Expiration Times for Data Management

Albrecht Schmidt   Christian S. Jensen   Simonas Šaltenis

Department of Computer Science
Aalborg University, Denmark
{*al,csj,simas*}*@cs.aau.dk*

Abstract

This paper describes an approach to incorporating the notion of expiration time *into data management based on the relational model. Expiration times indicate when tuples cease to be current in a database. The paper presents a formal data model and a query algebra that handle expiration times transparently and declaratively. In particular, expiration times are exposed to users only on insertion and update, and when triggers fire due to the expiration of a tuple; for queries, they are handled behind the scenes and do not concern the user. Notably, tuples are removed automatically from (materialised) query results as they expire in the (base) relations. For application developers, the benefits of using expiration times are leaner application code, lower transaction volume, smaller databases, and higher consistency for replicated data with lower overhead. Expiration times turn out to be especially useful in open architectures and loosely-coupled systems, which abound on the World Wide Web as well as in mobile networks, be it as Web Services or as ad hoc and intermittent networks of mobile devices.*

1 Introduction

Emerging distributed information systems based on technologies such as Web Services and mobile networks [2] exhibit characteristics that are not easily reconciled with traditional distributed data management systems. One important reason is that these former systems are often loosely coupled, which means that connectivity might be intermittent or that the clocks of different sub-systems are not synchronised. As a consequence, traditional transactional qualities of Database Management Systems (DBMS) in the spirit of ACID [16] are hard and expensive to achieve. Furthermore, determining cost factors and bottlenecks in the envisioned volatile settings are network traffic and latency, especially when certain quality constraints are to be met. This paper extends the relational model [10] with the concept of *expiration times*; the objective is facilitating the implementation of loosely-coupled information systems.

The notion of expiration time is implicit in data whose (approximate) lifetime is known when they are inserted into a DBMS. Lifetimes are frequently available in data warehousing applications and for Web and monitoring data such as session keys, credentials, tickets, cached copies, and temperature or location samples, to name but a few [24]. In these cases, we know the lifetime (or at least an upper bound) that denotes until when a tuple is considered current and thus part of the database [4]. In more traditional settings, an administrator or user would issue an explicit delete statement when or after a tuple's lifetime elapses. Expiration times automate this procedure and thus reduce both user interaction and application code. In this manner, expiration times facilitate the management of base relations as well as materialised views. In particular, in keeping with the assumed volatile settings, we propose to materialise and maintain query results as far as possible independently of, but in synchrony with their base relations.

Once query results are computed from base data, we are interested in maintaining them as independently of the base data as possible. Maintenance is an issue since tuples disappear from the base relations when their expiration times pass; to keep query results, which may reside on a remote device, in synchrony with the relations they were computed from, the query results must be modified so that they reflect the state of the base relations. Ideally, this should be done by looking only at the expiration times of the tuples of the query results and without referring back to the base relations [5]. Independent maintenance of query results is of paramount importance in loosely-coupled systems since accessing the source data potentially incurs high costs or may not be possible at all due to lack of connectivity or bandwidth. Therefore, policies are needed on (1) how to maintain the query results [23], whose tuples should expire just like the source data they are computed from, and, later, on (2) how to propagate updates to the source relations into already computed queries as described in [5]. In this paper, we start out by focusing on the former and assume that there are no updates to the source data. We provide techniques that enable us to extend the time for which query

results may be maintained independently of the base data.

Since there exist efficient ways to support expiration times with real-time performance guarantees [24], users of expiration time-enabled databases can retain the usual benefits of data management. For example, triggers can be supported that fire on expirations, as can integrity constraint checking. This leads to a seamless integration of expiration into database applications. Support for expiration time is particularly convenient in applications that concern, e.g., automatic session management in HTTP servers, short-lived credentials and keys in cryptographic protocols, and location status for moving objects.

The contributions of this paper are the following: (1) We *extend the relational model and algebra* with support for expiration time. (2) We *evaluate* how expiration times can be used in data management and how they affect the *query processing* when results are materialised. This is achieved through a (3) *classification* of queries into monotonic and non-monotonic ones; the former require no special action, whereas the latter require occasional recomputation. (4) An *in-depth analysis* of recomputation issues is presented, as are strategies for minimising the resources needed for recomputation. We conclude the coverage of expiration times with a discussion of (5) how to achieve an *integration* with other aspects of data management.

The remainder of this paper is laid out as follows: After considering a motivating example, we present an expiration time-enabled relational algebra in Section 2 and discuss two operators requiring recomputation in detail. Section 3 analyses aspects of recomputation in detail. Before concluding, we cover related work in Section 4. An associated technical report [27] offers additional coverage.

2 Data Model and Algebra

To support expiration times, the relational framework must undergo slight adjustments. It is our goal to handle expiration times transparently and automatically, so that a user who queries (rather than updates) a database does not have to be aware of expiration times. The only occasions expiration times concern users are on insertion and update, where an expiration time may be assigned to a tuple.

2.1 Motivating Scenario

The application scenario, to be used in subsequent examples, concerns a dynamic, personalised news service. At the heart of the service is an engine that maintains user profiles. For simplicity, user profiles are expressed as a pair of user ID and degree of interest; the relation in which a tuple is stored denotes the topic of interest. Figure 1 shows a part of an example database of user profiles Table 'Pol' (for politics) lists users, given by their IDs, along with the degrees to which they are interested in politics. Because politics is

a core topic in a news service, the expiration times are relatively large in comparison to those in the second table, 'El' (for elections), which records shorter term interests in the more specific topic, elections.

$t_{\text{Pol}}^{exp}(\cdot)$	UID	Deg
10	1	25
15	2	25
10	3	35

(a) Politics table Pol

$t_{\text{El}}^{exp}(\cdot)$	UID	Deg
5	1	75
3	2	85
2	4	90

(b) Elections table El

Figure 1. Example relations at time 0.

The expiration times describes the times until when a user's degree of interest in a topic is to remain in effect. For example, tuple $\langle 1, 25 \rangle$ in table Politics expires at time 10. After this time, we would either generate a new profile for this particular user based on past behaviour or ask the user to explicitly renew the profile. Table El similarly records levels of interest in the topic of elections. Tuples in both tables feature the expiration times denoted in the column labelled t^{exp}; this column is typeset different from the relation attributes, to indicate that the values are not user-accessible.

2.2 Data Model

We use the following definitions and notation. A relation R of arity $\alpha(R)$ is a subset of the Cartesian product $D^{\alpha(R)}$ of the attribute domain D; a tuple r is an element of a relation R. Given a relation R, we assume that its attributes are numbered $\{1, \ldots, \alpha(R)\}$. The i-th attribute is denoted by $r(i)$. Furthermore, let $\max : time^n \to time$ and $\min : time^n \to time$ be the maximum and minimum functions of arbitrary arity on the totally ordered *time domain* that comprises *times* or *timestamps* including the symbol ∞ that denotes infinity and is larger than any other time value; for simplicity, we identify finite times with the non-negative integers. We use the terms 'query' and (algebra) 'expression' interchangeably.

