Dégradation Progressive et Irréversible des Données

Nicolas Anciaux[*], Luc Bouganim[*], Harold van Heerde[***], Philippe Pucheral[*,**], Peter M.G. Apers[***]

* INRIA Rocquencourt
Le Chesnay, France
<Fname.Lname>@inria.fr

** PRiSM Laboratory
University of Versailles, France
<Fname.Lname>@prism.uvsq.fr

*** CTIT
University of Twente, The Netherlands
{h.j.w.vanheerde,apers}@ewi.utwente.nl

RESUME

Notre activité quotidienne laisse des traces digitales dans un nombre croissant de bases de données (sites Web commerciaux, fournisseurs de service Internet, moteurs de recherche, etc.). Ces traces sont souvent exposées à des divulgations accidentelles, résultats de négligence, de piratage ou d'interrogation abusive encouragée par des chartes de confidentialité peu précises. Personne n'est à l'abri car une situation particulière (la recherche d'un emploi, une demande de crédit) peut rendre un historique, a priori quelconque, soudainement intéressant. Par définition, le contrôle d'accès ne peut empêcher ce type de divulgation, ce qui a motivé l'intégration du principe de conservation limitée des données dans les législations protégeant les données personnelles. Ce principe vise à effacer physiquement les données des bases de données après une période de temps prédéfini. Toutefois, ce principe est difficile à mettre en œuvre, ce qui conduit souvent à conserver des informations sensibles pendant des années. Dans cet article, nous proposons un modèle de dégradation des données dans lequel les données sensibles sont soumises à une dégradation progressive et irréversible depuis leur collecte (état précis), en passant par des états intermédiaires dégradés mais partiellement exploitables, jusqu'à leur disparition totale lorsqu'elles deviennent inutiles. L'avantage de la dégradation de données est double: (i) en réduisant la quantité de données précises, l'impact de la divulgation d'un historique (dégradé) est considérablement réduit et (ii) la dégradation des données en conformité avec les besoins des applications offre un nouveau compromis entre fonctionnalité et préservation de l'intimité. Nous analysons ensuite l'impact de ce modèle sur les techniques de base des SGBD (stockage, indexation et gestion de transactions) et proposons des solutions adaptées.

MOTS CLES

Modèle de rétention, dégradation des données, confidentialité des bases de données.

1. INTRODUCTION

People give personal data all the time to commercial web sites, search engines, web browsers, ISPs and credit-card companies. Personal digital trails end up in databases somewhere, where they can be analyzed to serve new purposes (e.g., behavioural targeting). More insidiously and continuously, cell phones, GPS devices, RFID tags and sensors are giving even more accurate information about our daily life (location, journey, consumption habits, hobbies, relatives). If individuals could be tempted to fulfil Vannevar Bush's Memex vision by recording their complete life [22], there is an unprecedented threat on privacy if others try to do so.

Personal digital trails are difficult to protect in practice. As any data, they are exposed to accidental disclosures resulting from negligence or piracy. To cite a few, the personal details of 25 million UK citizens have been recently lost inadvertently [29] and some of the data published by AOL about Web search queries of 657,000 Americans have been deanonymized [17]. Regarding piracy, even the most defended servers (including those of Pentagon [27], FBI [28][11] and NASA [11]) are successfully attacked. But more, personal digital trails are often weakly protected by obscure and loose privacy policies which are presumed accepted when exercising a given service. This fosters ill-intentioned scrutinization 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 utmost interest for someone else. Companies like Intelius or ChoicePoint make scrutinization their business while others like ReputationDefender provide a lucrative service to destroy the sensitive part of personal digital trails subject to scrutinization.

In this paper, we call *trail disclosure* the leakage of data pertaining to a personal digital trail and resulting from negligence, attack or abusive scrutinization or usage. By definition, a trail disclosure cannot be tackled by any security mechanism because its occurrence assumes that all security mechanisms have been bypassed or that the access control policy has been defined too weakly. Limiting the data retention period is a means by which the impact of trail disclosure can be reduced. Promoted by most legislations protecting personal data [13][23], the limited data retention principle consists in attaching a lifetime to a data compliant with its acquisition purpose, after which it must be withdrawn from the system. The shorter the retention period is, the smaller the total amount of data needlessly exposed to disclosure. Beyond the protection of personal data, the limited data retention principle is also a cornerstone of the ISO/IEC 27002:2005 recommendation for protecting enterprise information systems.

The limited data retention principle is however difficult to put in practice. The first difficulty comes from the determination of the right retention period for each data item. Depending on the data category and the country, minimal retention periods can be fixed for law enforcement or legal processes purposes (*e.g.,* banking information in UK cannot be destroyed before 7 years). In this case, the retention limit is set to this same value for privacy preservation purpose, but such limits are usually large. For the large amount of data not covered by law, the retention limit is supposed to reflect the best compromise between privacy preservation and application purposes reach. In practice, the same data item is likely to serve different purposes, leading selecting the largest retention limit compatible with all purposes, as suggested in [2]. More, the purposes exposed in most privacy policies are fuzzy enough to defend very long retention limits (years or decades), denaturing the initial principle. As a consequence, retention limits are seen by civil rights organizations as a deceitful justification for long term storage of personal data by companies [14]. The retention problem has become so important and the civil pressure so high that practices start changing. For instance, Google announced that cookies will expire after two years instead of being retained up to 2038 as before and

search engines like Ask and Ixquick advertize retention limits expressed in terms of days. The second difficulty related to limited data retention is its effective implementation. As pointed out in [25], no existing database system can guarantee that data cannot be recovered after a regular delete. Indeed, every trace of a deleted data must be physically cleaned up in the data store, the indexes and the logs, a technical issue still open today.

The approach proposed in this paper opens up a new alternative to reason about and implement limited data retention. It is based on the assumption that long lasting purposes can often be satisfied with a less accurate, and therefore less sensitive, version of the data [21]. For example, online companies record accurate data about the client purchases in order to process the transactions and the delivery. Then these records are kept for years in databases to focus advertising actions and increase profits. This recording is valuable for the client too because she can benefit from recommendations and special offers related to her purchase history. However, the purchase category is usually enough to process recommendations and offers (e.g., recording Book/Religion/Buddhism for a purchase in a bookshop is as informative as the exact record for this purpose) and the category could in turn become less informative over time (e.g., Book/Religion) without penalty because user preferences evolve as well. Equivalent privacy benefits can be foreseen by degrading attributes of many types of recorded events. For example, the exact location of a driver acquired by a navigation system needs to be maintained during a short period for guiding purpose and then could be degraded at a lower accuracy (e.g., city) to plan and optimize next trips. An accurate web history could be retained for the duration of a work task and then visited sites could be degraded to their topic classification to help forming communities of users sharing the same interest.

As exemplified above, the objective of the proposed approach is to progressively degrade the data after a given time period so that (1) the intermediate states are informative enough to serve application purposes and (2) the accurate state cannot be recovered by anyone after this period, not even by the server[1]. To the best of our knowledge, this paper is the first attempt to implement the essence of the limited data retention principle, that is limiting the retention of any information to the period strictly necessary to accomplish the purpose for which it has been collected. Hence, if the same information is collected to serve different purposes, degraded states of this information and their respective retention limits are defined according to each application purpose.

The expected benefit of our data degradation model is twofold:

- *Increased privacy wrt trail disclosure*: no information is exposed to disclosure longer and in a state of higher accuracy than strictly necessary to accomplish the purpose motivating its retention. To this respect, and contrary to current practices, data degradation implements a strict interpretation of the limited data retention principle.
- *Preservation of application reach:* compared to data anonymization, data degradation keeps the identity of the users intact, allowing for user-oriented purposes. The intermediate states of a degraded information are defined according to the purposes the user opts-in rather than to a current data distribution in the database.

As exemplified above, data degradation attempts reducing retention limits to their minimum, leading to a dramatic reduction of the total amount of data needlessly exposed to trail disclosure. An important question is whether data degradation can be reasonably implemented in a DBMS.

