View Maintenance Issues for the Chronicle Data Model

(Extended Abstract)

H. V. Jagadish
AT&T Bell Laboratories
jag@research.att.com

Inderpal Singh Mumick
AT&T Bell Laboratories
mumick@research.att.com

Abraham Silberschatz
AT&T Bell Laboratories
avi@research.att.com

Abstract

To meet the stringent performance requirements of transaction recording systems, much of the recording and query processing functionality, which should preferably be in the database, is actually implemented in the procedural application code, with the attendant difficulties in development, modularization, maintenance, and evolution. To combat this deficiency, we propose a new data model, the *chronicle* model, which permits the capture, within the data model, of many computations common to transactional data recording systems. A central issue in our model is the incremental maintenance of materialized views in time independent of the size of the recorded stream.

Within the chronicle model we study the type of summary queries that can be answered by using persistent views. We measure the complexity of a chronicle model by the complexity of incrementally maintaining its persistent views, and develop languages that ensure a low maintenance complexity independent of the sequence sizes.

1 Introduction

Motivation: Many database systems are used to record a stream of transactional information, such as credit card transactions, telephone calls, stock trades, flights taken, sensor outputs in a control system, etc. Applications that deal primarily with transactional data have the following common characteristics:

- An incoming sequence of transaction records, each record having several attributes of the transaction to be recorded. The sequence of records can be very large, and grows in an unbounded fashion. The transaction records are stored in a database for some latest time window, as it is beyond the capacity of any database system to store and provide access to this sequence for an indefinite amount of time.

  For example, a major telecommunications company is known to collect 75GB of sequence data every day, or 27TB of sequence data every year. No current database system can even store so much data, far less make it accessible in an interactive manner.

- Queries over the stored sequence of transaction records, with stringent response time requirements. Of particular interest are **summary queries**, that access summarization, or aggregation information of past transactional activity.

  For example, a cellular phone company may want to provide a facility for a summary query that computes the total number of minutes of calls made in the current billing month from a phone number. This query could be executed whenever a cellular phone is turned on, and the result could be displayed on the customer's phone instrument. Another example of a summary query that a customer care agent in the cellular company may want to execute is: What is the total number of minutes of calls made from a given cellular number since the number was assigned to the current customer.

These applications can be (and are) implemented using commercially available relational databases. However, the relational model is not suitable to capture and exploit the peculiar characteristics of a transaction recording system. For example, there is no support for answering a summary query over a sequence that is not

PODS '95 San Jose CA USA

stored in the database in its entirety. Even if the sequence is stored, there is no support for answering a summary query over a large sequence, with the speed needed to process a banking transaction, or to display the answer on the customer's phone at power-on time.

These summary queries are therefore supported in today's systems by procedural application code. For example, an application program may define a few summary fields (*e.g.*, *minutes_called*, *dollar_balance*) for each customer, and update these fields whenever a new transaction is processed for this purpose. Summary queries are then answered by looking up the summary fields, rather than going to the sequence of transaction records. This gives the applications a fast response time, as well as independence from the need to lookup past transactional data. Some applications, such as ATM withdrawals, require that a summary field (dollar_balance) be updated as the transaction is executed, since the summary query needs to be made before the next ATM withdrawal. Some applications may choose to use triggers to invoke the updating code; others may update the summary fields as they process transaction in batch. In all cases, the logic to update the summary fields due to a transaction is encoded procedurally, and the burden of writing this code is with the application programmer. This updating code is known to be very tricky, and has been the cause of well-publicized banking disasters (*e.g.*, Chemical bank ATM withdrawals caused incorrect updates on February 18, 1994, leading to several bounced checks and frustrated customers [NYT94]).

The need to define summary fields and to write the update procedures within the application code is one of the reasons for the complexity of banking, billing, and other similar systems. Would it not be much better if these summary fields could be defined declaratively, and then updated automatically by the system as each transaction is processed? The *chronicle* data model, which we are advocating in this paper, has this goal in mind. In particular, we capture within the chronicle model the above needs of a transactional system, and thereby enable inexpensive, bug-free, and fast development of enhanced transactional systems. As the examples above illustrate, one feature that must be provided the chronicle model is support for summary queries that are specified declaratively (an SQL like language may be used), so that these queries can be answered without requiring the entire transactional history to be stored, and without accessing even the portion of the transactional history that is actually stored. The chronicle data model was inspired by our study of the complexities of a major transactional system within AT&T. However, our model is applicable to numerous other application domains, including credit cards, cellular telephone calls, stock trading, consumer banking, industrial control systems, retailing, frequent flyer programs, etc.

Results: We refer to the growing sequence of transaction records a *chronicle*. Our major contributions are:

- We present a new data model, called the chronicle data model.
- The model enables computation of summary queries over the chronicle, without requiring that the entire chronicle be stored in the database, and without requiring the application programmers to write procedural code.
- The model permits summary queries over arbitrarily large chronicles to be answered in subseconds, without requiring the application programmers to write procedural code.
- The chronicle model elevates persistent views to first class citizens in a database.
- We derive languages "Summarized Chronicle Algebra" and $\mathsf{SCA}_{\bowtie}$, such that views defined in these language can be maintained incrementally without accessing any of the chronicles. Further, $\mathsf{SCA}_{\bowtie}$ can be maintained incrementally in almost constant time, modulo index look ups. We have found $\mathsf{SCA}_{\bowtie}$ to be very useful in our application of the chronicle model within AT&T.
- We show that the language chronicle algebra, a component of the "Summarized Chronicle Algebra", and the language $\mathsf{CA}_{\bowtie}$, a component of $\mathsf{SCA}_{\bowtie}$, are the largest possible subsets of the relational algebra *operations* that derive chronicles and are in their respective incremental maintenance complexity classes.