To support expiration times properly, they must be integrated into the data model. We achieve this by leaving the main constituents of the relational data model unaltered. For each relation R, we add a function $t_R^{exp}(\cdot)$ that takes a tuple in relation R as argument and returns the expiration time associated with the tuple.

Next, we assume the existence of a function $t^{exp}(\cdot)$ that takes an algebraic expression, including a relation, as argument and returns an expiration time for the expression. The expiration time of an expression is a lower bound on the time when the materialised expression is no longer correct due to expiration of underlying tuples. The use of this function will become apparent when we introduce non-monotonic expressions.

Additionally, we require a function $\exp_\tau : R \to R$ that, for a time τ, takes a relation as argument and returns all

tuples of the relation that are unexpired at time τ. Thus, $\exp_\tau$ is defined as follows:

$$\exp_\tau(R) = \{r | r \in R \land t_R^{exp}(r) > \tau\}$$

In this paper, τ usually denotes a *current* time. We later elaborate on the choices that are possible for τ.

A further important desideratum is to retain the traditional semantics of the algebra when infinity is used as the expiration time. All query operators in this paper are defined in such a way that if all tuples are assigned expiration time ∞ then the algebra operators work like their textbook equivalents (*cf.* the SPCU algebra in [1], for example). Thus, expiration time ∞ is used for a tuple with no expiration time.

2.3 Algebra

When defining a relational algebra on the types of relations just defined, three issues must be dealt with. First, the different operators must in some manner address the expiration times associated with the argument tuples. Second, the operators must produce tuples with expiration time, to maintain closure. Third, also to maintain closure, the operators must specify an expiration time for their result relation.

With respect to the first issue, we gather there are two obvious approaches: to simply disregard the expiration times or to consider only tuples that have not yet expired at the time the operator under consideration is applied. The former approach offers maximum flexibility, but is counter to our objectives. So, we adopt an approach where we reuse the textbook relational algebra, but replace each argument relation R with $\exp_\tau(R)$, where τ is the time when the operator is applied. The second issue is addressed in the definition of each operator, and the third issue is covered at the end of this section.

We proceed to introduce an expiration time-aware algebra. We start out with an SPCU algebra [1] and consider the four basic operators *select*, *project*, *Cartesian product*, and *union*.

Selection. For the benefit of subsequent discussions, we distinguish between correlated and uncorrelated selections, *i.e.*, comparison between two attribute values of a tuple and comparison of a tuple's attribute value and a constant.

$$\sigma_p^{exp}(R) = \{t | t \in \exp_\tau(R) \land p(a)\}, \qquad (1)$$

where p is a predicate of the form $j = k$ or $j = a$, $a \in D$, $j, k \in \{1, \ldots, \alpha(R)\}$, or a $\land$- and $\lor$-connected composition of these. Result tuples simply retain their expiration times:

$$\forall t \in \sigma_p^{exp}(R) \; (t_*^{exp}(t) = t_R^{exp}(t)),$$

where the star '$*$' is a shorthand for the relation containing the result tuples, *i.e.*, $\sigma_p^{exp}(R)$ in this case.

Cartesian Product. The lifetime of a tuple produced by a cross-product is the minimum lifetime of the participating tuples.

$$R \times^{exp} S = \{t | r \in \exp_\tau(R) \land s \in \exp_\tau(S) \land t = \quad (2)$$
$$\langle r(1), \ldots, r(\alpha(R)), s(1), \ldots, s(\alpha(S)) \rangle\}$$
$$\forall t \in (R \times^{exp} S) \; (t_*^{exp}(t) =$$
$$\min\{t_R^{exp}(\langle t(1), \ldots, t(\alpha(R)) \rangle),$$
$$t_S^{exp}(\langle t(\alpha(R)+1), \ldots, t(\alpha(R)+\alpha(S)) \rangle)\})$$

Projection. Since projection includes elimination of duplicate tuples, a tuple is assigned the maximum expiration time of all its duplicates (including itself).

$$\pi_{j_1,\ldots,j_n}^{exp}(R) = \{t | t = \langle r(j_1), \ldots, r(j_n) \rangle \land r \in \exp_\tau(R)\}$$
$$(3)$$

Note that the expiration times of result tuples are calculated in a way that will recur in our discussion when we discuss grouping and aggregation, *e.g.*, see Equation (8):

$$\forall t \in \pi_{j_1,\ldots,j_n}^{exp}(R) \; (t_*^{exp}(t) =$$
$$\max\{t_R^{exp}(r) | r \in \exp_\tau(R) \land t = \langle r(j_1), \ldots, r(j_n) \rangle\})$$

Union. To compute the union of R and S we assign the maximum expiration of the participating tuples to the result tuples. We require R and S to be *union-compatible*, *i.e.*, $\alpha(R) = \alpha(S)$.

$$R \cup^{exp} S = \{r \mid r \in \exp_\tau(R) \lor r \in \exp_\tau(S)\} \quad (4)$$
$$\forall t \in (R \cup^{exp} S):$$
$$t_*^{exp}(t) = \begin{cases} \max\{t_R^{exp}(t), t_S^{exp}(t)\} & \text{if } t \in R \land t \in S \\ t_R^{exp}(t) & \text{if } t \in R \land t \notin S \\ t_S^{exp}(t) & \text{if } t \notin R \land t \in S \end{cases}$$

We now know which expiration times to assign to tuples produced by these four operators. The expiration times of the complete results (rather than the individual tuples) of the operations defined so far are given as follows: The expiration time of a selection, a Cartesian product, a projection, and a union is the minimum of the expiration time(s) of the argument(s). The expiration time of a base relation is defined to be infinity. It follows that the expiration times of all expressions that we can currently construct is infinity; thus $t^{exp}(\cdot)$ of any combination of operators and base relations defined so far always yields ∞. This will change as new operators are introduced in Section 2.6. For the time being, we formally define t^{exp} as follows:

$$t^{exp}(e) = \begin{cases} \infty & e \text{ is base relation} \\ t^{exp}(e') & \text{if } e = \sigma_p^{exp}(e') \text{ or} \\ & \quad e = \pi_{j_1,\ldots,j_n}^{exp}(e') \\ \min\{t^{exp}(e_1), t^{exp}(e_2)\} & \text{if } e = e_1 \times^{exp} e_2 \text{ or} \\ & \quad e = e_1 \cup^{exp} e_2 \end{cases}$$

2.4 Derived Algebraic Operators

We proceed to consider common derived operators.

Join. For joins, the expiration time semantics of the rewrite given below coincide with our intuition:

$$R \bowtie_p^{exp} S = \sigma_{p'}^{exp}(R \times^{exp} S), \qquad (5)$$

where p is a selection predicate on the attributes of R and S and p' the semantic equivalent of p on $R \times^{exp} S$.

