Discussed is a data base audit trail. It is defined here to be a generalized recording of "who did what to whom, when, and in what sequence." This information is to be used to satisfy system integrity, recovery, auditing, and security requirements of advanced integrated data base/data communications systems. This paper hypothesizes what information must be retained in the audit trail to permit recovery and audit later in time and a scheme of organizing the contents of the audit trail so as to provide the required functions at minimum overhead.

Introduced are the concepts of types of audit required, DB/DC audit assumptions, time domain addressing, time sequences required to support versions of data, what constitutes an audit trail, and implementation considerations.

Generalized audit trail requirements and concepts for data base applications

by L. A. Bjork, Jr.

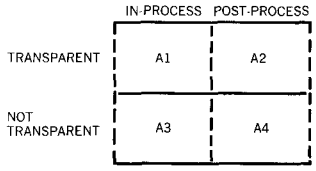
Auditing of a data base/data communication (DB/DC) application is defined as the act of monitoring the application for compliance with accounting rules and practices. Auditing an application is essentially certifying the integrity of the system by verifying that rules and policies dictated by laws, business agreements, etc., are being followed by the application.

There are three basic objects of interest for audit in a DB/DC environment, namely:

- The user – who entered what data from what terminal, etc.
- The program – on a given execution of a program what version was used, what branches were taken under what conditions, etc.
- The data – what was a particular field value before a specific transaction updated it and what was its value after update.

These objects and their interactions are usually identified as being within a common boundary for each invocation of the program. This common boundary is identified by a transaction name which usually serves multiple purposes within the system such as scheduling, recovery (backout), and auditing.[1]

Figure 1 Major types of audit



In advanced DB/DC systems, more manual processes and applications will be computerized. The audit support must be commensurate with the support to automate new processes. Otherwise, the inability to audit will limit new applications from being committed to an unauditable computerized environment.

Also, in advanced systems, commitments will be made more at terminals using on-line data, rather than after verification of the results of a batch run. This means that the audit function, e.g., recording of audit trails, the verification-of-results function, etc., must be system-supported to the same level as the dependency being placed on the on-line data by the terminal user.

Finally, in advanced systems, the sequence of processes interacting with the data base is less repeatable in an on-line interactive environment than in a batch environment. This sequence is due to the random arrival of incoming transactions rather than the preplanned processing sequence typical of batch processing. Therefore, a generalized audit trail facility must be provided that tracks data usage and captures the unrepeatable sequence of processes during the execution of the process itself.

Aspects of auditing

types of audit

Four major types of audit are of interest for advanced DB/DC systems as shown in Figure 1. In-process signifies that the monitoring of the application and verification for adherence to specified rules are performed while the process being audited is in execution. Post-process signifies that the recording of the audit trail is performed concurrently with the process to be audited, but the audit itself is performed after completion of the process. Transparent or not signifies whether the process to be audited is aware of the audit.

The four types of audit as seen in Figure 1 each have the following characteristics.

A1 is where the audit is being performed in real time transparently to the on-going process being audited. Examples of A1 are: (1) the auditor introduces test transactions into the system and verifies the process being audited by analyzing the outputs based on specific inputs, (2) two asynchronous processes with one monitoring the other at defined audit points, and (3) full interpretation with the audit process being the interpreter of the process being audited.

A characteristic of A1 is that the audit function, in addition to being transparent to the on-going process, does not alter the course of the process. That is, the audit is usually not in-line

with the day-to-day accounting practices of running the enterprise. If an in-process audit finds a violation of accounting practices, the usual procedure is not to stop the audited process but rather to have the person or group responsible for the erring process fix it and then issue adjustments to the incorrect outputs.

Additionally, A1 has the characteristic of not requiring an audit trail (not for audit purposes but may require one for other purposes) since the auditing function is performed in-process.

A2 is an "after-the-fact" audit in which a process or person looks back in time at the effects, actions, algorithms, etc., of an earlier process. This type requires a recording of a great deal more information than an audit may actually require and use since all the earlier processing is rarely audited. A sampling technique usually chooses which subset to actually audit out of all recorded data.

