)w	3 × (1|9)r + 4 × (1|9)w	3 × (1|9)r

Figure 4.8 This table shows a cost model based on the examples given in figure 4.6 and 4.7, and shows the amount of pages needed to store 50,000 tuples given the different storage structures, and the number of (random | sequential) I/O operations needed for inserting, degrading and reading 1000 tuples. One block in the figures stands for a set of 1000 attributes. The *baseline* strategy performs best for inserting and reading data, *fragmented eager* for degrading the data. Those numbers are only valid under the assumption that all bunches of sequential I/O can be performed without being interrupted. The abbreviation BL stands for baseline strategy.

strategy. If it comes to degradation, the fragmented eager strategy is most efficient; the cost at insertion time is however much higher.

If we consider that complying with the Δ-durability and the ρ-timeliness property has the highest priority in our context, and therefore the degradation process is the most time critical, we expect that the *fragmented eager* strategy is the best strategy to implement in a degradation supporting database system. As mentioned earlier, the costs for degradation using the baseline strategy is amplified by the amount of stable attributes, something which is not the case for the other strategies, while the read and insert costs will grow proportionally compared to the baseline strategy for all strategies. This makes the choice for a strategy other than the baseline strategy even more viable.

Although very suitable for degradation, the *fragmented eager* strategy might not be the best strategy when query load is high. For applications with a high query load, one might choose a strategy where query costs are lower, but the degradation costs are higher, such as the *cluster eager* strategy. When insert loads are high, *clustered lazy* can be a good option. Finally, *fragmented lazy* can be a good compromise with respect to insert, degradation and read performance. As said before, to validate our findings we will use a prototype to experimentally analyze the performance of the described storage structures. More about this in section 5.2.

4.3.2 Indexing

Indexing degradable attributes leads to two technical problems. First, attributes undergo transitions to multiple levels of precision and therefore their index key will change over time. Moreover, the selectivity of the access path decreases when the attributes are more generalized; an index suitable for the most accurate attribute state might not be the most efficient index

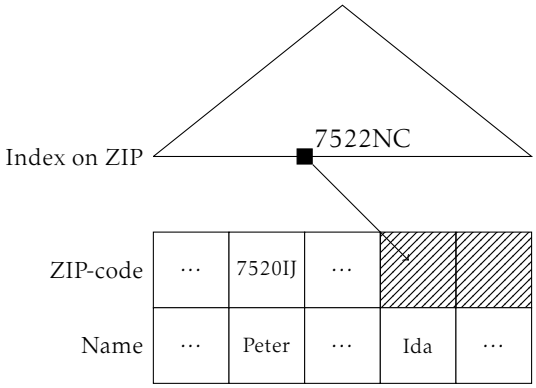


Figure 4.9 Example with two attributes and one index. As long as there is a pointer in the index on zip-code to the data file, Ida's zip-code can be reconstructed.

for higher attribute states. Since predicates on low accurate data—where the same attribute value will occur more frequent than in high accurate data—will result in larger result sets, we say in the following that queries on low accurate data are *low* (or not) selective, and queries on high accurate data are *high* selective.

Second, to comply with the Δ-durability property, data has not only to be removed from the storage structure, but from the indexes too, since those data can reveal information about previous attribute states, or even fully recover them. See figure 4.9 for an example. In traditional database systems it is not common usage to delete keys directly from the index; instead tombstones are used to invalidate a pointer in the index [43]. Hence, again changes to the database system kernel will be required to ensure Δ-durability.

In the following we will consider a number of indexes and investigate the cost to update the index, and how applicable the index is for high or low accurate data. The ρ-timeliness property can be exploited to make removal from the index less costly; however, to do this, data which can be removed together must be stored physically close together, something which is not the case in tree-based indexes. In this section we will give an overview of the various index techniques. An analysis and comparison of the B+tree, bitmap, Bloom filter and hash index will follow in section 5.4.

B+tree index

Traditionally, the B+tree index is the most frequently used index for high selective queries. Hence, B+trees are not only used as storage structure, but can also act as a secondary index, especially in cases where the data is stored in an unordered structure, as is the case with our storage structure. However, we have seen before that the B+tree as storage structure is inefficient for data degradation. In the same spirit, the B+tree as index is also inefficient for data degradation, since every delete or update results in a random I/O; a B+tree index cannot be sorted on degradation time, but is instead sorted on the key values themselves.

One solution to overcome the high cost of random I/O is the use of encryption. Assuming that the key values in the index themselves are not privacy-sensitive—they become sensitive when it becomes possible to relate them to full tuples—we do not have to remove those values immediately from the index. Instead, we make sure the pointers cannot be used anymore. We can achieve this by encrypting the pointers; after every period with length ρ we choose a new encryption key, and store this key in the degradation schedule. When data needs to be degraded, we can simply remove the key from the table. This is illustrated in figure 4.10.

Although not required to ensure Δ-durability, obsolete index entries should still be removed from the index, because the size of the index has an impact on performance of the index. This can be done in a lazy way; for example when the node in which an entry to be deleted is stored is updated with a new value, the node can be scanned for invalidated entries and these can be removed without requiring any additional I/O operations.

One can wonder why we would not use encryption to enable cheap removal from the data files itself, as suggested by Miklau et al. [81]. The reason is that to answer a question all data has to be decrypted. Since decryption costs are relatively high compared to the cost of a sequential I/O, querying encrypted data will be relatively costly. Moreover, indexing encrypted data is not an easy task; for further material on this topic we refer to [48, 98, 26].

Bitmap index

When data degrades, the domain of the degraded attribute becomes smaller and therefore the size of the result-set of queries larger. In the following we label such queries as *medium selective*. To answer medium selective queries, bitmap indexes can be used [34, 57]. Here we give a brief summary of what bitmaps are, so that we can compare them with other indexes, especially with regard to performance in a data degradation context.

In general, bitmaps are useful to index attributes of a domain with relative low cardinality (the number of possible attribute values is low). In

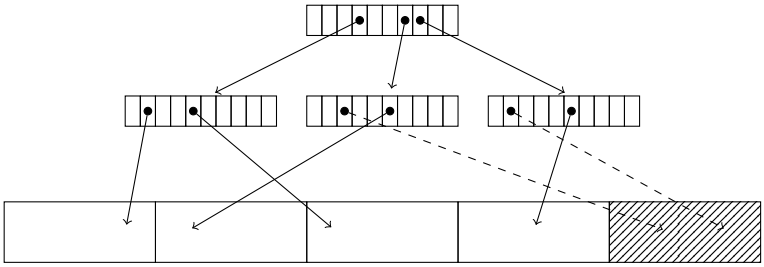


Figure 4.10 Degrading a B+tree index by means of encryption. The dashed lines represent pointers which cannot be decrypted anymore. The encryption keys can be stored in the degradation schedule, from which they will be removed at the same time as the tuples in the data files appointed by the schedule.

the most simple form, it uses one bit vector for each possible value of the attribute. Each position in the bit vector where the bit is set to 1, the tuple at the corresponding position in the table has the value associated with the bit vector (see figure 4.11).

For a table that consists of N tuples, with an indexed attribute with domain cardinality C, the bitmap consists of $C \times N$ bits. Referring to the terminology used in [34], we name such an index an *equality* encoded *one*-component bitmap. Equality encoded bitmaps are optimal for equality queries, but can also be used to answer range queries. Besides the equality encoded bitmap—which is the most common used—the *range* encoded bitmap can be used for range queries. For more information we refer to [34]. The term *n-component* stands for the amount of bit vectors needed to represent a value of the domain. Hence, for a *one*-component bitmap, only one vector is used per domain value, and therefore there are as many bit vectors as the size of the domain.

The *one-component* index is not space efficient if the cardinality C is high; for each possible attribute value it needs one bit vector. As a result, when C is large, the index might even be bigger than the actual data file itself. Nevertheless, to answer queries, the one-component index needs only one sequential scan on usually a small data file, and is therefore very time efficient.

To overcome the space problem, the bitmap can be decomposed into *multiple* components: we can decompose all possible values of a domain given a base sequence $\langle b_n, b_{n-1}, \ldots, b_1 \rangle$ to get a n-component index. More precisely, an attribute value v can be decomposed into a sequence of n digits

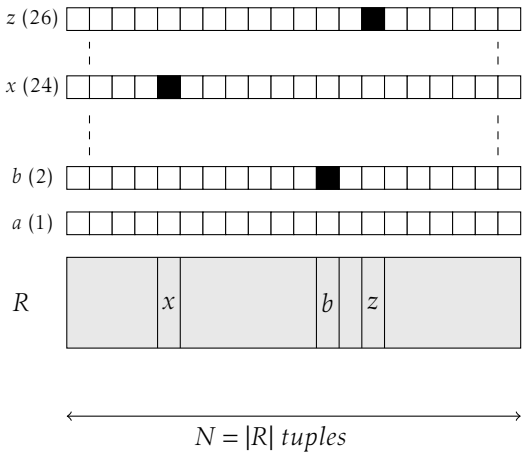


Figure 4.11 One component equality encoded bitmap

$v = v_n v_{n-1} \ldots v_1$, where each v_i is a base-b_i digit, as follows: [34]

$$v = v_n \left(\prod_{j=1}^{n-1} b_j \right) + \ldots + v_i \left(\prod_{j=1}^{i-1} b_j \right) + \ldots + v_2 b_1 + v_1 \tag{4.1}$$

For example, if we chose base sequence $\langle 7, 4 \rangle$, the value $v = 26$ would be decomposed as $v = v_2 \times b_1 + v_1 = 6 \times 4 + 2$. Using a base sequence $\langle 7, 4 \rangle$ we are able to encode $7 \times 4 = 28$ values: $\{0, \ldots, 27\} = \{0, 1, 2, 3, 4, 5, 6\} \times 4 + \{0, 1, 2, 3\}$. A graphical representation of a base-$\langle 7, 4 \rangle$ bitmap is shown in figure 4.12.

The benefit of such a decomposition is, that less bit vectors are needed to represent all values of a domain, and therefore it uses less space. However, more vectors have to be scanned to answer a query. More about performance considerations can be found in section 5.4.

Bitmap indexes are update—and therefore degradation—friendly due to their sequential nature. The indexes maintain the same order as the data files, and are therefore ordered on insert and degradation time. Hence, I/O operations can be shared by multiple tuples, and the degradation mechanism can fully benefit from the ρ-timeliness relaxation and the TIME TRIGGERED and UNIFORMITY simplifications.

Bloom filter index

Bloom filters are space efficient filters used to optimize so-called membership queries [27] and have been introduced by Bloom in 1970 [22]. Applied in database systems, sequential scans can be prevented, saving I/O costs.

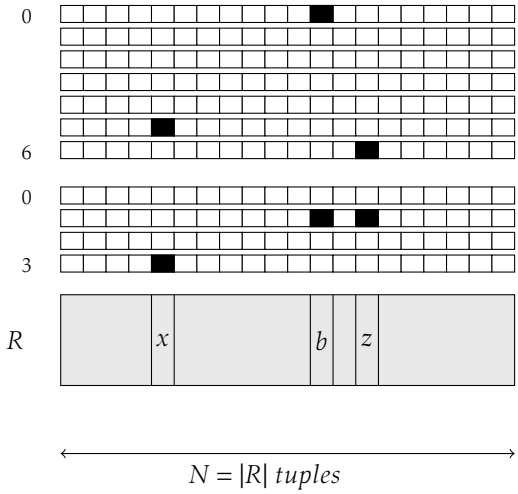


Figure 4.12 Base-$\langle 7, 4\rangle$-equality encoded bitmap, where only the mappings of the values x,b and $z \in R$ are shown. For the value x (numerical value is 23 ($a = 0$,$z = 25$), which can be decomposed in $5 \times 4 + 3$) the bits at the correct offsets in the 6th vector of the 1st component and the 4th vector of the 2nd are set to 1.

The size and lookup time of a Bloom filter are constant; however, false positives can occur.

A Bloom filter is basically an array of bits which are initially set to *zero*. When a value is inserted into the data set, k hash functions are applied on the inserted value; based on the outcome of the hash functions k positions in the Bloom filter are set to *one*. To check if an element is in the data set, the k hash functions have to be applied on the value. If the k positions in the Bloom filter are equal to 1, the value *might be* in the data set. A sequential scan–or whatever is the most appropriate access path–is required to decide whether the value indeed occurs in the data set. If one or more bits are equal to 0, the element is *not* in the data set, and a sequential scan has been avoided. Only false positives can occur, no false negatives. See for a graphical representation figure 4.13.

Bloom filters can be used to form an index on a data set in a similar fashion as bitmap indexes [100]. For each tuple in a dataset we create a Bloom filter using k hash functions and add the attribute value to be indexed to this Bloom filter. Hence, the Bloom *filter* contains only one element, and the whole Bloom filter *index* is build up from many Bloom filters. Note that a higher performance benefit—compared to the bitmap techniques described before—can be obtained if we create one Bloom filter *per page*.

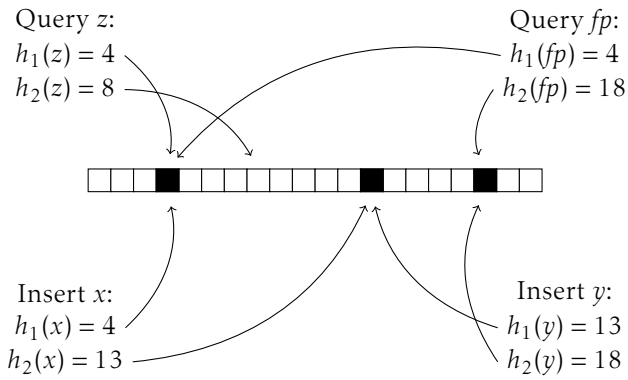


Figure 4.13 Example of a Bloom filter using two hash functions. A query on z results in a true negative, since the bit at place $h_2(z)$ is still zero. Bits on location 4 and 13 are set by hash functions h_1 on x and h_1 on y. This causes a false positive when querying fp, since $h_1(fp) = 4$ and $h_2(fp) = 13$.

Then, all tuples in a page are represented by one bloom filter, and added to the bloom filter as described in Figure 4.13. The disadvantage of such approach is that the probability of false positives will be higher; however, less storage space is needed to store the index.

Now, as illustrated in figure 4.14, we vertically fragment each Bloom filter so that each bit of the Bloom filter is stored in a separate file dedicated to a position in the Bloom filters, so that if there are N tuples, each fragment contains N bits (or, when page based bloom filters are used, N/p where p represents the number of tuples in a page). Hence, if the ith and j^{th} bit of the Bloom filter of a tuple at a certain offset in the dataset are set to one, the ith and j^{th} fragments contain a bit set to one at that offset.

To find the positions in the dataset of tuples (or pages containing one or more tuples) containing a specific attribute value x, we apply the hash functions on x and find the k fragments referenced by the output of the hash functions. By performing a bitwise AND operation on the k bit vectors and a scan of the resulting vector, we are able to find the positions of the corresponding tuples (or pages) in the dataset. See again figure 4.14 for an example.

The most interesting property of a Bloom filter index in the context of data degradation is, that the number of data files needed is more or less independent of the cardinality C. This makes the index interesting also for higher cardinality domains, although, as we will see in section 5.4, the number of false positives plays an import role in query efficiency.

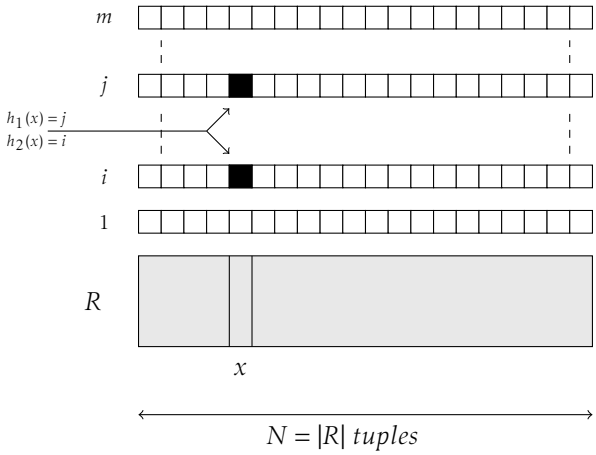


Figure 4.14 Example of a Bloom filter index using 2 hash functions. All Bloom filters (one per tuple) are drawn vertically, and fragmented into fragments. Each fragment is drawn horizontally. When a tuple x is inserted, the hash functions determine positions in a Bloom filter, for which the bits at the correct offset in the corresponding fragments (determined by the result of the hash functions) are set to 1.

Hash sequential list index

An hash sequential list index is basically the same as a normal hash index [43]. The basic idea behind the design of hash *sequential list index* is to benefit from sequential deletes. A hash sequential list index (HSL) consists of a set of hash buckets containing value/pointer-pairs, which are ordered on insert time. The bucket in which a value will be placed is determined by a hash function on that value; a bucket can contain multiple different values. See figure 4.15.

To answer a query on value x, the hash function is applied on x to determine the bucket which contains the value/pointer pairs containing x. A scan of this bucket will retrieve all pointers to the data file. This type of index is only useful for equality queries; for range queries, a range partitioning function can be used instead of the hash function.

The techniques to manage degradation, as discussed in section 4.3.1, can be applied to degrade data from the buckets, making this technique interesting for data degradation purposes.

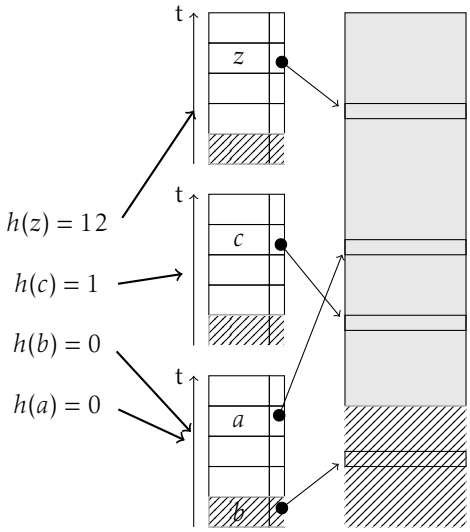


Figure 4.15 Hash sequential list index. The buckets are ordered on insert, and thus degradation time. The buckets consist of value/pointer pairs; multiple values can share the same buckets. The shaded area denotes already removed values.

4.3.3 Transaction management and recovery mechanisms

An important aspect of database system management is the way how (concurrent) transactions are managed. First, concurrent transactions need to run within an appropriate isolation level to keep the database in a consistent state, and second, in case of rollbacks and or media failure, the database has to be brought back in a consistent state. Due to the requirements of limited retention and data degradation, we have to revisit traditional transaction management techniques. In the following, we will discuss how degradation side-transactions can be synchronized with regular user transactions, and we will discuss how logs can be managed in an degradation efficient way.

Avoiding degradation side-transaction conflicts

An important aspect of a relational database system supporting data degradation is how to deal with regular user transactions and degradation side-transactions running concurrently. The Δ-atomicity property (see definition 4.1.1) states that every main transaction results in a set of degradation side-transactions, which are guaranteed to commit, and the Δ-durability property (see definition 4.1.2) requires that degradation side-transactions commit exactly at the end of the assigned retention periods. To guarantee

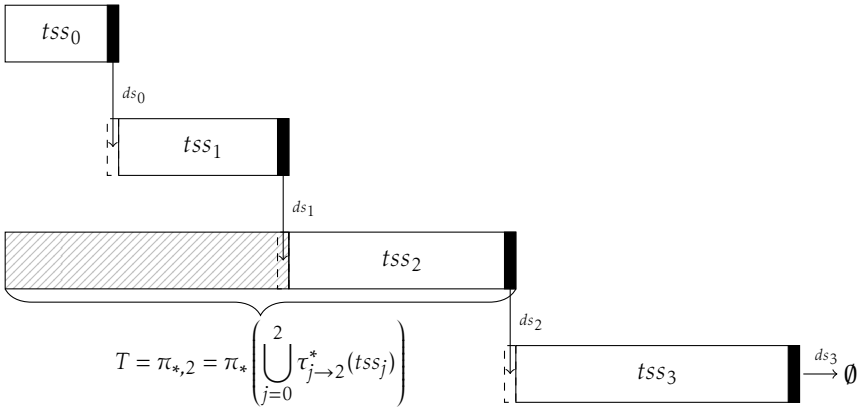


Figure 4.16 Schematic overview of an user transaction T querying all tuples with required precision $i = 2$, and degradation side-transactions $ds_0 \ldots ds_2$ degrading data from $tss_0 \ldots tss_2$ respectively. Transaction T is not allowed to update tuples. Since the result of ds_0 and ds_1 will have no impact on the *view* of T, and T has no impact on the tuples subject to the degradation steps, no serialization of T and ds_0, and T and ds_1 is needed. Only the result of ds_2 will impact T. Therefore, degradation side-transaction ds_0 requests a lock on all tuples which it will degrade after a time period ρ. Similarly, T requests a lock on all tuples it wants to access. If transaction T is still running at the end of period ρ, T will be aborted. If T is blocked by ds_2, waiting makes no sense, since the tuples will leave the view of T.

that every degradation side-transaction results in a consistent database state, normal isolation levels can be used; however, this will eventually lead to blocked or deadlocked degradation side-transactions.

Without any additional measures, to ensure degradation side-transactions run at the exact designated time, many regular user transactions have to be blocked or even aborted. Because each main transaction T triggers multiple degradation-sub transactions— where the number depends on the number of tuple states—relying on traditional isolation control will lead to poor performance.

To avoid as many transaction conflicts as possible, we propose a *least effort degradation* process, which takes benefit of the ρ-timeliness property introduced before (see definition 4.1.3). Thanks to the ρ-timeliness property, a degradation side-transaction can degrade multiple tuples at once, and can wait until the end of time interval with size ρ before data has to be degraded. In the following we consider how a regular user transaction T, working on a view including tuple state sets $tss_0 \ldots tss_i$ can be synchronized with degradation side-transactions $ds_0 \ldots ds_i$, each ds_j degrading (or removing) data from tss_j. We make the following two important observations:

- No degradation side-transaction will cause dirty-reads by T, since thanks to the Δ-atomicity property, a degradation side-transaction is guaranteed to commit. Therefore, also regular user transactions running with an isolation level equal to *read-committed* (and those with a lower isolation level), do not need to be synchronized.
- Degradation side-transactions $ds_0 \ldots ds_{i-1}$ do not impact the view of regular user transaction T. The view of T contains tuples which are transformed to the precision of tuple state i (see query semantics in section 4.1.5). Hence, for T it makes no difference if it accesses a tuple which is concurrently degraded by a degradation side-transaction, and therefore does not need to be synchronized with $ds_0 \ldots ds_{i-1}$. See figure 4.16.

Hence, only degradation side-transaction ds_i working on the oldest data in the view of transaction T has to be synchronized with T. To do this, ds_i requests an exclusive lock, and T requests a shared lock on the tuples which will be degraded by ds_i. Those tuples can be identified using the degradation schedule already introduced before, in section 4.3.1. The nth degradation side-transaction on tss_i will take place at $n \times \rho + \delta_{i+1}$ since startup time of the database system. All tuples inserted during the interval $(n-1 \times \rho, n \times \rho]$ will be degraded by the nth degradation side-transaction. If the lock has been acquired by T, degradation side-transaction ds_i can only be blocked until the final deadline at $n \times \rho + \delta_{i+1}$; at this time, the blocking regular transaction T will be aborted, otherwise the degradation side-transaction will miss its deadline. If a conflict occurs and T is blocked by the degradation side-transaction, it is useless for T to wait until the lock will be released, since the tuples will leave the view of T. Note that the tuples possibly locked by the degradation side-transaction are only still part of tss_i thanks to the ρ-timeliness property; without this property, the tuples would not be accessible by T either and this synchronization protocol therefore does not hurt data usability.

Note that the chance of T being aborted is relatively small given the fact that ρ is relatively large, at least much larger than average transaction durations. Therefore the impact of data degradation on transaction synchronization using above synchronization is limited, because regular transaction will almost never be aborted and tuples which should be accessible by a regular transaction will never be blocked by degradation side-transactions.