Intersection. Intersection can be expressed in terms of Cartesian product, selection, and projection. We again require R and S to be union-compatible as in (4), and define intersection as follows:

$$R \cap^{exp} S = \pi_{1,\ldots,\alpha(R)}^{exp}($$
$$\sigma_{1=\alpha(R)+1 \wedge \cdots \wedge \alpha(R)=\alpha(R)+\alpha(S)}^{exp}(R \times^{exp} S)) \quad (6)$$

Since the outer projection leaves the expiration times unaltered due to all tuples being unique and selections just pass on the expiration times of their arguments, new expiration times are only created in the inner Cartesian product. Thus, tuples that belong to the intersection are assigned the minima of the expiration times of the participating tuples.

Other common derived operators can be defined using the same schema. Note that the left-hand sides of the operator definitions contain no reference to expiration times. This is consistent with our goal of hiding expiration times from a querying user. The expiration times of joins and intersections follow straightforwardly by simple composition from the definitions of the expiration times of the operators from which they are derived.

We remark that a number of practically very relevant operators, such as outer joins, introduce attribute values which do *not* originate from the input relations. We do not discuss such operators in detail since we would have to extend our data model with null values and three-valued logic; while this is possible, the necessary machinery adds complications, and the introduction of the new operators does not lead to significant, new insights. Thus, we only note that operators which introduce new attribute values must ensure that the newly introduced attribute values do not contribute to the expiration time of a result tuple. The reasoning behind this is akin to the reasoning that null values should not contribute to aggregate values.

2.5 Monotonic Examples and Properties

Figure 2 illustrates the effects of expiration time on query processing. Here we cover exclusively queries that involve only monotonic operators as defined below.

Consider again the relations in Figure 1. Then Figures 2(a) and 2(b) illustrate the expiration of tuples from time of origin 0 on. For example, the tuple $\text{Pol}\langle 1, 25 \rangle$ expires at time 10; consequently, at time 5, its lifetime is the
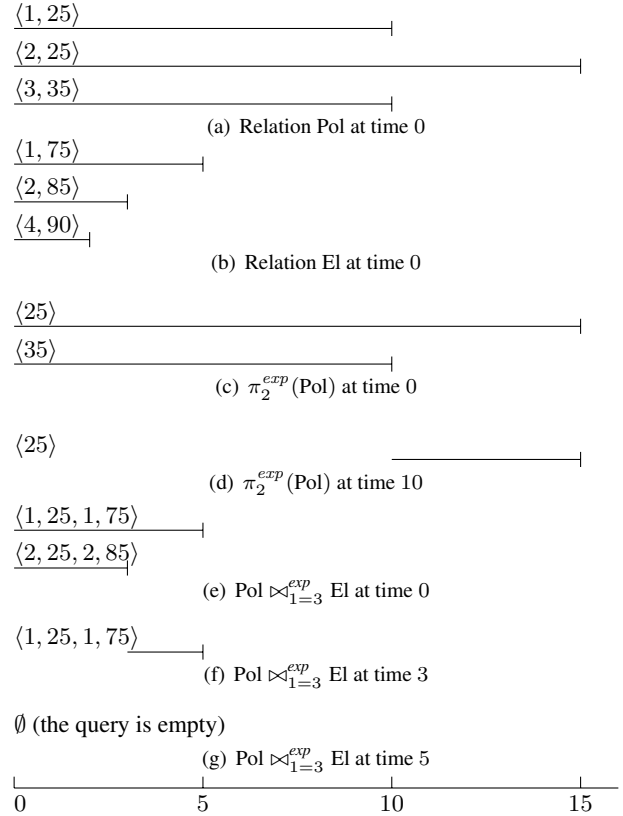

(a) Relation Pol at time 0

(b) Relation El at time 0

(c) $\pi_2^{exp}(\text{Pol})$ at time 0

(d) $\pi_2^{exp}(\text{Pol})$ at time 10

(e) Pol $\bowtie_{1=3}^{exp}$ El at time 0

(f) Pol $\bowtie_{1=3}^{exp}$ El at time 3

(g) Pol $\bowtie_{1=3}^{exp}$ El at time 5

Figure 2. Example monotonic expressions.

mere remaining five ticks until 10. Figure 2(c) presents a simple projection. Since the two tuples $\langle 1, 25 \rangle$ and $\langle 2, 25 \rangle$ coincide under π_2^{exp}, the result tuple inherits the maximum lifetime of the two, according to Formula (3). Note that the properly expired materialised query result 2(c) at any time $\tau > 0$ looks exactly as if the query had been computed at time τ, which is illustrated in Figure 2(d) for $\tau = 10$. Figures 2(e) through 2(g) illustrate a materialised expression containing a derived operator, namely a join expression. Again, the value of the derived expression at any time $\tau > 0$ coincides with the recomputation at time τ.

We remark that operators (1)–(4) are *monotonic* [1] in the sense that, if expiration times are ignored, $R_1' \times \cdots \times R_n' \subseteq R_1 \times \cdots \times R_n$ implies $e(R_1' \times \cdots \times R_n') \subseteq e(R_1 \times \cdots \times R_n)$ where e is a query expression consisting of (1)–(4) and $R_1, \ldots, R_n$. Algebraic expressions that only consist of monotonic operators inherit the monotonicity property from their constituents. In the following, the term *monotonic expression* is used for the expressions composed of the monotonic operators defined in this paper. The following property holds for such expressions:

Theorem 1. *Given a monotonic expression e and timestamps τ and τ', $\tau \le \tau'$, the following holds:*

$$\exp_{\tau'}(e) = \exp_{\tau'}(\exp_{\tau}(e))$$

Proof. We consider the selection operation. Assume that the materialised $\exp_\tau(e)$ and $\exp_{\tau'}(e)$ are correct. Let $t \in \exp_\tau(e)$. Then, for $\exp_{\tau'}(e)$ to be equal to $\exp_{\tau'}(\exp_\tau(e))$, the following must hold: $t \in \exp_{\tau'}(e) \Rightarrow \exp_{\tau'}(\exp_\tau(e))$ If $e = \sigma_p^{exp}(R)$ then $t_e^{exp}(t) = t_R^{exp}(t)$ according to (1). Thus, if $t \in \exp_{\tau'}(R)$ then also $t \in \exp_\tau(R)$. The correctness is proved for (2)–(4) analogously. This also implies that the theorem holds for (5) and (6). $\square$

Thus, if we materialise an expression e at time τ and then gradually expire tuples from this expression until time $\tau + m$, we obtain the same results as if e had been computed at time $\tau + m$, meaning that the materialised result of a monotonic expression remains valid (as long as no explicit updates occur on the underlying base relations). The use of expiration times on tuples and the operator definitions ensure this. For aggregation and difference, which are non-monotonic, this property does not hold.