The contribution of this paper is twofold:

- We propose a simple and effective data degradation model, providing a clean and intuitive semantics to SQL queries involving degraded data.
- We identify technical challenges introduced by this model in terms of data storage, indexation

1 Data degradation can be seen as footsteps in the sand fading away when time passes by, only leaving vague traces of the original footprint.

and transaction management, and propose preliminary solutions.

The remainder of this paper is organized as follows: Section 2 positions data degradation in relation to related works and gives an intuitive representation of the expected privacy benefit. Section 3 introduces the data degradation model and gives definitions used throughout this paper. Section 4 identifies technical challenges raised by our model. Section 5 details the impact of data degradation on core database technology and proposes preliminary solutions. Section 6 discusses open issues, gives hints for implementing data degradation in a database server and finally discusses briefly performance issues. Section 7 concludes.

2. DATA DEGRADATION POSITIONING

2.1 Related Work

Existing works on data privacy and security which are related to the problem tackled in this paper can be grouped in four classes. They are quickly described below and put in perspective with limited data retention.

***Access & usage control*:** Access control models like DAC (*Discretionary Access Control*) and RBAC (Role-Based Access Control) are part of the SQL standard and are widely used in the database context to protect sensitive data against unauthorized accesses. More recently, efforts have been put in increasing privacy by providing the means to let donors *themselves* express their privacy requirements, and to control how data can be accessed and used by service providers. The platform for privacy preservation [12] applies the well known need-to-know and consent policies to web sites, allowing encoding privacy policies into machine readable XML [12]. Web sites describe their practices and any P3P compliant browser can be parameterized to reject policies hurting the privacy of the owner. P3P itself only *describes* policies and does not *enforce* them, making it little more than a standardized complement to privacy laws [15]. Techniques like the *privacy aware database* (PawS) goes a step further, letting the system automatically interpret and apply the policies to the data [18]. The work on Hippocratic databases [2] has been inspired by the axiom that databases should be responsible for the privacy preservation of the data they manage. The architecture of Hippocratic DBMS is based on 10 guiding principles derived from privacy laws, including the Limited retention principle more deeply discussed in next section.

The access & usage control approach is based on the assumption that the control is never bypassed. Thus, while contributing to the protection of data privacy, this approach does not answer the same problem as limited data retention, that is limiting the impact of trail disclosure. However, both approaches are complementary, considering that access & usage control remains necessary to regulate the use of mandatory information.

Server-based protection: In addition to access & usage control, security measures like *database encryption, database audit* and *intrusion detection systems (IDS)* are often in place at the server. The increase in security brought by database encryption is limited when decryption occurs at the server [7][16] and encrypting a large portion of the database may bring important overheads [24]. IDS [9] and database audit [24] and are both based on a constant monitoring of the system. IDS use a system of rules, generally based on attack scenario signatures, to generate alerts when abnormal behaviour is detected.

While all those techniques limit the risks of attacks, they do not prevent them totally and are obviously ineffective with respect to other sources of trail disclosure (e.g., negligence, weak policies, governmental pressure). Conversely, IDS are very effective to detect repetitive attacks. Combined with data retention, IDS will thus make it very hard for an attacker to obtain a large consecutive history of accurate data by continuously spying the database.

Client-based protection: When the database server cannot be fully and permanently trusted, an alternative is to rely on the donor herself, making her responsible for protecting her own data. In the *p4p framework* [1], the donor keeps control about which information to release to service providers.

P4P targets at the ‘paranoid’ users who does not trust the collecting organizations, in contrast to the users of P3P frameworks. Other client-based approaches advocate the encryption/decryption of the data at the client device [7][16] to make the system robust against server attacks.

Both approaches are not general, forcing all updates and queries on the database to go through the client, thus putting strong constraints on how applications are developed and deployed. Limited data retention does not put these restrictions on applications. Data can still be stored at the server side and still meets the targeted application needs while the definition of retention periods allows controlling the privacy risk.

Data anonymization: Anonymization is good practice when datasets have to be made public without revealing personal sensitive data, for example, when used for disclosing datasets for research purposes. Datasets can then be anonymized such that the privacy sensitive data cannot be linked to there owners anymore. *k*-Anonymity [26] is based on the idea of *masking* (parts of) the (quasi) identifier of a partly privacy sensitive tuple, such that the privacy part of the tuple will be hidden between *k-1* potential candidate identifiers within the same dataset. For example, the zip-code, date of birth and gender may uniquely identify an individual and reveal its corresponding sensitive data. By masking the date of birth, the dataset should contain at least *k* occurrences of the same <zip-code, gender> combination.

While k-anonymisation shares some similarities with our data degradation model, both pursue different objectives. In practice, *k*-anonymisation could result in a strictly *k*-anonymous database with the cost of loosing much usability, where other purposes than statistics computations cannot be satisfied. In addition, correctly anonymizing the data is a hard problem [20], especially when considering incremental data sets or when background knowledge is taken into account [19][17], as exemplified by AOL scandal [17].

In contrast, limited data retention is performed on an individual base and is particularly useful when data needs to be accurate for some time to make well defined services possible. Moreover, limited data retention can keep the identifier of the donor intact; hence, user-oriented services can still exploit the information to the benefit of the donor.

2.2 Degradation vs retention

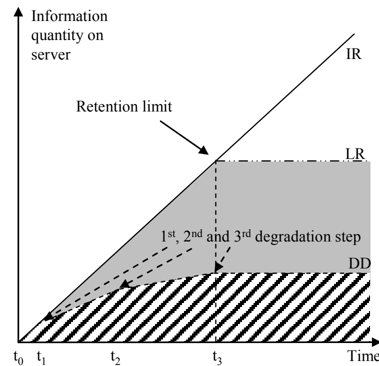

Figure 1. *Data degradation impact.*

Limiting data retention thus remains the ultimate barrier to trail disclosure. But to the best of our knowledge, this paper is the first to concretely and accurately address the implementation of this principle. [2] suggests including limited data retention in the design of Hippocratic databases but leaves it for future works. Moreover, the approach suggested in [2] is different since a data item that is likely to serve different purposes sees its retention limit extended to the duration required by the longest lasting purpose. We refer hereafter to this principle by the term lax limited data retention, or *lax retention* for short, to distinguish it from the *strict retention* implemented by data degradation. For a given information serving different purposes, data degradation fixes the level of accuracy and the retention limit strictly needed by each purpose, thereby organizing the lifecycle of this information from its acquisition up to its final destruction. Doing this, the amount of excessive information exposed to trail disclosure decreases over time much rapidly than with lax retention.

Figure 1 gives an intuitive representation of the privacy benefit provided by data degradation. Curve *IR* (Infinite Retention) plots the total amount of information gathered in a traditional database over time assuming a constant tuple

insertion rate. Curve *LR* (Lax Retention) shows that the amount of information kept available online, and then exposed to disclosure, remains constant once the retention limit has been reached (at t_3) instead of increasing linearly. Indeed, after this threshold, the tuple insertion and deletion rates are equal. Curve *DD* (Data Degradation) shows that the insertion rate in terms of "quantity" of information starts decreasing once the first degradation step is reached (at t_0). Actually, the number of tuples acquired by time unit is the same as with IR and LR but the accuracy of tuples acquired at time t_0 starts degrading at time t_1 and the degradation rate equals the insertion rate. The slope of *DD* decreases again after t_2 (second degradation step) up to become null at time t_3 (as for LR, deletion and insertion rates are equal after t_3). The integral of each curve can be interpreted as the total amount of information exposed to trail disclosure at any time. Though informal, this figure gives a clear intuition about the benefit provided by data degradation with respect to trail disclosure.

3. LCP DEGRADATION MODEL

3.1 Degradation policy and tuple states

In our data degradation model, called Life-Cycle Policy (LCP) model, data undergoes a progressive degradation from a precise state at data collection time to intermediate less accurate states, to elimination from the database. We capture the degradation of each piece of information (typically an attribute) by a Generalization Tree (GT). A generalization tree prescribes, given a domain generalization hierarchy of the corresponding attribute, the levels of accuracy the attribute can take during its life time.