- We list several research and systems issues that can add functionality, currently not available in databases, to a chronicle model and make the chronicle model even more appealing to the transactional systems.

The chronicle data model, in effect, is both an enhancement and a restriction of the relational data model, and can be built on top of the relational model. The manner of its realization is, however, orthogonal to the central theme that, in either case, we can reduce the complexity of a large class of applications.

Paper Outline The remainder of this extended abstract is organized as follows. In Section 2, we define the chronicle model. Section 3 defines complexity of a chronicle model as the complexity of incrementally maintaining its persistent views. We define a summarized chronicle algebra in Section 4, and derive several interesting results about this language, such as a low incremental complexity independent of the chronicle size, and maximal expressiveness while being limited to the relational algebra operations. We discuss some of the other research issues in the chronicle model in Section 5. Discussions concerning related work are offered in Section 6, which lead us to the conclusion in Section 7. Proofs of selected theorems are presented in the Appendix.

Preliminaries

We assume familiarity with relational algebra, and with grouping and aggregation operations of SQL. We use the following syntax [MPR90] to express the grouping operation:

$$\texttt{GROUPBY}(R, [G_1, \ldots, G_m], [A_1, \ldots, A_n]),$$

where R is the relation being grouped, $[G_1, \ldots, G_m]$ is the list of grouping attributes, and $[A_1, \ldots, A_n]$ is the list of aggregation functions. This expression defines a result relation with attributes in GL and with one attribute for each aggregation function. We will consider only those aggregation functions A_i that are incrementally computable, or are decomposable into incremental computation functions. For our complexity analysis, we will assume that each aggregation function can be computed in time $O(n)$ over a group of size n, and can be computed incrementally in time $O(1)$ over an increment of size 1. `MIN`, `MAX`, `SUM`, and `COUNT` are examples of such functions.

2 The Chronicle Data Model

We now define the chronicle model, discuss how queries may be posed on a chronicle database, and describe how updates to relations and chronicle must be handled in a chronicle database.

2.1 Model Definition

A chronicle database consists of *relations*, *chronicles*, and *persistent views*. *Relations* are standard, as in any relational database. Each relation may have several temporal versions (at least conceptually).

A *chronicle* is similar to a relation, except that a chronicle is a *sequence*, rather than an unordered set, of tuples. A chronicle can be represented by a relation with an extra *sequencing* attribute, whose values are drawn from an infinite ordered domain. The only update permissible to a chronicle is an insertion of tuples, with the sequence number of the inserted tuples being greater than any existing sequence number in the chronicle. There is no requirement that the sequence numbers be dense. Chronicles can be very large, and the entire chronicle may not be stored in the system.

There is a temporal instant (or chronon) associated with each sequence number. All operations on a tuple of a chronicle are with respect to the database as it was at that point in time. Thus, any join of a chronicle C and a relation R is a union of the corresponding joins of each tuple in C with the version of R that existed at the temporal instant of the tuple in C.

The database maintains a fixed number of *persistent views*, which are views that are materialized into relations, and are always maintained current in response to changes to the underlying database. Each persistent view is materialized when it is initially defined, and it is kept up-to-date, reflecting all the changes that occur in the database, as soon as these changes occur. Of particular concern is the maintenance of a persistent view after every append to a chronicle.

Persistent views are defined in a view definition language $\mathcal{L}$, and correspond to the procedurally computed

summary fields in the current transaction systems. Each choice of $\mathcal{L}$ derives a particular instance of the chronicle data model. Formally,

Definition 2.1: (Chronicle database system) A chronicle database system is a quadruple:

$$(\mathcal{C}, \mathcal{R}, \mathcal{L}, \mathcal{V}),$$

where $\mathcal{C} = \{C_0, C_1, \ldots, C_n\}$ is a set of chronicles, $\mathcal{R} = \{R_0, R_1, \ldots, R_m\}$ is a set of relations, and $\mathcal{V} = \{V_0, V_1, \ldots, V_l\}$ is a set of persistent views defined in a language $\mathcal{L}$. □

Example 2.1: Consider an airline database for tracking frequent flyer miles. There is one chronicle – the sequence of mileage transactions posted to the database. There is at least one relation containing information about customers, including their account number, name, and address. There are at least three persistent views to hold the mileage balance, the miles actually flown, and the premier status (bronze, silver, gold) of each customer. In order to define these persistent views, the language must allow for aggregation and joins between the chronicle and the relation. □

2.2 Queries

Queries that access the relations and persistent views can be written in any language — relational algebra, SQL, Datalog, etc. The choice of this language is orthogonal to the chronicle model. The chronicle model enables fast response to queries that access the persistent views; these queries may otherwise have been defined as complex SQL queries over the relations and chronicles and thus would not likely have been answerable with acceptable performance. Further, a system would typically provide detailed queries over some latest window on the chronicle; again the choice of the window and the query language are orthogonal to our discussion.