Logging and recovery management

Logging techniques are traditionally used to enforce atomicity and durability while permitting classical buffer management optimizations like writing in the database file before a transaction commits (using the so-called *steal strategy*), after a transaction commit (so-called *no force strategy*) as well as

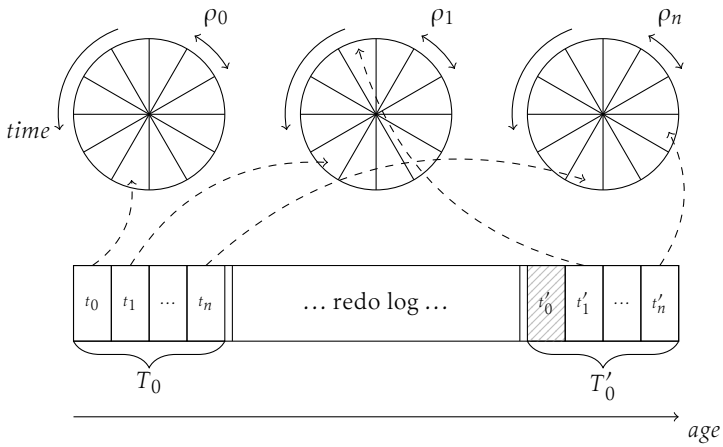


Figure 4.17 Both main transactions T_0 and T'_0 insert a single tuple. For each tuple state t_i, the tuple value will be degraded using transformation function $\tau_{0\rightarrow i}$, encrypted, and appended to the log. For each tuple state i and time period ρ_i, a new encryption key will be created and stored in a *circular list* of length Δ_i. This circular list will contain $\frac{\Delta_i}{\rho_i}$ keys. As long as the tuple should be retained in the database, the encryption key exists, making it possible to decrypt and recover the tuple only during that retention period. T'_0 took place more than Δ_0 before T_0, and therefore the original encryption key to decrypt t'_0 is overwritten and not available anymore.

making recovery possible in case of failure. However, in the context of data degradation, data should only be recoverable during a limited period, as defined by the Δ-durability property (see 4.1.2). Hence, after its retention period, data should disappear also from the log files.

Our objective is to keep the interesting optimizations made possible thanks to logging techniques, also valid on the degradable part of the database. Note that techniques used in traditional database management systems can be used for logging and recovery of *stable* data.

There are many different approaches to logging; in the following we consider an approach based on a combination of undo/redo logging. In our discussion we omit most details (they can be found in database textbooks [43]).

- *Redo log*. The redo log is traditionally used to implement *durability* [43]. Using a redo log, tuples do not have to be written to disk before a transaction commits, so that I/Os can be saved. Therefore, the redo log includes the images and references (needed to implement atomicity, see below) of tuples inserted by main transactions and the transaction begin and commit statements. In the context of data degradation, time-stamping the commit statements allows to replay, if

necessary, the degradation side-transactions and to rebuild data and index files.

To enforce Δ-durability, making sure that degradable data can only be recovered during its retention period, at every data degradation step, a degradation side-transaction should not only remove data from data files and indexes, but also from the log files. However, sequentially searching the log files and update those log files is expensive in terms of I/O, and should therefore be avoided. We suggest to encrypt the images of inserted tuples following the principle described in section 4.3.2; at degradation time, the encryption key will be erased so that the image in the log files cannot be decrypted anymore, effectively removing the data. This principle is illustrated and further explained in figure 4.17.

Now, the overhead of managing a redo log which is compliant with Δ-durability, induces only a negligible encryption overhead, and one single I/O for overwriting the encryption key for each time interval of size ρ for each tuple state.

- *Undo log*. The undo log is traditionally mainly used to implement *atomicity* [43]. For durability we rely on the redo log (see above). When updates are written to disk before the transaction commits (*steal strategy*), the pre-image of the updated tuple will be appended to the undo log. When the transaction rolls back, the undo log can be used to write the original value of the tuples back to disk.

  Degradation side-transactions are guaranteed not to roll back, so they do not have to be contained in the undo log. Moreover, as described in section 4.3.1, tuples are buffered in an insert buffer, even after the main transaction commits (enforcing a *no steal* strategy). Hence, inserted statements caused by main transactions also do not have to be contained in the undo log. Because regular user transactions are not allowed to update tuples (see the simplifications made in section 4.1.5), it only needs to be possible to undo deletes—delete operations by users themselves, not because of data degradation—in case the delete is written to disk before the transaction commits.

  The pre-image of the deleted tuples is already contained in the redo log to be able to implement Δ-durability (see above). Hence, containing a reference to the deleted tuple instead of the actual value is enough to implement atomicity, since in case of a roll back, the pre-image can be recovered from the redo log. This way, degradation of the undo log is not needed.

To recover the database in case of a media failure, where all data in the database has been lost (*cold recovery*), the redo log has to be used. The redo log contains the complete history of main and regular user transactions, including the time of commits. The redo log can be replayed in chronological

order to rebuild the data files, indexes, and the degradation schedule, where all tuples which cannot be decrypted are ignored. Of course, it is assumed that the file storing the keys is not destroyed (using the same assumption which holds for the log files themselves). It depends on the chosen storage structure and strategy how the data will be restored. When in lazy mode, for each tuple, the most precise tuple state for which an encryption key is available will recovered, all other states will be ignored. When in eager mode, all recoverable tuple states will be inserted in the corresponding data files.

However, since the database system has not been active since the media failure, no degradation steps have taken place on data which should have been degraded given the degradation policy. Therefore, before returning to normal usage, the database system must perform all necessary degradation updates to make the database state compliant with the degradation policy. To do this, the degradation schedule has to be used.

The undo log can be used to bring the database in a consistent state in case there has been uncommitted transactions. This is also the case after a database system failure where the data is still intact (*warm recovery*). Again, the degradation schedule has to be used to perform delayed degradation steps.

4.4 Revisiting the simplifications

In this section we discuss some of the simplifications we made to show the impact of data degradation on traditional database management systems. The simplifications we discuss now are simplifications on the implementation level; simplifications taken at the model level, although they have impact on the implementation, such as TIME TRIGGERED and UNIFORMITY, are discussed in chapter 6. In this section we concentrate on the query and update semantics. Note that many of the raised issues are left for future work.

4.4.1 Query semantics

In section 4.1.5 we defined the semantics of queries over degraded data. These query semantics are based on a *least* effort principle, or more specifically: only the tuples in a tuple state set with equal or higher precision than as defined in the purpose of the service are used to evaluate a predicate, and the result is degraded to the asked precision. For both projection and selection operation we could also have chosen a *best* effort strategy, each resulting in different query semantics. We discuss the options briefly.

A best effort *selection* strategy uses all the data which is still (and longer) available in the less precise states. The data in such states can be considered

as probabilistic data [16] using a probability distribution which takes the size of the domain into account. This additional information can be used to evaluate the query predicate. A part of the returned data will then be considered as uncertain data, which is up to the service provider to handle correctly. This strategy makes only sense when assuming that the probability distributions are not uniform, but based on the actual monitored environment. However, this is not the objective of our data degradation model in terms of privacy preservation, but can be used to increase the amount of usability for services.

Example 3. The attribute age can be degraded from precision level *Range1* to $[Child, Adult]$ (or represented differently as $[0,17],[18,\ldots]$). Given our query semantics based on the purpose specification, the predicate $age \geq 16$ would be evaluated only on the most precise set of tuple states. Using probabilistic database query semantics, it would evaluate the query also on the least precise sets, returning each tuple with $age = Adult$ with a probability of 1 attached to it, since each adult has an $age > 16$. The tuples with $age = Child$ will contain a probability lower than 1. Using an uniform probability distribution this value would be $\frac{2}{17}$, but with a probability distribution based on external information, this probability can be higher or lower for each individual tuple.

A best effort *projection* strategy returns each tuple in the highest precision available, instead of degrading the data to the required precision. The consequence of such a strategy is that the result of a query is unpredictable for the querying service; additional measures have to be taken to identify the precision of the resulting tuples. Moreover, a best effort projection strategy impacts the transaction synchronization protocol (see section 4.3.3). Not only degradation side-transactions working on the oldest data in a view of a regular transaction have to be synchronized, but also degradation side-transactions on all other tuple state sets, to avoid inconsistent results of subsequent queries within the same transaction.

4.4.2 Inserting in the past

An implicit simplification we made, is that a main transaction inserts a new tuple always in the most precise tuple state set tss_0. Hence, we assume that a tuple starts its life-cycle when it enters the database. This assumption may not always hold; firstly, there may be a delay between the monitoring of an event by sensors, and the actual insert into an underlying database. For example, records of monitored cars on a road may be first stored in a sensor buffer and flushed to a database system at the end of the day. It may be desirable to start the retention periods of the records at the time they where created by the sensors, and not when they enter the database. It might be that retention periods are shorter than a day, so that at the time

the record reaches the database, it should have been degraded and thus be inserted in a less precise tuple state set. Secondly, in distributed settings, a database system might take over the task of another database system; life-cycles should then simply be continued.

Inserting in the past adds additional complexity to degradation schedules and management of encryption keys. More troublesome is the ordering of the tuples on degradation time; many optimization are based on a strict ordering of tuples so that multiple tuples can be degraded by one single degradation side-transaction.

4.4.3 Update semantics

In traditional database management systems, a update is often treated as a delete of the old value followed by an insert of the new value. In the context of data degradation this is not a feasible approach, since retention periods are bound to tuples; a updated tuple should continue its life-cycle, and therefore cannot be considered as a new item. This behavior can only be enforced if we allow 'inserting in the past', as described above. Otherwise, a update should always result in a physical replacement of the value in all data files (depending on the storage structures), update of the indexes, and a record in the undo log for atomicity purposes and in the redo log for durability purposes (see below).

An additional difficulty of updates is the question how to update a tuple in a view in which tuples have been degraded to a lower precision. Consider the following:

Example 4. A service declares the following purpose, and issues an update statement to transfer all persons with a high salary to the Netherlands:

```
DECLARE PURPOSE Stat
SET PRECISION LEVEL Country for location, Range1000 for salary

UPDATE Person SET location = 'Netherlands' where salary > 5000
```

The problem is to which value tuples have to be updated if they have a higher precision than, in this case, 'country'. Simply updating high precision values with low precision values is undesirable if multiple services access the data, and leads to loss of data usability. A possibility is to restrict updates to only the most precise data.

Allowing updates leads to a revision of the logging mechanism. To ensure atomicity, updates have to be recorded in the undo log. To avoid degradation of the undo log, a reference to the updated tuple can be used, in a similar fashion as delete statements. In addition, the post-image of the updated tuple has to be added to the redo log, encrypted with the same encryption key as the original tuple.

4.5 Conclusion

This chapter zoomed in on the technical aspects of the data degradation model. Traditionally, design choices are focused only on finding a good balance between insert and query performance, which is highly application dependent. Data degradation adds a third dimension to the design considerations; we offered a set of storage strategies and indexes suitable for different application requirements, which all ensure Δ-durability and are all designed to increase degradation performance. Adding some flexibility to the retention periods results in major performance benefits; using an ordering on degradation time, tuples can be degraded together, resulting in a decrease of costly random I/O operations. In chapter 5 we will experimentally evaluate the proposed techniques using a prototype.

Data degradation puts not only requirements on storage structures and indexes, also transaction management and recoverability are effected. We proposed a synchronization protocol which takes benefit of the flexibility in retention periods and showed that the number of conflicts between degradation side-transactions and regular transactions can be minimized. Moreover, we provided a logging mechanism using encryption techniques so that Δ-durability and atomicity can be ensured using redo and undo logs, even in the context of data degradation.

Experiments and analysis

5

As we have seen in the previous chapter, data degradation requires changes to traditional database system techniques. We proposed new storage structures and index mechanisms, which all have their own characteristics when it comes to insert, update, and query performance. In this chapter we will—experimentally where possible—analyze the proposed techniques. To do so, we have basically three options:

- Adapt an existing database kernel.
- Using analytical methods.
- Building a degradation-aware database system from scratch.

To test the performance of the various storage structures we chose to build a (subset) of a database system from scratch. However, implementing a database system from scratch is complex, and therefore we limited ourselves to implement only what is minimally required to perform our experiments. This will make it hard to compare the performance of our system with traditional databases; however, this is not the objective of our study, which is to analyze the design options of a database system with respect to data degradation. Hence, we build a prototype implementation which we use to experimentally test the storage structure. Besides, we present an analytical study of the index structures.

The chapter is organized as follows. First, in section 5.1, we identify the objectives of evaluating the storage structure and indexes and motivate why we chose for our approach. Second, in section 5.2, we describe our prototype implementation to give insight in the more important implementation decisions, necessary to make a good interpretation of the test results possible. Then, in section 5.3, we describe the test-setup and present the results of our experiments. Finally, in section 5.4, we present an analysis of the previously proposed index structures.

5.1 Considerations

The most important consideration to make, is what the objectives of the experiments should be. As we have seen in the previous chapter, data degradation has a high impact on traditional database system techniques. Although we proposed several techniques which are supposed to limit the impact on performance as much as possible, the following two questions are still important:

- What is the penalty of degradation when implemented in a traditional database?
- Which application requirements lead to which choices of storage structure and indexes?

Data degradation adds an additional load to the system, besides the usual insert and query load. As there are insert-intensive database applications and query-intensive applications, we introduce *degradation*-intensive applications. How much degradation-intensive an application (or service) is, depends on the life-cycle policy it uses; for example, more degradation steps lead to a more degradation-intensive application. Also the amount of degradable attributes plays a role.

The storage structures and indexes we proposed all serve the purpose of speeding up data degradation. Every strategy has its drawbacks in terms of insert and query performance, where the most degradation-efficient strategy is expected to be the least efficient in terms of insert and query performance. Hence, the objective of our performance study will be to outline the performance penalties of each storage structure with respect to insert and query performance, and to give 'rules of thumb' for choosing the best strategy for various application requirements.

Answering the first question will require to adapt an existing database system kernel, which we decided not to do, as explained in the following. Nevertheless, to show the feasibility of data degradation, adapting an existing database system kernel remains important future work.

5.1.1 Adapting an existing database system kernel

The advantage of choosing to adapt an existing database system kernel is that we can show the feasibility of implementing data degradation as a *plug in* for traditional database systems such as MySQL, Oracle, PostgreSQL, *et cetera*. When we have such an implementation, we can easily indicate the performance loss due to data degradation, and show the benefits of our degradation-friendly storage structures under different conditions. Moreover, we get all optimizations and components which don't need to be adapted for data degradation for free. This way we can better indicate the actual performance penalty introduced by data degradation.

All common database management systems support a stored-procedure language to be able to add new functionality. Such a language can be used to write the procedures necessary to degrade the data. We can, for example, implement a degradation schedule which triggers transition functions degrading the data on the correct time. However, those languages only operate on a high level (such as SQL does), whereas, as we have seen in the previous chapter, data degradation requires at least a new implementation of the storage structure, such that write operations can be enforced and data can be correctly degraded. Moreover, because stored procedures cannot be used for low level operations, the necessary adaptations to transaction mechanism and logging cannot be implement with stored-procedures only.

The requirement of full control over all write and read operations makes that we did not choose for the option of adapting an existing database system kernel. Without this control, we cannot implement each data structure in such a way that a fair comparison between the storage structures is possible. To illustrate, it is hard to efficiently rebuild a database system kernel to support a fragmented storage structure while it has been designed to store data in a clustered way, and vice versa.

Moreover, to be able to implement data degradation using complete new storage structures—not only one, but at least four, one for each degradation-friendly structure we proposed in section 4.3.1—we need full understanding of and insight in the implementation details of an existing database system kernel, which we unfortunately have not. Moreover, the database system kernel should be *modular* enough to replace the storage structure. For extensive research on the feasibility of using one of the well-known open-source database systems such as MySQL, PostgreSQL and Derby, or one of the less well-known systems H2 and MonetDB, we refer to Mathijssen [108]. Although arbitrary, the outcome of this research was that the examined database systems show too much added complexity in their physical layer to be feasible for us to consider this option, given the available amount of time and available resources.

5.1.2 Using analytical methods

Due to time-constraints, we were not able to build a prototype which supports all the techniques discussed in chapter 4. Therefore we decide to analytically evaluate the index structures. For index structures this is a feasible approach, since the amount of parameters and dependencies is less than that of, for example, the storage structure.

5.1.3 Building a database system from scratch

Given the complexity of adapting an existing database kernel, and the limits of analytical methods to evaluate all our techniques, the only option

remaining is to build a database system from scratch. This option gives us two advantages:

- We have full control over all implementation details, making it possible to implement our proposed techniques to their full extent.
- We can make a fair comparison between all implemented techniques.

Building a database system which can compete with well-developed traditional database systems is not possible within a limited amount of time, and therefore we have to limit ourselves to implementing only the necessary components needed for a research platform which makes it possible to evaluate the storage structures and where possible in the future the index techniques. We describe this platform in the next section.

5.2 Prototype implementation

In the following we discuss the prototype we used to test and compare the performance of the storage structures. The prototype has been implemented with contributions by my Master student Mathijssen [108]; the source and executables can be found at our website [122].

5.2.1 Architecture

Our prototype consists of four components. Figure 5.1 gives an overview of those components. In short, SQL insert and schema definition statements are communicated through an *interface*, which parses them. For a schema and the corresponding LCP statement (describing the degradation policy), the *strategy logic* component builds—depending on the used storage strategy—a set of sub-relations containing either degradable or stable attributes (or both). It attaches, for each degradable attribute, transition definitions to each relation. Those transition definitions contain information about how and after which delay an attribute has to be degraded. Then, for each sub-relation, the storage engine will create a data file, including dedicated insert and update buffers. The degradation manager will process the transition definitions, and at given times send update statements to the storage engine.

Each individual component is explained in more detail in the following.

Storage engine

The storage engine is responsible for the actual storage, fetching and updating of data. Storage of data is page-based and follows as much as possible traditional database system storage techniques, and the techniques for degrading the data are implemented as described in the previous chapter (section 4.3.1). We recall here some of the important techniques which are relevant for a better understanding of the results of the experiments.

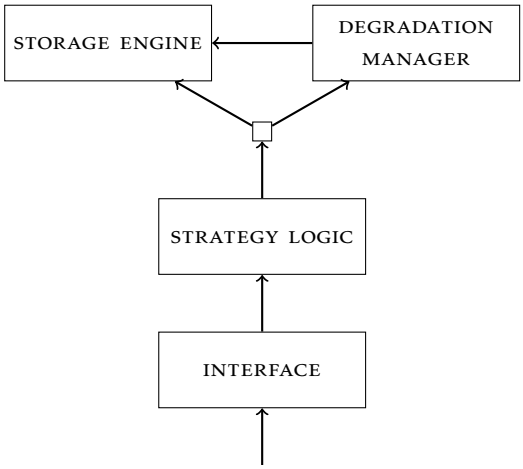


Figure 5.1 Schematic overview of the prototype, where arrows denote the direction of the flow of operations. First, the *interface* is responsible for parsing SQL insert and queries, and schema definition statements. Second, the *strategy logic* component transforms the original relation into sub-relations, depending on the chosen strategy. It attaches transition definitions for each degradable attribute in a sub-relation. Third, the degradation manager is responsible for executing those transitions at the correct time on the correct tuples. Finally, the storage engine is responsible for storing, updating and fetching the data when requested by either the strategy logic or the degradation manager.

- *Buffering*. For each data file there exists one insert, update and query buffer. The reason for keeping distinct buffers is simplicity; pages in each buffer are there for only one purpose, and can be treated as such. For examples, pages in the select buffer never have to be written back to disk when space has to be freed in the buffer. The insert buffer can be flushed to disk when it is full, and update buffers can be flushed to disk after each degradation step. However, before fetching a page from disk, the buffer manager will check if the requested page already exists in one of the other buffers, and *move* the page to the assigned buffer.
- *Page alignment*. Delete and update ranges can be aligned on page borders. This way, all tuples of a page will be updated or deleted at the same time, saving I/O cost. To degrade all tuples in a page to a next state, a page has to be read and written only once. The consequence is that some tuples might get degraded earlier than necessary. This might be not desirable behavior when insert rates and/or retention periods are so small that only a few tuples need to be degraded, and thus relatively many tuples are degraded too early. However, in such

cases performance will not be an issue anyway, making alignment of pages irrelevant.

- *Pre-fetching*. Pages are numbered with their offset in the data file. Hence, the physical address of a page is a combination of the name of the data file and the offset from the beginning of that data file. Now, if a page with offset x is requested, and it is not present in one of the buffers, it will be fetched from disk together with page $x+1, x+2, \ldots, x+n-1$, where n is a parameter, in the following called the *I/O size*. If those pages don't fit in the buffer, the buffer will be flushed. Hence, the *I/O size* used for a particular buffer can never be larger than the size of that buffer (counted in number of pages). This is an applicable and feasible strategy, because data is stored sorted on insert time, and will be degraded in the same order.
- *Deletes*. If we would not use page alignment (see above), situations can arise where only a subset of the tuples in a page need to be removed during one degradation step. However, all operations read and write full pages; it is not possible to overwrite individual tuples directly on disk with dummy values, such that they are removed. Hence, to delete tuples from such a page, the full page needs to be read to be able to 'save' the tuples which should not be deleted. The tuples which need to be deleted will be removed from the page in memory; the page can then be written back to disk, overwriting the original page.

  Thanks to page alignment, all tuples will be deleted without first reading the page which contains those tuples, because we know that all tuples can be deleted. To perform the delete, an empty page will be created, which will overwrite the old data in the data file.
- *Locking*. For the sake of simplicity, and to make sure there are no conflicts, each *transaction* is performed in isolation. In our prototype a transaction is either:
  - An insert of a single tuple. Depending on the used storage strategy, a tuple might need to be inserted into more than one data file. The transaction ends when the tuple has been inserted in all its data files.
  - Degradation of a range of tuples from *one single* sub-relation, initiated by the degradation schedule. If tuples from two sub-relations have to be degraded at the same time, the degradation will be performed by two subsequent transactions.
  - A single query; the transaction ends when all data is fetched and processed from all required data files.

  Once the lock is acquired, only one type of operation will use any buffer at the same time, and will release the lock and resources after full completion of the transaction. To enforce degradation, all writes

to pages will need to be committed to disk. Therefore we flush the update buffer to disk after the degradation transaction finishes. No matter which strategy we use, degradation of a range of tuples will always be performed on only one data file, and thus in practice only one *update* buffer will be used at the same time. This makes sharing of buffers possible.

Note that we did *not* implement the transaction synchronization protocol discussed in section 4.3.3. As we have seen, a strict locking mechanism as used in our prototype is not necessary in practice.

- *Fsync*. The prototype has been implemented in Java, which means that the code runs in a virtual machine. Hence, the prototype cannot *directly* communicate with the hardware and the disk in particular. For memory management and writes to the disk it has to rely on the operating system. However, to enforce that data is directly written to disk, and that the writes are not buffered in a operating system cache, Java supports the *fsync* system call. Still, we have to rely on the operating system for the actual implementation of the placement of data, and whether or not blocks are written sequentially to disk or not.

Strategy logic

The strategy logic component is responsible for three things:

- To transform an original relation R into sub-relations based on the chosen strategy (figure 5.2).
- To transform insert statements.
- To rewrite a query on R into sub-queries, and reconstruct the result.

In section 4.3.1 we discussed a *baseline* approach and four other data degradation-friendly approaches, of which two where LAZY and two where EAGER. In fact, the *baseline* approach is also a LAZY approach. To be able to compare the CLUSTERED and FRAGMENTED EAGER with a baseline version, we also implemented an eager version of the baseline strategy. In the following we refer to the *baseline approaches* as NONE LAZY and NONE EAGER, where NONE is suggestive for "neither CLUSTERED nor FRAGMENTED".

The strategy logic component will transform the original relation into sub-relations for each strategy. For example, consider a relation $R(S,A,B)$ and the LCP we used before in the previous chapter (see figure 4.2); then R is transformed into sub-relations with the following schema's:

NONE LAZY One relation containing both stable and degradable attributes $\Rightarrow \{S,A_0,B_0\}$

NONE EAGER For each tuple state a sub-relation containing all attributes $\Rightarrow \{S,A_0,B_0\}$, $\{S,A_0,B_1\}$, $\{S,A_1,B_1\}$, $\{S,B_1\}$

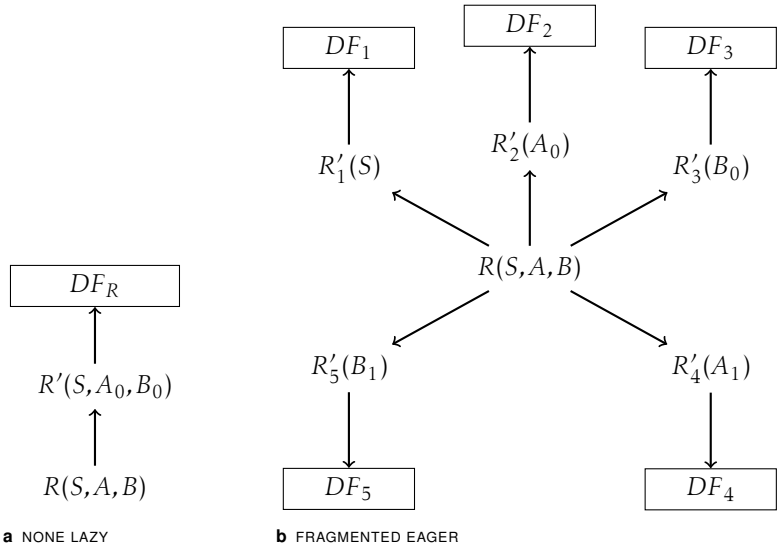


Figure 5.2 Decomposition (according to the LCP of figure 4.2) of an original relation R with a stable attribute S and two degradable attributes A and B for the NONE LAZY and FRAGMENTED EAGER strategies. With NONE LAZY the original relation stays intact, A_0 and B_0 denote that newly inserted tuples will initially be stored (only) in their most accurate state. For the FRAGMENTED EAGER strategy, the relation is split into 5 relations, all containing only one attribute. In relation R'_4 and R'_5, the attribute values will first be degraded to state A_1 and B_1 respectively before they are inserted.