For simplicity we chose here to use a crisp generalization tree (as defined more precisely below), although techniques for fuzzy generalization hierarchies exists and could be applicable to our degradation model [3].

Definition: Generalization Tree (GT)
A generalization tree is a rooted tree where:

- The leaves of the tree contain the most accurate values of the domain.
- The parents contain the value of the child node after one degradation step. Hence, a path to an ancestor in the GT expresses all forms a value can take in its domain.
- The root of the GT contains the null value.

What "more accurate" means depends on the domain, but for example if the domain is location then node n may contain the value *Los Angeles* and the parent of n may contain *California.*

Throughout this paper we will assume that a GT is defined for each domain of each degradable attribute d_i and is denoted $GT_{domain(di)}$. The parent value of value *v* of domain *D* is obtained using the function $GT_{D.Parent(v)}$. In practice, a GT can be implemented in different ways. Let us consider first a domain where all domain values are finite and identified (e.g., domain Location). The corresponding GT levels might correspond to data type extension, e.g.,: {address} → {city} → {province} → {country} → Ø. If the domain of values is infinite, each level of the GT could be represented as a range and the degradation steps could be implemented by means of functions, e.g.,: Range100(s) = round(s,100) → Range1000(s) = round(Range100(s), 1000) → Range5000(s) = round(Range1000(s),5000) → Ø, where round(x,y) is a function mapping a Salary value x to a set of intervals of accuracy y€.

Whatever the form of the GT, we assume that the degradation states match predefined application purposes. To this extent, data degradation pursues an orthogonal objective to data anonymization.

A Life-Cycle Policy (LCP) governs the degradation process by fixing how attribute values navigate from the GT leaves up to the root. While we may consider complex LCPs (see Section 6.1), we make the following simplifying assumptions: (1) LCP degradations are triggered by time, (2) LCP policies are defined on an attribute basis and (3) LCP policies apply to all tuples of the same table uniformly (rather than being user dependent). Typically, LCP policies may be defined by federal or national privacy agencies and imposed to database service providers or suggested by the service providers themselves to attract privacy conscious clients.

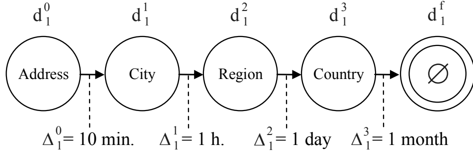

Figure 2. *An example of an attribute's LCP.*

Definition: Life-Cycle Policy (LCP)

A Life-Cycle Policy for an attribute is modeled by a Deterministic Finite Automaton as a set of attribute states (taken from the attribute domain GT) denoting the level of accuracy of the corresponding attribute, a set of transitions between those states and the time delay after which these transitions are triggered. More precisely:

- The initial state of an attribute d, denoted by d^0, is the value of this attribute acquired at insertion time of this tuple
- A transition $d^j \rightarrow d^{j+1}$ is triggered after a time delay Δ^j called attribute state duration.
- A transition $d^j \rightarrow d^{j+1}$ has the following effect: $d^{j+1} \rightarrow GT_{domain(di)}.Parent(d^j)$
- The final state, denoted by d^f, is the empty state corresponding to the root of GT, meaning that the value has been physically erased from the database.

LCP policies are defined per degradable attribute. A tuple is a composition of *stable attributes*, denoted by s_i, which do not participate in the degradation process and *degradable attributes*, denoted by d_i, which participate in the degradation process, each through its own LCP. A tuple carrying multiple degradable attributes will be subject to multiple degradation steps. This leads to define the notion of tuple state as follows.

Definition: Tuple State

The combination of LCPs of all degradable attributes of a tuple makes that, at each independent attribute transition, the tuple as a whole reaches a new *tuple state* t^k, until all attributes have reached there final state. As pictured in Figure 3, the life cycle of a tuple can thus be seen as a set of transitions between states derived from the combination of each individual attributes' LCP. More precisely:

- The initial state of a tuple t, denoted by t^0, is defined by $\forall i$, $t^0.d_i = d_i^0$.
- An attribute transition $t.d_i^j \rightarrow t.d_i^{j+1}$ results in a tuple state transition $t^k \rightarrow t^{k+1}$
- $t^k.d_i$ denotes the state of attribute d_i during the time period where t is in state t^k.
- Δt^k denotes the duration of a tuple state t^k
- $\Delta\Psi^k$ denotes the computability period of a tuple state t^k, representing the period (starting from the tuple insertion in state t^0) during which state t^k can be computed, i.e., during which every attribute is as accurate as or more accurate than in state k: $\Psi^k = \Sigma_{j=0 \text{ to } k}(\Delta t^j)$.

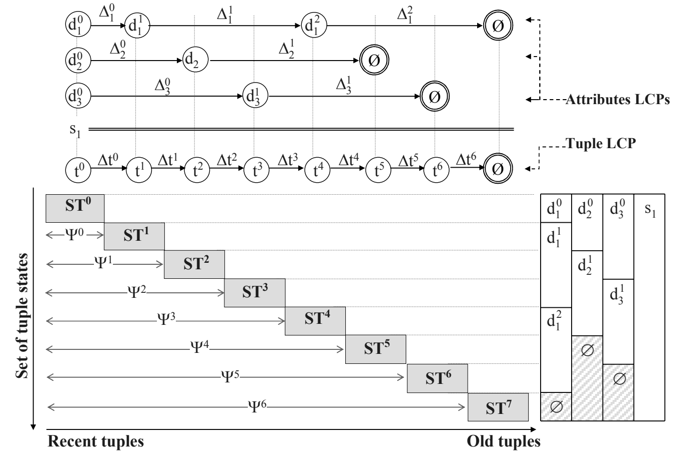

Figure 3. *Relationships between attribute states, tuple states and set of tuple states.*

Definition: Set of Tuple States (ST)

A set of tuple states, denoted by ST^i, is the set of all tuples t in state t^i. Hence, all tuples in a set of tuple state ST share the same levels of accuracy for all their attributes.

3.2 Impact on the Query Language

As introduced earlier, the LCP degradation model has been designed such that the degradation states match identified application purposes. This information is captured in each GT. Thus, we do the natural assumption that applications are GT aware and express query predicates according to the level of each GT they need to accomplish their purpose.

Our objective is to modify the SQL syntax as little as possible while providing a clear semantics for both selection and update queries in this context.

For the sake of simplicity, we consider below queries expressed over a single relational table.

The basic principle is, for a given application purpose, to declare the levels of accuracy required for each degradable attribute of interest through a specific statement of the form:

DECLARE PURPOSE MYPURPOSE

SET ACCURACY LEVEL L_j FOR R.d_i, ..

L_j refers here to the jth level of GT.domain(d_i). Based on this principle of defining purposes, the semantics of SQL queries is captured by the following definitions.

Definition: Domain of a purpose
Let A be a set of $<d, L>$ pairs capturing the accuracy level of all degradable attributes of interest for a purpose P. The domain of a purpose $P_A=\{<d_i,L_j>\}$, denoted by $D(P_A)$, is the union of all sets of tuple states containing tuples having an accuracy level equal to or greater than the requested one for all their degradable attributes. More formally:

$D(P_A) = \bigcup_{i=0}^{k} \{ f_{i\to k}(t) / t \in ST^i \}$ where $f_{i\to k}$ is a function degrading tuples t in ST^i to ST^k for $i \leq k$

Definition: Database view of a purpose
The database view of a purpose P_A, denoted by $V(P_A)$, corresponds to the projection of all tuples belonging to the domain of a purpose, over the degraded attributes of interest with the requested accuracy. This means that attribute values of tuples belonging to $D(P_A)$ are degraded in $V(P_A)$ if their accuracy is higher than the one expected by the purpose.