2.3 Updates

There are two types of updates in our model: those that modify the relations and those that append to chronicles. An update to a relation R can be an insert, delete, or modification of a tuple in R. An update to a chronicle C can only be the insertion of a new tuple (or tuples) to C with a sequence number greater than the sequence number of all existing tuples in C. We consider these updates in turn below.

Each relation conceptually has multiple temporal versions, one after every update. In any persistent view defined in language $\mathcal{L}$, any joins between the relations and chronicles have an implicit temporal join on the sequencing attribute: We can associate a temporal version of the relations with each sequence number in the chronicles. Each tuple of a chronicle is then joined with the version of the relations associated with the same sequence number.

If an update to a relation affects only the versions corresponding to sequence numbers not seen as yet, then it is a *proactive* update; such an update does not affect the persistent views. Only subsequent chronicle updates see the new relation values. Since maintainability of persistent views is critical in the chronicle model, we have chosen to limit the language $\mathcal{L}$ so that only proactive updates to relations are allowed.

In contrast, a *retroactive* update to a relation would require older tuples in the chronicle to be re-processed. Such updates, when necessary, are computationally expensive to maintain, may not be maintainable if the entire past chronicle is unavailable, and are not included as part of the chronicle model.

Example 2.2: Consider again the frequent flyer example. Suppose that each customer living in New Jersey gets a bonus of 500 miles on each flight. The customer relation is updated whenever a customer changes his/her address. A flight tuple in the chronicle qualifies for the bonus only if the flight was made during the period of residence in New Jersey. Thus, the join between the chronicle and the relation is based on the temporal version of the relation associated with the sequence number in the chronicle. An update to the relation is proactive if the address update occurs before the associated tuples are appended to the chronicle. □

An update to a chronicle may cause a change in each persistent view, and we discuss maintenance of persistent views in the next two sections.

3 Complexity of a Chronicle Model

Each time a transaction completes, a record for the transaction is appended to the chronicle, and one or

more persistent views may have to be maintained. The transaction rate that can be supported by a chronicle system is determined by the complexity of incremental maintenance of its persistent views. Thus, it is important to choose a language $\mathcal{L}$ that ensures that incremental view maintenance can be done efficiently. Moreover, since chronicles may not be stored in the system, the language $\mathcal{L}$ should allow incremental maintenance without having access to the entire chronicles. Ideally, the complexity of maintaining a view defined in $\mathcal{L}$ should be low – independent of the size of the relations and the view itself, modulo the overhead of index lookups.

We define the complexity of a chronicle system as the complexity of incremental computation of the language $\mathcal{L}$ used to express the persistent views. A class IM-T means that all persistent views defined in the language can be maintained in time $O(T)$ in response to a single append into a chronicle (IM for incremental maintenance). The incremental complexity classes are similar to the dynamic complexity classes of Patnaik and Immerman [PI94].

The following incremental complexity classes may be defined:

IM-Constant: A language is in the class IM-Constant if any persistent view defined in the language can be maintained incrementally in response to a single append into the chronicle in constant time.

IM-log(R): A language is in the class IM-log(R) if any persistent view defined in the language can be maintained incrementally in response to a single append into the chronicle in time logarithmic in the size of the relations.

IM-R^k: A language is in the class IM-R^k if any persistent view defined in the language can be maintained incrementally in response to a single append into the chronicle in time polynomial in the size of the relations.

IM-C^k: A language is in the class IM-C^k if any persistent view defined in the language can be maintained incrementally in response to a single append into the chronicle in time polynomial in the size of the chronicle and the relations.

It is easy to show that the following relationships hold between the sets of views that can be described by languages in each class:

$$\text{IM-Constant} \subset \text{IM-log}(R) \subset \text{IM-}R^k \subset \text{IM-}C^k$$

In a high throughput system, a complexity of IM-Constant is desired, which implies that even index lookups are not permitted, and is thus difficult to achieve. At the other end of the spectrum, a chronicle model with complexity IM-C^k would permit arbitrary access to the chronicle. Such a complexity is totally impractical for an operation to be executed after each append into each chronicle. The size of the relations, $\mid R \mid$, is assumed to be much smaller than the size of the chronicle $\mid C \mid$, so complexity class IM-R^k is the largest that has the possibility of being manageable.

The choice of language $\mathcal{L}$ for defining persistent views with a low IM complexity is crucial. One obvious candidate is relational algebra with grouping and incrementally computable aggregate operators. However, relational algebra is not an acceptable choice for $\mathcal{L}$, as the following result indicates.

Proposition 3.1: Relational algebra, extended with grouping and aggregation, applied to chronicles and relations, is in the class IM-C^k, and is not in the class IM-R^k. □

4 Summarized Chronicle Algebra

In this section, we derive the largest subsets of a set of relation algebra operators that define languages in the first three IM complexity classes, which are all independent of the size of the chronicle. We present our development in two steps. First, we define an intermediate chronicle algebra that maps chronicles and relations into chronicles. Then, we add a summarization step that maps chronicle algebra expressions into relations by projecting out the sequencing attribute, possibly doing a grouping and aggregation alongside.