CLUSTERED LAZY One relation for all stable attributes, one relation for all degradable attributes $\Rightarrow \{S\}, \{A_0, B_0\}$

FRAGMENTED LAZY One relation per degradable attribute, one relation for all stable attributes $\Rightarrow \{S\}, \{A_0\}, \{B_0\}$

CLUSTERED EAGER One relation per tuple state containing only the degradable attributes, one relation for all stable attributes $\Rightarrow \{S\}, \{A_0, B_0\}, \{A_0, B_1\}, \{A_1, B_1\}, \{B_1\}$

FRAGMENTED EAGER One relation per attribute state, one relation for all stable attributes $\Rightarrow \{S\}, \{A_0\}, \{B_0\}, \{A_1\}, \{B_1\}$

Each sub-relation will be assigned its own data file, as pictured in figure 5.2. When an insert statement arrives it has to be split into multiple insert statements, one for each data file. For the EAGER strategies, a *transformation* function has to be called to pre-process all attribute states. In our prototype, such a function can be implemented by either a *generalization tree* or by a *generalization function*.

For example, when the strategy is FRAGMENTED EAGER, the original insert statement on relation R will be split into five statements, one for each sub-relation. The original values a and b will first be degraded using transformation functions f and g respectively:

$$\text{insert into } R \text{ values } (s,a,b) \quad \Rightarrow \quad \begin{cases} \text{insert into } R'_1 \text{ values } (s) \\ \text{insert into } R'_2 \text{ values } (a) \\ \text{insert into } R'_3 \text{ values } (b) \\ \text{insert into } R'_4 \text{ values } (f(a)) \\ \text{insert into } R'_5 \text{ values } (g(b)) \end{cases}$$

Finally, we choose in our prototype to let a query always return the most accurate version of a tuple. This means that the query SELECT * FROM R will be translated into a union of queries which scan *all tuple state sets*. It is important to note that following this semantics, a tuple will never be fetched more than once, representing the correct costs of a sequential scan for any strategy. Strictly following the query semantics described in section 4.1.5 would require to also apply a transformation function to each produced tuple. This is not necessary for analyzing the performance of the storage structure, and is therefore omitted.

Because a relation R is now possibly stored in multiple data files, queries on R have to be re-written, where each strategy needs a different re-write rule, so that no more tuples are fetched than required to produce the result. This is also a task for the strategy logic.

Interface

Our prototype accepts basic SQL-like statements to define a schema and to insert data. As an extension to the standard SQL language we added a syntax to define degradable attributes and a policy. For example, to define a schema $R(S,A,B)$ and our running example policy:

```
1 CREATE TABLE R (S 10, A 20 degradable_tree, B 20
    degradable_function)
2 CREATE LCP R
    0,0;10000;0,1;15000;1,1;10000;-1,1:15000:-1,-1
```

In the first line we create a table named R and three attributes. The number behind the attribute name defines the fixed size of the attribute (although the prototype supports variable-size attributes). The keywords *degradable_tree* and *degradable_function* indicate that the attributes A and B

will be degraded with a generalization tree and function respectively. Those trees and functions are implemented elsewhere, and the tree (or function) can be identified given the attributes name.

The second line defines the life-cycle policy. Each pair indicates the attribute state of each attribute, followed by the tuple state duration (in milliseconds). The number −1 denotes removal of the attribute. The prototype currently supports only one table and one policy at a time, which is sufficient given the purposes of our experiments.

Insert statements are specified with standard SQL syntax and can be written in a *script file* which will be read by the interface. The interface will read the script file line by line, and artificially waits between each line such that a fixed insert rate is simulated. To support precise insert rates, it takes into account that this insert rate cannot always be reached due to temporary high loads (for example because of a degradation run). When some inserts are delayed, the following inserts will be processed with a higher insert rate to make up the lost time.

Degradation manager

The degradation manager keeps track of the degradation schedules, and triggers degradation of data with intervals of size $\rho \times \Delta$, as suggested in section 4.3.1.

Each degradation schedule is a table (kept in RAM) containing entries of the form $\langle n, \textit{row offset} \rangle$, where n is the nth interval of size $\rho \times \Delta$ since the start-up of the database. The *row offset* points to an offset in the data file, which is the position of the last tuple which needs to be degraded.

When the degradation schedule is triggered—at the end of each time interval—an update statement or a delete statement will be sent to the storage engine, depending on the chosen strategy, containing the range of tuples to be degraded. Note that the storage engine might decide to extend this range to perform *page alignment* as discussed before. The degradation schedule only acts on a logical level, the actual degradation is performed in the storage engine.

5.2.2 Capabilities and non-capabilities

Although our prototype supports the storage and retrieval of data, and most important, degradation of data using six different strategies, it has a number of shortcomings. We first list what we *can* do with our prototype:

- Implementation of storage structures based on *heap* files using six different strategies.
- Implementation of degradation schedules directing the timely degradation of data, including support of the ρ-timeliness property.

- Support of generalization trees and functions to specify how data should be degraded.
- An interface to easily create schemas with a variable number of stable and/or degradable attributes.
- Support of life-cycle policies with which tuple states and retention periods can be defined.
- Possibility to execute a sequential scan of *tuple state sets*.

Moreover, the prototype has been designed as a research platform. For this purpose, it supports:

- Collection of runtime statistics such as number of random and sequential I/Os, total time needed to execute an I/O operation, *et cetera*, all per type of operation.
- Support of a fixed insert rate.
- Possibility to configure the system in detail; for example, it is possible to configure buffer sizes, page sizes, I/O size, whether or not buffers need to be flushed, page alignment, and many other parameters.

Although the prototype has been implemented such that it easily can be extended with more features, in its current state it cannot be considered as a full-fledge database system. That has never been the purpose of building this prototype. As a consequence, it misses many components which can be found in existing database systems, which makes that the performance of this prototype cannot be compared with such systems. However, in the previous chapter we described many techniques which need to be implemented in order to fully support data degradation in practice. Those techniques are:

- *Transaction control*. The prototype uses strict isolation of transactions, to make sure that the storage structures are tested correctly. We are not able to evaluate the synchronization protocol described in section 4.3.3.
- *Logging*. Traditional database systems support logging for durability purposes. Our prototype lacks a logging mechanism. Hence, we also did not implement the encryption techniques proposed to manage data degradation from log files.
- *Indexes and query optimization*. Due to a lack of time indexes are currently *not* supported, although the prototype is modular enough to support access paths other than sequential scans in the future.

5.3 Degradation-friendly storage structures

The difficulty of our performance analysis compared to traditional performance analysis is the additional dimension introduced by data degradation.

Whereas traditionally a balance has to be found between query and insert loads, data degradation adds an additional load to the database. As we will see, techniques to optimize degradation performance will lead to a decrease of insert and/or query performance. Application requirements, such as the number of stable and degradable attributes, the number of degradation steps and retention periods determine the degradation load.

As explained in the previous section, the current state of our prototype has limitations and can therefore only be used to extensively test the storage structure. We focus the experiments on insert and degradation cost. To include the cost of query overhead due to fragmentation, we perform sequential scans on the data which show the cost of the reconstruction of tuples. For more realistic queries, especially point queries, and thus to analyze the access paths, we will use a theoretical analysis presented in section 5.4.

In section 4.3.1 we introduced a set of storage structures, which are derivatives of a storage structure based on heap files, as an alternative to traditional storage structures. In this section we will investigate the characteristics of each derivative of the storage structure under different conditions. We will use our prototype to perform a set of experiments. The final objective of these experiments is to be able to indicate the best choice of storage structure for specific application requirements.

In the following, we will study the behavior when varying a set of parameters, one by one. This means that for each parameter we will look to how well the cost—especially the inserts and degradation cost—scales when we increase, for example, the number of degradation steps. After studying those parameters, we will form a general picture and provide conclusions, in section 5.3.5.

5.3.1 Test setup

The parameters we will *vary* to test the storage structures are:

- *Number of stable attributes*. The baseline approach suffers from the fact that many data have to be transferred from disk at each degradation operation. We expect that the CLUSTERED and FRAGMENTED storage structures perform better when the number of stable attributes grows.
- *Number of degradable attributes*. More degradable attributes means a higher degradation load. We expect that the strategies most focused on decreasing the degradation cost are able to scale better on a higher degradation load, but will also suffer from the increased insert cost.
- *Number of degradation steps*. The fragmented eager strategy creates a data file for each attribute state, and the clustered eager strategy creates a data file for each tuple state. Hence, for those eager strategies,

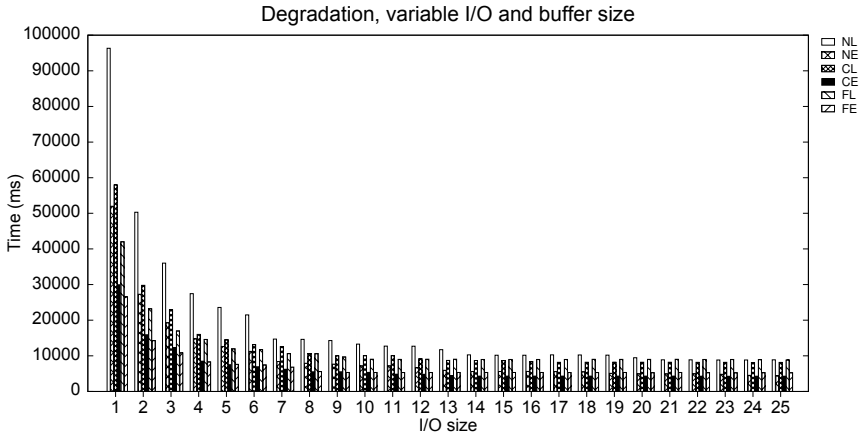
we expect that the insertion cost will grow proportionally to the number of degradation steps while lazy strategies are not influenced by this parameter (with respect to inserts). However, eager strategies are expected to be more efficient when degradation loads increase.

Some system properties have a high impact on the performance of data degradation. The most important property is the number of pages which can be sequentially fetched from disk after having randomly accessed the initial page. We name this property the *I/O size*. This property is tightly bound to the *buffer size*; the system can only benefit from a large I/O size if the to be flushed buffer is large enough. Furthermore, a large I/O size is only useful when there *are* enough pages to be flushed to disk at once. With respect to data degradation, this depends on several factors, such as the insertion rate, retention period Δ, and ρ.

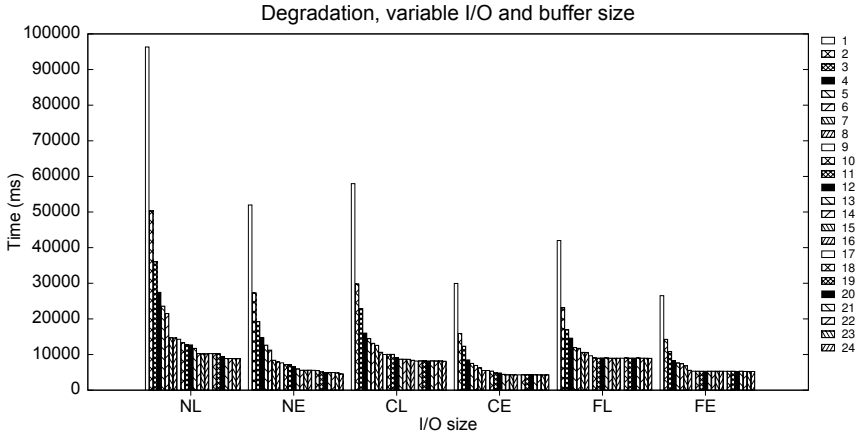
The degradation cost highly depends on the amount of data which has to be processed given each storage structure strategy. The CLUSTERED and FRAGMENTED storage structures are designed to lower the amount of data processed during each degradation operation. The amount of random I/O needed to fetch and write the data to disk mainly determines the total cost; since sequential I/O cost is relatively low compared to random I/O cost, its contributes much less to the total cost. If the I/O size and buffer size is large enough to fit all data required for the baseline approach, the higher degradation cost only comes from the higher sequential I/O cost. Hence, to make a good comparison possible between the strategies, by clearly reflecting and stressing the cost of disk based data processing, we limit the buffer size such that only a small portion of the to be degraded pages fits in the buffer.

Another consideration is whether or not we specify the amount of available buffer space *per file* or *per storage structure*. Although easier to implement, the first option would mean that strategies using more data files can use more memory, which might look unfair at first glance. Actually, in general, fragmented strategies will not benefit from the additional buffer space. Although each buffer is as large as the single buffer used by the clustered strategy, also less data needs to be processed per buffer, and therefore the additional space is useless. This can be concluded from figure 5.3. Besides, all transactions in our prototype are *synchronized*, so that no more than one degradation process can run at the same time, and thus only one update buffer will be used at a time. Therefore, and for simplicity reasons, we chose the option to specify buffer space per data file.

Unless specified otherwise, we will use the values for the various parameters as shown in figure 5.4 throughout the experiments. The choice for those parameters is more or less arbitrary, every value for each parameter can be more or less beneficial for a certain strategy. It is therefore important to notice that each outcome of each test with respect to the comparison of



a Degradation cost grouped by I/O size, and per I/O size grouped by strategy.



b Degradation cost grouped by strategy, and per strategy grouped by I/O size.

Figure 5.3 These plots show the cost for degradation given the I/O size (and a buffer size which is equal to the I/O size) for the different strategies. Since the NONE LAZY strategy processes the most data *per data file*, it benefits much from increasing I/O sizes until the I/O size is 14. Similarly, a strategy such as FE benefits until the size is 8. With this I/O and buffer size, all the to be degraded data fits in the buffer and can be fetched from disk using only one random I/O. These numbers are highly dependent on the chosen insert rate (here 250 inserts per second), number of stable attributes (3) and degradable attributes (3), ρ, retention periods, *et cetera*.

Parameter	*Value*
Page size (p)	4096 bytes
IO size (y)	16 pages
Buffer size (updates) (bu)	5 pages
Buffer size (inserts) (bi)	5 pages
Buffer size (selects) (bs)	16 pages
Approximate attribute size	15 bytes
Random I/O weight (α)	200
Inserts (i)	50000 tuples
Insert rate (x)	250 inserts per second
Degradable attributes (d)	3
Stable attributes (s)	3
Retention period (Δ)	10 seconds
Steps per attribute (a)	2
Tolerance (ρ)	10%

Figure 5.4 Parameters as used in the experiments. The update buffer size is chosen so that the data processing cost is still manifest, even given the tolerance ρ and the short retention period Δ at an insert rate of 250 inserts per second. The insert rate is slow enough to be supported by all strategies under all conditions as used in our experiments.

the performance of strategies, is an outcome on its own. From each test the only thing we can conclude with high confidence is the *effect of the changing value of a parameter on the performance of a strategy compared to the performance of the other strategies under the same conditions*. For example, we expect that a high number of stable attribute is bad for the baseline approach (NONE strategies); this effect will be magnified when the number of degradation step increases. When the outcome of an experiment with variable degradation steps shows that the baseline approach performs less than the other approaches, this might be the case because in that particular setting the number of stable attributes is high. Nevertheless, we assume that the choice of default parameters is fair enough to draw reasonable conclusions.

For the experiments we construct a life-cycle policy with d degradable attributes, with for each attribute one degradation step. The total retention period of a tuple is $(d+1)\times\Delta$—see figure 5.5. This period is divided into a period $i\times\Delta$ for the state duration of the attribute state A_0^i, and in the remainder, $(d+1-i)\times\Delta$, for state A_1^i. So, after all degradable attributes have been degraded, all attributes will be removed at the same time.

The final consideration is what to measure, and for how long. As discussed earlier, random I/O contributes most to the overall cost of database operations, followed by sequential I/O, and to a smaller extent, the CPU cost. However, although our prototype is implemented—where possible—

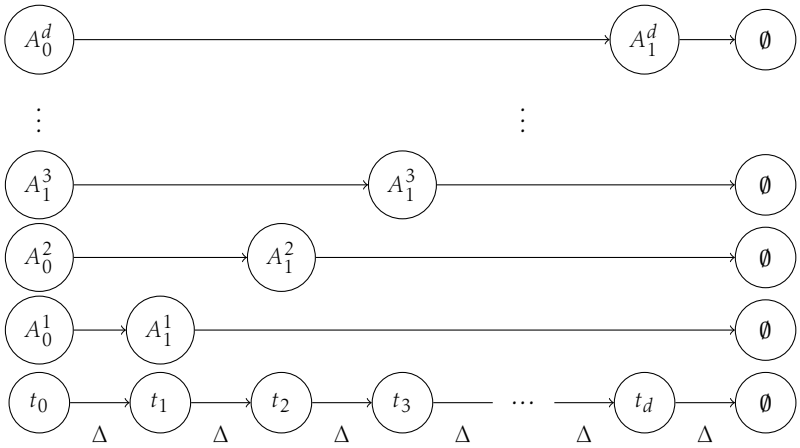


Figure 5.5 The tuples undergo d updates before they are fully removed, where d is the number of degradable attributes. Each tuple state duration is Δ seconds.

using traditional database techniques, it lacks optimizations which might be found in well-developed databases, and is written in Java which uses a virtual machine, adding additional computational overhead. Moreover, although the prototype can force the operation system to pass the I/O requests to disk, the actual execution and cost are hard to control. Hence, measuring time can only give an indication of the performance, and is not fully reliable. To overcome this, we count the number of random and sequential I/O and multiply the random I/O with a weight, to simulate the actual cost of each operation. To be precise, $random\ I/O = \alpha \times seq.\ I/O$; we assume that random I/O are approximately 200 times more expensive than sequential I/O ($\alpha = 200$).

For each run of an experiment, we count the amount of random and sequential I/O needed to execute a test script. A test script consists of a number of SQL insert statements with synthetic data, a statement specifying the schema containing degradable and/or stable attributes, and the lifecycle policy. After the database has been created, and the degradation schedules initialized, the insert statements will be executed one by one, with a constant insert rate, until all degradation schedules are active, namely after $(d + 1) \times \Delta$ seconds. At that moment, the database has reached its maximum size, so we start counting the I/O needed to insert and degrade tuples. We continue inserting tuples from the test script (starting at the first insert statement) until each insert statement has been processed. At that moment we stop counting and stop the run. Hence, we count the I/O needed to insert i tuples with an insert rate of x tuples/sec, and the amount

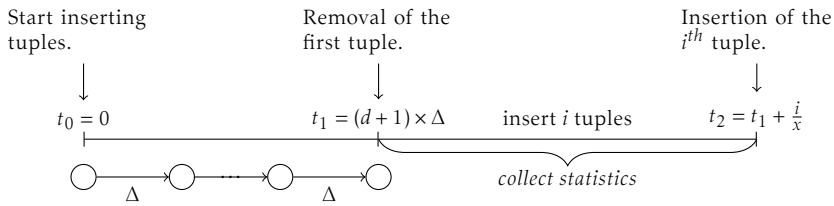


Figure 5.6 Schematical overview of an experiment run. At t_0, the first tuple will be inserted and the first entry will be placed in the degradation schedules. At t_1, the first tuple is at the end of its life-time and will be removed from the database. At this moment, all *tuple state sets* have their maximum size (constant insert rate). From this point, i *additional* tuples will be inserted. This process will be finished at t_2. Statistics will be collected during the interval $[t_1, t_2]$, because during this interval all degradation processes are active. The insert rate is x (tuples/sec).

of I/O needed to degrade the previously inserted data during a period of $\frac{i}{x}$ seconds. The process has been pictured in figure 5.6.

5.3.2 Varying the number of stable attributes

The CLUSTERED and FRAGMENTED storage structures are especially designed to limit the amount of data which has to be transferred for each degradation operation; see section 4.3.1. The less data to be transferred, the more efficient the operation. Therefore, the first optimization proposed was to detach the stable attributes, and store them in a separate data file. In the following experiment, we vary over the number of stable attributes.

Expectations

We expect to observe that the CLUSTERED and FRAGMENTED storage structures perform better than the baseline approach with respect to degradation, and that the performance gain is proportional to the number of stable attributes. Moreover, we can give a first indication of the performance of each strategy compared to the other strategies. We expect that the results verify the preliminary comparison we presented in section 4.3.1, figure 4.8.

Results

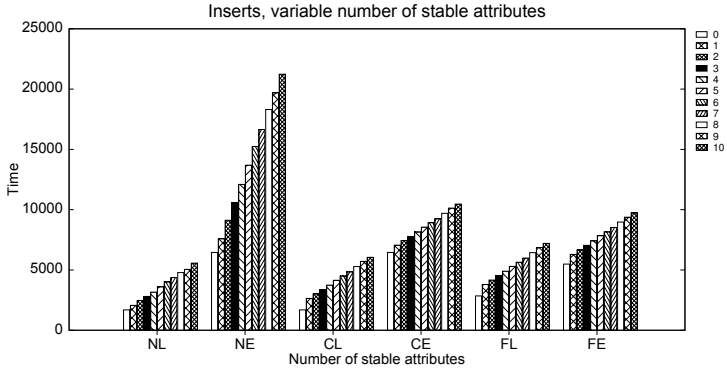
In figure 5.7a we see that the insert cost for each strategy grows linearly with equal gradient, except for the NONE EAGER strategy. The reason for the higher gradient is that the cost for storing one additional *stable* attribute is multiplied by the number of *degradable* attributes, because each data file created for each tuple state will also contain the stable attributes.

The cost for degradation (see figure 5.7b) is constant with respect to the number of stable attributes for the CLUSTERED and FRAGMENTED strategies, while it is linearly increasing for the baseline strategies. The reason for this is, that for the CLUSTERED and FRAGMENTED strategies, the data file storing the stable attributes remains untouched during each degradation operation, and therefore the stable attributes don't contribute to the cost of degradation.

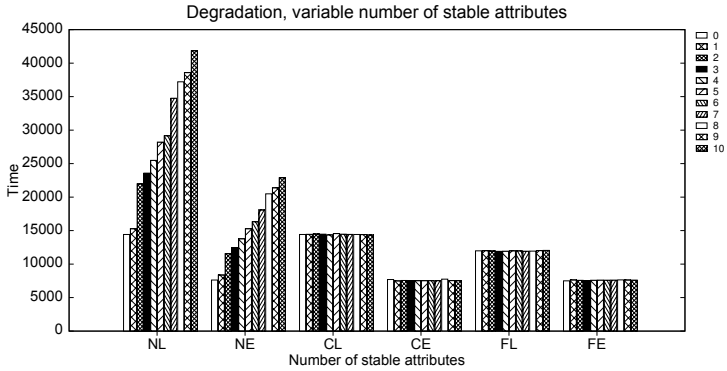
If we relate insert and degradation costs to each other by dividing them (see figure 5.8a), we see that degradation takes a smaller part of the total cost when the number of stable attributes increases. In this figure, a ratio equal to 1 means that insert cost is equal to the degradation cost, as is more or less the case with the NONE EAGER strategy. For the NONE LAZY strategy, we see that the degradation cost takes a much higher share of the total cost, although this ratio stays constant, no matter what the number of stable attributes is. Also for the CLUSTERED LAZY strategy, the degradation cost is higher than the insert cost. Nevertheless, since the degradation cost is constant, the gap is getting smaller when the number of stable attributes increases. For the EAGER strategies, the insert cost is higher than the degradation cost. The insert cost increases when adding more stable attributes, whereas the degradation cost stays constant. Note that this figure can *not* be used to compare the performance of each strategy to the other strategies.

Although we are aware that we should not draw a conclusion only based on a particular, arbitrary set of parameters, we added insert and degradation costs together, and for each strategy, relate those costs to the NONE LAZY strategy. This gives at least some insight in the performance of each storage structure, taking both inserts and degradation into account. Query performance is left out of consideration here; we concentrate on insert and degradation performance.