A3 is an in-line audit of the application's process. What audit rules, when to apply them, and their results must be preplanned as part of the application. A3 is different from A1 in that the audit rules in A3 cannot be changed without reprogramming the process. An example of A3 is the application displaying certain data (to an auditor) when a specific transaction type or instance is encountered.

A4 is the case of the process to be audited explicitly saving what is required for an audit of the process later in time. A4 is typically used in the debugging mode in which the debug tools (e.g., trace, storage dumps, etc.) are invoked in-line for analysis later in time. The data saved may also be used for recovery purposes such as determining what the initial values were during process execution.

DB/DC audit assumptions

In developing general audit trail concepts, certain assumptions were made regarding the audit environment of advanced DB/DC systems. These assumptions are summarized below.

1. Auditing application systems must be permitted by the following classes of objects: (a) application transaction levels (where one level may be nested within a higher level), (b) procedure, (c) data type—fields, records, files, (d) user, (e) terminal, and (f) any Boolean combination of the above.
2. The auditor requires the system to retain for addressability later in time (after process completion) the following (maximal case): (a) identity of transaction by occurrence, (b) name and version of procedure, (c) name and version of

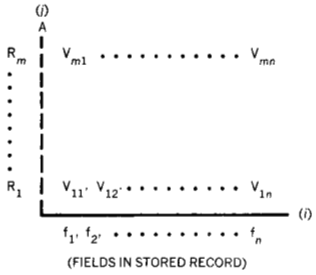
interpreter (e.g., OS/360, version 21), (d) name and version of transaction input by field (where a new field version is created by each transaction changing the field value), (e) name and version of transaction outputs by field, (f) time stamp unique with respect to sequence, (g) sequence flow between related transactions (predecessor and successor), (h) user submitting originating transaction, (i) terminal and/or node in network that originated transaction, and (j) operations, and sequence of operations, within a transaction on the inputs and outputs.

3. The act of auditing must be capable of being transparent to the process being audited as well as to what is audited.
4. The act of establishing and activating audit hooks must be capable of being dynamic (as well as static) and performed against an on-going process without logically interrupting the process.
5. The system must support "transparent event descriptors" whose content defines the conditions under which a recording is to be made for later audit and recovery purposes. Examples of what the descriptor must contain are: (a) process to be audited, (b) when (time), (c) what data and values, (d) what operators, and combinations of data and operators, (e) what level and sequence of transactions are to be audited, and (f) what information to record when the event is either true or false.
6. The audit function (detection and recording) must not be disabled for that period of time for which auditability will ever be required later in time.
7. The audit support must provide for the tracing of a sequence of transactions across the man-machine interface.
8. The audit support must fulfill the "after-the-fact" reconstruction and repeatability legal requirements on a data base of the anticipated laws in the time frame of the later 1970s and 1980s. That is, the interactions between processes and data and the consequences thereof must be traceable.
9. The system must support the acts of audit event detection and recording of the captured data. In a data-independent environment, the sharing of a physical resource must be transparent to an application program.
10. The system support for the audit function must be standardized across application systems—not a special audit facility for each application subsystem.
11. The auditor must be able to use the same terminology and names for an "after-the-fact" audit as were used during the process itself. Examples are: (a) field names and (b) procedure names.
12. The auditor (by command and procedure) requires the following type of update capability by version of field. (This requirement assumes the system has retained prior versions of fields.)

- The ability to logically supersede the "now" (most recently recorded) version (a) prior to any dependent use (i.e., use for the purpose of making a decision) by another process and (b) after dependent use by another process. The requirement is recursive in that multiple logical supersedures must be supported.
- The ability to logically supersede an earlier version later in time (a) prior to any dependent use of the earlier version and (b) after dependent use of the earlier version. This requirement is also recursive.

13. The installation must be able to control or choose between cost/performance tradeoffs for functional support of the detection and recording of audit events and information.