All inserted tuples into a chronicle must have a sequence number greater than all existing sequence numbers, but multiple tuples with the same sequence number can be inserted simultaneously. For instance, when a tuple is inserted into a base chronicle, each of the two operands of a union may derive a distinct tuple with

the same sequence number. The union expression can then have two distinct tuples with the same sequence number.

We define a *chronicle group* as a collection of chronicles whose sequence numbers are drawn from the same domain, along with the requirement that an insert into any chronicle in a chronicle group must have a sequence number greater than the sequence number of any tuple in the chronicle group. Operations like union, difference, and join are permitted amongst chronicles of the same chronicle group.

Definition 4.1: The chronicle algebra (CA) consists of the following operators (C is a chronicle or a chronicle algebra expression, $A_1, \ldots, A_n$ are attributes of the chronicle, and p is a predicate):

- A selection on a chronicle, $\sigma_p(C)$, where p is a predicate of the form $A_1 \theta A_2$, or $A_1 \theta k$, or a disjunction of such terms, k is a constant, and θ is one of $\{=, \neq, \leq, <, >, \geq\}$. $\sigma_p(C)$ selects chronicle tuples that satisfy the predicate p. The resulting chronicle has the same type as the chronicle C.
- Projection of a chronicle on attributes that include the sequencing attribute, $\Pi_{A_1, \ldots, A_n}(C)$.
- A natural equijoin between two chronicles on the sequencing attribute, $C_1 \bowtie_{C_1.SN=C_2.SN} C_2$, where SN is the sequencing attribute, C_1 and C_2 are chronicles in the same chronicle group, and one of the sequencing attributes is projected out from the result.
- Union of two chronicles, $C_1 \cup C_2$, where C_1 and C_2 are chronicles in the same chronicle group, and have the same type.
- Difference of two chronicles, $C_1 - C_2$, where C_1 and C_2 are chronicles in the same chronicle group, and have the same type.
- A groupby with aggregation, with the sequence number as one of the grouping attributes: GROUPBY(C, GL, AL), where C is a chronicle being grouped, GL is the list of grouping attributes (which must include the sequencing attribute), and AL is the list of aggregation functions.
- A cartesian product between a chronicle C, and any relation R, $C \times R$. Though this operation is written as a cross product, recall (from Sec. 2.3 and Example 2.2) that an implicit temporal join on the sequencing attribute exists between C and R. □

Theorem 4.1: A view defined by the chronicle algebra is monotonic with respect to insertions into the base chronicles. Whenever tuples with sequence numbers greater than all existing sequence numbers are added to the base chronicles, the effect is to add tuples with some of these new sequence numbers to the view. □

Lemma 4.1: Each view defined using a chronicle algebra expression is a chronicle in the same chronicle group as the operand chronicles. □

Definition 4.2:

- CA_1 is chronicle algebra, without the cross product operation between chronicles and relations.
- $CA_{\bowtie}$ is chronicle algebra, where the cross product operation between chronicles and relations is replaced by a join, with a guarantee (based on the schema and integrity constraints on the database) that at most a constant number of relation tuples join with each chronicle tuple. A sufficient condition for the guarantee is that the join be on a key of the relation R. □

Theorem 4.2: The changes, due to insertions into the base chronicles, for a chronicle view defined by:

- CA can be computed in time and space independent of the size of the chronicles and independent of the size of the view. Time $= O((u \mid R \mid)^j \log(\mid R \mid))$, and Space $= O((u \mid R \mid)^j)$, where u is the number of unions in the expression defining the view, $\mid R \mid$ is the size of the relation R, and j is the number of equijoins and cross products in the expression defining the view.
- $CA_{\bowtie}$ can be computed in Time $= O(u^j \log(\mid R \mid))$, and Space $= O(u^j)$.
- CA_1 can be computed in Time $= O(u^j)$, and Space $= O(u^j)$ □

In all cases, neither the chronicle view nor the chronicles need to be stored or accessed for the view maintenance; and this is the reason for obtaining a complexity which is independent of both the sizes of the chronicle and the sizes of the view.

Theorem 4.3: An extension of the chronicle algebra with either of (1) projection without including the sequencing attribute, or (2) a groupby operation without including the sequencing attribute as a grouping attribute leads to an algebra that can define an expression that is not a chronicle. Further, an extension of the chronicle algebra with either of (1) cross product between chronicles, or (2) a non-equijoin between two chronicles leads to an algebra that can define an expression for which the time for incremental view maintenance is dependent on the size of a chronicle. □

Note that this theorem implies that the chronicle algebra is the largest subset of relational algebra operations that is in IM-R^k, and that $\mathsf{CA}_{\bowtie}$ is the largest subset of relational algebra operations that is in IM-log(R). It is important to note that we can define expressions using the cross product or non-equijoin between chronicles that are in IM-R^k. Theorem 4.3 simply states that there also exist expressions one can define using the cross product or non-equijoins that are not in IM-R^k.

Next, we present a summarization step that maps chronicles produced by chronicle algebra into persistent views, which are relations without the sequencing attributes. The persistent view is then stored, and it is updated whenever an insert occurs into the chronicles on which the persistent view depends.

Definition 4.3: The *summarized chronicle algebra* (SCA), has the two basic operations that can eliminate the sequence attribute and map a chronicle algebra expression χ into a relation.