2.6 Aggregation and Difference

We now add *aggregation* and *difference* operators to the algebra. These operators differ from the operators discussed so far in that they are *not* monotonic. The definition of aggregation is based on the framework of Klug [19].

2.6.1 Aggregation

The simple way to assign an expiration time to a tuple produced by an aggregation is to set it to the minimum expiration time of the tuples in the partition that the result tuple belongs to. To make this statement precise, we define the auxiliary function ϕ^{exp}:

$$\phi_{j_1,\ldots,j_n}^{exp}(R,r) = \{r'|r' \in \exp_\tau(R) \wedge$$
$$\langle r'(j_1),\ldots,r'(j_n)\rangle = \langle r(j_1),\ldots,r(j_n)\rangle\} \quad (7)$$

This function returns all tuples in R equal to r under the projection of attributes $j_1,\ldots,j_n$, *i.e.*, ϕ^{exp} returns the *partition* of which r is an element.

In this paper, we allow only this kind of partitioning of relations by tuple-wise equality of one or more attribute values. This corresponds to SQL's GROUP BY clause. This type of 'stable' partitioning ensures that when tuples in a partition expire, the overall partitioning scheme is retained, *i.e.*, pairs of the remaining tuples belong to the 'same' partition as they belonged to before the expiration. Thus, stable partitioning functions are also monotonic. This results in the following definition:

Definition 1. *A partitioning function ϕ is called* stable *if it is total (i.e., it is defined for every input) and monotonic.*

An example of an 'unstable' partitioning function is one that partitions tuples in ordered bands so that the first band consists of, say, the top 10% of the tuples according to some attribute, the second band consists of the next 10% of the

tuples, *etc*. Although such a function would be total, it is not monotonic since tuples may fall into a different band after a recomputation.

Using ϕ^{exp}, we define aggregation as follows:

$$agg_{j_1,\ldots,j_n,f}^{exp}(R) = \{t|t = \langle r(1),\ldots,r(\alpha(R)),a\rangle \wedge$$
$$r \in \exp_\tau(R) \wedge a = f(\phi_{j_1,\ldots,j_n}^{exp}(R,r))\} \quad (8)$$

$$\forall t \in agg_{j_1,\ldots,j_n,f}^{exp}(R)(t_*^{exp}(t) =$$
$$\min\{t_R^{exp}(r)|r \in \phi_{j_1,\ldots,j_n}^{exp}(R,\langle t(1),\ldots,t(\alpha(R))\rangle)\})$$

Here, $f \in F$, which is a family of *aggregate functions* such as $\min_1$, $\max_2$, sum_1, count_3, or avg_2, where the subscripts identify the attributes in the argument relation to which the functions are applied.

Note that by assigning the minimum expiration time of *all* tuples in a given partition to the result tuples for that partition, (8) yields a quite conservative bound on the possible lifetime of the tuple. For example, when calculating a $\min$ aggregate, a tuple that is not minimal may have the minimum expiration time; according to (8), the result tuples from the partition inherit the expiration time of a tuple that does not contribute to the aggregate value. In the sequel, we prolong the expiration time of the results by taking into account special properties of the standard SQL aggregate functions. The idea is that, to obtain less conservative expiration times of tuples, we ignore the lifetimes of all *time-sliced, neutral* sets of tuples. We proceed to define these concepts.

First, a *time-sliced* set of tuples is a set of tuples with identical expiration times. Next, a *neutral* set of tuples with respect to a given aggregate function is one that, if removed from the partition under consideration, neither changes the aggregate value nor its expiration time. For example, a set of tuples with values that add up to zero is neutral with respect to a sum aggregate. Given a partition P, Table 1 defines neutral subsets of P for the standard SQL aggregate functions

f	$N \subseteq P$ is neutral with respect to f if:
$\min_i$	$\forall t \in N\ (t(i) > f(P) \vee$ $t_*^{exp}(t) < \max\{t_*^{exp}(r)|r \in P \wedge r(i) = f(P)\})$
$\max_i$	$\forall t \in N\ (t(i) < f(P) \vee$ $t_*^{exp}(t) < \max\{t_*^{exp}(r)|r \in P \wedge r(i) = f(P)\})$
avg_i	$\sum_{t \in N} t(i) = (|N|/|P|) \sum_{r \in P} r(i)$
sum_i	$\sum_{t \in N} t(i) = 0$
count_i	$N = \emptyset$

Table 1. Neutral Subsets.

The definitions of neutral sets for the avg_i, sum_i, and count_i aggregate functions are intuitive. There may be two types of tuples in a neutral set for the $\min_i$ aggregate function. First, there are tuples that have the aggregated attribute

value larger than the minimum value in the partition. Expiration of such tuples obviously does not change the aggregate value. Next, there are tuples that have the aggregated attribute value equal to the minimum value in the partition. All such tuples, except those with the largest expiration time, can expire without changing the aggregate value. A neutral set for the $\max_i$ aggregate function has an analogous structure.

The concepts of neutral and time-sliced sets are useful for describing the validity of aggregate attribute values. The idea is that an aggregate attribute value does not change (meaning that the tuple to which it belongs does not need to expire) as long as every time-sliced set that has expired so far is neutral. As long as this condition holds, we can rely on aggregate attribute values to remain correct. In this way, we reduce the size of the set of tuples in the partition that determine the expiration time for the tuples of this partition in the result of the aggregate. We call this set the *contributing* set.

Definition 2. *Let P be a partition and f be an aggregate function. Let $\mathcal{N}_{f,P}$ be a set of time-sliced, neutral subsets of P. Then the contributing set of tuples in P is given as:*

$$\mathcal{C}_{f,P} = P - \bigcup_{N \in \mathcal{N}_{f,P}} N$$

Then, given a partition P and an aggregate function f, the expiration time of the aggregation result tuple t corresponding to tuple $r \in P$ ($r = \langle t(1), \ldots, t(\alpha(R)) \rangle$) is defined as follows:

$$t_*^{exp}(t) = \begin{cases} \min\{t_R^{exp}(l) | l \in \mathcal{C}_{f,P}\} & \text{if } \mathcal{C}_{f,P} \neq \emptyset \\ \max\{t_R^{exp}(l) | l \in P\} & \text{if } \mathcal{C}_{f,P} = \emptyset \end{cases}$$

This formula handles the special case where $\mathcal{C}_{f,P} = \emptyset$. This case occurs when an aggregate attribute value in the partition P remains valid until all tuples in the partition expire. This may happen, for example, if all attribute values to be aggregated are zero and the aggregate function is sum. The new definition of the expiration time improves on the expiration times of all aggregates except count which strictly follows (8).

Having offered operational mechanisms that extend the expiration times for aggregation result tuples for the five standard SQL aggregate functions, we proceed to consider in more abstract terms how to extend the expiration times of result tuples without making specific assumptions about the aggregate functions used.