Figure 2 Stored record occurrences



Definition of an audit trail

The preceding discussion has sought to present the environment and requirements of a generalized audit trail for advanced DB/DC systems. This section defines and characterizes such an audit trail.

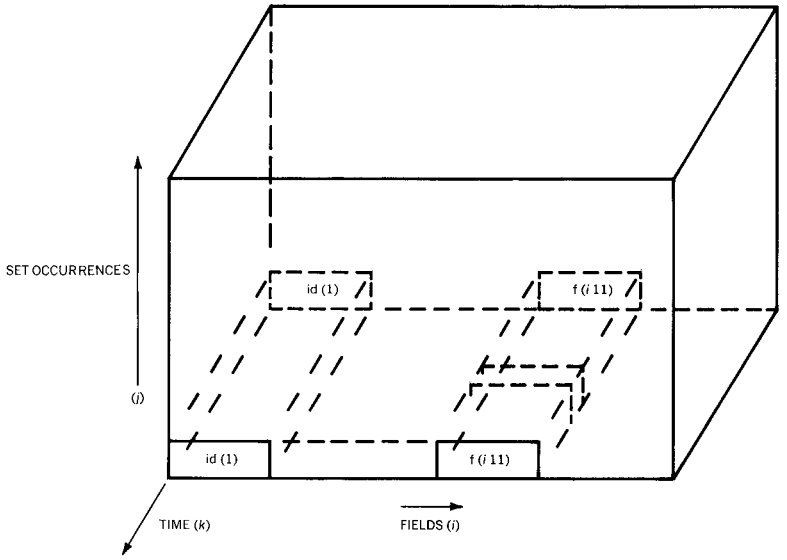
Our definition of an *audit trail* is: A history of activities by transaction, posted because of operations on specific data; operations are those functions that are defined as events (via transparent event descriptors) to be noted in the audit trail as a consequence of a particular interaction with the data base. For example, operations might be updates as is required for simple backout or may include references to data as well to be used for scoping the bounds of propagated errors (see Reference 2).

graphical definition of audit trail

Figure 2 shows the traditional representation of a data set consisting of one stored record type comprised of n fields containing m stored record occurrences. On a two-dimensional graph, the fields are denoted on the horizontal axis, the record occurrences are represented on the vertical axis, and an (i, j) coordinate selects the value of field i from record j. This two-dimensional set of values represents the set of "now" or current values of all fields in all stored records in the data set.

An audit trail concept adds the time dimension as a third coordinate to Figure 2. Figure 3 shows the time dimension with the "now" values being the closest to the origin and prior values (versions) going chronologically back through time as one traverses the time coordinate. To locate a unique value, an (i, j, k) coordinate must be specified where i is the field name (assumed unique through all versions of values), j is the unique stored record identifier (also assumed unique through time), and k is the point in time for the desired value. If, in addition, supple-

Figure 3 Time domain addressing concept



1. f(*ijk*) IS FIELD *i*, VERSION *k* (WHERE *k*-1 PRIOR VALUES HAVE BEEN PREVIOUSLY RECORDED) WITHIN SET OCCURRENCES IDENTIFIED BY id (*j*).
2. ids ARE UNIQUE WITHIN SET THROUGH TIME.
3. VALUES OF UNCHANGED FIELDS ARE LOGICALLY REPLICATED FROM VERSION TO VERSION.

mental information is kept about each version of the field (who created it, when, etc.), we have the complete audit trail concept.

Methodology of an audit trail

A proposal for an audit trail organization is presented to demonstrate the feasibility of meeting the requirements that were described in the first section. This example explains in a generalized implementation sense a typical sequence of operations on "now" and "prior" field values. The following assumptions are made:

- The audit trail event descriptor specifies that the operations of create, reference, and update for field x are to be recorded in the audit trail.
- The time stamp is unique with respect to audit trail entries per field.
- The field name x remains constant for all generations of values of x.
- The stored-record identifier remains constant through all generations of nonidentifier fields.
- For simplicity, the additional audit trail contents have been ignored.

Following are the steps in the sequence.