- Projection, with the sequencing attribute projected out; that is, $\Pi_{A_1,\ldots,A_n}(\chi)$, where the attributes $A_1, \ldots, A_n$ do not include the sequencing attribute.
- Grouping with aggregation, where the sequencing attribute is not included in the grouping list, and where the aggregation functions are incrementally computable (or decomposable into incrementally computable functions); represented as GROUPBY(χ, GL, AL) where the grouping list GL does not include the sequencing attribute of χ.

If the expression χ is in CA_1, then the resulting language is called SCA_1; if the expression χ is in $\mathsf{CA}_{\bowtie}$, then the resulting language is called $\mathsf{SCA}_{\bowtie}$. □

From Definition 4.3, it follows that every persistent view expressed in SCA produces a relation (not a chronicle) that does not have the sequence number as an attribute. Once a relation is defined using SCA, it could be further manipulated by using relational algebra and the other relations in the system, to define a persistent view. However, since incremental maintenance is the key, we have to be careful to store a persistent view that can be maintained without accessing the full chronicles over which the summarization step is defined.

Theorem 4.4: Given a set of changes to chronicle algebra expression, incremental maintenance of a persistent view written in SCA in response to insertions to a chronicle can be done in

- Space equal to the size of the view.
- Time = $O(t \log(\mid V \mid))$, where $\mid V \mid$ is the size of the persistent view V, and t is the number of tuples inserted into the chronicle algebra (or CA_1 or $\mathsf{CA}_{\bowtie}$) expression χ. □

Theorem 4.5: SCA is contained in class IM-R^k, $\mathsf{SCA}_{\bowtie}$ is contained in class IM-log(R), and SCA_1 is contained in class IM-Constant. □

Thus, though the result of a chronicle algebra expression contains sequence numbers, and therefore may have a size that is polynomial in $\mid C \mid$, incremental maintenance of summarized chronicle algebra expressions can be done in time independent of $\mid C \mid$, since the chronicle view is not accessed during maintenance.

5 Issues in the Chronicle Model

We briefly outline several additional research issues that need to be considered in designing a chronicle system.

5.1 Periodic Persistent Views

The applications targeted by the chronicle model require the definition of a view that is computed over several,

potentially overlapping, intervals on a chronicle, one view computation for each interval. To address this need, we introduce a *periodic summarized chronicle algebra* by adding, to the chronicle algebra, features in the spirit of [SS92, CSS94] to construct sets of time intervals over which the persistent views can be computed. A mapping from sequence numbers in a chronicle to time intervals must be made for the periodic summarized chronicle algebra to be defined.

Given a view V in summary algebra, and a calendar D (*i.e.*, a set of time intervals), $V < D >$ specifies a set of views $V_1, \ldots, V_k$, one for each interval in the calendar D. The view V_i for interval i is defined as in V, but with a selection on the chronicle, which requires that all chronicle tuples be within the interval i, under the mapping defined from sequence numbers to time intervals. If the calendar D has an infinite number of intervals, there will be an infinite number of views V_i. The view expression $V < D >$ is called a *periodic view*. When the calendar D has only one interval, the periodic view corresponds to a single view defined using an extra selection on the chronicle.

The periodic summarized chronicle algebra also provides for an expiration time for a view, after which the view is not needed. Expiration dates allow the system to implement an infinite number of periodic views, provided only a finite number of them are current at any one instant.

Many applications require periodic views over non-overlapping intervals. (For instance, a new billing statement is generated each month, by banks, telephone companies, credit card companies, etc.) The evaluation of these can be optimized by starting to maintain a view as soon as its time interval starts, and stopping its maintenance as soon as its interval ends. Periodic views over overlapping intervals can be defined to compute moving averages over the transactions in a chronicle. Optimization of such views is even more challenging. For example, consider a periodic view for every day that computes the total number of shares of a stock sold during the 30 days preceding that day. The computation of these views can be optimized by noticing that the sum of shares is an incrementally computable function. This suggests that we should keep the total number of shares sold for each of the last 30 days separately, and derive the view as the sum of these 30 numbers. Moving from one periodic view to the next one involves shifting a cyclic buffer of these 30 numbers. Further, if an expiration date is given, the space for the periodic view can be reused. How would such a computation be derived automatically by the system for a generic periodic view expressed over any given set of overlapping time intervals?

5.2 Identifying affected persistent views

When multiple views are to be maintained over the same chronicle, each update to the chronicle would require checking all the views to determine if they need to be updated. To do so, we must:

- Identify the persistent views V that will be affected. We need to filter these out early so as not to waste computation resources. This problem is similar to determining when a query is independent of an update [LS93]. The problem is similar to detecting the active rules that must be checked after a database update.
- For each persistent view V, identify the tuples that will be affected. Thus, the persistent views need to have indices. What indices should be constructed?
- When periodic views are used, we must be able to easily identify the persistent views that are *active* – these are the views defined for the *current* time interval, and only these periodic views need to be maintained upon insertions into the chronicle.

Efficient storage structures are needed for fast access to the updated persistent view tuples. We have developed an efficient storage structure in our implementation of the chronicle model at AT&T Bell Labs.