To do so, we look at how the contributing sets of tuples just introduced develop over time. As a helper, we define the following predicate, where P is a partition and f is an aggregate function:

$$\chi(\tau, P, f) \equiv f(\exp_\tau(P)) \neq f(\exp_{\tau+1}(P))$$

This predicate is true if applications of f to P at times τ and $\tau + 1$ yield different results. This is helpful because we will need to expire a tuple that contains such an aggregate value when the value changes.

Before we can express the expiration times of tuples in an aggregation result, we need a function ν that tells when the aggregate value computed by aggregate function f on partition P first changes.

$$\nu(\tau, P, f) = \min\{\tau' | \tau' \geq \tau \wedge \chi(\tau', P, f)\}$$

We are now able to define the expiration times of tuples in aggregation results for any aggregation function f:

$$\forall t \in agg_{j_1,\ldots,j_n,f}^{exp}(R)($$
$$t_*^{exp}(t) = \nu(\tau, \phi_{j_1,\ldots,j_n}^{exp}(R, \langle t(1), \ldots, t(\alpha(R)) \rangle), f)) \quad (9)$$

Thus, all tuples in a partition are assigned the same expiration time, and they expire when the aggregate value changes. We note that, in practise, the functions χ and ν are best calculated when the actual aggregate values to a given aggregate function are computed. This is more promising than a naive translation of the above formulae for χ and ν into code, which would be overly complex. Since we took a similar avenue with the description and optimisation of the standard SQL aggregates earlier in this section, we omit the discussion of the more general optimisations for the sake of brevity.

Moving on to defining the expiration time of a materialised aggregate expression, we note that predicate χ may return true in two different cases. In the first case, the aggregate value, which is contained in some result tuple, should be replaced by another aggregate value that is, however, unknown to us. So we have a tuple that should be replaced by an unknown tuple. This forces us to expire the entire materialised expression. In the second case, the entire partition from which the tuple containing the aggregate attribute value originates has expired. So the aggregate value in the tuple should not be replaced by another value. Rather, the entire tuples should simply be disregarded, or expired, which is taken care of by the function above. In this case, the materialised aggregate expression remains correct and needs not expire.

Following this analysis, we can see that the following formula tells us when all tuples in a partition have expired (P is again a partition):

$$\min\{\tau' | \tau' \geq \tau \wedge \exp_{\tau'}(P) = \emptyset\}$$
$$= \max\{\tau' | \tau' = t_P^{exp}(t) \wedge t \in P\}$$

We now are also able to assign an expiration time to a ma-

terialised aggregation at time τ:

$$t^{exp}(agg_{j_1,\ldots,j_n,f}^{exp}(R)) = \min\{t^{exp}(R),$$
$$\min\{\tau'\,|\,\tau' = \nu(\tau,P,f)\wedge$$
$$P = \phi_{j_1,\ldots,j_n}^{exp}(R,r)\wedge r \in R\wedge$$
$$\max\{\tau''|\tau'' = t_P^{exp}(t)\wedge t \in P\} > \tau'\}\}$$

Thus, the result of an aggregations becomes invalid whatever happens earlier: (1) The argument relation R, on which the aggregation is computed, expires, or (2) the aggregate attribute value of tuples forming a partition changes before the corresponding tuples in the input relation have all expired.

2.6.2 Difference

The difference operator is yet another primitive operator, *i.e.*, it cannot be expressed by composing other operators. When we compute $R -^{exp} S$, a tuple $r \in R$ retains its expiration if it is not in S. Tuples that are only in S are disregarded:

$$R -^{exp} S = \{r|r \in \exp_\tau(R) \wedge r \notin \exp_\tau(S)\} \quad (10)$$

The respective expiration times are calculated as follows:

$$\forall t \in (R -^{exp} S)\,(t_*^{exp}(t) = t_R^{exp}(t))$$

We now turn our attention to expressions involving the difference operator. To see when exactly we need to take action and recompute, consider the case analysis in Table 2, where t is a tuple.

condition	$t_*^{exp}(t)$	$t^{exp}(e)$
(1) $t \in R \wedge t \notin S$	$t_R^{exp}(t)$	∞
(2) $t \notin R \wedge t \in S$	n.a.	∞
(3) $t \in R \wedge t \in S$ and		
(3a) $t_R^{exp}(t) > t_S^{exp}(t)$	n.a.	$t_S^{exp}(t)$
(3b) $t_R^{exp}(t) \le t_S^{exp}(t)$	n.a.	∞

Table 2. Lifetime analysis of $e = R -^{exp} S$.

Case (3a) is when a whole query result becomes invalid. Informally, if $t_R^{exp}(t) > t_S^{exp}(t)$ then $t \in R$ should show up in $R -^{exp} S$ after $t \in S$ expires. Thus, the result becomes invalid at $t_S^{exp}(t)$ since it contains (at least) one tuple too few.

The materialised expression $R -^{exp} S$ expires at time τ_R, which is given as follows:

$$\tau_R = \min\{t_S^{exp}(t)|t \in R \wedge t \in S \wedge t_R^{exp}(t) > t_S^{exp}(t)\}$$
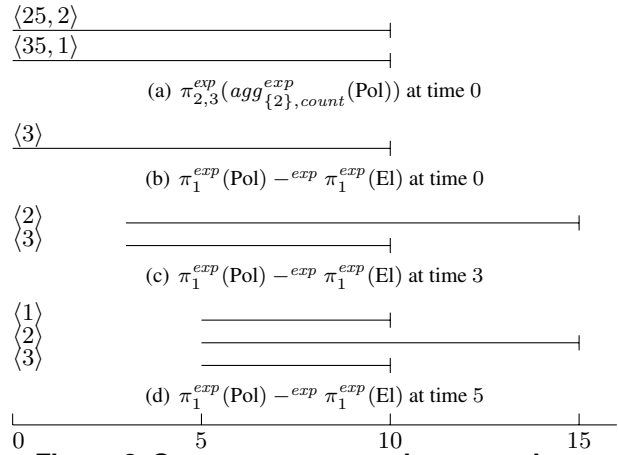
Thus, τ_R is the minimum time when case (3a) happens, *i.e.*, when a tuple from $r \in R$ should re-appear in the result since the matching $r \in S$ has expired. We can now define t^{exp} for difference expressions as follows.

$$t^{exp}(R -^{exp} S) = \min\{t^{exp}(R), t^{exp}(S),$$
$$\min\{t_R^{exp}(t)|t \in R \wedge t \in S \wedge t_R^{exp}(t) > t_S^{exp}(t)\}\} \quad (11)$$

Thus, like for aggregation, a materialised difference expression becomes invalid, when one of its arguments becomes invalid, or when a tuple should reappear in the result.

2.7 Further Examples and Properties

We now look at some examples that illustrate what happens when non-monotonic operators are included in queries. Let Pol and El be as in Figure 2(a) and 2(b). Then expression $\pi_{2,3}^{exp}(agg_{\{2\},count}^{exp}(\text{Pol}))$ in Figure 3(a) computes a 'histogram' of the values and, from time 10, should contain (only) the tuple $\langle 25, 1\rangle$, but according to (8), it does not. Instead, $\langle 25, 2\rangle$ expires. Thus, from time 10 on, the result is invalid.