A. *Create a value for field x at time* t_0—Create $x = 10$ at t_0. Only the audit trail for field x is considered. The audit trail consists of two parts: the "now" value and the history of prior activities. The "now" value is the latest version of field x and contains (in this example) only the value. Each history (audit trail entry) consists of a four-way relation (v, t, op, p) where v is the value at time t, op is the operation that caused the audit trail entry and p is a set of linkages to prior audit trail entries. Thus, the audit trail contains:

$10|(10, t_0, C, -)$

where | separates the two parts into the "now" value and the history of former values. C denotes the create operation.

Also note that the "now" value 10 is associated with the most recent (and only) audit trail entry.

B. *Reference x*—Reference x at time t_1. Note that the version of x is assumed to be the "now" value since no "as of" time is specified. The audit trail, after the reference, appears as:

$10|(10, t_1, R,)(10, t_0, C, -)$

where the underscore arrow is the real-time sequence of operations on field x.

C. *Update x*—Update x to 12 at time t_2.

$12|(12, t_2, U,)(10, t_1, R,)(10, t_0, C, -)$

D. *Refer to prior generation*—Refer to $x(t_1)$ at time t_3. This reference is to a prior version of the value, namely t_1, being asked at time t_3. The value of x returned is 10.

$12|(10, t_3, R,)(12, t_2, U,)(10, t_1, R,)(10, t_0, C, -)$

The overscore arrow indicates which generation of x was referenced as contrasted with Step B, above, where the assumption was the "now" value. This linkage gives the capability to record references to prior values in the same audit trail as references to the "now" value.

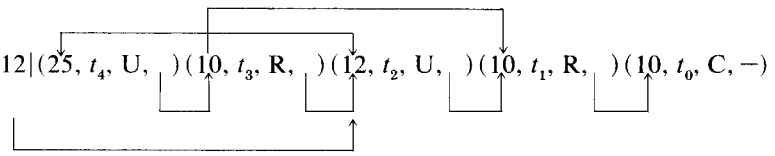
In effect, this is a two-level audit trail: the first level is the audit trail of activities of the "now" value; the second is the audit trail

of the first-level audit trail (the activities "now" against the "then" now values).

The underscore linkage from the "now" value, 12, to the t_2 audit trail entry is required to provide addressability to the supplemental information recorded when x was updated to 12. In other words, the head of the push-down stack shows activity on a prior generation of x, not the "now" value.

The motivation of having the "now" value be a separate part of the audit trail is to provide compatibility with today's data set formats. Also, the content is selected by the data base administrator. The audit trail may reside on different devices and be accessed by different access methods compared with the "now" values.

E. *Update a prior generation of* x—Update $x(t_2)$ to 25 at time t_4. This is the case of updating an earlier generation of x when an error has been detected "after the fact."



The t_4 audit trail entry now indicates that the update to x at t_2 has been superseded by the t_4 entry.

At least two cases of error propagation are apparent. The first involves a blind fix wherein the value of the record or field is changed as specified independently of subsequent usage. The second case is an application-dependent repair to later-generation values—such as adding the increment 13 to all generations of x after t_3.

It is the responsibility of the user of the audit trail interface to take the necessary corrective action to repair later values and determine who depended on the incorrect values. The "who" is omitted in this example.

The security controls required to prevent unauthorized usage of the audit trail are assumed to exist but not addressed in this paper.

The problem that now arises is: who gets to see either the uncorrected version of x at time t_2 or the logically updated version of x at time t_4 as it should have been at time t_2? The auditor or person using the audit trail for debugging purposes may want to see the real history of activities. Otherwise, an application program may want the logically correct value at time t_2 ($x = 25$). One

way of solving this potential ambiguity (which has always existed) is to have a "user intent" code in the interface denoting intended audit trail usage of this particular user.