Once a purpose P has been declared, queries can be expressed with no change on the SQL syntax. The set of tuples considered by a Select statement related to PA is simply V(PA). Let us illustrate this with the following example given a table Person(name, location, salary, …).

DECLARE PURPOSE STAT SET ACCURACY LEVEL COUNTRY FOR PERSON.LOCATION, RANGE1000 FOR PERSON.SALARY

SELECT * FROM PERSON WHERE LOCATION LIKE"%FRANCE%" AND SALARY = '2000-3000'

V(Stat) will contain all tuples from the PERSON table for which both attributes location and salary have at least the accuracy Country and Range1000. All other tuples are discarded. Before evaluating a predicate or projecting a tuple on a degraded attribute, the value of this attribute is automatically degraded up to the requested level of accuracy (thanks to the GTs) if required.

The semantics of update queries is as follows: delete query semantics is unchanged compared to a traditional database, except for the selection predicates evaluated over $V(P_A)$. Hence, 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. We made the assumption that insertions of new elements are granted only in the most accurate state (ST^0). Finally, we make the assumption 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.

The primary objective of this section is introducing a simple and intuitive language to manipulate a database implementing the degradation model defined in Section 3.1. More sophisticated semantics could be devised to allow direct insertions and updates into ST^i with $i>0$. More sophisticated query semantics could also be devised, taking advantage of previous works conducted in fuzzy databases [8] and probabilistic databases [5]. For instance, queries could consider tuples outside the domain of a purpose as defined above (i.e., tuples having an accuracy level less than expected by the query purpose) and deliver probabilistic results. We left such studies for future work.

3.3 Impact on transaction semantics

User transaction inserting tuples with degradable attributes generates effects (i.e., database updates) all along the lifetime of the degradation process, that is from the transaction commit up to the time where all inserted tuples have reached a final LCP state for all their degradable attributes. This significantly impacts the transaction semantics since a transaction commit implicitly binds a

contract for future updates. Conceptually, a transaction T inserting new tuples can be split into a main transaction T^0 modifying ST^0 followed by a succession of degradation subtransactions T^k, each T^k degrading the state generated by T^{k-1} after a time interval fixed by the LCP. Thus degradation subtransactions work on the behalf of their main transaction. We could do a parallel with nested transactions, sagas or other advanced transaction models but the comparison stops here. Indeed, the peculiarities of degradation subtransactions are that they implement a part of an already committed transaction, then their own commit is mandatory and must respect time constraints. Transactions which do not insert new tuples in ST^0 (i.e., reader transactions, writer transactions performing only deletes and/or updating only stable attributes) are called regular transactions. We revisit the definition of the usual ACID properties in this light.

Δ-Atomicity: a regular transaction is atomic in the usual sense meaning that either all or none of its effects are integrated in the database. Let T^0 be a main transaction, T^0 is said to be Δ-atomic, meaning: (1) T^0 is atomic with respect to all its effects in ST^0 and (2) . $k>0$, all T^k effects must be integrated in ST^k in any situation. Δ-Atomicity assumes that no reason other than a failure can cause an abort of T^k and that the recovery process will enforce atomicity even in this case.

Consistency: consistency has the usual meaning that no integrity constraints are violated. Δ-Atomicity precludes aborts of degradation subtransactions due to a runtime violation of integrity. Hence, integrity constraints must be checked by the main transaction for all subsequent updates generated by the degradation process. To enforce this property, we make the assumption that integrity constraints are compiled into each GT so that each degradation step is certified consistent a priori.

Isolation: conflicts may occur between regular transactions, main transactions and degradation subtransactions. Regular and main transactions can use traditional SQL isolation levels to protect their execution properly and get the expected view of the database. Degradation subtransactions need simply to be protected against concurrent deletes generated by uncommitted regular and main transactions in the same ST.

Δ -Durability: the effects of regular transactions are durable in the usual sense. The effects of main transactions are said Δ-Durable. Δ-Durability means that, for each tuple t inserted by a committed main transaction, the history $t^0 < t^1 < \ldots < t^i < t^f$ is guaranteed in spite of any subsequent failures, where:

t^f denotes the empty state produced by a tuple deletion.

. $\forall k$, the states t^k and t^{k+1} are exclusive meaning that after Δt^k, t^k is atomically replaced by t^{k+1} and cannot be recovered.

Degradation subtransactions have no transactional properties on their own, other than a requirement for a degraded form of isolation. However, they play an important role in the enforcement of the Δ-ACID properties of the main transactions. Notably, they must enforce a timeliness property underlying Δ -Durability. Timeliness is more precisely defined as follows.

δ-Timeliness: To enforce Δ-Durability, a degradation subtransaction T^k is assumed to degrade the state generated by T^{k-1} after a time delay equal to Δt^k. The time delay is initialized at T^{k-1} commit. Respecting this time delay strictly (e.g., in the second) would incur severe performance penalty with no foreseeable benefit in practice. Thus, we introduce a slightly weaker property called δ-Timeliness where δ is a time tolerance associated to the degradation process. Under this property, Δ-Durability guarantees state changes within a time window ($\Delta \pm \delta$).

4. TECHNICAL CHALLENGES

Whenever an extension is proposed to a database model, and whatever the merits of this extension is, the first and legitimate question which comes in mind is how complex will the technology be to support it. Can existing DBMSs be extended with no impact on the kernel, should a few well identified core database techniques be revisited or is a complete redesign of the DBMS mandatory?

Identifying the exact impact of making a DBMS data-degradation aware leads to three more precise questions.

How to enforce Δ-Durability and Δ-Atomicity over degradable data? As stated in Section 3.3, updates over stable data must be atomic and durable, as usual. The novelty is thus in the management of degradable attributes. Δ-Durability enforces that the ith state of a tuple remains durable during Δt^i and can in no way be recovered after this period. As pointed out in [25], traditional DBMSs 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 candidate techniques can be used to tackle this issue in our context. The first one is overwriting the data with its degraded value at each degradation step, using a dummy value when the data reaches the final state of its LCP. The second one is precomputing all degraded versions of a data at insertion time and storing them encrypted with a different key (along with an identification of this key in plaintext). At degradation time the corresponding key(s) will be destroyed, making the data undecryptable. These two techniques exhibit opposite behaviors in terms of access efficiency (depending on whether the data must be decrypted) and degradation efficiency. The storage of degradable attributes, indexes and logs have to be revisited in this light. The performance problem is particularly acute considering that each tuple inserted in the database undergoes as many degradations as tuple states. The second impact of Δ-Durability and Δ-Atomicity is on the recovery protocol itself.

How to speed up queries involving degradable attributes? Traditional DBMS have been designed to speed up either OLTP or OLAP applications. In OLTP workloads, insertions are massive, queries are simple and usually highly selective, and transaction throughput is the main concern. This leads to the construction of relatively few indexes on the most selective attributes to get the best trade-off between selection performance and insertion/update/deletion cost. In OLAP workloads conversely, 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 thanks to bitmap-like indexes. Data degradation can be useful in both contexts. However, data degradation changes the workload characteristics in the sense that OLTP queries become less selective when applied to degradable attributes and OLAP must take care of updates incurred by degradation. This introduces the need for indexing techniques supporting efficiently degradations. Query optimization may also impact tuple storage and index management because queries apply to a purpose view potentially built from several sets of tuple states of different accuracies.

How to guarantee δ-Timeliness? Timeliness is a fundamental property of a degradation model and the δ-tolerance is introduced for the sole purpose of performance. We believe that δ should remain application dependent, but our intuition suggests a direct relationship between Ψ and δ (i.e., the shorter the computability period Ψ^k of a tuple state t^k, the smaller the tolerance δ^k to degrade it). For this reason, and for the sake of simplicity, we consider in the following that δ is directly proportional to Ψ, that is $\delta^k=\rho\Psi^k$, where ρ is a constant for the system (e.g., ρ = *1%*). Whatever the degradation strategy, ensuring δ-Timeliness forces degradation subtransactions to be executed and completed in the time window (Δ ± δ). Implementing degradation subtransactions in a traditional DBMS by means of normal transactions may lead to conflicts, then to deadline misses and even to deadlocks. On the other hand, degradation subtransactions cannot run without any isolation control, forcing a new synchronization protocol to be designed.