(a) $\pi_{2,3}^{exp}(agg_{\{2\},count}^{exp}(\text{Pol}))$ at time 0

(b) $\pi_1^{exp}(\text{Pol}) -^{exp} \pi_1^{exp}(\text{El})$ at time 0

(c) $\pi_1^{exp}(\text{Pol}) -^{exp} \pi_1^{exp}(\text{El})$ at time 3

(d) $\pi_1^{exp}(\text{Pol}) -^{exp} \pi_1^{exp}(\text{El})$ at time 5

Figure 3. Some non-monotonic expressions.

The following example illustrates that expressions might, for certain time intervals, even monotonically grow in cardinality rather than shrink because of expirations.

The difference Pol $-^{exp}$ El at time 5 does not contain $\langle 1\rangle$ although the recomputation of $R - S$ contains the element. Even worse, Pol $-^{exp}$ El increases monotonically before time 10. Thus, the expression is invalid from time 3 onwards. See Figures 3(b) to 3(d).

This example illustrates that whether the expiration time of a non-monotonic expression is actually less than ∞ depends on the contents of the relation. For example, operations on empty relations or on relations all of whose tuples have the same expiration time always result in expressions with infinite expiration time and which thus will never become invalid. In other cases recomputation may be required.

The following theorem states concisely when recomputation of a materialised expression is needed.

Theorem 2. *Let an expression e that is materialised at time τ be given that has expiration time $t^{exp}(e)$ and that consists of operations (1)–(10). Also let a time τ' be given such that $\tau \le \tau' < t^{exp}(e)$. Then:*
$$\exp_{\tau'}(e) = \exp_{\tau'}(\exp_\tau(e))$$

Proof. The proof of Theorem 1 provides evidence for operators (1)–(6). To see whether (8) holds, we have to consider two things: First, a result never contains wrong values. For this to hold, tuples must expire before the aggregate attribute value contains an invalid value. Without any further information, a valid value is either the current value or, if the partition expires, no value at all. This is exactly what the function ν does. Second, a result becomes invalid before any tuples are missing. This implies that all aggregate tuples must either expire or remain valid before the result expires, which is made sure by $t^{exp}(e)$.

To prove the theorem for (10), we again have to check that the result never contains wrong values. This is ensured by (10). Second, we must ensure that the expression expires before it contains too few tuples, *i.e.*, in $R -^{exp} S$ there is no $t \in R$ with $t_R^{exp}(t) > t_S^{exp}(t)$. This is ensured by Case (3a) in Table 2.

In this proof we implicitly assumed that the arguments never expire. It can be verified that t^{exp} for (8) and (10) takes care of this case as well. $\square$

After $t^{exp}(e)$ passes, the query result has to be (incrementally) recomputed, as set forth earlier in this section.

3 Recomputation Aspects

Having presented various approaches to extending the expiration times associated with query results, we proceed to consider three kinds of techniques that also aim to reduce recomputation. We first consider different options for performing recomputations when materialised expressions expire. We then consider when to remove expired tuples. Finally, we consider a more radical approach where we associate time intervals with expressions rather than simply a single expiration time.

3.1 Recomputation Alternatives

When a materialised expression becomes invalid, we can try to incrementally update it given the instance and possibly some helper information we stored in anticipation of the expiry, or we can simply recompute the expression. Which option to choose is beyond the scope of this paper; the authors instead refer the reader to the extensive literature on updating views (*e.g.*, [5] and [29]) which provides general techniques.

Since the circumstances when an expression expires are well-defined, one can take different steps to keeping the expression valid without requiring a user to request an update explicitly. One option is to recompute the expression once it becomes invalid; other than for recomputation and transmission, no additional resources are required. Another option is to act on a per-operator basis. This is dicussed in more detail in Section 3.4.2.

There are also opportunities for postponing the time when a recomputation may have to take place. The idea is to use algebraic equivalences [23] to rewrite query plans; the objective is to reduce the following set of tuples, which causes recomputations to happen:

$$\{t | t \in R \wedge t \in S \wedge t_R^{exp}(t) > t_S^{exp}(t)\}$$

A second opportunity is to apply rewriting to pull up non-monotonic operators in query plans to reduce the effects of recomputations on operators that depend on them. In a DBMS, the cost estimation mechanisms can be made use of to estimate the impact of a rewrite-rule application.

3.2 Eager Versus Lazy Removal of Expired Tuples

With *eager removal*, expired tuples are removed from a materialised expression or base relation as soon as possible. This strategy is useful when events should be triggered as soon as a tuple expires. In contrast, *lazy removal* allows more freedom. Expired tuples are kept invisible to the user [26], but may be removed physically in a delayed fashion. Thus, lazy expiration provides more optimisation opportunities than eager expiration [24].

3.3 Queries and Observers

To reduce recomputation, be it incremental or not, of the materialised result of an expression that has expired, we may modify the data model to not only contain a single expiration time for the materialised result of an expression, but instead to contain a set of time intervals during which the result is valid.

To see why this is useful, consider the difference expression $R -^{exp} S$, where both R and S contain a single tuple t such that $t_R^{exp}(t) > t_S^{exp}(t)$. Tuple t is first absent from the materialised result of this expression. Then, when it expires in S, it should appear in the result. And, when it later expires in R, it should again be absent from the result. Using a single expiration time, the expiration time for the difference is $t_S^{exp}(t)$. With the new arrangement, we would associate two intervals with the materialised expression, namely one that extends from the current time until $t_S^{exp}(t)$ and one that extends from $t_R^{exp}(t)$ and onwards. Similar reasoning is possible for other operators. As an extreme case, we can even state that, if all tuples carry finite expiration times, a future time exists where every materialised result of any expression is valid, namely when all tuples have expired.

Queries that are issued against a materialised expression during the intervals specified for the materialised expression can be answered readily, without the need for recomputation of the expression. Other queries need special handling. Recomputation is one option. In other cases, it may be appropriate to either move the query backward in time (intuitively returning a slightly outdated result) or forward

in time (intuitively delaying the query), to a time where the materialised expression is correct.

Being somewhat philosophical, we may equate queries with observers and then observe the following: *An (materialised) expression is only required to contain correct values when a user queries it.* We refer to this point of view as *Schrödinger's cat semantics* [25].

3.4 Schrödinger Semantics

To define basic Schrödinger's cat semantics for the non-monotonic operators, we define two functions analogously to t_*^{exp} and t^{exp}: $\mathcal{I}_*$ tells us in which intervals a tuple is valid in the expression being computed, and $\mathcal{I}$ tells when the expression is valid. Thus, for a relation R,

$$\mathcal{I}_R : R \rightarrow 2^{intervals}$$

where *intervals* is the set of intervals $[\tau_1, \tau_2[, \tau_1 < \tau_2$. Thus, for a query $\sigma_p^{exp}(R)$ issued at time τ, $\mathcal{I}_R$ would return $[\tau, t_*^{exp}(t)[$ for every tuple $t \in R$. This is also the case for the other monotonic operators.