5.3 Batch to Incremental Updates

Applications often define computations applying to a batch of transactions. For example, a bank may charge a fee based upon the total number of transactions within a period, a telephone company may offer a discount based upon the total calls within a period, an airline may give bonus miles based upon total activity within a certain period. For instance, a popular telephone discounting plan in the USA gives a discount of 10% on all calls

made if the monthly undiscounted expenses exceed \$10, a discount of 20% if the expenses exceed \$25, and so on. In such applications, a common assumption is that the computations are performed once at the end of the period. This leads to two problems:

- The results of these computations are either out-of-date, or inaccurate before the end of the period over which the discount applies.
- The transactions need to be processed, for computing these attributes, in batch.

Converting computations on a batch of records to an equivalent incremental computation on individual records is an exercise akin to devising algorithms for incremental view maintenance. For example, for the telephone discount plan, there is a nontrivial mapping for incrementally computing a persistent view for total_expenses.

6 Related Work

The notion of incremental differences has been studied extensively, beginning with [SL76], continuing with work on delayed updates, and most recently with work on languages (*e.g.*, [GHJ94]). Incremental differences have been used effectively in the active database context [BW94], and in the constraint maintenance context [Qia88]. Incrementally computable aggregation functions are used in [DAJ91, GMS93, RRSS94]. Our work here deals with incremental view maintenance. However, we differ from the past view maintenance work [BLT86, CW91, GMS93, JMR91, JM93] in that (1) we assume that the modified relation (the chronicle) is not accessible during maintenance, (2) we require that all intermediate views defined in chronicle algebra not be materialized, and (3) we make special use of the sequencing and temporal join properties of a chronicle.

The temporal data model [TCG+93] builds in a model of time into the data, and allows complex queries that relate data across time. Several versions of data are stored to support these queries. However, as in the relational model, queries or persistent views over data that is not completely stored in the database are not modeled. The concept of time in the chronicle model is much simpler – a sequence number for chronicle records, and an implicit temporal join with the relations when defining views. The implicit temporal join is always with the most *current* version of the relations, and versions of relations do not need to be stored. Seshadri *et al.* [SLR94] look at optimizing queries over sequences, assuming the full sequence is stored.

Patnaik and Immerman [PI94] define the dynamic complexity of a query as the complexity of maintaining a materialized view against which the query can be answered with the same complexity. They focused on a class of queries that can be incrementally maintained using a relational algebra expression (the Dyn-FO class), and show that several queries that cannot be expressed in relational algebra can be maintained using a relational algebra expression. The notion of dynamic complexity is similar to the idea of incremental complexity (Section 3) of a persistent view; however, the later does not include the complexity of querying the view, and is a computational measure, while the former is a descriptive measure. We focus on incremental complexity classes that are more efficient than Dyn-FO, and we exploit the special properties of a chronicle to make the incremental complexity independent of the chronicle size.

Incarnations of the chronicle model may be applicable to domains other than transactional systems. For example, in active databases, the recognition of complex events to be fired is done on a chronicle of events. The notion of history-less evaluation [Cho92a, Cho92b, GJS92b, GJS92a] is simply the idea of incremental maintenance of the persistent views defined by the event algebra. The language $\mathcal{L}$ in these cases is either a variant of temporal logic [Cho92a, Cho92b], or a variant of regular expressions [GJS92b, GJS92a].

7 Conclusions

Transaction recording systems are an important class of database applications. To meet the stringent performance requirements of these applications, much of the recording and query processing functionality, which should preferably be in the database, is actually implemented in the procedural application code, with the attendant difficulties in development, modularization, maintenance, and evolution.

To combat this deficiency, we proposed a new data

model, the *chronicle* model, which permits the capture, within the data model, of many computations common to transactional recording systems. The database maintains a set of chronicles, each of which is a log of the transactions records of a particular type. A chronicle, in contrast to a regular relation, need not be stored in its entirety in the system. To capture the essential information about the records in the chronicles we allow one to define a materialized (persistent) view over a chronicle. The persistent views must be maintained in response to each new transaction. One major concern is the efficient and automatic maintenance of persistent views in order to meet the performance requirements of transaction recording systems. With this in mind, we defined incremental complexity classes. A language belongs to an incremental class if all views defined using the language can be maintained incrementally in the time specified by the class. The complexity of the chronicle model was itself defined in terms of the incremental complexity of its language for defining persistent views.

We introduced a summarized chronicle algebra (SCA) that can be used to define persistent views. Aggregation is permitted, and is an important operation for the applications. We showed that all SCA views can be maintained incrementally without reference to the chronicle, in time polynomial in the size of the relations. We further derived a condition under which the language $\mathsf{SCA}_{\bowtie}$, by restricting joins to be on a key, can be incrementally maintained in time logarithmic in the size of the relations. We showed that our proposed summarized chronicle algebra and its variants are in fact the largest fragments of relational algebra with these incremental complexities.

The chronicle data model has been implemented for a large transactional system in AT&T. We believe that our model is applicable to numerous other application domains, including credit cards, billing for telephone calls, cellular phones, advanced telephone services, stock trading, consumer banking, industrial control systems, retailing, and frequent flyer programs.

Acknowledgments: We thank Jan Chomicki and Kenneth Ross for valuable suggestions.

References

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of ACM SIGMOD 1986 International Conference on Management of Data*, pages 61–71. ACM SIGMOD, 1986.