Next section focuses on technical issues related to the three questions above. Alternatives regarding the degradation process and its synchronization, the storage model, the indexing model and the logging and recovery management will be discussed in the next sections. For the sake of simplicity, we focus the discussion on the degradable part of the database, as if a vertical

partitioning was made between stable and degradable attributes[2].

5. IMPACT OF DATA DEGRADATION ON CORE DATABASE TECHNIQUES

As stated in Section 4, classical transactional protocols can be used to synchronize the read/write activity of regular and main transactions and deliver the desired isolation level [6] between them. The novelty introduced by the LCP model is that degradation subtransactions change the database state steadily and may also generate conflicts. Solving these conflicts by executing degradation subtransactions in the scope of standard transactions has been shown a poor alternative both in terms of blocking and deadlock probability and in terms of performance (there is n times more such degradation subtransactions than main transactions where n is the number of tuple states).

To decrease the total degradation cost, the idea is taking advantage of the time tolerance brought by δ-Timeliness to group a set of degradation subtransactions into a single large degradation step. At first glance, this solution seems counter-productive by increasing the duration of the degradation step and thereby the conflict probability. In fact, the benefit is high considering that: (1) the number of execution threads is divided by the grouping factor, (2) the I/O generated in the data space, index space and log space can be grouped and produce sequential I/O and (3) the guarantee of success of degradation subtransactions can be exploited to avoid most conflicts. Point 1 is self-explanatory; point 2 will be more deeply discussed in Sections 5.2 to 5.4 so that this section focuses on point 3.

5.1 Avoiding subtransactions conflicts

We propose a *Least Effort Degradation* process where the degradation is performed at the coarsest granularity authorized by the δ tolerance (i.e., the laziest interpretation of Timeliness). Let us consider one set of tuple state ST^k with a degradation tolerance δ^k. For ST^k, a degradation step *DS* will be triggered at every δ^k time interval. The nth *DS* triggered will enforces the effects of all degradation subtransactions T^k planned during the interval $[\Psi^k+n\delta^k, \Psi^k+(n+1)\delta^k[$. Tuples, index entries and log records have to be synchronized to make each set of tuple state consistent. This is exactly what a degradation subtransaction guarantees. Hence, a *Degradation Agenda* DA^k is used to record the degradations to be performed in every files participating in ST^k. DA^k is a queue filled by main transaction commits and consumed by degradation steps. To know which records are actually relevant to a degradation step in a given file, we make the simplifying assumption that every record is time-stamped with the commit date of the main transaction having inserted it (better solutions dependent of the file organization will be discussed next).

Let us now consider how a transaction *T* (regular or main) working on a database view involving the sets of tuple states ST^0 to ST^k must be synchronized with a degradation step DS. The first observation is that only a degradation of the oldest tuples, i.e., those in ST^k, may change the database view of *T*. Indeed, *DS* is done on behalf of Δ-atomic degradation subtransactions (commit is guaranteed). Hence, the value delivered to T of a tuple *t* in ST^i or in ST^{i+1} (with $i<k$) is guaranteed to be identical after its projection on state *k*. Thus *T* and *DS* do not need to be synchronized on ST^0 .. ST^{k-1}. The second important remark is that synchronization is still not necessary if *T* selects an isolation level lower or equal to the SQL Read Committed level. Indeed, *DS* cannot generate dirty reads since *DS* effects are done on behalf of Δ-atomic degradation subtransactions.

Hence, synchronization is necessary only with isolation levels higher than Read Committed and when *DS* degrades ST^k. We propose a protocol where locks are requested on time intervals. When the nth *DS* is triggered, it requests an exclusive lock on the interval $[n\delta^k, (n+1)\delta^k[$ since it will degrade all data time-stamped within this interval. Similarly, *T* must request a shared lock on the intervals the accessed data belong to. If a conflict

[2] Such partitioning could make sense in practice, with the benefit to keep standard behaviour and performance on the stable part of the database.

occurs and *DS* is blocked, it will remain blocked at most until its deadline δ^k is met[3]. At this time the blocking transaction is aborted to enforce Δ-atomicity and δ-Timeliness of all degradation subtransactions. This situation is rather unlikely considering that δ^k is supposed to be much larger than a transaction duration. If a conflict occurs and *T* should be blocked, it is useless for *T* to wait since the accessed data will leave *T*'s database view.

In addition to the performance benefit brought by *Least Effort Degradation*, the synchronization protocol presented above exhibits the following interesting features: (i) it is independent of the way data, index and logs are managed, assuming they are time-stamped; (ii) it minimizes the impact on main and regular transactions, never blocking them and never aborting transactions shorter than δ^k.

5.2 Storage model for degradable attributes

The storage model selected for the degradable attributes must cope with two contradictory objectives: (1) minimizing the cost of implementing an unrecoverable degradation and (2) optimizing queries. The second objective disqualifies degradation by encryption since this would incur a decryption every time a degradable attribute participates in a query and since this would make it difficult to index encrypted attributes. Partial solutions exist for the latter point [16] but the loss of index accuracy is usually high, making these solutions not relevant in contexts other than privacy preservation. The remaining solution, that is overwriting attributes at degradation time, can be implemented in various ways: shall we store degradable attributes altogether, separately, ordered by degradation date, can the degradation be prepared by a precomputing phase as suggested for degradation by encryption? There are actually two main dimensions dictating the storage model:

Clustered vs. Fragmented: Clustered means that all degradable attributes of the same tuple are stored together while Fragmented means that degradable attributes are vertically partitioned. The benefit of clustering is optimizing the execution of queries involving several degradable attributes. The benefit of fragmentation is minimizing the quantity of data to be degraded at each tuple state change.

Lazy vs. Eager: Lazy means that the degradation overwrites a data item at the time of a state change according to the δ-Timeliness. Eager means that the degradation is precomputed, leading to store all states of the same data item at insertion time and to destroy them one by one at degradation time. The benefit of Lazy is avoiding data duplication among states while the benefit of Eager is implementing degradation by deletions rather than by updates, assuming that deletions could be physically performed more efficiently than updates[4].

Both dimensions can be combined leading to four possible storage models: Clustered-Lazy Storage (CLS), Clustered-Eager Storage (CES), Fragmented-Lazy Storage (FLS) and Fragmented-Eager Storage (FES). Figure 4 illustrates these four alternatives and show how the files images evolve over time in each model.

Whatever the data format in a file (single attributes or group of attributes depending on the Clustered/Fragmented option) and the number of files impacted when inserting new tuples (depending on the Lazy/Eager option), there is a high benefit of ordering a file's items according to the data degradation date. Following this principle, degradation can be performed in a set-oriented way taking advantage of sequential I/O. Also, since data items share the same degradation delay, ordering the file on the degradation date is equivalent to respecting the commit ordering. However, taking full advantage of sequential I/O

[3] We do the assumption that the time spent to physically perform the degradation is insignificant with respect to δ and can be neglected.

[4] Deletes are less costly than updates by definition since they do not need to read the existing value. Deletes can be further optimized by organizing the file in a circular way such that new insertions naturally erase old values. We do not discuss this optimization further since it apply only to specific situations (i.e., a constant throughput is required).

requires ad-hoc buffering and the degradation policies detailed below.

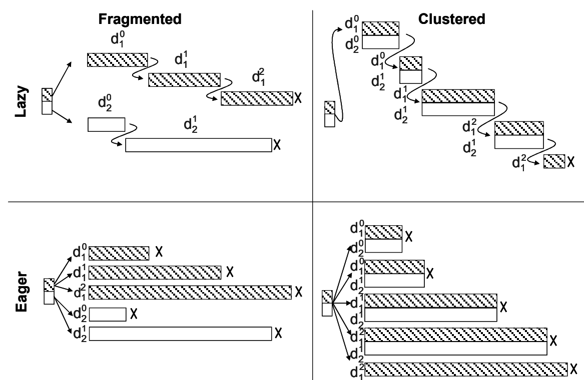

Figure 4. *Storage model alternatives.*

Whatever the storage model, an LCP generating *k* tuple states leads always to manage *k* data files and induces always *k* deletions or degradations until reaching the tuple final state (see Figure 4). In the following, we denote by f^k the data file containing the tuple state ST^k so that, degrading physically file f^i is logically equivalent to degrading ST^i.