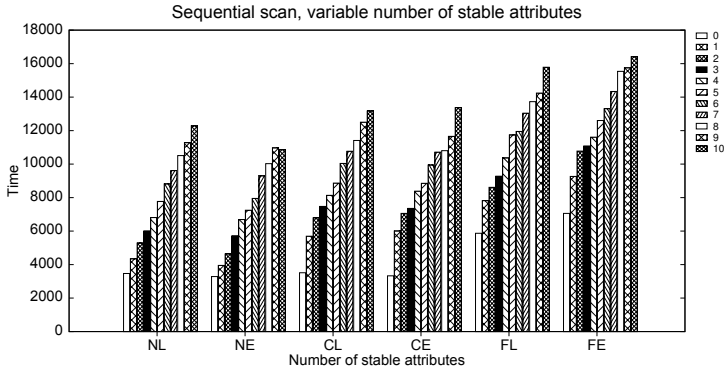
In figure 5.8b we group the cost for each number of stable attributes together per strategy, and in figure 5.8c we group the cost for a particular number of stable attributes for each strategy together. If the ratio is 1, the total cost of inserts and degradation for a strategy is equal to that of the NONE LAZY strategy. If the ratio is smaller than 1, the cost is lower and thus the performance better. We see that the CLUSTERED and FRAGMENTED structures perform at least as good as, but in most cases better than, the NONE LAZY strategy. Especially when the number of stable attributes increases, the CLUSTERED and FRAGMENTED strategies benefit from separating stable and degradable attributes. Using these plots, we also observe that the cost comparisons between the CLUSTERED and FRAGMENTED strategies are not related to the number of stable attributes.



a Inserts



b Degradation



c Sequential scan

Figure 5.7 Insert, degradation and sequential scan cost with varying number of *stable* attributes. For the CL,CE,FL and FE strategies, the degradation cost remains constant (see plot b). The insert and scan cost increases proportional for each strategy (see plot a and c). An exception is the NE strategy (see plot a), where the insert cost increases faster.

Conclusions

When we only vary the number of stable attributes, and use the parameters listed above, the FRAGMENTED EAGER strategy performs best, thanks to its degradation performance. This confirms the prediction we made in section 4.3.1. However, the query cost and the insertion cost do play an important role; we do not take query cost into consideration, although we can see that for sequential scans, as expected, the FRAGMENTED EAGER strategy has the highest cost (see figure 5.7c).

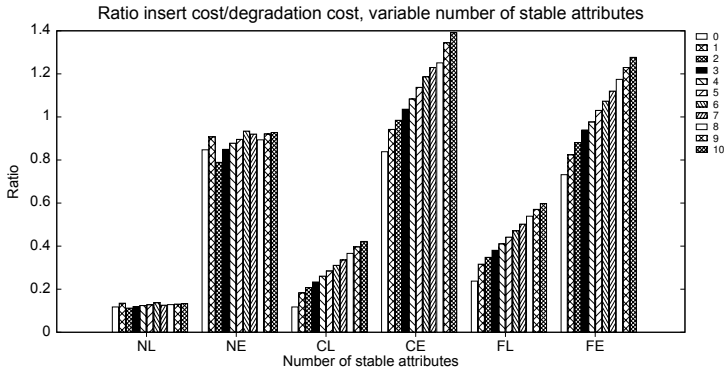
5.3.3 Varying the number of degradable attributes

By varying the number of degradable attributes we increase the degradation load; each additional degradable attribute adds one additional degradation step to the life-cycle policy.

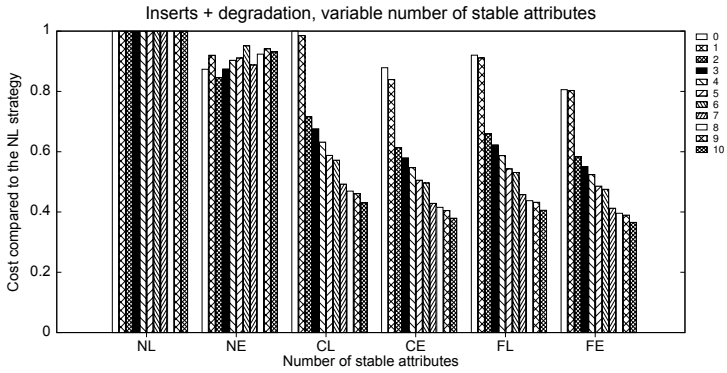
Expectations

We expect that for the different types of storage structures, varying the number of degradable attributes has the following consequences:

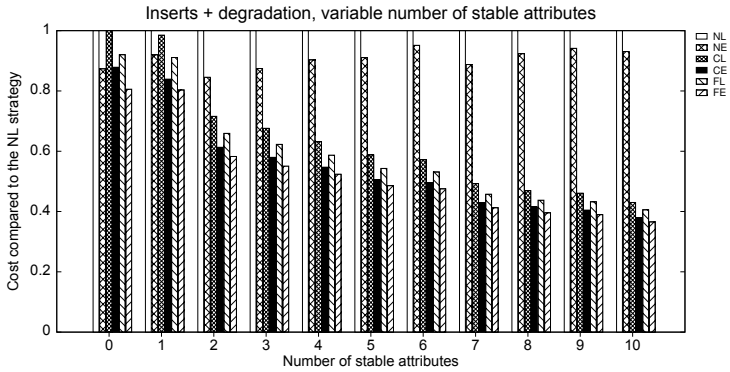
- NL Adding a degradable attribute has the same effect as adding a stable attribute: the amount of data to be transferred at each degradation step increases. Hence, besides the additional cost of the additional degradation step(s), the cost of all other steps also increases.
- NE The effect is the same as for the NONE LAZY strategy: more data has to be processed per degradation step. Moreover, the additional tuple states lead to additional data files, leading to an increase of insertion cost, which is more than linear.
- CL The increase in tuple size is the same as for the NONE LAZY strategy, and therefore the increase in degradation cost has the same cause. The increase of degradable attributes leads to an increase of cost *per degradation step*.
- CE As for NONE EAGER, the insert cost will increase fast because of the additional redundancy caused by an additional degradable attribute. Also degradation time will increase in a similar fashion as NONE EAGER.
- FL Because of fragmentation, an additional degradable attribute has no effect on the cost of each degradation step. The increase of degradation cost is only due to the additional degradation steps. The cost of inserts will increase faster than for NONE LAZY and CLUSTERED LAZY, since an additional degradable attribute requires an additional data file, and thus additional random I/O.
- FE Again, the additional cost for degradation only comes from the fact that there are more degradation steps to be executed. An additional



a Ratio inserts/degradation with varying number of stable attributes.This plot shows that, for the CLUSTERED and FRAGMENTED strategies, insert cost gets the upperhand (the ratio increases) when more stable attributes are added.



b Insert and degradation are added, and displayed relatively to the NL strategy. Again, CL,CE, FL and FE perform better relatively to NL when the number of stable attributes increases.



c The same values are plotted as in (b), but with the strategies grouped together. The FE strategy performs best.

Figure 5.8 Comparison of strategies with respect to insert and degradation cost, when not taken the query cost into consideration.

degradable attribute has no effect on the cost per degradation step. For insert time, the negative effect of an additional degradable attribute is multiplied by the amount of degradation steps per attribute; each degradation step requires an additional data file. Therefore the insert cost will increase faster than for FRAGMENTED LAZY.

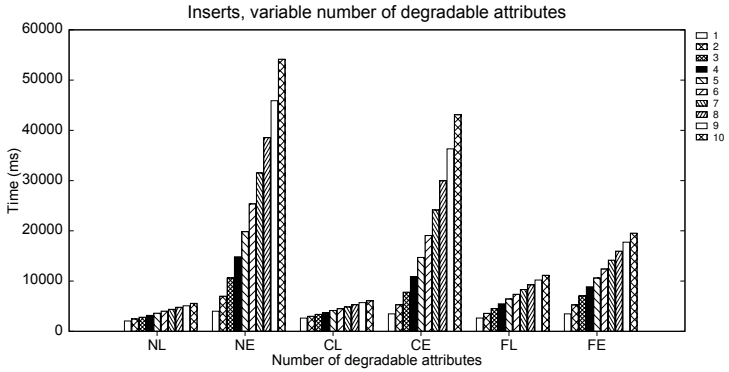
Results

If we look to figure 5.9a we see that the NONE EAGER strategy suffers most from the additional degradable attribute with respect to the insert cost. Again the NONE LAZY performs best when it comes to inserts, which is as expected. The CLUSTERED LAZY performs slightly less than NONE LAZY due to the additional random I/O required to store the stable attributes, but the gradient with which the cost increases is equal to that of NONE LAZY. The FRAGMENTED LAZY strategy performs relatively good with respect to inserts, with performance scaling less good with the number of degradable attributes than NONE LAZY and CLUSTERED LAZY, but better than FRAGMENTED EAGER, and much better than CLUSTERED EAGER and NONE EAGER.

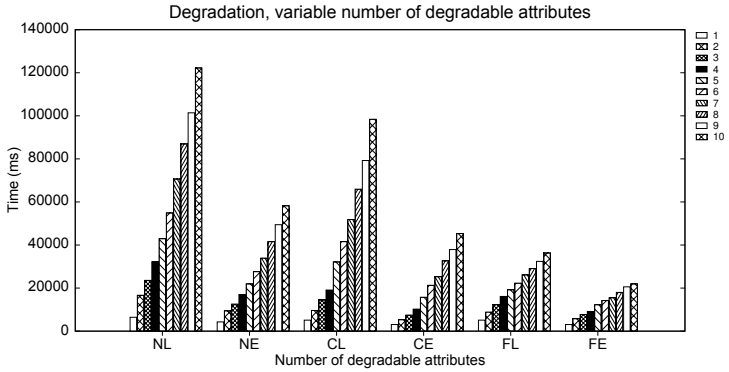
As expected, the NONE LAZY strategy performs worst with respect to data degradation, and the FRAGMENTED EAGER strategy performs best, and also scales best, together with the other fragmented strategy. The CLUSTERED EAGER and NONE EAGER strategies perform good too, but scale less good than the FRAGMENTED LAZY strategy, making the latter the better choice when the number of degradable attributes increases. The CLUSTERED LAZY strategy performs relatively bad, but at least better than NONE LAZY.

As said before, combining inserts and degradation (figure 5.10b and figure 5.10c) is dangerous, although it gives some indication of the performance given those two dimensions. We see that, with the settings we used here, the FRAGMENTED EAGER strategy is the best *all-round* strategy (thanks to its degradation performance), followed by CLUSTERED EAGER (when the number of degradable attributes is small) and FRAGMENTED LAZY (when the number of degradable attributes is higher).

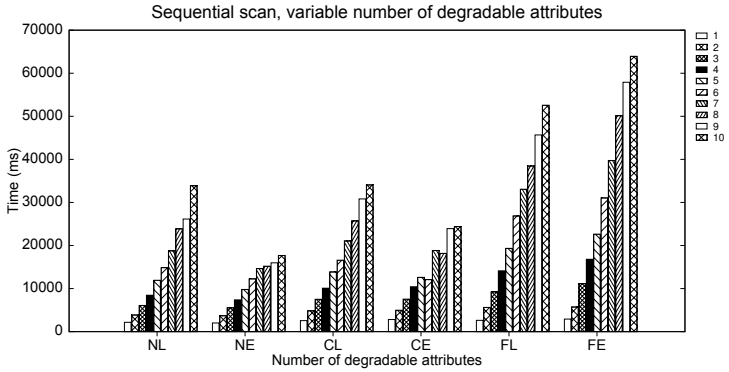
For the FRAGMENTED EAGER strategy, the ratio between inserts and degradation cost stays more or less constant given the number of degradable attributes, indicating that those strategies scale in a similar fashion with respect to the insert and the degradation cost. Especially for the LAZY strategies, they scale less good with respect to degradation cost than to the insert cost. This is mainly because additional degradable attributes have only a limited effect on the insert cost, while it has a severe effect on the degradation cost. See figure 5.10a.



a Inserts

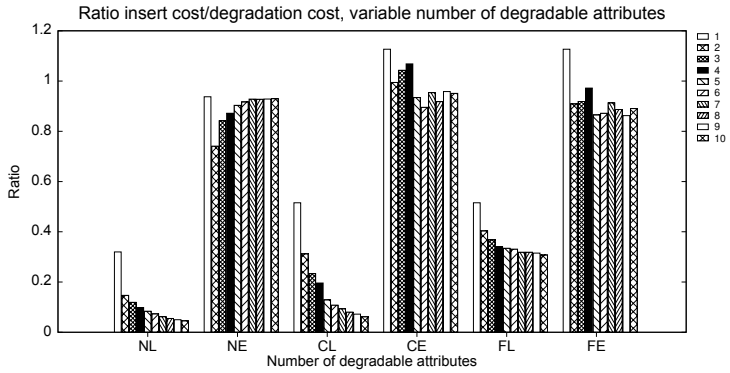


b Degradation

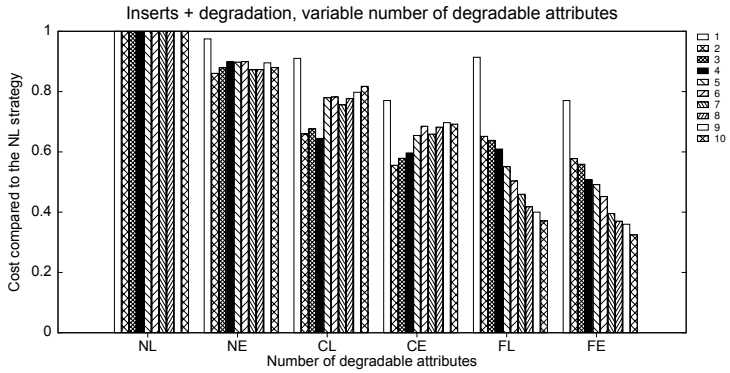


c Sequential scan

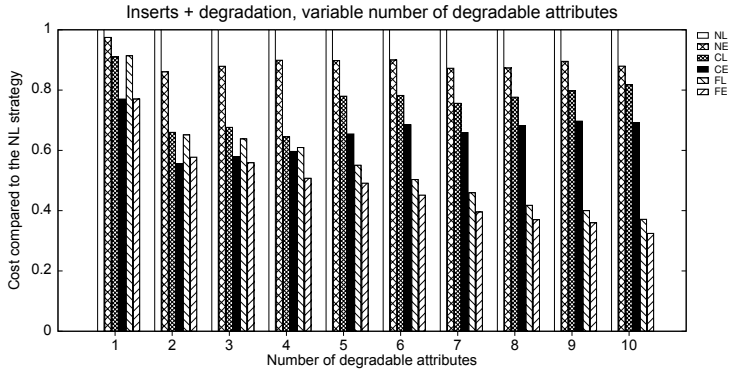
Figure 5.9 Insert, degradation and sequential scan cost with varying number of *degradable* attributes.



a Ratio inserts/degradation with varying number of degradable attributes.



b Insert and degradation are added, and displayed relatively to the NL strategy.



c The same values are plotted as in (b), but with the strategies grouped together.

Figure 5.10 Comparison of strategies with respect to the insert and degradation cost. Interesting to see is that the LAZY strategies scale better with respect to the insert cost (insert cost increases slower than the degradation cost), while for the EAGER strategies the degradation and insert cost increase proportionally (see plot a). The FRAGMENTED EAGER strategy performs also best if we take inserts and degradation together (plot b), followed by the FRAGMENTED LAZY strategy, but only when the number of degradable attributes increases.

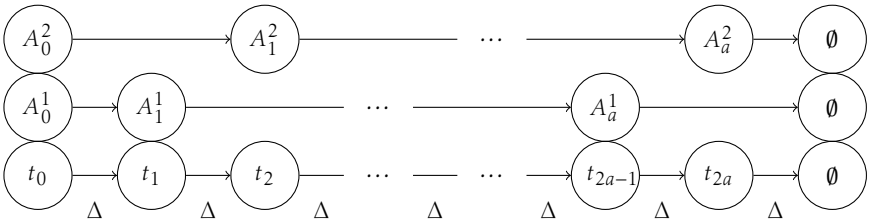


Figure 5.11 The tuples undergo $d \times a$ updates before they are fully removed, where a is the number of degradation steps per attribute. Each tuple state duration is Δ seconds, there are two degradable attributes ($d = 2$).

Conclusions

As expected, the LAZY strategies scale better in terms of insert cost, and the EAGER strategies in terms of degradation cost. When the number of degradable attributes increases, and thus the degradation load gets higher, FRAGMENTED EAGER is the best strategy.

5.3.4 Varying the number of degradation steps

Another way to increase the degradation load is to increase the number of degradation steps per attribute. However, the consequences for the different strategies will be different, especially for the EAGER strategies. In the following we will investigate the performance of the different strategies with a life-cycle policy as shown in figure 5.11, with $d = 2$ degradable attributes, and a degradation steps *per* attribute. We let a vary over $[1 \ldots 10]$, where $a = 1$ indicates that both degradable attributes will be removed after retention period Δ, without undergoing a degradation step.

Expectations

As is the case with increasing the number of degradable attributes, by adding degradation steps the number of tuple states increases. Hence, the expectation is again that the CLUSTERED and FRAGMENTED strategies will perform better than the baseline strategies, because the cost per degradation operation is lower. Adding a tuple state results for all EAGER strategies in an additional data file, which has an immediate effect on insert performance. More precisely, the foreseeable consequences for each strategy are:

NL Since the strategy is lazy, and thus no future states are pre-computed, the number of degradation steps will not have any influence on insert performance. Degradation performance will decrease fast because of

the relatively high cost per degradation step, so any additional step will have a severe effect on degradation performance.

NE For each tuple state a data file will be used, so for two degradable attributes, adding a degradation step results in two additional data files. The NONE EAGER strategy causes the highest amount of redundancy, so insert cost will increase fast. Degradation cost *per* degradation step will remain stable, so the only decrease in performance will be because of the additional steps themselves.

CL As with all lazy strategies, insert performance will not be influenced by additional degradation steps. Degradation performance will decrease, similar to the NONE LAZY strategy, though slightly less due to the separation of stable and degradable attributes.

CE The strategy is EAGER, and therefore insert performance will decrease, though less severe than for the NONE EAGER strategy, because there is less redundancy. Degradation performance will be better than the previous strategies.

FL Fragmentation itself will not lead to a decrease of performance, since the amount of fragments is only dependent on the number of degradable attributes. Hence, insert performance will be constant for all numbers of degradation steps. Also here the degradation performance per degradation step is independent of the number of degradation steps.

FE For each attribute state there will be one data file, so insert performance will decrease with each additional degradation step. Since there is no redundancy, insert performance will decrease less fast than for the other EAGER strategies. The strategy should perform best in terms of degradation performance.

In this experiment, we include the *limited retention* case, where the number of degradation steps a is one, indicating that all data will be immediately removed after the retention period Δ. For this case the benefit of the eager strategies compared to the lazy strategies will be negligible, since there is no need for read operations to degrade the data. Moreover, there will be no benefit for fragmentation, because all data has to be removed at the same time, making clustered strategies a better choice.

Results

What we can see in figure 5.12a is that for the LAZY strategies indeed the insert performance is not related to the number of degradation steps, and stays constant. The insert cost increases for the eager strategies, the NONE EAGER strategy performs worst, followed by CLUSTERED EAGER and FRAGMENTED EAGER. The fact that NONE EAGER and CLUSTERED EAGER perform less good than FRAGMENTED EAGER is due to data redundancy.

Figure 5.12b shows, like in all previous experiments, that NONE LAZY performs worst, and FRAGMENTED EAGER performs best when it comes to data degradation. What should be noted however, and this effect can be best seen from figure 5.13a, is that when $a = 1$, there is no difference in performance between the LAZY and EAGER versions of the strategies. This is because only deletes have to be performed, and no intermediate degradation steps. When we combine the insert and degradation cost, we see indeed that the FRAGMENTED strategies have the same performance, and that this performance is lower than the CLUSTERED strategies. As said before, this behavior can be explained by the fact that all data has to be removed at once, an operation which performs best when all data that is to be removed, is stored in a single data file.

When $a > 1$, we see that the FRAGMENTED EAGER strategy wins again; for the used settings, the degradation cost is more important than the insert cost, and therefore the strategy with the best degradation performance wins. What is interesting to see is that the CLUSTERED strategies perform better than FRAGMENTED LAZY when $a = 1$, but FRAGMENTED LAZY scales better on the number of degradation steps. Looking to the ratios between inserts and degradation in figure 5.13a, we see that for the CLUSTERED and FRAGMENTED strategies the degradation cost is at least as high as the insert cost when $a = 1$, while this is not the case for the baseline strategies, showing again that those CLUSTERED, and especially FRAGMENTED strategies are not suitable for implementing limited retention. Note that NONE LAZY is the fastest strategy concerning inserts; if the degradation cost is not high, which is the case when $a = 1$, the insert cost is the most important factor.

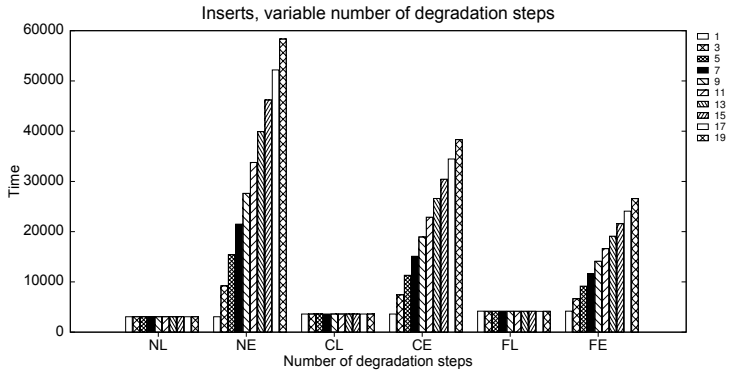
Conclusion

Although for the LAZY strategies, the insert cost is independent of the number of degradation steps and thus those strategies scale good with respect to the insert cost, the EAGER strategies scale slightly better with respect to the degradation cost. However, the degradation cost has the higher impact (when $a > 1$), making that in the end there is almost no difference between them. The fragmented strategies perform better than the clustered strategies, especially when the number of degradation steps increases.

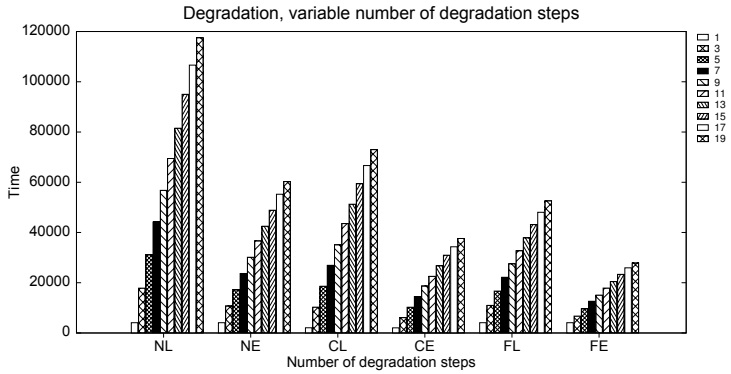
5.3.5 Overall conclusions

The experiments show the effect on performance when a particular parameter is varied. Those experiments validate the hypothesis of section 4.3.1. We repeat the main conclusions:

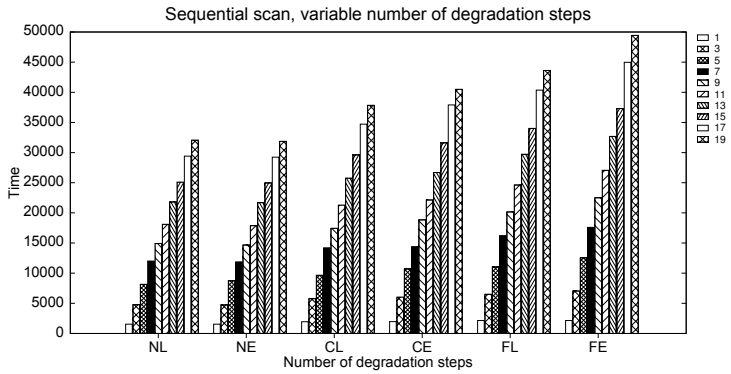
- Strategy FRAGMENTED EAGER scales best with the degradation load, but is bad for inserts.



a Inserts

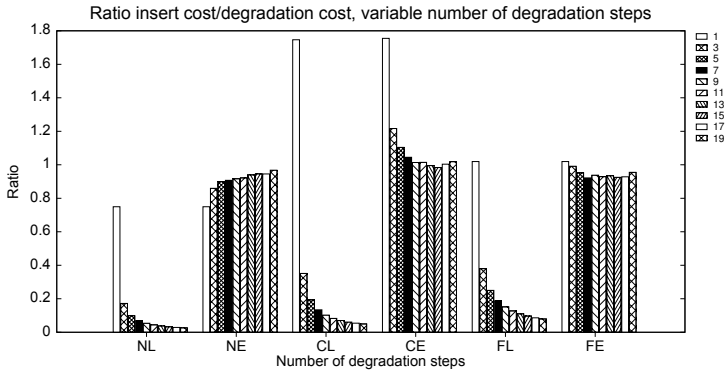


b Degradation

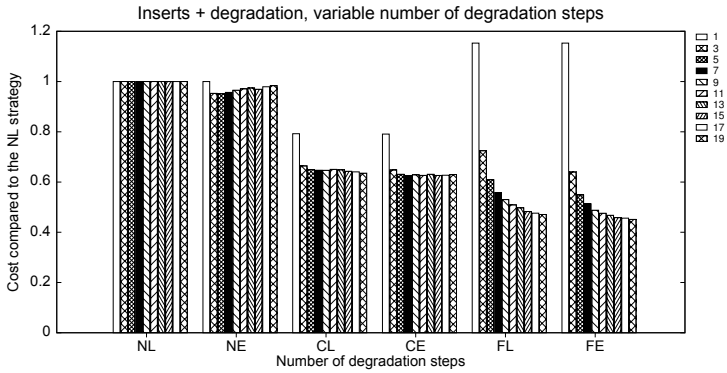


c Sequential scan

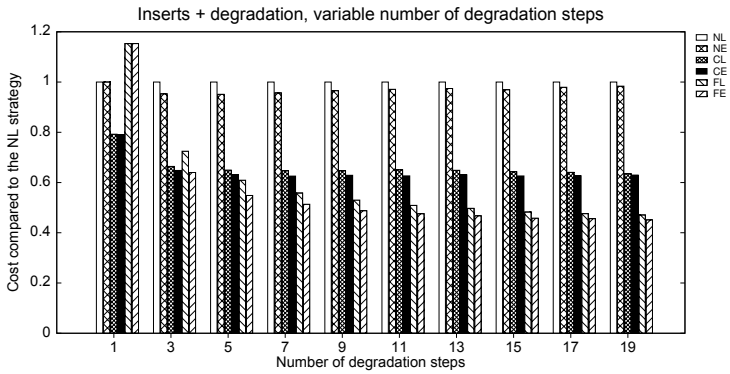
Figure 5.12 Insert, degradation and sequential scan cost with varying number of degradation steps per attribute.



a Ratio inserts/degradation with varying number of degradable attributes.



b Insert and degradation are added, and displayed relatively to the NL strategy.



c The same values are plotted as in (b), but with the strategies grouped together.