[BW94] Elena Baralis and Jennifer Widom. An algebraic approach to rule analysis in expert database systems. In Jorge Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of the 20^{th} International Conference on Very Large Databases*, pages 475–486, 1994.

[Cho92a] Jan Chomicki. History-less checking of dynamic integrity constraints. In *Proceedings of the Seventh IEEE International Conference on Data Engineering*, 1992.

[Cho92b] Jan Chomicki. Real-time integrity constraints. In *Proceedings of the Eleventh Symposium on Principles of Database Systems (PODS)*, pages 274–281, 1992.

[CSS94] Rakesh Chandra, Arie Segev, and Michael Stonebraker. Implementing calendars and temporal rules in next-generation databases. In *Proceedings of the Tenth IEEE International Conference on Data Engineering*, 1994.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB)*, 1991.

[DAJ91] Shaul Dar, Rakesh Agrawal, and H. V. Jagadish. Optimization of generalized transitive closure. In *Proceedings of the Seventh IEEE International Conference on Data Engineering*, 1991.

[GHJ94] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[alg,c]: Elevating deltas to be first class citizens in a database programming language. Unpublished Manuscript, 1994.

[GJS92a] Narain Gehani, H. V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB)*, pages 327–338, 1992.

[GJS92b] Narain Gehani, H. V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, pages 81–90, 1992.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, 1993.

[JM93] Christian Jensen and Leo Mark. Differential query processing in transaction time databases. In *In [TCG+93]*, pages 457–491. The Benjamin/Cummings Publishing Company, Inc., 1993.

[JMR91] Christian Jensen, Leo Mark, and Nick Roussopoulos. Incremental implementation model for relational databases with transaction time. *ACM Transactions on Knowledge and Data Engineering*, 3(4):461-473, 1991.

[LS93] Alon Levy and Yehoshua Sagiv. Queries independent of updates. In Rakesh Agrawal, Sean Baker, and David Bell, editors, *Proceedings of the Nineteenth International Conference on Very Large Databases*, pages 171–181, 1993.

[MPR90] Inderpal Singh Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. The magic of duplicates and aggregates. In *Proceedings of the Sixteenth International Conference on Very Large Databases (VLDB)*, pages 264–277, 1990.

[NYT94] New York Times. February 18, 1994. Front page article on mistakes made in updating the balance fields in an account after ATM withdrawals from Chemical Bank ATM machines in NY and NJ. The mistakes were due to buggy updating software.

[PI94] Sushant Patnaik and Neil Immerman. Dyn-fo: A parallel, dynamic complexity class. In *Proceedings of the Thirteenth Symposium on Principles of Database Systems (PODS)*, pages 210–221, 1994.

[Qia88] X. Qian. An effective method for integrity constraint simplification. In *Proceedings of the IEEE 4th International Conference on Data Engineering*, pages 338–345, 1988.

[RRSS94] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *International Logic Programming Symposium*, pages 204–218, 1994.

[SL76] D. G. Severance and Guy Lohman. Differential files: Their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3), 1976.

[SLR94] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In *Proceedings of ACM SIGMOD 1994 International Conference on Management of Data*, 1994.

[SS92] M. Soo and Richard Snodgrass. Mixed calendar query language support for temporal constants. Technical Report 29, Computer Science Department, University of Arizona, Tucson, Arizona, May 1992.

[TCG+93] Abdullah Uz Tansel, James Clifford, Shashi Gadia, Sushil Jajodia, Arie Segev, and Richard Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* The Benjamin/Cummings Publishing Company, 1993.

A Proofs of Selected Theorems

Proof of Theorem 4.1 We prove the monotonicity result by induction on the length of the chronicle algebra expressions. The critical step is to show that the difference and groupby operators are monotonic with respect to insertions into chronicles. Consider each possible operator:

- selection: $E = \sigma E_1$, where E_1 involves a chronicle. Compute $\Delta E = \sigma \Delta E_1$. When E includes sequence numbers, so must E, and by the inductive hypothesis, ΔE_1 contains only new sequence number tuples. Hence ΔE contains only new sequence number tuples.

- projection: $E = \pi E_1$, where E_1 involves a chronicle. We want to project the new E_1 tuple, and check if this already exists in the projection. If E contains sequence numbers, so must E_1, and ΔE_1 contains only new sequence number tuples. Hence, $\Delta E = \pi \Delta E_1$, and ΔE contains only new sequence number tuples.

- union: $E = E_1 \cup E_2$, where E_1 and/or E_2 involves a chronicle. We want to discard tuples common to E_1 and E_2. If E contains sequence numbers, so must E_1 and E_2, and ΔE_1, ΔE_2 contain only new sequence number tuples. Hence, $\Delta E = \Delta E_1 \cup \Delta E_2$, and ΔE contains only new sequence number tuples.

- difference: $E = E_1 - E_2$, where E_1 and/or E_2 involves a chronicle. If E contains sequence numbers, so must E_1 and E_2, and ΔE_1, ΔE_2 contain only new sequence number tuples. The chronicle data model also requires that the new sequence number could not be present in any other chronicle in the chronicle group before the current update. Hence, $\Delta E = \Delta E_1 - \Delta E_2$, and ΔE contains only new sequence number tuples.