Insertions: Inserted tuples are buffered in an *Uncommitted Inserts Buffer (UIB)* in RAM until transaction commit. Buffering uncommitted tuples is necessary to guarantee a correct ordering in the file in case of transaction abort. At commit time, tuples move from the UIB buffer to *Insert Buffers* associated with each file[5], potentially suffering degradations (e.g, in CES/FES models). When an *Insert Buffer* is full, it is flushed to disk, generating sequential I/Os.

Degradation Agenda: As already mentioned DA^k records the degradations to be performed in every files participating in ST^k, the data file f^k being one of them. Since degradation is performed at δ^k granularity (Least Effort Degradation) and since f^k is ordered on the degradation date, DA^k cardinality can be limited to $\Psi^k/\delta^k + 1$ entries, thus allowing it to be kept in RAM. Each DA^k entry simply stores the offset of the most recent data stored in f^k that must be degraded/deleted by the corresponding degradation step. More precisely, the nth instance of a degradation step DS refers to $DA^k[n]$ to retrieve the offset of the last data item inserted in f^k by the last main transaction committed in the interval $[\Psi^k+n\delta^k, \Psi^k+(n+1)\delta^k[$. Note that registering offsets in DA makes time-stamping the data useless.

Degradation buffers: For CLS and FLS, a *degradation buffer* is used as follows. When the nth *DS* is triggered, the data stored in the range $]DA^k[n-1], DA^k[n]]$ are loaded in the *degradation buffer*, then degraded, written to their destination file, and the range $]DA^k[n-1], DA^k[n]]$ is physically erased in f^k. All these operations can be done by sequential I/O. Obviously, a data range $]DA^k[n-1], DA^k[n]]$ is likely not to be aligned to disk page frontiers (this is particularly true for ranges smaller than a page). To avoid repetitive I/O of a same page in that case, a page-aligned superset of the range is read in the *degradation buffer* and degraded data are also produced on a page basis[6]. Unaligned data ranges introduce a requirement for a buffer even for CES and FES, but in that case, the buffer can be fairly small (up to one I/O page).

5.3 Indexing Model

The distinguishing characteristics of indexes over degradable attributes are that the same attribute may be considered with different accuracy levels and that indexes must be degradable as well.

Let us consider first the multi-accuracy problem. Mixing key values of different accuracy levels in the same index will increase its size and then decrease its performance with no benefit at query time. So we suggest that each index contain keys related to a single accuracy level. The second point is that an index built over attribute state d_i^k must index all the tuples containing this attribute state to avoid maintaining one index per tuple state and to avoid scanning several indexes to evaluate a single predicate. The third point is whether there is a benefit to continue indexing attributes entering a low accurate state. Indeed, given the nature of degradation, the lower the accuracy, the lower the

[5] Insert buffers can be shared by several files (e.g., with CES, one insert buffer can be use for performing all the inserts in the different files).

[6] This introduces a slight complexity in the recovery process since the database state on disk is no longer consistent wrt the degradation steps.

selectivity of the index. We believe that indexing highly degraded attributes (i.e. creating a multi-dimensional index) may make sense to benefit from combined selectivities of several non-selective predicates.

Let us now consider the index degradation problem, starting the study by traditional B+Trees. While B+Trees scale gracefully in terms of number of indexing elements, their tree-like structure makes them badly adapted to degradation, precluding any locality of updates and then generating random I/O. Encryption could be considered as a solution. Assuming that index keys themselves are not sensitive and remain in clear text, the references to the indexed tuples could be encrypted using degradable encryption keys, following a principle similar to the one described in Section 4. The *Least Effort Degradation* mechanism suggests using the same key for all references to be degraded by the same degradation step. Additional information is added to the Degradation Agenda of the index file in order to destroy the adequate key at degradation time. To prevent information disclosure which could occur by joining different indexes applied to the same data file on their encrypted references, distinct encryption keys must be used by degradation step and by index. Encryption keys are stored in an unordered array and are referenced by the index entries. The garbage collection problem, that is eliminating the index entries corresponding to degraded data, is more acute than on the data files simply because stale index entries augment the size of the index and decrease its performance. We suggest cleaning up index nodes lazily at the time of the next index node update. All node entries are then scanned to try to decrypt their reference. If the decryption does not success (a constant marker is associated with each reference to make this test possible), this means that the decryption key has been degraded and the index entry can be removed.

Low cardinality domains can be indexed thanks to bitmaps as usual. Bitmaps are sequential data structures and thus support insertion and degradation gracefully. Medium cardinality domains can be indexed by Value-List indexes [10]. This more sophisticated bitmap encoding introduces an interesting trade-off between the number of bitmaps to be maintained and updated at tuple insertion time and the number of bitmaps to be read at lookup time. However, this strategy remains inoperative for large domains.

To deal with high domain cardinality (i.e., accurate levels of the GT) in an insert/erase intensive context, we propose an alternative to encrypted B+Trees and Value-List indexes called Hash Sequential Lists (HSL). Roughly speaking, HSL are hash buckets containing sequential lists (i.e., ordered by insertion date) of <value, pointer> where *value* is the value of the indexed attribute and *pointer* a reference to the tuple having this value. When a tuple is created, a new pair <value, pointer> is simply inserted into the bucket determined by the hash function, with no additional computation. For exact match queries, a single bucket is fully scanned to find all the tuples matching the predicate. To enable range queries, Range Sequential Lists (RSL) can be designed using a range partitioning function.

Note that the buffering and degradation strategies discussed in Section 5.2 apply to HSL, RSL and bitmaps as well since they are all sequential data structures.

5.4 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 commit (Steal strategy), after a transaction commit (No Force strategy) as well as optimizing the recovery process in case of failure (Checkpoint and fast recovery techniques). Our goal is to keep, whenever possible, these interesting optimizations on the degradable part of the database. Note that logging and recovery for stable data are assumed to be ensured classically.

5.4.1 Undo Log

Degradation subtransactions are guaranteed to never rollback even in case of failure (see the recovery process described below). Thus, the undo

log is used to ensure the atomicity of main and/or regular transactions only. Since updates are not allowed on degradable attributes and inserts are buffered in RAM (in the UIB) until transaction commits (thus enforcing *No Steal*), the sole operation that needs to be undone is DELETE. To avoid requiring a degradation of the undo log we propose to log only the transaction id and the references of the deleted tuple rather than before images. In case of a rollback, the deleted tuple can be recovered from the redo log (see below) which still contains an image of this tuple in the correct accuracy (Δ-Durability).

5.4.2 Redo Log

The redo log includes (i) the images of tuples inserted by main transactions; (ii) the references to tuples deleted by regular or main transactions; and (iii) the transaction begin and commit statements. Time-stamping the commit statements allows to replay, if necessary, degradation subtransactions and to rebuild data and index files (see below). We suggest encrypting the images of inserted tuples following the principle described in Section 4 since the redo log is not subject to queries. This enables fast degradation without any access to the redo log (encryption keys are simply erased). The overhead of managing a redo log compliantly with Δ-durability induces thus a negligible encryption overhead [25], and one I/O for key overwriting each time interval δ for each attribute state.