Figure 5.13 In the limited retention case ($a = 1$), fragmented strategies perform much less than clustered strategies. For $a > 1$, fragmented strategies are better. The number of degradation steps does not lead to performance differences between lazy and eager strategies.

- Strategy NONE LAZY scales best with inserts, but the decrease in degradation performance is in general higher than the increase of its insert performance.
- When the insert cost grows faster than the degradation cost, the LAZY strategies can be a better alternative to the EAGER strategies; however, the insert cost and degradation cost are in general proportional to each other. A reason for the relatively higher insert cost might be, for example, expensive integrity checks which have to be performed before inserting a value in each data file.
- Under high query load, clustered strategies perform better than the fragmented strategies.
- When a database system only has to support limited retention, and no intermediate data degradation steps, a strategy other than the baseline strategy is not recommended. If for some reason another strategy has to be chosen, the CLUSTERED strategies perform better than the FRAGMENTED strategies in this case.

Basically, the experiments show that it is indeed useful to implement data degradation using fragmented storage structures. However, the performance gain compared to that of the baseline (NONE LAZY) strategy is not very high; the most important factor which makes that data degradation does not lead to a huge performance loss is that data is ordered on degradation time, and not in B+tree-like structures (see section 4.3.1). This can be easily concluded from the fact that the cost for degrading a single tuple using such a structure is in the order of $O(log(n))$, compared to $O(1)$ for a *set* of tuples for degradation friendly structures; therefore we did not implement B+tree structures for an experimental comparison.

5.4 Degradation-friendly indexes

In this section we analyze the various indexes we discussed in section 4.3.2. Our target is to make suggestions, analogously to our discussion of the storage structures, about which index to use in which situation. Apart from the usual considerations concerning the choice of an index—the balance between query and insert load—we will take into account that data is subject to data degradation. Hence, we will investigate how suitable an index is given the following characteristics:

- *Cardinality of the domain*, i.e., the amount of distinct values that an attribute possibly can have (we consider only degradable attributes). When a generalization tree is used to determine the degradation path of the attribute, this domain is limited and for the most accurate attribute equal to the number of leaves in the tree. For each degradation step, the cardinality of the domain decreases until $C = 1$ when the

attribute is fully degraded. We assume that when C is low, a point query will result in a large result set; the selectivity of a query is lower when C is low than when C is high.

- *Degradation load*. The higher the degradation load, the more important a degradation-friendly index is. For some indexes, a balance can be made between query efficiency and update (degradation) efficiency.

In the following we limit the discussion to point queries, also known as equality queries. Typically, the number of results will depend on the amount of tuples N and the cardinality of the domain C. In our context, a query will result always in $\frac{N}{C}$ results. When we assume that the retention period of precise tuples is shorter than that of less precise tuples, the amount of tuples in a precise tuple state set will be lower than in a more degraded tuple state set. Moreover, because the cardinality of the domain of a set containing highly precise tuples will be higher, the number of tuples in the result of a query on precise data will be smaller than that of a query on less precise data.

5.4.1 B+tree

The cost of traditional B+trees concerning inserts and queries has been studied thoroughly in literature and therefore does not need much discussion; for a complete study we refer to [99, 43]. Here we limit ourselves to the observation that since the index is tree-based, accessing a particular attribute value will take a number of random I/O which is dependent on the depth of the B+tree. This cost counts for both accessing the value for answering a query and for updating or removing the value. We disregard the additional cost needed when the tree needs to be reorganized.

Although the cost for inserting values into the index is high (typically one or more random I/O per tuple), the use of this index can highly reduce query cost when the cardinality C is high, and thus the query selective. For lower cardinality domains, the cost for maintaining the index might be too high compared to the query cost reduction.

When we use encryption, as suggested in section 4.3.2, data degradation comes for free, since only decryption keys have to be removed. However, this assumes that removal of <value,pointer> pairs from the tree can be done using a lazy method, or that the <value,pointer> pairs do not have to be removed at all. However, at each lookup in the tree, it is necessary to test the existence of a valid encryption key to make sure the <value,pointer> pair does not belong to a removed tuple. Sometimes the index is used to answer queries without requiring accessing the data files themselves; in such situations, it is important that the B+tree does not contain invalid <key,pointer> pairs.

The comparison of the B+tree index to other indexes is done in section 5.4.5.

5.4.2 Bitmap

In the most simple form, a bitmap index is a set of bit vectors, where each bit vector represents one of the values v in the domain of an attribute. For each position i, the tuple at position i in the data file contains v if, and only if, the bit vector for v has a 1 at position i (see figure 4.11 in section 4.3.2).

In general, a one-component index, where each bit vector represents one value in the domain of the indexed attribute, is space-inefficient but time-efficient; see [34]. The more components (and thus the less bit vectors needed to represent all values in a domain), the more space-efficient, and less time-efficient.

To be precise, for a base-$< b_n, b_{n-1}, \ldots, b_1 >$ bitmap only $\sum_{i=1}^{n-1} b_i$ bit vectors are needed and those bitmaps are therefore more space-efficient than one-component bitmaps, in which the single component contains much more bit vectors. For example, when the cardinality of a domain is $C = 26$, a $< 7, 4 >$-bitmap requires 11 bit vectors compared with 26 in case of a one-component bitmap. However, finding the positions of a value requires also $n = 2$ scans, making those indexes less time-efficient. In general, Chan et al. [33] conclude:

Time optimal: 1-component index with base sequence $< C >$. Only one scan is needed, but the space consumption is highest ($C \times N$ bits).

Space optimal: $\lceil \log_2 C \rceil$-component index with base sequence $< \ldots, 8, 4, 2 >$. $\lceil \log_2 C \rceil$ scans are needed, whereas the space consumption is $\lceil \log_2 C \rceil \times N$ bits.

Time / space optimal: 2-component index with base sequence $< b_2 - \delta, b_1 + \delta >$, where:

$$b_1 = \left\lceil \sqrt{C} \right\rceil, b_2 = \left\lceil \frac{C}{b_1} \right\rceil, \delta = \left\lfloor \frac{b_2 - b_1 + \sqrt{(b_1 + b_2)^2 - 4C}}{2} \right\rfloor \tag{5.1}$$

Note that the base of the time/space optimal index is an *approximation*; for details and a comprehensive study about finding this optimum we refer to Chan et al. [33].

In the context of data degradation, it is important to take into account the number of data files needed for the index, because this gives an indication of the cost to update it. The space optimal index requires less vectors; considering that we have to store each component (or even each bitmap) in a separate data file—although it is possible to combine components, or the whole index, into one single data file, which we ignore in this

discussion—this type of bitmap index will be most efficient for data degradation. However, since the amount of components, and thus the amount of scans needed to answer a query, is higher, this index will be less efficient for answering queries. Hence, the one-component is time optimal (only one scan needed for answering questions), but less efficient for data degradation (more bit vectors or components to update). The best trade-off can therefore most probably be found by using the time/space optimal index, which is a 2-component index.

5.4.3 Bloom filter index

A Bloom filter is an array of bits which are initially set to *zero*. Hash functions are used to set k positions in the Bloom filter to *one*, making it possible to check if an element is possibly in the data set, or certainly not. Because hash functions are used, false positives can occur when the index is queried. See for a more detailed description section 4.3.2.

As for any index, for Bloom filters it is important to know how costly it is to update the index, especially in the context of data degradation. As for bitmap indexes, this cost depends on how many files need to be accessed. In the case of Bloom filters, to determine the number of files needed for the Bloom filter, we have to make a trade-off between the acceptable number of *false positives* and update cost. The number of false positives determine the effectiveness of the index for answering queries, and has to be limited, especially when query loads are high. For simplicity reasons, the following cost estimation is based on a situation where one bloom filter *per tuple* is used. When one bloom filter per page would be used, the estimation of false positives would become more complex; the number of files to be managed would not change, although the size of each file would be smaller.

When the length m of each Bloom filter (see also figure 4.14) is small, less data files have to be updated, limiting the amount of costly random I/O. However, when m is small, the domain of the hash functions is also small, so that the chance of false positives will be higher. This can only be partially solved with increasing the number of hash functions, assuming that using more hash function does not lead to an unacceptable computational overhead [38]. False positives lead to more I/O at query time since tuples are fetched which don't belong to the query result.

To find the relation between false positives and the length m of the Bloom filters, we consider the following:

- The number of possible Bloom filters of length m containing k ones and $m-k$ zeros is $\binom{m}{k}$. So, if we take two different attribute values, and assume that the hash functions are fully random, the chance that the same k bits are set is $1/\binom{m}{k}$.
- The chance that an arbitrary attribute value is *not* equal to another

attribute value is $\frac{C-1}{C}$, where C is the cardinality of the attribute domain.

Hence, the chance of a false positive (two values are different, but the index returns a positive) is:

$$P(\textit{false positive}) \quad = \quad \frac{1}{\binom{m}{k}} \times \frac{C-1}{C}$$

Then the expected number of false positives when querying a dataset with N tuples is:

$$N \times \frac{1}{\binom{m}{k}} \times \frac{C-1}{C}$$

Note that this is only valid under the simplifying assumption that each hash function sets a different bit of the Bloom filter, which in practice cannot be guaranteed due to the limited size m.

Using the above equation, we conclude that with $m = 40$, $k = 3$, $N = 1,000,000$, $C = 10,000$ there are about 100 false positives. However, we can calculate that there are also 100 *true* positives given the size of the domain and the number of tuples and assuming that the attribute values are uniformly distributed over the dataset. This number of true positives is already too high to justify the use of an index since scanning the whole table using a sequential scan will be much cheaper. Hence, with lower cardinalities, an index is only useful when the selection predicate is a conjunction of equations—for example, SELECT * FROM R WHERE $A = a$ AND $B = b$—so that the number of true positives will be much lower. In such a scenario the number of false positives is also automatically reduced, because the chance of a false positive for the same position in both indexes is small.

Unlike Bitmap indexes, the number of data files m is independent of the cardinality C (apart from keeping the number of false positives low). This makes an index using Bloom filters interesting for larger domains. We give a full comparison and indications for the applicability of the indexes later, in section 5.4.5.

5.4.4 Hash Sequential list

A hash sequential list index (HSL) consists of a set of hash buckets containing value/pointer-pairs, which are ordered on insert time. The bucket in which a value will be placed is determined by a hash function on that value; a bucket can contain different values. Buckets can be maintained as heap files, in a similar fashion as the sequential data structures discussed before. Hence, multiple tuples can be removed using one I/O operation per hash bucket, making the index interesting for data degradation.

The efficiency of the HSL index depends on the size and number of buckets. However, there is a clear balance between query efficiency and

degradation efficiency: if the buckets are large, more data has to be accessed to answer a query. If the buckets are small, more random I/O operations are needed to remove data from all buckets, since each bucket will be stored in its own data file. The techniques to manage degradation, as discussed in section 4.3.1, can be applied to degrade data from the buckets.

5.4.5 Comparison of indexes

Now we have discussed four types of indexes, we will compare them and indicate:

- Which index is most suitable to use, at which level of precision?
- Which index performs best under which condition with respect to query and insert load, taking data degradation into account.

To answer the first question we have to look at how dependent the indexes are on the cardinality of the domain, in terms of insert, degradation and query cost. B+trees are little dependent on this cardinality, and are therefore suitable for high cardinality domains—typically the lowest level of the generalization tree. The hash sequential list index can also manage high cardinality domains; the hash buckets will only be shared by more different values. Hence, both indexes are good candidates for data with high precision.

Bitmap indexes do not scale well in terms of the cardinality of the domains they can handle. We saw that for an one-component index, we have to manage C data files if we choose for the space-optimal index, 'only' $\lceil \log_2 C \rceil$ data files, and if we choose for the time/space-optimal index the number of data files will be—see section 5.4.2 and equation 5.1—$\sum_{i=1}^{n-1} b_i \approx \left\lceil \sqrt{C} \right\rceil + \left\lceil \frac{C}{\sqrt{C}} \right\rceil = 2 \times \left\lceil \sqrt{C} \right\rceil$. When the cardinality is high, more bit vectors have to be managed which makes the index less efficient for inserts and degradation.

Bloom filters are hardly dependent on the cardinality. Still, when the cardinality is lower, the efficiency of the index will suffer from the higher amount of false positives, which depends on the size of the dataset N, the choice of the size of the Bloom filters m, and the number of hash functions. The amount of false positives is:

$$N \times \frac{C-1}{\binom{m}{k} \times C}$$

So, for lower cardinality domains, to keep the number of false positive limited, we have to maintain more data files, which is more costly for data degradation.

Taken those characteristics into account, we conclude that B+trees (with encrypted pointers) and hash sequential list indexes are good for high

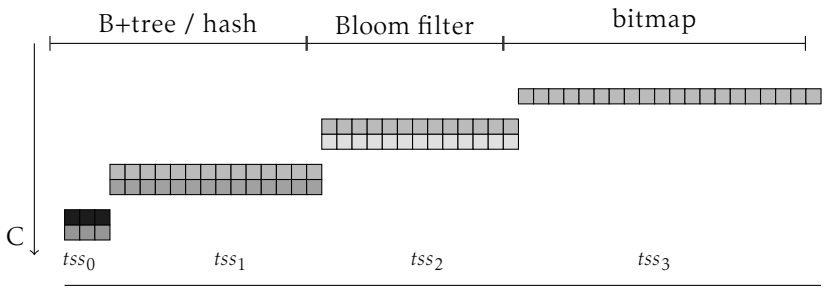


Figure 5.14 Choice of indexes given the precision (measured in the domain cardinality C) of the data. Where the data would normally have been indexed using one single index per attribute, with data degradation we choose a suitable index for each tuple state set.

cardinality domains—the lower level of the generalization tree where data is highly precise—Bloom filter indexes for medium cardinality domains, and bit map indexes for the low cardinality domains—where the data is highly degraded.

Traditionally, the use of an index, and the choice of an index is always based on the expected insert and query load. In our context, degradation adds an additional parameter which has to be taken into account. Especially when there are many degradation steps, the degradation-friendliness is important.

Basically, indexes based on sequential ordering of the data file on insert and degradation time are well suited for degradation. Hence, those indexes—especially bitmaps and Bloom filter indexes—are a good choice when inserts play an important role, while the cost for data degradation can be limited. The low query cost of B+trees, especially when the queries are highly selective, make that this index is interesting when query load is high compared to the insert load. Moreover, thanks to the encryption technique, data degradation comes for free. With the hash sequential list, a good balance is possible between query, insert and degradation cost by playing with the size of the hash buckets. Hence, this index can be considered as a good 'all-round' index. For a schematic overview of this conclusion, see figure 5.14.

5.5 Conclusion

In chapter 4 we identified the technical implications of data degradation on traditional database management systems, and proposed solutions to speed up data degradation. In this chapter we evaluated the proposed storage structures using a prototype implementation, and the degradation-friendly

indexes using an analytical study.

The prototype facilitates experimental evaluation of the storage structures, so that a comparison of the storage structures is possible. The prototype cannot be used to show the impact of data degradation compared to traditional database systems without data degradation. The comparison showed that using fragmentation, combined with an eager storage strategy results in the best trade-off between insert and degradation cost, especially when degradation load is high due to a high amount of degradation steps. The performance study also showed that when only limited retention is used, and no intermediate degradation steps, the baseline strategy performs almost as good as all other strategies. Moreover, the main performance gain is possible because all storage structures are based on the *heap file* structure; the discussed variations of the basic heap structure are useful to tweak performance given specific application requirements.

The analysis of index structures showed that for different levels of precision, different indexation strategies are useful. For high cardinality domains, where the precision of data is high, the B+tree index with encrypted pointers is the most obvious choice, together with the hash sequential list index. For intermediate levels of precision, the Bloom filter index is the best choice, and for the least precise data bitmap indexes are most applicable.

The prototype we built from scratch is already capable of testing the storage structures. A more extensive performance study could include an implementation of the index structures; the prototype is modular enough to include this. Moreover, the prototype can be extended to test the transaction synchronization protocol, and the logging mechanism. Such a prototype can show the feasibility of data degradation, as a proof-of-concept that data degradation can indeed be applied in practice.

As a final remark, we recommend that data degradation should be implemented in a traditional database system, which we therefore consider as an important future work direction. This would make it possible to measure the performance loss of data degradation in a real-world setting.

Future research directions

6

Data degradation is an interesting approach within the field of privacy-aware data management, and necessary to lower the impact of unauthorized data disclosure. The approach is new, and therefore opens up many interesting research directions, especially on how to use the concept of data degradation in practice.

We describe some instantiations of the degradation model. For two of those models we do this in more depth, to provide a strong foundation for future research on this topic. In the first model, the *service-oriented model* (section 6.1), we drop the statement that privacy and usability should be negotiated, and put the service-provider in control of specifying which data it needs for which purposes. Data degradation will make sure that no data is kept in the system with a higher precision than required to fulfill those purposes.

The second model, the *ability-oriented model* (section 6.2), goes one step further; here the objective is to manipulate the data such that the data can only be used for a limited set of queries.

We conclude with some additional instantiations and usages of data degradation, paving the way to new privacy solutions based on the limited retention of data, in section 6.3.

6.1 Service-oriented data degradation

A single data item is often collected for multiple purposes; we assume that a single *service* is assigned to fulfill one of the purposes, and all purposes are assigned to a service. To fulfill its purpose, a service requires the data to have a specific level of precision. In the service-oriented data degradation model, the objective is to store each data item with the lowest precision needed, such that each service can fulfill its purpose. Once one of the services has fulfilled its purpose, the precision needed to serve all other services will be evaluated again. If the precision can be decreased, the data item will be degraded.

6.1.1 Formalization

We assume there is a set *Service* of services. At any time, a subset is allowed to access a particular data item. For each service A, the level of precision required by that service, is denoted with $Prec_A$. At some point in time, a service should *not* be able anymore to use the data item, typically when its purpose has been fulfilled. We name the condition determining that a service A can no longer use the data item the *destruction condition*, denoted by A_{DC}. Hence, at any point in time, there is a set of services for which the destruction condition has not become true yet; we call this set the *state*. So, we define $State = \mathbb{P}\ Service$ and the effect of the event *that a destruction condition A_{DC} becomes true* on a state S, is a transition from S to $S \setminus \{A\}$.

The above description can be phrased in terms of automata theory [82]. A control automaton is a deterministic finite automaton responsible to choose the next *level of precision* given the *current* level of precision and an *event*. Every data item will be controlled by its own control automaton; each data item is at any time in one of the states of its automaton.

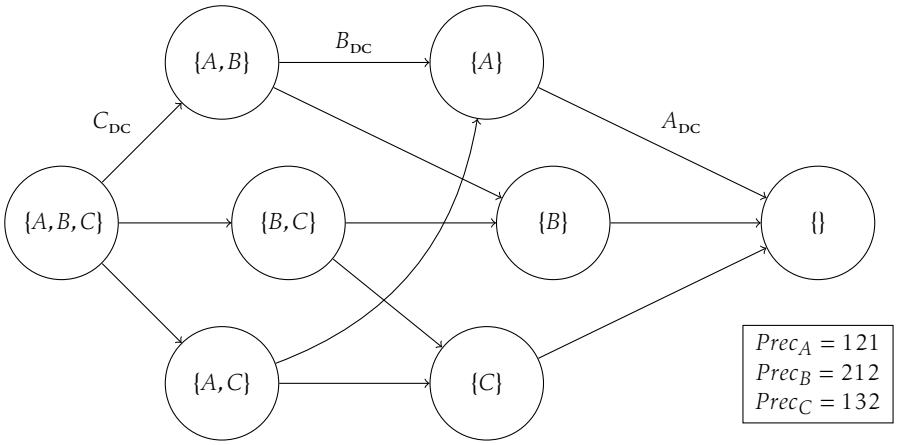
A control automaton is defined as follows:

$$
\begin{array}{lcl}
Control & = & < State, Event, transition, init, final > \\
\quad\text{where} & & \\
State & = & \mathbb{P}\ Service \\
Event & = & \{A_{DC} \mid A \in Service\} \\
init & = & Service \in State \\
final & = & \{\} \in State \\
\quad\text{and} & & \\
transition & : & State \times Event \rightarrow State \\
transition(S, A_{DC}) & = & S \setminus \{A\}
\end{array}
$$

Now, the service-oriented degradation model requires that for each state S, the system has to have the data available in precisely the level of precision needed—and not more—for the services in S to fulfill their purpose. Hence, the level of precision of a state S is defined as:

$$\text{Level of precision of state } S = \prod_{A \in S} Prec_A \tag{6.1}$$

The control automaton itself is fixed; it is *independent* from the services and can be reused for any application with services requiring different levels of precision. In figure 6.1a we give the control automaton for a set of three services A,B,C. When the precisions of A,B, ... are known, the levels of precision belonging to each state can be filled in, with a transition table as shown in figure 6.1b as a result. As in section 3.3, the level of precision is



a Control Automaton for 3 applications.

State	*Event* (DC)	*Next state*
$\mathbb{P}$ *Service*, *Level*	*Service*	$\mathbb{P}$ *Service*, *Level*
$\{A,B,C\},111$	A	$\{B,C\},112$
$\{A,B,C\},111$	B	$\{A,C\},121$
$\{A,B,C\},111$	C	$\{A,B\},111$
$\{A,B\},111$	A	$\{B\},212$
$\{B,C\},112$	B	$\{C\},132$
$\{B,C\},112$	C	$\{B\},212$
$\{A,C\},121$	A	$\{C\},132$
$\{A,B\},111$	B	$\{A\},121$
$\{A,C\},121$	C	$\{A\},121$
$\{A\},121$	A	$\{\},null$
$\{B\},212$	B	$\{\},null$
$\{C\},132$	C	$\{\},null$

b Transition table

Figure 6.1 Example of *the* control automaton for three services A, B and C. In this particular example, the services require precisions 121, 212 and 132 respectively. The control automation, combined with those precision requirements, define a *transition table* which describes the life-cycle of the data. For example, if the control automaton is in state $\{A,B\}$, and the destruction condition of service B is true, the data will be degraded to precision 121.

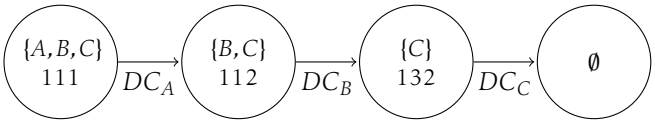


Figure 6.2 A life-cycle from the basic model, now expressed as a control automaton, where degradation times δ_1,δ_2,δ_3 are identified with events A_{DC},B_{DC},C_{DC}.

denoted by a list of digits, each denoting a level of precision for a single attribute. The level of the highest precision is denoted with number 1. The lower a precision, the higher its level is numbered. The transition table gives, for each state and event, the level of precision to which the controlled data item should be degraded.