Similarly, we define the function $\mathcal{I}$ that tells us when a query expression is valid. Again, for an expression e consisting solely of monotonic operators, $\mathcal{I}(e)$ returns $[\tau, \infty[$, if the query is issued at time τ. In the sequel, we discuss what happens in the case of non-monotonic operators.

3.4.1 Aggregation

The validity intervals of an aggregation are constructed in a two-step process: First, the validity times of individual tuples are computed. Second, the intersection of the validity times of all tuples constitutes the validity time of the complete materialised expression.

To determine the validity time of a tuple, we extend the function χ defined earlier to also cover intervals by adding a second timestamp: An element of the results of an aggregation $agg_{j_1,\ldots,j_n,f}^{exp}(R)$ on a relation R is valid in an interval $[\tau', \tau''[, \tau' < \tau''$ iff

$$\chi(\tau', \tau'', P, f) = \neg\chi(\tau', P, f) \wedge \cdots \wedge \neg\chi(\tau'' - 1, P, f)$$

holds; that is, there is either no change in the aggregate attribute value such that $f(\exp_\tau(P)) = f(\exp_{\tau'}(P)) = \cdots = f(\exp_{\tau''}(P))$, or, alternatively, the partition expires (τ is still the query time). Thus,

$$\mathcal{I}_R(t) = \bigcup_{\chi(\tau', \tau'', P, f) \wedge f(\exp_\tau(P)) = f(\exp_{\tau'}(P))} [\tau', \tau''[.$$

For an expression, the Schrödinger validity is defined as the intersection of the validity intervals of the member tuples:

$$\mathcal{I}(agg_{j_1,\ldots,j_n,f}^{exp}(R)) = \bigcap_{t \in R} \mathcal{I}_R(t).$$

An interesting question in this context is how many possible values there are for an aggregation as it develops over time. In the sequel, we assume that the aggregate function is *deterministic*, *i.e.*, $f(P) = f(P)$ is always true. If f is deterministic, it can deliver, for a given partition P at most $|P|$ different values before the partition expires. Thus, for a partitioning function $\phi_{j_1,\ldots,j_n}^{exp}(R, \cdot)$, the number of different aggregate values is at most

$$\sum_{t \in \pi_{j_1,\ldots,j_n}^{exp}(R)} |\phi_{j_1,\ldots,j_n}^{exp}(R, t)| = |R|,$$

i.e., the number of tuples in the relation. For a more precise calculation, we can use the function χ, which indicates when an aggregate attribute value really changes. Thus, at time τ,

$$\sum_{t \in \pi_{j_1,\ldots,j_n}^{exp}(R)} |\{\tau' | \chi(\tau', \phi_{j_1,\ldots,j_n}^{exp}(R, t), f) \text{ is true} \wedge \tau \leq \tau'\}|$$

tells us how often an aggregate attribute value changes due to expiring partitions. This is also the amount of memory we need to store the future states of an aggregation.

3.4.2 Difference

A difference $R -^{exp} S$ is definitely valid after all $t \in R$ with $t \in S, t_R^{exp}(t) > t_S^{exp}(t)$ have expired, *i.e.*, after all tuples $t \in R$ which should later appear in the result have expired. The difference expression is also definitely valid until the first tuple $t \in R, t \in S$ with $t_R^{exp}(t) > t_S^{exp}(t)$ should appear at time, *i.e.*, until time $t_S^{exp}(t)$, and after all critical tuples have expired. Thus:

$$\begin{aligned}
\mathcal{I}(R -^{exp} S) = [\tau, \infty[- \\
[\min\{t_S^{exp}(t) | t \in R \wedge t \in S \wedge t_R^{exp}(t) > t_S^{exp}(t)\}, \quad (12) \\
\max\{t_S^{exp}(t) | t \in R \wedge t \in S \wedge t_R^{exp}(t) > t_S^{exp}(t)\}[
\end{aligned}$$

To compute the validity intervals for the difference operator, we take into account the case analysis of Table 2. Additionally, by keeping a priority queue of those $r \in R$ that are to be added at a certain point in time to $e = R -^{exp} S$, one can improve the independence between the materialised e and R, S mainly at the expense of additional storage cost and insertion operations. Finding out which $r \in R$ qualify involves a policy for deciding how many r to keep in the queue and an algorithm for extracting them from the arguments. The former point is a classic trade-off decision between saving future communication and time/space as well as up-front communication cost. The latter point is a query processing issue in the sense that we would like to able to integrate the creating of the priority queue into the difference operator to reduce the additional overhead. The difference operator can be implemented in a variety of ways, most notably as a left outer anti-semijoin [17], which may be executed as a hash join, a nested-loop join, or a sort-merge join.

Whichever method we use, we can always gather the information necessary to build the priority queue in $O(n \log n)$ time, n being the number of elements in the queue, with standard algorithms [20].

The idea of using a priority queue is compatible with the notion of expiration time since we can interpret this priority queue as a helper relation whose tuples expire; when they expire, they should simply be inserted into the materialised difference expression. Using this idea we, can extend the validity time of the difference expression by building on the machinery we defined earlier as laid out in the sequel.

In addition to the expression $R -^{exp} S$ defined in Equation (10), we define the following helper relation $\mathcal{R}$ whose expired tuples are to be added to the result of Equation (10):

$$\mathcal{R}(R -^{exp} S) = \{r | r \in \exp_\tau(R) \wedge r \in \exp_\tau(S)\}$$
$$(\forall t \in \mathcal{R}(R -^{exp} S)) : t_*^{exp}(t) = t_S^{exp}(t)$$

Theorem 3. *Given a helper relation $\mathcal{R}(R -^{exp} S)$, the non-monotonic expression $R -^{exp} S$ can be patched with the helper relation's expiring tuples so that recomputation is avoided, i.e., the expression's expiration time is ∞. The expiration time of a patching tuple t is $t_R^{exp}(t)$.*

Proof. We look at the events which involve a $t \in R$. First, if $t \notin S$ then $t_*^{exp}(t) = t_R^{exp}(t)$ according to (10); in this case (12) tells us that its expiration time does not affect the validity of the materialised expression. Second, if $t \in S$ then we distinguish between $t_R^{exp}(t) > t_S^{exp}(t)$ and $t_R^{exp}(t) \le t_S^{exp}(t)$ (*cf.* Case (3a) in Table 2). The latter case does not cause problems. In the former case, events happen at the following times: $t_S^{exp}(t)$ and $t_R^{exp}(t)$. At time $t_S^{exp}(t)$, t should appear in the result. Since this is also the time when t expires in $\mathcal{R}(R -^{exp} S)$, it indeed does appear; since it disappears at time $t_R^{exp}(t)$ in the argument relation R, the theorem also assigns the correct expiration time to the inserted $t \in R -^{exp} S$. $\square$

Note that the preceding theorem takes only differences into account. This is mainly because, for difference, the size of the priority queue depends only on the size of the input relations, *i.e.*, it contains at most $|R \cap S|$ elements. A similar theorem about aggregation would have to take f and the set of applicable attribute values into account, *i.e.*, it could require an infinite amount of storage needed for storing future values.