5.4.3 Recovery

Let us first consider a cold recovery process rebuilding entirely the database state using the redo log file. Since the redo log includes the complete history of main and regular transactions, along with commit timestamps and key references, this history can be replayed in the same order leading to rebuild the data files, the index files and the Degradation Agenda. However, since some date keys have potentially been erased, insertions corresponding to erased keys are ignored, leading to recover the database in the same state as it was just before the system failure. Thus, before returning to a normal usage, the database must perform all necessary degradations to make the database state compliant with the LCP policy, considering the current date. This can be done by applying the degradation planned in the Degradation Agenda. For a warm recovery after a system failure, additional information is required to synchronize the log content with the data files and index files content. This information is precisely the one contained in the Degradation Agenda which must then be logged, similarly to traditional checkpoint information.

6. OPEN ISSUES

This section reviews the choices made so far for the model and discusses other alternatives and the interest of considering them in future works. It discusses different alternatives to implement a data degradation enabled DBMS. Finally, it shows a rough estimate of the performance of the storage and indexing techniques proposed in this paper. A real performance study is premature in this work. Thus, the objective is more to anticipate potential bottlenecks.

6.1 Discussion about the model

The data degradation model proposed in Section 3 is based on Life-Cycle Policies where (1) state transitions are fired at predetermined time intervals and (2) all tuples populating the database are uniformly ruled by the same LCP. This model inherits this from the limited data retention principle, today well accepted. Time degradation reflects well the fact that the value of an information decreases over time (in terms of usability, not in terms of privacy). Uniform LCP have the benefit of simplicity and reflects the fact that LCP should be preferably defined by civil rights organizations or agencies rather than by individuals for a better protection. However, other forms of data degradation make sense and could be the target of future work.

Event-based degradation. State transitions could be caused by events like those traditionally captured by database triggers. For example, an online book shop could degrade the delivery address of a customer order (e.g., down to city) straight after the order status is turned to “delivered” in the database.

Value-based LCP. In the same spirit, state transitions could be conditioned by predicates applied to the data to be degraded. For example, web searches containing illness related keywords could be considered as more sensitive than others and thus being degrading more quickly.

User-defined LCP. Users do not have the same perception of their privacy and do not attach either the same value to the services which can be offered to them in return for their data. Hence, letting paranoid users defining their own LCP makes sense. However, the observation of user's practice shows that few people actually (try to) understand and use configurable privacy protection tools [17] with a final negative impact on protection.

Whatever the form of the degradation, the foundation of the model presented in Section 3 remains valid (though slight adaptations are required). A query still works on a *database view* containing the projection of all tuples belonging to the domain of purpose of interest, over the degraded attributes of interest with the requested accuracy. Similarly, the transaction semantics still guarantees that the effects of a transaction are made atomic and durable in the LCP sense (i.e., a LCP automaton continues its execution after a commit as long as the related data is alive, even in case of crash, and previous states can never be recovered after a transition has been fired). The form of the degradation simply impacts which tuples are actually degraded and when. This may introduce however new technical challenges because degradation steps cannot always be managed in a set-oriented way.

6.2 Discussion about implementation

An important question is whether data degradation could be developed on top of an existing DBMS without modification in the kernel or must be tightly integrated within the DBMS kernel.

The first option seems to us not realistic for two main reasons: First, physical deletion is not supported by existing DBMS. Miklau [25] proposes a set of solutions to tackle this issue, like (1) overwriting deleted data in the table and index area before linking them in the free list and (2) encrypting the log, two techniques to be integrated in the DBMS kernel. Second, this option would lead to bad performance since traditional database techniques: (1) do not try to optimize deletes and updates since they are considered as rare; (2) are designed to favour either OLTP like queries (minimal indexation of few selective attributes to maintain a low insertion cost) or OLAP like queries (maximal indexation on non selective attributes – insertion cost is not a concern); and (3) may lead to enforce a stronger, useless and thus too costly transaction semantics (the protocol proposed in Section 5.1 allows avoiding conflicts between degradation subtransactions and regular and main transactions). Finally, this option would incur unclean hooks to enforce Δ-Durability.

We thus suggest to integrate data degradation into existing DBMSs either by ad-hoc modifications of traditional DBMSs kernels (as [25] does for handling physical deletions), or by developing plugs-in for traditional DBMSs.

6.3 Performance Estimates

The objective of this section is not to provide a detailed performance analysis but rather to identify potential bottlenecks and get a rough idea of the performance impact of the candidate storage and indexing techniques suggested in this paper. This preliminary study will help making design choices and focus our effort in developing the most accurate techniques.

To this end, we use a simple simulation, allowing us to change easily parameters and to simulate steady state performance across those parameter settings. After reaching a stable state, we count I/O requests generated during an experimentation period sufficiently large to observe degradation of low accuracy data. Then, we compute the corresponding disk time consumption according to the disk parameters. The results are obtained considering a single database table composed of three degradable attributes called d1, d2, and d3 regulated by the LCP described in Figure 2. Table 1 gives the simulation parameters (Disk, LCP, and experiment dependant parameters).

Table 1. Parameters of the simulation.

Parameters		Value	
	Exp. duration (sec.)	*1800*	
Disk	*Page size (KB)*	*4*	
	Disk Latency (ms)	*10*	
	Transfer rate (MB/s)	*50*	
Data files	*Attribute size (B)*	*10*	
	Pointer size (B)	*4*	
	Insert buffer size (KB)	*16*	
LCP	*Transition time delay for d_1: Δ_1^0, Δ_1^1, Δ_1^2 (h)*	*0.5, 4, 24*	
	Transition time delay for d_2: Δ_2^0, Δ_2^1 (h)	*2, 8*	
	Transition time delay for d_3: Δ_3^0, Δ_3^1 (h)	*3, 12*	
	LCP precision ρ	*1%*	
Indexes	*BTree node size (KB)*	*4*	
	HSL buffers size (KB)	*4*	
Varying parameters		**Fig. 7**	**– Fig. 8**
	RAM size (KB)	*64*	*– 256 KB/index*
	Inserts per sec. (Ips)	*20*	*– varying*
	Queries per sec. (Qps)	*0*	*– varying*
	Storage model	*varying*	*– clustered*
	Feeding strategy	*varying*	*– eager*
	Indexes	*No*	*– varying*

Figure 5 shows the overhead of degradation considering the four storage models introduced in Section 5.2 with a constant insert rate. This overhead is computed as the ratio between the disk time consumption of each storage model with degradation and their counterpart[7] without degradation, i.e., sequential raw data files with accurate attribute values (note that indexes are not considered here). Figure 5 leads to two remarks. First, *Eager* degradation performs better than *Lazy*. *Lazy* avoids some redundancy but increases disk accesses: at degradation time, *Lazy* induces reading, erasing, writing back degraded data, while *Eager* requires only erasing. With the fragmented storage model, each degradation step involves a small quantity of data (typically less than a page), leading to the introduction of a delete buffer and thus similar costs for Eager and Lazy. Second, *Fragmented* is twice as efficient as *Clustered* since *Fragmented* generates fewer disk read/write operations at insert and degradation time. However, the impact of vertical partitioning on query performance (tuple reconstruction) might render this approach unattractive for some applications.

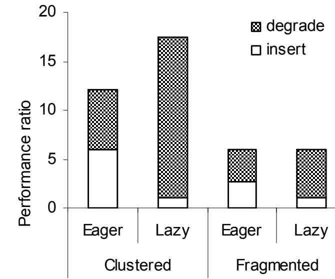

Figure 5. *Storage degradation overhead.*

To determine a good indexation scheme for each attribute's accuracy, we compare the B-tree with encrypted pointers denoted by *BTree* and Hash Sequential Lists denoted by *HSL*. We consider an index build on d_1^1. Queries[8] consist in equi-selections on d_1^1 evaluated using the index. Figure 6 plots the performance of both strategies for different query rate (Qps is 10, 20, 30, 40) with an increasing rate of inserts per second. Two main conclusions can be raised from this figure. First, *HSL* scales better at a high insert rate since it is based on sequential insert/degrade operations. *BTree* is more sensitive to insert rate increase (random I/Os). Second, *HSL* suffers a higher penalty while increasing the query load. This Figure shows clearly that each indexing technique has its own area of interest in term of insert/query workloads. Mainly, *HSL* behaves better at high insert/low query ratios, while *BTree* is better at the opposite load point (low insert/high query ratios).