As an additional, practical consideration, we foresee the benefits of *acquisition conditions*, denoted A_{AC} for a service A, which states under which condition a collected data item can be accessed by a service. An *acquisition condition* has exactly the opposite effect as a *destruction condition*: the initial set of services is taken to be those services A for which the acquisition condition A_{AC} is true.

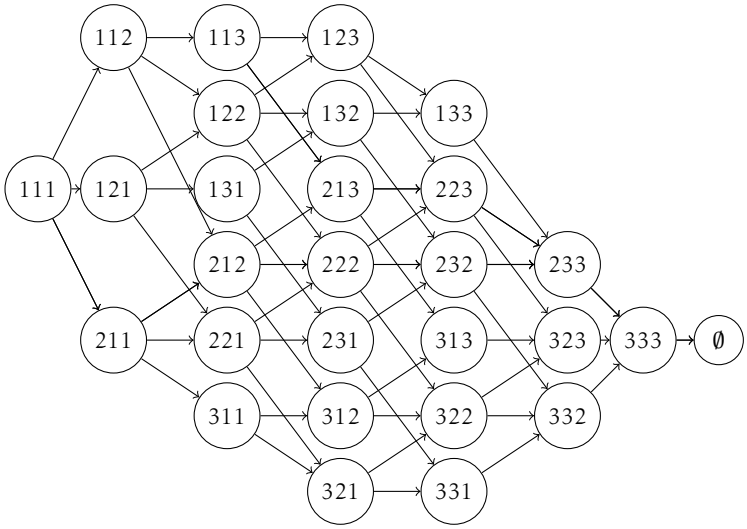
6.1.2 Comparison with the basic model

In the previous chapters, we presented and discussed the basic data degradation model. The main difference of the basic model, compared to the service-oriented model, is that one policy—containing only transitions triggered by time—is specified for all tuples in the database, so that all tuples have the same life-cycle. Note that this model can be expressed using service-oriented model notations. To do so we define that destruction conditions A_{DC},B_{DC}, and C_{DC} become true exactly after time periods δ_1,δ_2 and δ_3. Figure 6.2 shows the resulting control automaton, where all transitions which cannot take place (because the order of transitions is already predefined) are omitted.

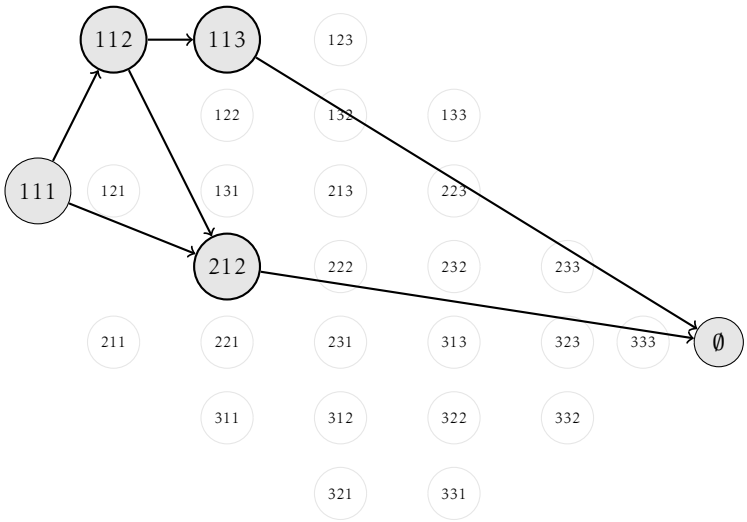
In the service-oriented model, the life-cycle is not known in advance; therefore two different tuples can have two different life-cycles, although the policy itself may be the same. This is because destruction conditions do *not* become true after predefined time periods.

6.1.3 Considerations for implementing the model

In the service-oriented model, the life-cycle of a single data item is regulated by its own control automaton. The automaton takes events as input. Those events signal the fulfillment of the purpose of a service. After such an event, the automaton determines the next level of precision of the data item. Apart from the question how to monitor events (we can simply assume that



a



b

Figure 6.3 For 3 attributes which can have 3 different levels of precision $(1 \ldots 3)$, there are 3^3 different levels of precision possible (a). Given a set of services and their required level of precisions, the life-cycle of a tuple will be a subset of those levels (b). The path which the tuple will eventually follow depends on the acquisition and destruction conditions.

services themselves report that their purpose has been fulfilled), the main difficulty is to manage the life-cycle of each individual tuple. In chapter 4 we already discussed in detail the implementation of the life-cycle of a data item; compared to that work, the service-oriented model introduces two main difficulties:

- The precise life-cycle is not known in advance and depends on the order of events. Managing which tuples have to be degraded using the same degradation schedule is therefore not possible.
- Tuples which have been inserted at the same time do not necessarily degrade at the same time. Optimizations based on the grouped degradation of tuples need to be more sophisticated.

To manage the degradation of each individual tuple, the degradation schedule will be replaced by a control automaton. The following issues, which increase the complexity of regulating the life-cycle of each tuple, come to mind and have to be taken into account:

- *Storage*. The current *level of precision* of a tuple is determined by the set of services for which the acquisition condition was true *and* for which the destruction condition has not become false yet. Since the *order* in which the destruction conditions become true is unknown—unless all destruction conditions are based on a time event—and the acquisition conditions can for example be based on the current database population, the actual *life-cycle* of a tuple cannot be determined in advance. What is known, however, is that the life-cycle (the set of levels of precision of a tuple during its life-time) is a *subset* of all combinations of precision of all attributes (see figure 6.3). Hence, we can make the following observations about the complexity of managing the life-cycle of a tuple:
  - An automaton, controlling the life-cycle of a tuple for n services, consists of 2^n states. However, the size of the used subset of the automaton depends on the number of acquisition conditions which will be false (m). Hence, the actual control automaton regulating the life-cycle of a single data item consists of 2^{n-m} states.
  - The number of *distinct* levels of precision in a life-cycle of a tuple can also be bound by the number of attributes (d) and number of levels of precision (l). The actual number of *possible* states in the life-cycle of a tuple is $min(l^d, 2^{n-m})$. The number of *degradation steps* is limited by the number of services n, and can be less if multiple services share the same required level of precision.

  A possible implementation is to maintain a table containing a copy of each tuple in the data set for all its possible future states. With each event, all versions of the tuple which are in a state which cannot

be reached anymore and for which the precision is higher than the current state, will be removed. However, the amount of additional storage required due to redundancy can be very high, and depends on n, m, l and d (see above); the number of tables to be maintained is $min(l^d, 2^{n-m})$.

- *I/O cost*. Thanks to the simplification we made for the basic model, that all tuples share the same life-cycle, and that transitions can only be triggered by time, updates to a tuple in the *basic model* can be implemented efficiently by storing those tuples close to each other. Although this simplification does not hold anymore in the service-oriented model, we might assume that tuples which are collected close to each other in time, have been collected for similar purposes. If so, those purposes are also fulfilled close in time, making it still useful to store the data ordered by time. Especially when we reuse the ρ-timeliness flexibility, tuples can be marked for degradation, such that as many tuples as possible can be degraded at each degradation interval, sharing I/O costs.

6.2 Ability-oriented data degradation

In the basic and service-oriented models, we looked at the degradation of data as a *natural* process; degradation of data means that the data becomes less precise. The assumption was that data can still be used when it is less precise, although the usability has been decreased. In the service-oriented model, we assumed that services *require* a particular precision, and cannot fulfill the purpose when the data is not precise enough.

In the following, we propose an *ability-oriented* data degradation model; the objective is to degrade the *ability* of what can be done with the stored data, while guaranteeing that the queries required by the services can be performed on the degraded data with the intended result. Other queries which require data which are less manipulated than the allowed queries, are not guaranteed to give correct or meaningful results anymore.

For example, time can be represented in various forms. Time t_1 and t_2 can be represented as '9 a.m.' and '10 p.m' respectively, but also as 'morning' and 'evening', or as '18842' and '23123'. The value 'morning' can be derived from the absolute time value '9 a.m', but the fact that 10 p.m is later in time than 9 a.m. cannot be derived from this time representation, because the date part is missing. The values '188842' and '23123' represent an order number, and therefore can be used to order the two time values.

In the following we introduce the ideas behind the ability-oriented degradation model by means of a formalization of these concepts. This type of data degradation is related to the management of materialized views.

6.2.1 Formalization

In the following we use S to denote a *schema* and R to denote a *relation*. Traditionally, a schema is a set of attributes: $S = \{A_1, A_2, \ldots\}$. Here, we identify a schema with the products of the domains of attributes; so $S = \{domA_1 \times domA_2 \times \ldots\}$, so that R is a subset of S. Tuples t are elements of R or S. A query Q is a function, mapping relations to relations; so its type is $\mathbb{P}\, S \rightarrow \mathbb{P}S'$, where S and S' are schemas. To simplify the discussion, we only consider *bulk inserts*: $R \mapsto R \cup R'$ is a bulk insert, where the newly inserted tuples form the set R'. Both R and R' have to have the same schema S.

Informally, the objective of ability-oriented data degradation is to degrade a data set in such a way, that it is possible to define a query Q' on the degraded data which results in *the same* answer as a specific query Q on the original data. We call such a degraded set a *degraded view* of a relation R. When the degraded view V complies with the above property, it is Q, Q'-*adequate*, which we will define more precisely below. Different queries lead to different degraded views of the original data. To comply with the limited retention principle, only the tuples in those degraded views must be stored, and not the original tuples in R.

Let S and S' be a schema. A degraded view *function* V is, like a query, a function of type $\mathbb{P}S \rightarrow \mathbb{P}S'$. We name the result of a degraded view function on a relation a *degraded view*. We require a degraded view function to be Q, Q'-*adequate*, which we define as follows:

Definition 6.2.1. *Adequacy*. Let S and S' be a schema, and Q,Q' queries on S, and V a degraded view function on S. V is Q, Q'-*adequate* when for all R, R' with schema S:

$$\begin{array}{lcl} Q(R) & = & Q'(V(R)) \\ V(R \cup R') & = & V(R) \cup V(R') \end{array}$$

Note that it is trivial to find, for arbitrary Q, a Q' and a degraded view function V which is Q, Q'-*adequate*, namely $Q' = Q$ and V is the identity function. Nevertheless, the objective of the ability-oriented model is to make the degraded view as minimal as possible, so that it contains the minimum amount of information to be able to produce the result of Q'. In some cases, such a minimal set is the original query Q itself.

However, let V be a degraded view function on R, which is Q, Q'-adequate. Than, updating $V(R)$ is not always possible without the presence of the base relation R. Updating an Q, Q'-adequate degraded view would be easy *if* the base relation R would be available; first R can be updated to R' with the new tuples, after which $V(R')$ can be computed. However, the idea behind data degradation and the limited retention principle is that R itself is *not* available.

We give two degraded view functions that are minimal; the queries involved are the selection and projection. Let S be a schema and S' a subschema of S, and consider relations and queries over S, with P a predicate on S. For every of the following choices of Q,Q' and V we have that V is Q,Q'-adequate:

Q	Q'	V
σ_P	Q	Q
$\pi_{S'}$	Q	Q

For the selection operation, the proof reads as follows; for the projection the proof is similar.

$$\begin{aligned} & Q(R) \\ = & \\ & \sigma_P(R) \\ = & \\ & \sigma_P(\sigma_P(R)) \\ = & \\ & Q'(V(R)) \end{aligned}$$

and

$$\begin{aligned} & V(R \cup R') \\ = & \\ & \sigma_P(R \cup R') \\ = & \\ & \sigma_P(R) \cup \sigma_P(R') \\ = & \\ & V(R) \cup V(R') \end{aligned}$$

However, we make here the assumption that the predicate can be evaluated on the newly inserted tuples, and that the result of this predicate will not change if the predicate would be evaluated later in time, which is in practice not always the case. Take for example the predicate $t > now() - 10 minutes$ on a time attribute. This predicate may evaluate to true at insert time, but to false after 10 minutes.

Finally, we consider a query Q containing the join operator, for example a self-join on attribute U. Let $Q = \pi_{S-\{U\}} \circ (\bowtie_U)$ be the query that maps R to $\pi_{S-\{U\}}(R \bowtie_U R)$. Let $f : U \to U'$ be an injective function for which it is computationally hard to calculate f^{-1}, so that it is nearly impossible to derive an original U-value u from U'-value $f(u)$. We define $[U/U']S$ as schema S where attribute U has been replaced by U', and $[U/fU]t$ as t where the U component of t, say u, has been replaced by $f(u)$. Now, take $Q' = \pi_{S-\{U'\}} \circ (\bowtie_{U'})$ and $V = [U/fU]$, then V is Q,Q'-adequate. The proof is a similar calculation as above.

In practice, f can be a one-way hash-function. Note however that a hash function is not an injective function by definition, although—when used properly in practical situations—it can be assumed to be nearly injective,

with low chance of duplicate hash values. By replacing the original join-attribute value u by a so-called *join-key* value $f(u)$, the join can be performed without knowing the original values; the only requirement is that this join-key is unique for each attribute value, which is implied by injectivity of f.

We believe that for many types of queries, for example involving a sort operator, a degraded view can be found which is adequate. The benefits are clear: the less *information* has to be stored to answer particular queries, the lower the impact is in cases of unauthorized data disclosure. Relating adequacy to other privacy preserving techniques, and providing degraded views for various operators or abilities, is future work. The ability-oriented degradation model is an interesting future research direction, and an addition to the basic and/or service-oriented degradation model.

6.2.2 Considerations for implementing the model

Similar to our proposals in the context of the service-oriented model, one way to implement this model is to maintain for each service, for each of its queries, a degraded view. Inserts will be inserted directly into the view, and data will be removed from the view if there is a retention period attached to the data.

To some extent, maintaining a view of degraded data is similar to maintaining traditional materialized views. Much research has been done on maintaining materialized views, of which most deals with the relevancy of updates for the view. For example, it is useful to know if an insert is independent of the query for which the view has been built [19]. When there is only partial or no information—there is no base relation from which the view can be recalculated—those algorithms can determine if the view needs to be updated. Efficient algorithms exist to perform the update itself [19, 20, 46]. The views which do not need a base relation are called self-maintainable views. The requirement of not needing a base relation is important in the context of limited retention.

According to Levy [63], knowing whether a query result on a view is complete and correct, can be expressed in terms of *query dependency*. An inserted tuple is query dependent if the tuple changes the result of the query. This means that the performance issues can be concentrated on how costly it is to check if an incoming tuple results in an update of a view. Moreover, if a tuple is *query dependent* for k different queries, the tuple must be inserted in k separate views generating k I/Os (instead of just one I/O in the original base relation). Further research must resolve if this really is an issue. Conceptually, the tuple must be inserted in k separate data sets, but this does not necessarily mean that the data needs to be replicated k times. Attributes which are not subject to any degradation might be stored in separate data files, and used in multiple materialized views. Fragmented

storage structures might prove useful in such case.

6.3 Other models

6.3.1 User-oriented data degradation

In a user-oriented approach, the user is supposed to have full control over what part of the privacy-sensitive data should be retained and degraded by the service provider. In a basic implementation, a user can be given the possibility to manually mark tuples to be deleted, or attach a policy to each individual tuple, which then has to be enforced by the service provider. More sophisticated methods to give users control over their own digital trails are explained using the following scenario.

Let us consider a scenario in which Google gives users control over what will be stored in their query log, comparable with Google's own Web History service [115]. With this service, Google gives their users the possibility to browse through their personal query log, and mark items for deletion.

A more sophisticated system would allow users to provide a set of privacy-sensitive keywords. All queries containing those keywords are marked for deletion, or will have a shorter life-cycle. To go a step further, all queries in the query log leading to websites containing keywords from the privacy-sensitive set, are removed from the log.

Finally, data-mining algorithms might be used to identify certain clusters of users sharing the same characteristic, e.g., a cluster C in which all users suffer from cancer. All queries in the query log, which make that a user is member of C, have to be removed from the query log, if the user requires this.

The benefit of a model in which users mark tuples for degradation, is that in many cases only a small subset of a data set will be subject to data degradation, or limited retention. If services are not particularly interested in this very privacy-sensitive data, then usability will not suffer from this model.

6.3.2 Upgradeable data degradation

Data degradation, as a limited retention model, is supposed to be irreversible; once a data item has been degraded, it should not be possible to *upgrade* the data item to its previous level of precision. However, in some situations it might be desirable to be able to have access to a more precise version of a data item.

In chapter 2.3 we argued that most existing security techniques cannot prevent attacks as described in our threat model. However, client-side protection methods are possible solutions to prevent unauthorized data

disclosure, although they are hard to use in practice and require severe changes in the way service-providers can use the data. Nevertheless, client-side protection can be a tool to make it possible to upgrade degraded data on a temporary basis to its precise, original value.

We foresee three possible implementations. In these implementations, the service provider maintains the data at its own servers, and applies data degradation. Each user is represented by a user agent.

- An original, precise copy of each data item is stored at the client, and managed by the user agent. When the service provider needs a more precise version, it can ask the user's agent for it.
- The service provider keeps a precise copy of each data item, but in an encrypted form. The service provider can ask the user's agent for a license, to decrypt the item. Such an approach is similar to digital-right management [124]; the rights are now managed by the user in a fine-grained way instead of the service provider. For example, this allows the user to provide temporary access to a particular data item.
- The user agent can provide, when requested by the service provider, the 'missing part' of a degraded data item. For example, the user agent provides the minute and second part of a time stamp, so that the service provider can reconstruct the original time value.

Those particular implementations have benefits in terms of communication costs, storage costs and user control abilities, making it an interesting future research direction.

6.3.3 External data degradation

So far, we focused on data-degradation as an integral part of an autonomous storage system, such as a database management system. We made the assumption that such a system is *honest*, and as such, respects and enforces the degradation policies. We provided techniques to make sure that data will be irreversibly degraded.

However, in practice, a database system is only one of the components in a complex architecture. Data may flow around in a network, distributed around many nodes. Data will be queried and transferred to applications running on machines somewhere in the network. Until now we assumed that queries don't keep local copies of the data, or at least remove them as soon as possible, but in practice, without additional measures, data might end up in local caches and then escape data degradation.

An interesting approach is to bind the degradation policy to the data while the data is traveling through the network, and make network components degradation-aware. For example, network switches or routers check the policy attached to each data item, and block (or remove the data item from the stream) if the data item does not comply with its policy. If the

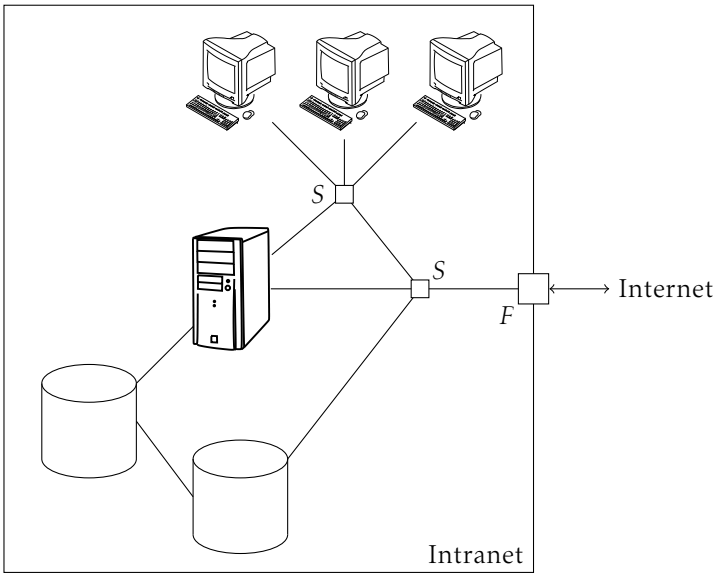


Figure 6.4 An infrastructure with clients, a server, and databases, connected with each other in a network. All components are connected with the Internet through a firewall (point F in the diagram). Only data items which are not subject to data degradation are allowed to pass the firewall. Within the network, network components, such as switches (points S), check if the stream of data contains data which should have been degraded, and degrade this data when necessary.

precision of the data item is too high, the network component can degrade the data on the fly. For an interesting starting point to implement such an architecture, we refer to Kodeswaran et al. [58].

Still, a degradation-aware network as pictured in figure 6.4, in which each network component performs degradation operations on the data, does not prevent that data can still be kept in local storages without being degraded. Physical attacks on those local storages will still result in the disclosure of data which is more precise than should be. Although it is possible to implement data degradation in storage systems, it is not feasible to implement the technique in all types of devices which might connect to the network (such as mobile phones, PDAs, *et cetera*). Future research should resolve how to deal with this issue.

6.4 Conclusion

Data degradation can be used in various ways; we presented several models using the principle of decreasing the precision of data over time to decrease the impact of unauthorized data disclosure. Both service-oriented and ability-oriented models take the limited retention principle literally: only data is kept which is useful for a certain purpose. Those models are therefore particularly useful in scenarios where purposes can be clearly identified as fulfilled or not.

A question arising from each model instantiation is how to implement the model, and what the performance penalty will be. We investigated the impact of the basic model on database management systems; some of the techniques described for the basic model can be used in other models. However, an open issue is what the price will be of implementing a privacy model.

Finally, an open question is to what extent each proposed model impacts both usability and privacy. To be able to give clear indications, each model needs to be further formalized and analyzed.

Conclusions and Future work

7

Limited retention of data is an approach to limit the impact of unauthorized data disclosure, and is based on the principle that data should only be retained as long as there is a purpose for it. Limited retention is a fundamental privacy principle, described in various privacy laws. Within the field of privacy-aware data management, limited retention is orthogonal to disclosure prevention techniques—security mechanisms such as access control—and anonymization techniques. Those other techniques do not protect against unauthorized disclosure, something which has shown to be inevitable, making limited retention a necessary addition to privacy-aware data management.

This thesis has focused on various aspects of limited retention, and gives answers to the following research questions. Where possible, we will give an indication of future work which has still to be done to get more insight in the particular subject.

7.1 Revisiting the research questions

Research question 1. How to model the interest of both service provider and user, to find the best retention period of privacy-sensitive data?

Limited retention periods are often overstated because fulfillment of a purpose cannot always be clearly defined. To find reasonable retention periods, not only in favor of service providers, but also taking the privacy concerns of users into account, we proposed in chapter 3 a framework to balance usability and privacy. In this framework, we model the interest of both service provider and user, and show how those interests can be combined in what we name the *common interest*. The framework helps to maximize this common interest.

The interest of the service provider is to maximize the usability of the data, expressed in the *worth* of the data. Although the worth of a single data item depends on many factors, we made the abstraction that the worth

of a data item only depends on its age. Moreover, we made the assumption that the worth of a data item decreases when the item gets older.

The interest of the user is to maximize its privacy, and minimize the risk of a privacy violation. Again, although also the privacy sensitivity of a data item depends on many factors, we made the abstraction that privacy depends on the amount of data stored at the service provider, which in turn depends on the retention period.

The common interest of service provider and user is the combination of the individual interests. We expressed common interest as a single function with only one argument, which is the retention period. We showed that, under the assumption that the worth of a data item decreases with age, the common interest function has a maximum, which it takes on a argument which can be considered as the *optimal retention period*.

The existence of this maximum can be interpreted in the following way. By setting the retention period smaller than the optimum, the user will gain more privacy, but the common interest will be lower because the decrease in stored data induces a greater loss of total worth for the service provider. Similarly, by setting the retention period larger than the optimum, the service provider will gain more total worth of the stored data, but the common interest will be lower because of a larger decrease of the user's privacy.

Summarizing, we succeeded to model the interest of service provider and user in an abstract, qualitative way; when such a model is quantitatively specified, we can indeed calculate the optimal retention period.

Research question 2. How to refine the limited retention principle, to better balance the interests of service provider and user?

In chapter 3 we introduced the concept of data degradation. Instead of removing a data item in one single step, as is the case with limited retention, the precision of the data item will be progressively decreased until the data item will be finally removed. The assumptions we make is that the privacy sensitivity of a data item is proportional to the precision of the data item, and that a data item with decreased precision can still be useful for the service provider.

We included data degradation in our framework, and showed that a higher common interest can be achieved by sacrificing some precision to get more privacy in return, especially when the privacy increase is higher than the loss in worth. In our analysis, we showed cases where it is indeed useful to progressively degrade the data from precise states to less precise states until final destruction of the data. This makes it possible to make a better balance between the interests of service provider and user, resulting in a higher common interest.

Thus, by progressively degrading data instead of removing data in one single step, a higher common interest can be achieved. Data degradation makes it possible to balance the interests of service provider and user better than limited retention can do.