- cross product: $E = E_1 \times E_2$, where exactly one of E_1 and E_2 involves a chronicle. Say E_1 involves a chronicle. If E contains sequence numbers, so must E_1, and ΔE_1 contains only new sequence number tuples. Hence, $\Delta E = \Delta E_1 \times E_2$, and ΔE contains only new sequence number tuples.

- equijoin between chronicles:
  $C = C_1 \bowtie_{C_1.SN==C_2.SN} C_2$. By induction, ΔC_1 and ΔC_2 contain only the new sequence numbers.
  $\Delta C = \Delta C_1 \bowtie_{\Delta C_1.SN==C_2.SN} C_2 \cup C_1 \bowtie_{C_1.SN==\Delta C_2.SN} \Delta C_2 \cup \Delta C_1 \bowtie_{\Delta C_1.SN==\Delta C_2.SN} \Delta C_2$, where C_1 is the old value of C_1. The first two terms in the union are guaranteed to be empty, so that

  $$\Delta C = \Delta C_1 \bowtie_{\Delta C_1.SN==\Delta C_2.SN} \Delta C_2,$$

  and ΔC contains only the new sequence numbers.

- aggregation: $E = \texttt{GROUPBY}(E_1, GL, AL)$, where E_1 involves a chronicle. If E contains sequence numbers, so must E_1, and ΔE_1 contains only new sequence number tuples. Further, the grouping list GL must contain sequence number. Then, $\Delta E = \texttt{GROUPBY}(\Delta E_1, [GL], [AL])$, and ΔE contains only new sequence number tuples. □

Proof of Theorem 4.2 *Time Complexity*: By induction on the length of the expression in CA. Hypothesis: For any subexpression g, the number of tuples in Δg is exponential in number of joins in the expression g, and can be computed in time exponential in number of joins in the expression g:

$$\text{Time} = O((u \mid R \mid)^j \log(\mid R \mid)),$$

$$\text{Space} = O((u \mid R \mid)^j),$$

where u is the number of unions in expression g, $\mid R \mid$ is the size of the relation R, and j is the number of equijoins and cross products in the expression g.

Consider any subexpression $g = op(h)$ or $g = op(h_1, h_2)$:

Select, Project: For each tuple in Δh, constant time per tuple to derive new changed tuple. Size of g = size of h.

Union: $g = h_1 \cup h_2$: For each tuple in Δh_1 and Δh_2, constant time to derive each tuple in Δg. Size of g = size of h_1 + Size of h_2. Thus, each union operator can at most add the size of two Δ relations.

Difference: $g = h_1 - h_2$: For each tuple in Δh_1, $O(\log(\mid \Delta h_2 \mid))$ time to derive each tuple in Δg. Size of $g <$ size of h_1. Since size $\mid \Delta h_2 \mid$ is exponential in number of unions and joins in the expression h_2 as given by formula above, $\log(\mid \Delta h_2 \mid)$ is proportional $uj + j\log(\mid R \mid)$. Thus the time to compute Δg is $O(u_2^{j_2}(\mid R \mid)^{j_2} \log(\mid R \mid))$, where u_2 and j_2 are the number of unions and joins/cross-products in expression h_2.

Aggregation: Since the sequence number is included in the groupby list, the new inserted tuples form one or more brand new groups. A sort on these can be done in time $\log(bigo(\text{number of inserted tuples}))$, and the aggregation function computation can be done in time $O(\text{number of inserted tuples})$.

Cross Product: Each cross product with relation R adds $\mid R \mid$ tuples into the result for every tuple added to the operand chronicle. Thus, the size of the result grows by a multiplicative factor of R with each cross product. The time taken is one unit for each tuple in the output size, so the inductive hypothesis on time also holds. (Consequently, j cross products can grow the result size by $(\mid R \mid)^j$.)

Equijoin: $g = h_1 \bowtie h_2$. Each equijoin can increase the size of the result to a product of the sizes of the two chronicles. If the number of joins in h_1 and h_2 is j_1 and j_2, the number of joins in g is $j = j_1 + j_2 + 1$. The size can grow to

$$O(u^{j_1}(\mid R \mid)^{j_1} \times u^{j_2}(\mid R \mid)^{j_2}) = O(u^j(\mid R \mid)^j$$

One computation step is needed to output each of the above tuples.

Space Complexity: In the chronicle algebra, discounting the subexpression for the relational view on relations, all subexpressions include the sequence number. The monotonicity analysis shows that the change to each such subexpression can be computed without storing the corresponding view. Thus the space complexity is the number of tuples that are inserted into each chronicle expression.

Complexities for $\text{CA}_{\bowtie}$: Join with relational view: For each tuple in Δh, $\log(\mid R \mid)$ to find the single matching tuple. Size of g = size of h. □

Proof of Theorem 4.3 (Outline): A projection without including the sequencing attribute, or a groupby operation without including the sequencing attribute as a grouping attribute leads to a view that does not include the sequencing attribute, and is thus not a chronicle. Further, a simple cross product $C_1 \times C_2$ between two chronicles would require us to look up all old values of one chronicle upon insertion into the second chronicle. This would increase the complexity of change computation to be in class IM-C^k. Similarly, an expression that contains only a non equi-join (on the sequence number) between two chronicles would require that old chronicle tuples be looked up for at least some database state. □