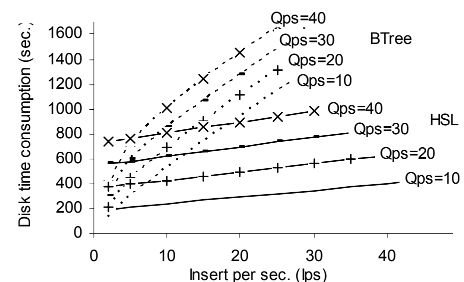

Figure 6. *Index access time consumption.*

Experiments on the simulator have also allowed delivering the following interesting remarks:

[7] The *Clustered* (resp. *Fragmented*) storage without degradation stores inserted tuples in a single sequential file (resp. in 3 sequential files) without applying any degradation operation. They obtain the same results because insertion into file(s) is buffered in both cases, and queries are not considered in this experiment (which would introduce a difference since data is not organized on disk in the same way). As a side effect, in Figure 5, we can compare fragmented versus clustered ratios.

[8] We only considered the index lookup costs first tuples are not always accessed, and second to avoid interfering with the data cache behaviour.

- *Redo Log cost is almost not impacted by degradation*. While degradation increases the redo log footprint (roughly by a factor of the number of attribute states), the number of I/O requests can be kept constant but not their length, allocating a larger redo buffer.
- *Increases of LCP precision have small impact on performance*. Increasing the LCP precision induces more frequent degradations, but on less data. Delete buffers are then used to minimize overheads for sequential structures (raw data files and HSL). We observe the convergence of *Lazy* and *Eager* strategies for raw data. HSL remains unchanged, a delete buffer per Hash bucket being required anyway. Regarding BTrees and redo logs, a larger number of keys must be managed, though remaining relatively small (e.g., ρ=0.1% leads to manage 1000 keys).
- *Each index has its area of interest.* While we did not implement the value list indexes [10], experiments with Btree, HSL and classical bitmap indexes show that the important parameters are (i) the size *s* of the time window covered by the index, (ii) the insertion rate *i*, (iii) the query rate *q*, and (iv) the number *d* of distinct values. Considering HSL and Btree, HSL improves when either *s* or *q* decrease and *i* increases while Btree improves when *s* or *q* increases, *i* decreases. Both are independent of *d*. When *d* is rather small, bitmap becomes interesting. This suggests for a given database workload and LCP: use HSL for small time window (probably for the most accurate attribute states i.e., the bottom levels of the GT), then Btrees for large time window (probably for average accuracy attribute state, i.e., medium levels of the GT) and finally, Bitmap for highly generalized attribute states, i.e., upper levels of the GT.

7. CONCLUSION

Data degradation is still an unexplored area and we believe it should deserve a stronger interest for the new opportunities it opens in terms of data protection. Applications to the safeguard of personal data are obvious, in particular within automated data monitoring environments, but corporate, administrative or military applications can be targeted as well. Data degradation provides guarantees orthogonal and complementary to those brought by traditional security services like access control, intrusion detection systems, etc. The benefits of a progressive and irreversible data degradation is twofold: (i) by reducing the amount of online accurate data, the privacy offence resulting from a trail disclosure is drastically reduced and (ii) degrading the data in line with the application purposes offers a new compromise between privacy preservation and application reach. More, by degrading the data repeatedly, attacks are forced to be repeated as well and become more easily detectable.

To the best of our knowledge, this paper is the first to propose a degradation model for databases. An intuitive semantics has been defined for this model. A first analysis of the impact of this model over the storage, indexation and transaction management has been conducted, and new techniques have been proposed. This must be considered as a first step towards the definition of more sophisticated and accurate solutions we plan to experiment.

8. REFERENCES

[1] Aggarwal, G., Bawa, M., Ganesan, P., Garcia-Molina, H., Kenthapadi, K., Mishra, N., Motwani, R., Srivastava, U., Thomas, D., Widom, J., Xu, Y. Vision Paper: Enabling Privacy for the Paranoids. In VLDB, 2004.

[2] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y. Hippocratic databases. In VLDB, 2002.

[3] Angryk R.A., Petry F.E. Mining Multi-Level Associations with Fuzzy Hierarchies. In Fuzzy Systems, 2005

[4] AOL search data scandal. http://en.wikipedia.org/wiki/AOL_search_data_scandal, 2006.

[5] Barbara D., Garcia-Molina H., Porter D. A probabilistic relational data model. In Int. Conf. on Extending Database Technology, 1990.

[6] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E. A

Critique of ANSI SQL Isolation Levels. In SIGMOD, 1995.
[7] Bouganim, L., Pucheral, P. Chip-Secured Data Access: Confidential Data on Untrusted Servers. In VLDB, 2002.
[8] Buckles B. P., Petry F. E. Fuzzy databases in the new era. In ACM Symposium on Applied Computing, 1995.
[9] Cavusoglu, H., Mishra, B., Raghunathan, S. The value of intrusion detection systems in information technology security architecture. Info. Sys. Research, 16, 1 (Mar. 2005).
[10] Chan, C. Y., Ioannidis, Y. E. An Efficient Bitmap Encoding Scheme for Selection Queries. In SIGMOD, 1999.
[11] Computer World. NASA sites hacked. Dec. 2003. http://www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,88348,00.html
[12] Cranor L., Langheinrich M., Marchiori M., Presler-Marshall M., and Reagle J. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C Recom., 2002.
[13] Directive 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, 1995. http://ec.europa.eu/ justice_home/fsj/privacy/law/index_en.htm
[14] EDRI. Telecommunication data retention. 2007. http://www.edri.org/issues/privacy/dataretention
[15] Grimm, R., Rossnagel, A. Can p3p help to protect privacy worldwide? In ACM workshops on Multimedia, 2000.
[16] Hacigumus H., Iyer B. R., Li C., Mehrotra S. Executing SQL over encrypted data in the database service-provider model. In SIGMOD, 2002.
[17] Hillyard, D., Gauen, M. Issues around the protection or revelation of personal information. Knowledge, Technology, and Policy, 20, 2 (July 2007).
[18] Langheinrich, M. A privacy awareness system for ubiquitous computing environments. In Ubiquitous Computing, 2002.
[19] Machanavajjhala, A., Gehrke, J., Kifer, D., Venkitasubramaniam, M. L-diversity: Privacy beyond k-anonymity. In ICDE, 2006.
[20] Meyerson, A., Williams, R. On the complexity of optimal k-anonymity. In PODS, 2004.
[21] Mills, E. Google adding search privacy protections. news.com, http://news.com.com/2100-1038_3-6167333.html
[22] MyLifeBits Project. http://research.microsoft.com/barc/mediapresence/MyLifeBits.aspx
[23] Organisation for Economic Co-operation and Development (OECD), OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data, 23rd Sept., 1980. http://www.it.ojp.gov/documents/OECD_FIPs.pdf
[24] Oracle Corp. Oracle Database, Advanced Security Administrator's Guide, 10g Release 2 (10.2). Oracle doc. B14268-02, 2005.
[25] Stahlberg, P., Miklau, G., Levine, B.N. Threats to privacy in the forensic analysis of database systems. In SIGMOD, 2007.
[26] Sweeney, L. K-anonymity: A model for protecting privacy. Int. Journal on Uncertainty Fuzziness and Knowledge-based Systems, 10, 5 (Oct. 2002).
[27] The Financial Times. Chinese military hacked into Pentagon. Sept. 2007. http://www.ft.com/cms/s/0/9dba9ba2-5a3b-11dc-9bcd-0000779fd2ac.html
[28] The Washington Post. Consultant Breached FBI's Computers. July 2007. http://www.washingtonpost.com/wp-dyn/content/article/2006/07/05/AR2006070501489_pf.html
[29] UK government loses personal data on 25 million citizens. http://www.edri.org/ edrigram/number5.22/personal-data-lost-uk