Research question 3. What is the impact of data degradation on traditional database systems, and is it feasible to implement the technique?

Data degradation has a high impact on almost every component of traditional database systems, namely on storage, indexing, transaction management and logging:

- *Storage*. Tree-based storage structures, by default used to speed up queries, will inevitably lead to poor degradation performance. Each degradation of a tuple will result in one or more I/O operations, which is costly. To overcome this problem, several degradation friendly storage structures have been proposed, all based on heap structures. Tuples are stored in insert order, and therefore also on degradation order. By allowing some flexibility in the retention periods, multiple tuples can be degraded together, sharing I/O costs. By fragmenting tables over several data files, the amount of data transferred at each degradation step can be minimized, leading to an improvement of degradation performance. Moreover, by pre-computing each degradation step, costly read/write updates can be prevented, so that degradation can be performed by writes only. However, every proposed structure has its benefits in terms of insert and degradation performance.

  In chapter 5 we experimentally analyzed the proposed storage structures using a prototype implementation, and gave an analysis of the index structures. We chose to implement a prototype from scratch, instead of adapting a traditional database management system, to be able to make a fair comparison between the storage structures, and to have full control over all operations.

  The experiments show that fragmented storage structures perform better in the context of heavy data degradation, although the differences between the various strategies are small. Especially when there are only a small number of degradation steps, or when only limited retention has to be implemented, a basic heap structure performs well enough without requiring additional optimizations.

- *Indexing*. We investigated the impact on index structures. Not only data stored in the data files have to be degraded, at each degradation step, but also the indexes have to be updated. We therefore proposed several data degradation friendly indexes. Moreover, since data will be degraded, queries will become less selective; some indexes are more useful with respect to low-selective queries than others. In

section 5.4.5 we compared the proposed indexes, and argued that a b+tree index, using encryption methods to efficiently remove tuples from the tree, is most suitable for highly precise attributes. The other proposed indexes, such as bitmap indexes and bloom filter indexes, benefit from the fact that data is stored in degradation order, and are especially useful for smaller domains, and thus for less precise data.

- *Transaction management.* Data degradation adds complexity to transaction mechanisms. Each insert transaction leads to several degradation side-transactions that have to be performed on time, and cannot be rolled back or aborted. We proposed a synchronization protocol which minimizes the number of regular user transactions that have to be blocked or aborted. We showed that only some long-running transactions, which use the oldest to be degraded data, might have to be aborted; something which is unlikely to happen when allowing a sufficiently large flexibility in retention periods.
- *Logging.* Data degradation has an impact on logging mechanisms. To avoid that data items have to be removed from log files, we proposed to pre-compute every degradation step of every tuple, and encrypt every degraded representation of the tuple with a different encryption key, and store the encrypted values in the redo log. Every encryption key, which can be shared by a set of tuples, will be overwritten at the end of its retention period, so that the log entries cannot be decrypted anymore.

The question remains whether or not it is feasible to implement data degradation. We proposed several techniques to implement data degradation on top of a relational database system. The objective of all techniques is first to degrade the data in an irreversible way, and second, to minimize the performance penalty of data degradation. So, it is feasible to implement data degradation so that it complies with the limited retention principle, although we cannot conclude from the experiments whether or not the performance penalty is acceptable.

Summarizing, data degradation has impact on the storage structure, indexing, transaction management, and logging mechanisms. We provided several techniques to implement data degradation, showing the feasibility of data degradation, although we cannot yet decide whether the performance penalty of data degradation is acceptable.

Research question 4. How can the concept of data degradation be further exploited when some of the simplifications are released?

The basic model is based on two simplifications: first, all transitions are triggered by time, and second, every tuple is subject to the same degradation policy. As an outlook to future research, we proposed in chapter 6 various models in which those simplifications are released.

- In the *service-oriented* data degradation model, the objective is to store each data item with the lowest precision needed, with which each service can fulfill its purpose. Once one of the services has fulfilled its purpose, the precision needed to serve all other services will be evaluated again. If the precision can be decreased, the data item will be degraded. In this model, the life-cycle is not known in advance; therefore two different tuples can have two different life-cycles, although the policy itself may be the same. This is because purposes are not always fulfilled after predefined time periods.
- In the *ability-oriented* data degradation model, the objective is to degrade the *ability* of what can be done with the stored data, while still guaranteeing that the queries required by the services can be performed on the degraded data with same result. Other queries which require data other, or in another form than used by the allowed queries, are not guaranteed to give correct or meaningful results anymore.

Besides, we proposed two additional models: first, with *upgradeable data degradation*, degraded data can temporarily be upgraded, using information explicitly provided by the user himself. Second, in the *external data degradation* model, data is not only subject to degradation in an autonomous storage system, but will be degraded while traveling through a network infrastructure.

In summary, by releasing the simplifications underlying the basic model, new directions open up for exploitation: service-orientation and ability-orientation. These need further investigation.

7.2 Future work

In chapter 6 we gave research directions for new degradation models. Here we list additional work that needs to be done to complete the *theory* and *practice* of data degradation.

Privacy metric for data degradation. Throughout this thesis, we assumed that the privacy sensitivity of a data item is proportional to its precision. We have not been able to capture the privacy benefits in a metric, making it hard to quantify to what extent data degradation contributes in lowering the impact of unauthorized data disclosure.

Defining interest functions. The interest functions for a single data item, which we defined in chapter 3, are only proportional to the retention period of the data item. Both privacy and worth functions can depend on various parameters other than the age of a data item. For example, *users* can choose different 'risk profiles' depending on the nature of the

data items, and *service providers* can attach a lower worth to specific data based on the user, location, and time of the day.

Moreover, for practical reasons, an important question is *if* and *how* service providers will be able to express their worth functions. To put our framework into practice, it is necessary to provide tools which enable service providers to give transparency about their need to collect personal data.

Integration with a traditional DBMS. To be able to show the feasibility of data degradation, it is recommended to implement data degradation as an extension to an existing database management system. This way data degradation can be used and tested within real-world scenario's, and experiments can show the performance impact of data degradation compared to traditional database management systems.

Extending the prototype implementation The prototype implementation, in this thesis used to compare storage structures, needs to be extended with indexes and logging mechanisms, so that a complete research platform is created to test all aspects of data degradation. We hope that techniques developed on such a prototype can find their way into traditional database management systems.

7.3 Concluding remarks

Privacy-aware data management has become a popular research topic, and deserves much attention. Triggered by a growing risk on privacy violations due to the collection of huge sets of personal information, new techniques have to be found to limit the impact of unauthorized data disclosure. We believe that the work in this thesis paves the way for solutions in this important area.

The framework we presented in this thesis shows that it is indeed possible to reason about retention periods so that not only service providers, but also users of those services will be satisfied. The techniques we proposed show that data degradation is feasible; they should be an integral part of any storage system handling privacy-sensitive data. So, we presented a promising approach to close the huge gap between the enormous amount of collection and storage of personal data, and the risk users have to take to be able to profit from all the new and exciting services offered to us today and in the future.

Bibliography

[1] O. Abul, F. Bonchi, and M. Nanni. Never walk alone: Uncertainty for anonymity in moving objects databases. In *Proceedings of the 24th IEEE International Conference on Data Engineering*, pages 376–385, 2008.

[2] A. Acquisti, A. Friedman, and R. Telang. Is there a cost to privacy breaches? An event study. In *Fifth Workshop on the Economics of Information Security*, 2006.

[3] G. Aggarwal, M. Bawa, P.Ganesan, H. Garcia-Molina, K. Kenthapadi, N. Mishra, R. Motwani, U. Srivastava, D. Thomas, J. Widom, and Y. Xu. Vision Paper: Enabling Privacy for the Paranoids. In *Proceedings of the 30th VLDB Conference*, 2004.

[4] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 86–97, New York, NY, USA, 2003. ACM Press.

[5] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th Int'l Conf. on Very Large Databases (VLDB), Hong Kong*, 2002.

[6] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Implementing p3p using database technology. In *ICDE*, pages 595–606, 2003.

[7] N. Anciaux, L. Bouganim, Y. Guo, P. Pucheral, J.-J. Vandewalle, and S. Yin. Pluggable personal data servers. *To be published in Sigmod*, 2010.

[8] N. Anciaux, L. Bouganim, H. Van Heerde, P. Pucheral, and P. Apers. Dégradation progressive et irréversible des données. *24emes journées Bases de Données Avancées (BDA)*, 2008.

[9] N. Anciaux, L. Bouganim, H. J. W. van Heerde, P. Pucheral, and P. M. G. Apers. Instantdb: Enforcing timely degradation of sensitive data. In *Proceedings of the 24th International Conference on Data Engineering*. IEEE Computer Society Press, April 2008.

[10] N. L. G. Anciaux, L. Bouganim, H. J. W. van Heerde, P. Pucheral, and P. M. G. Apers. Data degradation: Making private data less sensitive over time. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM2008), Napa valley, California, USA*, pages 1401–1402, New York, October 2008. ACM.

[11] N. L. G. Anciaux, L. Bouganim, H. J. W. van Heerde, P. Pucheral, and P. M. G. Apers. The life-cycle policy model. Research report RR-6577, INRIA, Rocquencourt, France, July 2008.

[12] N. L. G. Anciaux, H. J. W. van Heerde, L. Feng, and P. M. G. Apers. Implanting life-cycle privacy policies in a context database. Technical Report TR-CTIT-06-03, University of Twente, Enschede, January 2006.

[13] R. Angryk and F. Petry. Mining multi-level associations with fuzzy hierarchies. *Fuzzy Systems*, 2005.

[14] P. Ashley, S. Hada, G. Karjoth, and M. Schunter. E-p3p privacy policies and privacy authorization. In *WPES '02: Proceedings of the 2002 ACM workshop on Privacy in the Electronic Society*, pages 103–109, New York, NY, USA, 2002. ACM Press.

[15] A. A. Ataullah, A. Aboulnaga, and F. W. Tompa. Records retention in relational database systems. In *CIKM*, pages 873–882. ACM, 2008.

[16] D. Barbara, H. Garcia-Molina, and D. Porter. A probabilistic relational data model. *Advances in Database Technology-EDBT*, 90:0–74, 1990.

[17] E. Bertino, J.-W. Byun, and N. Li. Privacy-preserving database systems. *Lecture Notes in Computer Science*, 3655:178–206, 2005.

[18] F. Bignami. Protecting Privacy against the Police in the European Union: The Data Retention Directive. *Chicago Journal of International Law*, 2007.

[19] J. A. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[20] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[21] J. Blanchette and D. Johnson. Data retention and the panoptic society: The social benefits of forgetfulness. *The Information Society*, 18(1):33–45, 2002.

[22] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[23] P. Boncz and M. Kersten. Monet: an impressionist sketch of an advanced database system, 1995.

[24] D. Boneh and R. Lipton. A revocable backup system. In *USENIX Security Symposium*, pages 91–96, 1996.

[25] L. Bouganim and P. Pucheral. Chip-secured data access: Confidential data on untrusted servers. In *Proceedings of the 28th international conference on Very Large Data Bases*, page 142. VLDB Endowment, 2002.

[26] R. Brinkman, J. Doumen, and W. Jonker. Using secret sharing for searching in encrypted data. *Lecture notes in computer science*, pages 18–27, 2004.

[27] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004.

[28] B. Buckles and F. Petry. Fuzzy databases in the new era. In *Proceedings of the 1995 ACM symposium on Applied computing*, pages 497–502. ACM New York, NY, USA, 1995.

[29] J.-W. Byun and E. Bertino. Micro-views, or on how to protect privacy while enhancing data usability: concepts and challenges. *SIGMOD Rec.*, 35(1):9–13, 2006.

[30] J.-W. Byun, E. Bertino, and N. Li. Purpose based access control of complex data for privacy protection. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 102–110, New York, NY, USA, 2005. ACM.

[31] J.-W. Byun, Y. Sohn, E. Bertino, and N. Li. Secure anonymization for incremental datasets. In *Secure Data Management*, pages 48–63, 2006.

[32] H. Cavusoglu, B. Mishra, and S. Raghunathan. The value of intrusion detection systems in information technology security architecture. *Info. Sys. Research*, 16(1):28–46, 2005.

[33] C. Chan and Y. Ioannidis. Bitmap index design and evaluation. *ACM SIGMOD Record*, 27(2):355–366, 1998.

[34] C. Chan and Y. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 215–226. ACM New York, NY, USA, 1999.

[35] G. Conti. *Googling Security: How Much Does Google Know About You?* Addison-Wesley Professional, 2008.

[36] A. Cooper. A survey of query log privacy-enhancing techniques from a policy perspective. *ACM Trans. Web*, 2(4):1–27, 2008.

[37] L. Cranor. P3p: making privacy policies more useful. *Security & Privacy Magazine, IEEE Security & Privacy Magazine, IEEE*, 1(6):50–55, 2003.

[38] P. Dillinger and P. Manolios. Fast and accurate bitstate verification for SPIN. *Lecture notes in computer science*, pages 57–75, 2004.

[39] C. Dwork et al. Differential privacy. *Lecture notes in computer science*, 4052:1, 2006.

[40] European Parliament. 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the EC*, 23, 1995.

[41] D. Ferraiolo and D. Kuhn. Role-based access control. In *15th National Computer Security Conference*, 1992.

[42] B. Fung, K. Wang, R. Chen, and P. Yu. Privacy-preserving data publishing: A survey on recent developments. *ACM Computing Surveys*, 2009.

[43] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2008.

[44] G. Gordon and N. Willox. *Identity Fraud: a Critical National and Global Threat*. Economic Crime Institute, 2003.

[45] R. Grimm and A. Rossnagel. Can p3p help to protect privacy worldwide? In *MULTIMEDIA '00: Proceedings of the 2000 ACM workshops on Multimedia*, pages 157–160, New York, NY, USA, 2000. ACM Press.

[46] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques and applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[47] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, volume 7, pages 77–90, 1996.

[48] H. Hacigümüş, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.

[49] J. Halpern and K. O'Neill. Anonymity and information hiding in multiagent systems. *Journal of Computer Security*, 13(3):483–514, 2005.

[50] J. He and M. Wang. Cryptography and relational database management systems. In *IDEAS*, pages 273–284, 2001.

[51] R. J. Hilderman, H. J. Hamilton, and N. Cercone. Data mining in large databases using domain generalization graphs. *J. Intell. Inf. Syst.*, 13(3):195–234, 1999.

[52] D. Hillyard and M. Gauen. Issues around the protection or revelation of personal information. *Knowledge, Technology, and Policy*, 2007.

[53] J. I. Hong and J. A. Landay. An architecture for privacy-sensitive ubiquitous computing. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 177–189, New York, NY, USA, 2004. ACM Press.

[54] L. Introna and A. Pouloudi. Privacy in the information age: Stakeholders, interests and values. *Journal of Business Ethics*, 22(1):27–38, 1999.

[55] W. Jiang, M. Murugesan, C. Clifton, and L. Si. t-plausibility: Semantic preserving text sanitization. *Computational Science and Engineering, IEEE International Conference on*, 3:68–75, 2009.

[56] X. Jiang, J. I. Hong, and J. A. Landay. Approximate information flows: Socially-based modeling of privacy in ubiquitous computing. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 176–193, London, UK, 2002. Springer-Verlag.

[57] M. Jurgens and H. Lenz. Tree based indexes vs. bitmap indexes-a performance study. In *Proceedings of the Intl. Workshop on Design and Management of Data Warehouses, DMDW*, volume 99, pages 14–15, 1999.

[58] P. Kodeswaran, S. Kodeswaran, A. Joshi, and T. Finin. Enforcing security in semantics driven policy based networks. *ICDE Workshops 2008*, pages 490–497.

[59] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, 1974.

[60] M. Langheinrich. A privacy awareness system for ubiquitous computing environments. In *UbiComp '02: Proceedings of the 4th international conference on Ubiquitous Computing*, pages 237–245, London, UK, 2002. Springer-Verlag.

[61] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 108–119. VLDB Endowment, 2004.

[62] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Incognito: efficient full-domain k-anonymity. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 49–60, New York, NY, USA, 2005. ACM Press.

[63] A. Y. Levy. Obtaining complete answers from incomplete databases. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 402–412, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[64] P. Lewis, A. Bernstein, and M. Kifer. *Databases and transaction processing: an application-oriented approach*. Addison-Wesley Boston, MA, 2001.

[65] T. Li and N. Li. On the tradeoff between privacy and utility in data publishing. In *KDD '09: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 517–526, New York, NY, USA, 2009. ACM.

[66] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 24, Washington, DC, USA, 2006. IEEE Computer Society.

[67] D. Martin, D. Kifer, A. Machanavajjhala, J. Gehrke, and J. Halpern. Worst-case background knowledge for privacy-preserving data publishing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, volume 162, 2007.

[68] F. Massacci, J. Mylopoulos, and N. Zannone. Minimal disclosure in hierarchical hippocratic databases with delegation. *Lecture notes in computer science*, 3679:438, 2005.

[69] V. Mayer-Schöberger. Useful Void: The Art of Forgetting in the Age of Ubiquitous Computing. *Working Paper Series*, 2007.

[70] J. Medina, M. Vila, J. Cubero, and O. Pons. Towards the implementation of a generalized fuzzy relational database model. *Fuzzy Sets and Systems*, 75(3):273–289, 1995.

[71] G. Miklau, B. N. Levine, and P. Stahlberg. Securing history: Privacy and accountability in database systems. In *CIDR*, pages 387–396, 2007.

[72] M. E. Nergiz, M. Atzori, and C. Clifton. Hiding the presence of individuals from shared databases. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 665–676, New York, NY, USA, 2007. ACM.

[73] C. of the European communities. Directive of the european parliament and of the council on the retention of data processed in connection with the provision of public electronic communication services and amending, 2005.

[74] S. Preibusch. Implementing privacy negotiations in e-commerce. In *APWeb*, pages 604–615, 2006.

[75] J. Reid, J. M. G. Nieto, E. Dawson, and E. Okamoto. Privacy and trusted computing. In *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, page 383, Washington, DC, USA, 2003. IEEE Computer Society.

[76] R. Richardson. CSI Computer Crime and Security Survey. 2008.

[77] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–49, 1994.

[78] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[79] A. Schmidt, C. S. Jensen, and S. Saltenis. Expiration times for data management. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 36, Washington, DC, USA, 2006. IEEE Computer Society.

[80] D. J. Solove. *Understanding Privacy*. Harvard University Press, Cambridge, Massuchusetts, 2008.

[81] P. Stahlberg, G. Miklau, and B. N. Levine. Threats to privacy in the forensic analysis of database systems. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 91–102, New York, NY, USA, 2007. ACM Press.

[82] T. A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[83] L. Sweeney. Datafly: A system for providing anonymity in medical data. In *Proceedings of the IFIP TC11 WG11. 3 Eleventh International Conference on Database Securty XI: Status and Prospects*, pages 356–381. Chapman & Hall, Ltd. London, UK, UK, 1997.

[84] L. Sweeney. Achieving *k*-anonymity privacy protection using generalization and suppression. *International Journal on Uncertainty Fuzziness and Knowledge-based Systems*, pages 571–588, 2002.

[85] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty Fuzziness and Knowledge-based Systems*, pages 557–570, 2002.

[86] Y. Takahashi. Fuzzy database query languages and their relational completeness theorem. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):122–125, 1993.

[87] Q. Tang. Timed-ephemerizer: Make assured data appear and disappear. In *Sixth European Workshop on Public Key Services, Applications and Infrastructures, Pisa, Italy*, Lecture Notes in Computer Science, London, 2009. Springer Verlag.

[88] T. M. Truta and B. Vinay. Privacy protection: p-sensitive k-anonymity property. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*, page 94, Washington, DC, USA, 2006. IEEE Computer Society.

[89] Y. Tsukada, K. Mano, H. Sakurada, and Y. Kawabe. Anonymity, privacy, onymity, and identity: A modal logic approach. *Computational Science and Engineering, IEEE International Conference on*, 3:42–51, 2009.

[90] C. Valli and A. Woodward. Oops they did it again: The 2007 australian study of remnant data contained on 2nd hand hard disks. In *Proceedings of the 5th Australian Digital Forensics Conference*, 2007.

[91] H. J. W. van Heerde, N. L. G. Anciaux, M. M. Fokkinga, and P. M. G. Apers. Exploring personalized life cycle policies. Technical Report TR-CTIT-07-85, Enschede, December 2007.

[92] H. J. W. van Heerde, M. M. Fokkinga, and N. L. G. Anciaux. A framework to balance privacy and data usability using data degradation. In *Proceedings of the International Conference on Computational Science and Engineering (CSE2009), Vancouver, Canada*, pages 146–153, Los Alamitos, August 2009. IEEE Computer Society Press.

[93] K. Wang, B. Fung, and P. Yu. Handicapping attacker's confidence: an alternative to k-anonymization. *Knowledge and Information Systems*, 11(3):345–368, 2007.

[94] K. Wang and B. C. M. Fung. Anonymizing sequential releases. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 414–423, New York, NY, USA, 2006. ACM.

[95] M.-A. Williams. Privacy management, the law & business strategies: A case for privacy driven design. *IEEE International Conference on Computational Science and Engineering*, 3:60–67, 2009.

[96] X. Xiao and Y. Tao. Anatomy: Simple and effective privacy preservation. In *VLDB*, pages 139–150, 2006.

[97] X. Xiao and Y. Tao. Personalized privacy preservation. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 229–240, New York, NY, USA, 2006. ACM Press.

[98] Z. Yang, S. Zhong, and R. Wright. Privacy-preserving queries on encrypted data. *Lecture Notes in Computer Science*, 4189:479, 2006.

[99] S. B. Yao. An attribute based model for database access cost analysis. *ACM Trans. Database Syst.*, 2(1):45–67, 1977.

[100] S. Yin, P. Pucheral, and X. Meng. A sequential indexing scheme for flash-based embedded systems. In *EDBT '09: Proceedings of the 12th International Conference on Extending Database Technology*, pages 588–599, New York, NY, USA, 2009. ACM.

Referenced web sources

[101] BBC news. T-mobile staff sold personal data. `http://news.bbc.co.uk/2/hi/uk_news/8364421.stm`, November 2009.

[102] CNN.com. Hackers stole data on pentagon's newest fighter jet. `http://www.cnn.com/2009/US/04/21/pentagon.hacked/`, April 2009.

[103] Council of Europe. Convention for the protection of individuals with regard to automatic processing of personal data. `http://conventions.coe.int/treaty/EN/Treaties/Html/108.htm`, 1981.

[104] Counsil of the European Union. SWIFT agreement. `http://www.netzpolitik.org/wp-upload/SWIFT-Abkommen-2009-11-10.pdf`, November 2009.

[105] EDRI. Uk government loses personal data on 25 million citizens. `http://www.edri.org/edrigram/number5.22/personal-data-lost-uk`, November 2007.

[106] Google. About google web directory. `http://www.google.com/intl/en/dirhelp.html#about`, 2009.

[107] Google Mobile Blog. Search with my location for iphone 3.0. and all that jazz. `http://googlemobile.blogspot.com/2009/07/search-with-my-location-for-iphone-30.html`, July 2009.

[108] S. Mathijssen. Feasibility study for implementing life-cycle policies in an rdbms. `http://wwwhome.ewi.utwente.nl/~heerdehjw/feasibility_study_mathijssen2007.pdf`, 2007.

[109] Ministry of Transport. Road pricing: Different payment for mobility. `http://www.verkeerenwaterstaat.nl/english/topics/mobility_and_accessibility/roadpricing/index.aspx`, 2009.

[110] NRC Handelsblad. Fingerprints in passports can't be used by the police - yet. http://www.nrc.nl/international/Features/article2363938.ece, September 2009.

[111] Oracle. Oracle virtual private database, an oracle database 10g release 2 white paper. http://www.databasesecurity.com/oracle/twp_security_db_vpd_10gr2.pdf, 2005.

[112] Platform for Privacy Preferences (P3P) Project. http://www.w3.org/P3P/implementations.html, May 2007.

[113] Reuters. Facebook privacy revamp draws fire. http://www.reuters.com/article/idUSTRE5B82F320091210, December 2009.

[114] Techworld. Eu parliament rejects us bank data deal. http://news.techworld.com/security/3212523/eu-parliament-rejects-us-bank-data-deal/, 2010.

[115] Google. Google web history. http://www.google.com/history.

[116] W3C. Platform for privacy preferences (P3P) project. http://www.w3.org/P3P/, June 2005.

[117] The Register. Prime minister's health records breached in database attack. http://www.theregister.co.uk/2009/03/02/nhs_database_breach/, March 2009.

[118] The Washington Post. Consultant breached fbi's computers. http://www.washingtonpost.com/wpdyn/content/article/2006/07/05/AR2006070501489_pf.html, July 2007.