4 Related Work

At a data model and query language level, several lines of research involve notions that relate to, but also differ from, expiration time.

In particular, a kind of automatic data invalidation is implicit in sliding window-based processing of data streams [3], in that stream data with a time that exceeds the lower end of a window is no longer visible. In this context, it may even be of interest to attach semantics to expiration times by exposing the semantics of the timestamps to users, for example, to assign decreasing weights to data items according to their age [11]. The core conceptual difference between stream databases and our concept of expiration time is that, for the former, the user is required to specify a window of interest whereas, for expiration time, the data sources specify how long a certain tuple is to be considered current; thus, the validity of a data item is user-defined, whereas the expiration time is given by the data source and not the user.

As expiration time concerns the deletion of data from the current database state, expiration time relates to transaction time, which concerns the evolution of the database [22]. While support for expiration time may thus be useful for supporting transaction time, support for expiration time does not imply support for transaction time because past states are not retained. In a degenerate temporal database where transaction and valid time coincide, expiration time then also relates to valid time [18]. The notion of expiration somewhat relates to approaches to the vacuuming of time-referenced data [26]. In vacuuming, current-time dependent algebraic expressions are introduced that specify tuples to be deleted. In contrast, we attach explicit expiration times to tuples allowing expiration to be data-driven rather than user-driven. Other than this, we believe that expiration time has not previously been studied in the area of temporal data models and query languages.

Next, this paper proposes techniques that make it possible to extend the time during which a materialised query result may be maintained independently of the underlying base relations. This aspect of the paper can be complemented with existing techniques for incremental view maintenance [5,6,23]. Expiration time-enabled databases, on the other hand, exploit the knowledge about future events in the life of data to improve the maintainability of views.

Finally, data validity in the context of (mobile) networks has been investigated, and several notions of validity have been proposed [4]. Expiration times can be used to provide certain guarantees for all of these notions, disregarding approximate validity, which is not discussed in this paper. When monitoring Web data, the notions of time-to-live can be used to model and optimise latency and recency [7, 13].

5 Summary and Research Directions

The paper initially builds support for expiration time into the relational algebra, the main purpose being to facilitate data management in loosely-coupled systems, which are becoming increasingly widespread. The resulting framework enables transparent and declarative handling of expiration times, which are only exposed to the user at insertion and updates. The framework distinguishes between monotonic

and non-monotonic operators and algebra expressions. The results of the former always have unlimited lifetimes and never require recomputation, whereas the results of the latter may have limited lifetimes and thus may require recomputation. In particular, materialised non-monotonic expressions may become invalid after data expire in the base relations and, thus, may have to be recomputed or patched. The two non-monotonic operators, aggregation and difference, were discussed in detail. We concluded by motivating the formal setting of the work and by discussing the relationship to temporal and stream databases.

Several promising directions for future work exist. This paper assumes that the base relations are not updated, so that only expiring tuples are removed; it would be interesting to lift this restriction and integrate view update techniques. Next, the introduction of techniques that offer approximate query answers is reasonable in our setting and may yield performance improvements; if we are interested in maintaining, *e.g.*, aggregate values with certain error bounds, we might be able to improve performance. Finally, we plan to incorporate expiration into query processing with (approximate) quality of service guarantees as well as into the SQL framework.

Acknowledgments

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer, 2004.

[3] A. Arasu, S. Babu, and J. Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *Proc. International Workshop on Database Programming Languages*, pp. 1–19, 2003.

[4] M. Bawa, A. Gionis, H. García-Molina, and R. Motwani. The Price of Validity in Dynamic Networks. In *Proc. ACM SIGMOD*, pp. 515–526, 2004.

[5] J. Blakeley, N. Coburn, and P. Larson. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *ACM TODS*, 14(3):369–400, 1989.

[6] J. Blakeley, P. Larson, and F. Tompa. Efficiently Updating Materialized Views. In *Proc. ACM SIGMOD*, pp. 61–71, 1986.

[7] L. Bright and L. Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. In *Proc. VLDB*, pp. 550–561, 2002.

[8] A. Campbell and K. Nahrstedt. *Building QoS into Distributed Systems*. Chapman & Hall, 1997.

[9] J. Cho and H. Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *Proc. VLDB*, pp. 200–209, 2000.

[10] E. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.

[11] E. Cohen and M. Strauss. Maintaining time-decaying stream aggregates. In *Proc. ACM PODS*, pp. 223–233, 2003.

[12] G. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. ACM SIGMOD*, pp. 268–279, 1985.

[13] A. Gal and J. Eckstein. Managing Periodically Updated Data in Relational Databases: A Stochastic Modeling Approach. *JACM*, 48(6):1141–1183, 2001.

[14] L. Golab and M. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[17] T. Griffin and B. Kumar. Algebraic Change Propagation for Semijoin and Outerjoin Queries. *ACM SIGMOD Record*, 27(3):22–27, 1998.

[18] C. S. Jensen and R. T. Snodgrass. Temporal Specialization and Generalization. *IEEE TKDE*, 6(6):954–974, 1994.

[19] A. Klug. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions. *JACM*, 29(3), 1982.

[20] D. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 2nd edition, 1998.

[21] J. Melton. *Understanding the New SQL: A Complete Guide*. Morgan-Kaufmann, 2000.

[22] G. Özsoyoglu and R. Snodgrass. Temporal and Real-Time Databases: A Survey. *IEEE TKDE*, 7(4):513–532, 1995.

[23] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE TKDE*, 3(3):337–341, 1991.

[24] A. Schmidt and C. S. Jensen. Efficient Management of Short-Lived Data. TimeCenter TR, 2005. http://arxiv.org/abs/cs.DB/0505038.

[25] E. Schrödinger. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*, 23, 1935. For English translation see also: J. Wheeler and W. Zurek, Quantum Theory and Measurement, Princeton University Press, New Jersey, 1983.

[26] J. Skyt, C. S. Jensen, and L. Mark. A Foundation for Vacuuming Temporal Databases. *DKE*, 44(1):463–472, 2002.

[27] A. Schmidt, C. S. Jensen, and S. Šaltenis. Expiration Times for Data Management. Technical Report, Aalborg University, 2005.

[28] K. Torp, C. S. Jensen, and R. Snodgrass. Effective Timestamping in Databases. *The VLDB Journal*, 8(3-4):267–288, 2000.

[29] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proc. ACM SIGMOD*, pp. 316–327, 1995.