[119] The Washington Post. Payment processor breach may be largest ever. http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html, January 2009.

[120] Times Online. Terror police to track capital's cars. http://www.timesonline.co.uk/tol/news/uk/crime/article2091023.ece, July 2007.

[121] A. Türk. Letter from the article 29 working party addressed to search engine operators (google, microsoft, yahoo!). http://ec.europa.eu/justice_home/fsj/privacy/workinggroup/wpdocs/2009-others_en.htm, October 2009.

[122] H. van Heerde and S. Mathijssen. Instantdb prototype implementation. http://www.vanheerde.eu/instantdb/, 2010.

[123] VNUNet.com. T-mobile loses 17 million customer details. http://www.vnunet.com/vnunet/news/2227691/loss-headaches-mobile, October 2008.

[124] Wikipedia. Digital rights management. http://en.wikipedia.org/wiki/Digital_rights_management.

[125] Wikipedia. Congestion pricing. http://en.wikipedia.org/w/index.php?title=Congestion_pricing, 2009.

[126] Yahoo! Yahoo directory. http://dir.yahoo.com/, 2009.

[127] Yahoo! Finance. Key statistics for google inc. http://finance.yahoo.com/q/ks?s=GOOG, April 2009.

Siks dissertations

1998

1998-1	Johan van den Akker (CWI) DEGAS - An Active, Temporal Database of Autonomous Objects
1998-2	Floris Wiesman (UM) Information Retrieval by Graphically Browsing Meta-Information
1998-3	Ans Steuten (TUD) A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective
1998-4	Dennis Breuker (UM) Memory versus Search in Games
1998-5	E.W.Oskamp (RUL) Computerondersteuning bij Straftoemeting

1999

1999-1	Mark Sloof (VU) Physiology of Quality Change Modelling; Automated modelling ofQuality Change of Agricultural Products
1999-2	Rob Potharst (EUR) Classification using decision trees and neural nets
1999-3	Don Beal (UM) The Nature of Minimax Search
1999-4	Jacques Penders (UM) The practical Art of Moving Physical Objects
1999-5	Aldo de Moor (KUB) Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems
1999-6	Niek J.E. Wijngaards (VU) Re-design of compositional systems
1999-7	David Spelt (UT) Verification support for object database design
1999-8	Jacques H.J. Lenting (UM) Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation.

2000

2000-1	Frank Niessink (VU) Perspectives on Improving Software Maintenance
2000-2	Koen Holtman (TUE) Prototyping of CMS Storage Management
2000-3	Carolien M.T. Metselaar (UVA)

	Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief.
2000-4	Geert de Haan (VU) ETAG, A Formal Model of Competence Knowledge for User Interface Design
2000-5	Ruud van der Pol (UM) Knowledge-based Query Formulation in Information Retrieval.
2000-6	Rogier van Eijk (UU) Programming Languages for Agent Communication
2000-7	Niels Peek (UU) Decision-theoretic Planning of Clinical Patient Management
2000-8	Veerle Coup (EUR) Sensitivity Analyis of Decision-Theoretic Networks
2000-9	Florian Waas (CWI) Principles of Probabilistic Query Optimization
2000-10	Niels Nes (CWI) Image Database Management System Design Considerations, Algorithms and Architecture
2000-11	Jonas Karlsson (CWI) Scalable Distributed Data Structures for Database Management

2001

2001-1	Silja Renooij (UU) Qualitative Approaches to Quantifying Probabilistic Networks
2001-2	Koen Hindriks (UU) Agent Programming Languages: Programming with Mental Models
2001-3	Maarten van Someren (UvA) Learning as problem solving
2001-4	Evgueni Smirnov (UM) Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets
2001-5	Jacco van Ossenbruggen (VU) Processing Structured Hypermedia: A Matter of Style
2001-6	Martijn van Welie (VU) Task-based User Interface Design
2001-7	Bastiaan Schonhage (VU) Diva: Architectural Perspectives on Information Visualization
2001-8	Pascal van Eck (VU) A Compositional Semantic Structure for Multi-Agent Systems Dynamics.
2001-9	Pieter Jan 't Hoen (RUL) Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes
2001-10	Maarten Sierhuis (UvA) Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design
2001-11	Tom M. van Engers (VUA) Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01	Nico Lassing (VU) Architecture-Level Modifiability Analysis
2002-02	Roelof van Zwol (UT) Modelling and searching web-based document collections
2002-03	Henk Ernst Blok (UT) Database Optimization Aspects for Information Retrieval
2002-04	Juan Roberto Castelo Valdueza (UU) The Discrete Acyclic Digraph Markov Model in Data Mining
2002-05	Radu Serban (VU)

The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel(KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UVA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UVA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14	Stijn Hoppenbrouwers (KUN) Freezing Language: Conceptualisation Processes across ICT-Supported Organisations
2003-15	Mathijs de Weerdt (TUD) Plan Merging in Multi-Agent Systems
2003-16	Menzo Windhouwer (CWI) Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses
2003-17	David Jansen (UT) Extensions of Statecharts with Probability, Time, and Stochastic Timing
2003-18	Levente Kocsis (UM) Learning Search Decisions

2004

2004-01	Virginia Dignum (UU) A Model for Organizational Interaction: Based on Agents, Founded in Logic
2004-02	Lai Xu (UvT) Monitoring Multi-party Contracts for E-business
2004-03	Perry Groot (VU) A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving
2004-04	Chris van Aart (UVA) Organizational Principles for Multi-Agent Architectures
2004-05	Viara Popova (EUR) Knowledge discovery and monotonicity
2004-06	Bart-Jan Hommes (TUD) The Evaluation of Business Process Modeling Techniques
2004-07	Elise Boltjes (UM) Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes
2004-08	Joop Verbeek(UM) Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise
2004-09	Martin Caminada (VU) For the Sake of the Argument; explorations into argument-based reasoning
2004-10	Suzanne Kabel (UVA) Knowledge-rich indexing of learning-objects
2004-11	Michel Klein (VU) Change Management for Distributed Ontologies
2004-12	The Duy Bui (UT) Creating emotions and facial expressions for embodied agents
2004-13	Wojciech Jamroga (UT) Using Multiple Models of Reality: On Agents who Know how to Play
2004-14	Paul Harrenstein (UU) Logic in Conflict. Logical Explorations in Strategic Equilibrium
2004-15	Arno Knobbe (UU) Multi-Relational Data Mining
2004-16	Federico Divina (VU) Hybrid Genetic Relational Search for Inductive Learning
2004-17	Mark Winands (UM) Informed Search in Complex Games
2004-18	Vania Bessa Machado (UvA) Supporting the Construction of Qualitative Knowledge Models
2004-19	Thijs Westerveld (UT) Using generative probabilistic models for multimedia retrieval
2004-20	Madelon Evers (Nyenrode) Learning from Design: facilitating multidisciplinary design teams

2005

2005-01	Floor Verdenius (UVA) Methodological Aspects of Designing Induction-Based Applications
2005-02	Erik van der Werf (UM)) AI techniques for the game of Go
2005-03	Franc Grootjen (RUN) A Pragmatic Approach to the Conceptualisation of Language
2005-04	Nirvana Meratnia (UT) Towards Database Support for Moving Object data
2005-05	Gabriel Infante-Lopez (UVA) Two-Level Probabilistic Grammars for Natural Language Parsing
2005-06	Pieter Spronck (UM) Adaptive Game AI
2005-07	Flavius Frasincar (TUE) Hypermedia Presentation Generation for Semantic Web Information Systems
2005-08	Richard Vdovjak (TUE) A Model-driven Approach for Building Distributed Ontology-based Web Applications
2005-09	Jeen Broekstra (VU) Storage, Querying and Inferencing for Semantic Web Languages
2005-10	Anders Bouwer (UVA) Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments
2005-11	Elth Ogston (VU) Agent Based Matchmaking and Clustering - A Decentralized Approach to Search
2005-12	Csaba Boer (EUR) Distributed Simulation in Industry
2005-13	Fred Hamburg (UL) Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen
2005-14	Borys Omelayenko (VU) Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics
2005-15	Tibor Bosse (VU) Analysis of the Dynamics of Cognitive Processes
2005-16	Joris Graaumans (UU) Usability of XML Query Languages
2005-17	Boris Shishkov (TUD) Software Specification Based on Re-usable Business Components
2005-18	Danielle Sent (UU) Test-selection strategies for probabilistic networks
2005-19	Michel van Dartel (UM) Situated Representation
2005-20	Cristina Coteanu (UL) Cyber Consumer Law, State of the Art and Perspectives
2005-21	Wijnand Derks (UT) Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01	Samuil Angelov (TUE) Foundations of B2B Electronic Contracting
2006-02	Cristina Chisalita (VU) Contextual issues in the design and use of information technology in organizations
2006-03	Noor Christoph (UVA) The role of metacognitive skills in learning to solve problems
2006-04	Marta Sabou (VU) Building Web Service Ontologies
2006-05	Cees Pierik (UU)

	Validation Techniques for Object-Oriented Proof Outlines
2006-06	Ziv Baida (VU) Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling
2006-07	Marko Smiljanic (UT) XML schema matching – balancing efficiency and effectiveness by means of clustering
2006-08	Eelco Herder (UT) Forward, Back and Home Again - Analyzing User Behavior on the Web
2006-09	Mohamed Wahdan (UM) Automatic Formulation of the Auditor's Opinion
2006-10	Ronny Siebes (VU) Semantic Routing in Peer-to-Peer Systems
2006-11	Joeri van Ruth (UT) Flattening Queries over Nested Data Types
2006-12	Bert Bongers (VU) Interactivation - Towards an e-cology of people, our technological environment, and the arts
2006-13	Henk-Jan Lebbink (UU) Dialogue and Decision Games for Information Exchanging Agents
2006-14	Johan Hoorn (VU) Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change
2006-15	Rainer Malik (UU) CONAN: Text Mining in the Biomedical Domain
2006-16	Carsten Riggelsen (UU) Approximation Methods for Efficient Learning of Bayesian Networks
2006-17	Stacey Nagata (UU) User Assistance for Multitasking with Interruptions on a Mobile Device
2006-18	Valentin Zhizhkun (UVA) Graph transformation for Natural Language Processing
2006-19	Birna van Riemsdijk (UU) Cognitive Agent Programming: A Semantic Approach
2006-20	Marina Velikova (UvT) Monotone models for prediction in data mining
2006-21	Bas van Gils (RUN) Aptness on the Web
2006-22	Paul de Vrieze (RUN) Fundaments of Adaptive Personalisation
2006-23	Ion Juvina (UU) Development of Cognitive Model for Navigating on the Web
2006-24	Laura Hollink (VU) Semantic Annotation for Retrieval of Visual Resources
2006-25	Madalina Drugan (UU) Conditional log-likelihood MDL and Evolutionary MCMC
2006-26	Vojkan Mihajlovic (UT) Score Region Algebra: A Flexible Framework for Structured Information Retrieval
2006-27	Stefano Bocconi (CWI) Vox Populi: generating video documentaries from semantically annotated media repositories
2006-28	Borkur Sigurbjornsson (UVA) Focused Information Access using XML Element Retrieval

2007

2007-01	Kees Leune (UvT) Access Control and Service-Oriented Architectures
2007-02	Wouter Teepe (RUG) Reconciling Information Exchange and Confidentiality: A Formal Approach
2007-03	Peter Mika (VU) Social Networks and the Semantic Web
2007-04	Jurriaan van Diggelen (UU)

Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UVA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramírez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbán (EUR)
Agent-Based Simulation of Financial Markets: A modular, continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UVA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06	Arjen Hommersom (RUN) On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective
2008-07	Peter van Rosmalen (OU) Supporting the tutor in the design and support of adaptive e-learning
2008-08	Janneke Bolt (UU) Bayesian Networks: Aspects of Approximate Inference
2008-09	Christof van Nimwegen (UU) The paradox of the guided user: assistance can be counter-effective
2008-10	Wauter Bosma (UT) Discourse oriented summarization
2008-11	Vera Kartseva (VU) Designing Controls for Network Organizations: A Value-Based Approach
2008-12	Jozsef Farkas (RUN) A Semiotically Oriented Cognitive Model of Knowledge Representation
2008-13	Caterina Carraciolo (UVA) Topic Driven Access to Scientific Handbooks
2008-14	Arthur van Bunningen (UT) Context-Aware Querying; Better Answers with Less Effort
2008-15	Martijn van Otterlo (UT) The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains.
2008-16	Henriette van Vugt (VU) Embodied agents from a user's perspective
2008-17	Martin Op 't Land (TUD) Applying Architecture and Ontology to the Splitting and Allying of Enterprises
2008-18	Guido de Croon (UM) Adaptive Active Vision
2008-19	Henning Rode (UT) From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search
2008-20	Rex Arendsen (UVA) Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven.
2008-21	Krisztian Balog (UVA) People Search in the Enterprise
2008-22	Henk Koning (UU) Communication of IT-Architecture
2008-23	Stefan Visscher (UU) Bayesian network models for the management of ventilator-associated pneumonia
2008-24	Zharko Aleksovski (VU) Using background knowledge in ontology matching
2008-25	Geert Jonker (UU) Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency
2008-26	Marijn Huijbregts (UT) Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled
2008-27	Hubert Vogten (OU) Design and Implementation Strategies for IMS Learning Design
2008-28	Ildiko Flesch (RUN) On the Use of Independence Relations in Bayesian Networks
2008-29	Dennis Reidsma (UT) Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans
2008-30	Wouter van Atteveldt (VU) Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content
2008-31	Loes Braun (UM) Pro-Active Medical Information Retrieval
2008-32	Trung H. Bui (UT) Toward Affective Dialogue Management using Partially Observable Markov Decision Processes
2008-33	Frank Terpstra (UVA)

Scientific Workflow Design; theoretical and practical issues
2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining
2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models
2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques
2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT
2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering
2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality
2009-06 Muhammad Subianto (UU)
Understanding Classification
2009-07 Ronald Poppe (UT)
Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08 Volker Nannen (VU)
Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09 Benjamin Kanagwa (RUN)
Design, Discovery and Construction of Service-oriented Systems
2009-10 Jan Wielemaker (UVA)
Logic programming for knowledge-intensive interactive applications
2009-11 Alexander Boer (UVA)
Legal Theory, Sources of Law & the Semantic Web
2009-12 Peter Massuthe (TUE, Humboldt-Universitaet zu Berlin)
Operating Guidelines for Services
2009-13 Steven de Jong (UM)
Fairness in Multi-Agent Systems
2009-14 Maksym Korotkiy (VU)
From ontology-enabled services to service-enabled ontologies (making ontologies work in e-science with ONTO-SOA)
2009-15 Rinke Hoekstra (UVA)
Ontology Representation - Design Patterns and Ontologies that Make Sense
2009-16 Fritz Reul (UvT)
New Architectures in Computer Chess
2009-17 Laurens van der Maaten (UvT)
Feature Extraction from Visual Data
2009-18 Fabian Groffen (CWI)
Armada, An Evolving Database System
2009-19 Valentin Robu (CWI)
Modeling Preferences, Strategic Reasoning and Collaboration in Agent-Mediated Electronic Markets
2009-20 Bob van der Vecht (UU)
Adjustable Autonomy: Controling Influences on Decision Making
2009-21 Stijn Vanderlooy (UM)
Ranking and Reliable Classification
2009-22 Pavel Serdyukov (UT)
Search For Expertise: Going beyond direct evidence
2009-23 Peter Hofgesang (VU)
Modelling Web Usage in a Changing Environment
2009-24 Annerieke Heuvelink (VUA)
Cognitive Models for Training Simulations
2009-25 Alex van Ballegooij (CWI)
"RAM: Array Database Management through Relational Mapping"
2009-26 Fernando Koch (UU)

An Agent-Based Model for the Development of Intelligent Mobile Services
2009-27 Christian Glahn (OU)
Contextual Support of social Engagement and Reflection on the Web
2009-28 Sander Evers (UT)
Sensor Data Management with Probabilistic Models
2009-29 Stanislav Pokraev (UT)
Model-Driven Semantic Integration of Service-Oriented Applications
2009-30 Marcin Zukowski (CWI)
Balancing vectorized query execution with bandwidth-optimized storage
2009-31 Sofiya Katrenko (UVA)
A Closer Look at Learning Relations from Text
2009-32 Rik Farenhorst (VU) and Remco de Boer (VU)
Architectural Knowledge Management: Supporting Architects and Auditors
2009-33 Khiet Truong (UT)
How Does Real Affect Affect Affect Recognition In Speech?
2009-34 Inge van de Weerd (UU)
Advancing in Software Product Management: An Incremental Method Engineering Approach
2009-35 Wouter Koelewijn (UL)
Privacy en Politiegegevens; Over geautomatiseerde normatieve informatieuitwisseling
2009-36 Marco Kalz (OUN)
Placement Support for Learners in Learning Networks
2009-37 Hendrik Drachsler (OUN)
Navigation Support for Learners in Informal Learning Networks
2009-38 Riina Vuorikari (OU)
Tags and self-organisation: a metadata ecology for learning resources in a multilingual context
2009-39 Christian Stahl (TUE, Humboldt-Universitaet zu Berlin)
Service Substitution – A Behavioral Approach Based on Petri Nets
2009-40 Stephan Raaijmakers (UvT)
Multinomial Language Learning: Investigations into the Geometry of Language
2009-41 Igor Berezhnyy (UvT)
Digital Analysis of Paintings
2009-42 Toine Bogers (UvT)
Recommender Systems for Social Bookmarking
2009-43 Virginia Nunes Leal Franqueira (UT)
Finding Multi-step Attacks in Computer Networks using Heuristic Search and Mobile Ambients
2009-44 Roberto Santana Tapia (UT)
Assessing Business-IT Alignment in Networked Organizations
2009-45 Jilles Vreeken (UU)
Making Pattern Mining Useful
2009-46 Loredana Afanasiev (UvA)
Querying XML: Benchmarks and Recursion

2010

2010-01 Matthijs van Leeuwen (UU)
Patterns that Matter
2010-02 Ingo Wassink (UT)
Work flows in Life Science
2010-03 Joost Geurts (CWI)
A Document Engineering Model and Processing Framework for Multimedia documents
2010-04 Olga Kulyk (UT)
Do You Know What I Know? Situational Awareness of Co-located Teams in Multidisplay Environments
2010-05 Claudia Hauff (UT)
Predicting the Effectiveness of Queries and Retrieval Systems
2010-06 Sander Bakkes (UvT)
Rapid Adaptation of Video Game AI
2010-07 Wim Fikkert (UT)

	A Gesture interaction at a Distance
2010-08	Krzysztof Siewicz (UL)
	Towards an Improved Regulatory Framework of Free Software. Protecting user freedoms in a world of software communities and eGovernments
2010-09	Hugo Kielman (UL)
	A Politiele gegevensverwerking en Privacy, Naar een effectieve waarborging
2010-10	Rebecca Ong (UL)
	Mobile Communication and Protection of Children
2010-11	Adriaan Ter Mors (TUD)
	The world according to MARP: Multi-Agent Route Planning
2010-12	Susan van den Braak (UU)
	Sensemaking software for crime analysis
2010-13	Gianluigi Folino (RUN)
	High Performance Data Mining using Bio-inspired techniques
2010-14	Sander van Splunter (VU)
	Automated Web Service Reconfiguration
2010-15	Lianne Bodenstaff (UT)
	Managing Dependency Relations in Inter-Organizational Models
2010-16	Sicco Verwer (TUD)
	Efficient Identification of Timed Automata, theory and practice
2010-17	Spyros Kotoulas (VU)
	Scalable Discovery of Networked Resources: Algorithms, Infrastructure, Applications
2010-18	Charlotte Gerritsen (VU)
	Caught in the Act: Investigating Crime by Agent-Based Simulation
2010-19	Henriette Cramer (UvA)
	People's Responses to Autonomous and Adaptive Systems
2010-20	Ivo Swartjes (UT)
	Whose Story Is It Anyway? How Improv Informs Agency and Authorship in Emergent Narrative

Summary

Service-providers collect more and more privacy-sensitive information, even though it is hard to protect this information against hackers, abuse of weak privacy policies, negligence, and malicious database administrators. In this thesis, we take the position that endless retention of privacy-sensitive information will inevitably lead to unauthorized data-disclosure. Limiting the retention of privacy-sensitive information limits the amount of stored data and therefore the impact of such a disclosure.

A problem of limited retention is that the retention period is often overstated in advantage of the service-provider. We model the interests of service-provider and user in an abstract, qualitative way; when such a model is quantitatively specified, it is possible to calculate the optimal retention period for the *joint* interest of service-provider and user.

The all-or-nothing behavior of limited retention is too rigorous: the data will be completely destroyed, also destroying any possible use of the data. *Progressively degrading* a data item instead of removing the data item in one single step makes it possible to balance the interests of service-provider and user better than limited retention can do. At every degradation step, the precision of a data item will be decreased. Degraded data is supposed to be less privacy-sensitive, and still usable enough to motivate a longer storage.

However, removing data from a database system is not a straightforward task; data degradation has an impact on the storage structure, indexing, transaction management, and logging mechanisms. To show the feasibility of data degradation, we provide several techniques to implement it; mainly, a combination of keeping data sorted on degradation time and using encryption techniques where possible. The techniques are founded with a prototype implementation and a theoretical analysis.

Finally, we explore several models which build further on the basic data degradation model. This gives different perspectives from which data degradation can be used.

Samenvatting

Dienstverleners verzamelen steeds meer privacygevoelige gegevens, ondanks het feit dat het moeilijk is om deze gegevens te beschermen tegen krakers, zwak privacybeleid, nalatigheid, en kwaadwillende gegevensbeheerders. In dit proefschrift nemen we de positie in dat het eindeloos bewaren van privacygevoelige gegevens onvermijdelijk zal leiden tot het op straat komen te liggen van deze gegevens. Door het beperken van de opslagperiode kan de privacyschending bij dergelijke gevallen ook beperkt worden.

Vaak wordt er een lange opslagperiode gehanteerd in het voordeel van de dienstverlener. Wij modelleren het belang van zowel de dienstverlener als de gebruiker op een abstracte, kwalitatieve manier; met een dergelijk model is het mogelijk een opslagperiode te bepalen die optimaal is voor het belang van zowel de dienstverlener als de gebruiker.

Door gegevens ineens volledig te verwijderen gaat ook elk mogelijk nut van de gegevens verloren. Door de gegevens langzaam te *vervagen*, wordt het mogelijk om een betere afweging te maken tussen het belang van dienstverleners en gebruikers. Bij elke vervagingsstap neemt de nauwkeurigheid van een gegeven af; wij nemen aan dat gegevens met een lagere nauwkeurigheid nog steeds bruikbaar zijn voor dienstverleners, maar ook minder privacygevoelig, zodat een langere opslagperiode acceptabel is.

Het vervagen van gegevens heeft invloed op de manier hoe gegevens intern technisch opgeslagen worden, het gebruik van indices, transactiebeheer, en logmechanismen. Om aan te tonen dat het vervagen van gegevens practisch doenlijk is, beschrijven we verschillende technieken om gegevensvervaging te implementeren. Dit komt hoofdzakelijk neer op het gesorteerd houden van gegevens op het tijdstip van vervaging, en het gebruik van encryptiemethoden. Onze technieken worden onderbouwd door een prototype en een theoretische analyse.

Daarnaast verkennen we verschillende modellen die voortborduren op het concept gegevensvervaging. Dit leidt tot verschillende perspectieven van waaruit het vervagen van gegevens in de toekomst verder ingezet kan worden.

Résumé

Les fournisseurs de services recueillent de plus en plus d'informations personnelles sensibles, bien qu'il soit réputé comme très difficile de protéger efficacement ces informations contre le piratage, la fuite d'information par négligence, le contournement de chartes de confidentialité peu précises, et les usages abusifs d'administrateurs de données peu scrupuleux. Dans cette thèse, nous conjecturons qu'une rétention sans limite de données sensibles dans une base de données mènera inévitablement à une divulgation non autorisée de ces données. Limiter dans le temps la rétention d'informations sensibles réduit la quantité de données emmagasinées et donc l'impact d'une telle divulgation. La première contribution de cette thèse porte sur la proposition d'un modèle particulier de rétention basé sur une dégradation progressive et irréversible de données sensibles.

Effacer les données d'une base de données est une tâche difficile à mettre en œuvre techniquement; la dégradation de données a en effet un impact sur les structures de stockage, l'indexation, la gestion de transactions et les mécanismes de journalisation. Pour permettre une dégradation irréversible des données, nous proposons plusieurs techniques telles que le stockage des données ordonnées par le temps de dégradation et l'utilisation de techniques ad-hoc de chiffrement. Les techniques proposées sont validées par une analyse théorique ainsi que par l'implémentation d'un prototype.