F. *Concept of a cutoff period*—A cutoff period is a distinct real-time interval in the time-ordered sequence of audit trail entries that has been useful to record in a catalog. The time intervals may be (t_2, t_0) and (t_4, t_3) in the above example (e.g., t_2 = April 1, t_0 = January 1). The cutoff periods will generally be chosen to coincide with some legal or accounting date requirements such as end-of-month or end-of-year. A cutoff period permits the user/system to have multiple entry points to the time-ordered sequence of audit trail entries. Without a cutoff period concept, a sequential scan is implied from the latest activity of the field serially back through time through all preceding activities until the desired field generation is found. A cutoff period concept implies better performance by permitting the user/system to choose the cutoff period entry point to the audit trail that is later than and nearest to the desired earlier value.

A cutoff period is also useful in the area of data purging and reduction. A simple purging algorithm might be to purge all audit trail entries that are more than y years old, or created prior to cutoff period 3. Or the data might be summarized such as average quantity-on-hand for the cutoff period 3. Purging is the process of deleting audit trail entries and making their contents no longer addressable under system control. (This is probably a data reduction operation with the original data being retained for x years).

A problem with the cutoff period concept is that operations such as update in one cutoff period may be logically superseded later in another cutoff period. Case E, above, is such an example. One discovers later what the then "now" value should have been. One cannot simply go to that desired cutoff period and start searching backward for the desired update (for example) since a later cutoff period contains the logically corrected value which, in turn, may have been still later superseded, etc. Each cutoff period must be able, therefore, to have an indication of whether its audit trail entries have been corrected logically in a later cutoff period—and possibly which entry in which later cutoff period.

Content and format tradeoffs of an audit trail

This section examines the detailed parts of an audit trail entry and some of the format considerations and attendant tradeoffs. Following is a list of candidates that have been identified as being useful or required to be recorded in the audit trail.

audit trail entry

The name of the data being operated upon. Whether or not the name of the data is recorded explicitly in the audit trail is a function of the naming convention and scope of the audit trail entries. The naming convention assumed in this paper is that the data name remains constant for all versions of its values. If the audit trail has a scope of many data sets and the same field can appear in more than one data set, then obviously at least a two-level naming scheme is required (e.g., data set name, field name). Both data set and field name can be factored out of each audit trail entry and placed into a dictionary. The dictionary would contain descriptive information, constant across a cutoff period, such as data name, representation, version, etc., needed to fully interpret the audit trail entries. In reality, an audit trail is also needed on the dictionary to track name changes and synonyms of a field.

Not addressed in this paper are the problems of a data name being changed (or deleted) between generations, synonyms, and how to know which name to use in the first place (as well as what it means).

The new value after the operation. The new value is recorded after the operation (as defined in the audit trail event descriptor). The value prior to the operation is available as the prior operation's result. One possibility is that the prior value and the new operation are recorded in the same audit trail entry, thereby making the prior value immediately addressable. This gives high-performance capability to the data restore of the original value at the expense of redundancy of the value in the audit trail.

Operation causing audit trail entry. This operation is the recording of what interaction with the data caused the entry to be made in the audit trail. Those that are ready candidates are as follows: (a) Create or insert new data, (b) Delete data, (c) Update, (d) Reference data for the purpose of commitment (using this data as the basis of future actions), and (e) References for any reason (such as debugging purposes).

Time stamp. The actual time that the given operation occurs is recorded. The granularity of the time stamp must be fine enough so that no two operations have the same time stamp. Otherwise, the real sequence of operations is not guaranteed reproducible.

The representation. The format of the field at the time of the operation upon the value is possibly recorded. This provides the capability of being able to change the field representation from one version to the next. The audit trail interface could permit viewing prior values through the "now" value's descriptor (current representation of the field). Therefore, a possible format

conversion is required when operating upon prior values. This conversion would be transparent to the user of the interface. If the representation is constant through all generations, as the name of the data is assumed to be, the representation also may be factored out of each audit trail entry and placed into a dictionary.

Status flags for each generation. A status flag will optionally be included (as specified by a descriptor defining what is to be recorded) with each audit trail entry. Some of the usages of the status flags might be:

a. Deleted value—On the delete verb, the status code is set to the delete state. On retrieve, a no-data-found condition would be raised.
b. "Bad" data indicator—A program detecting a wrong or suspected wrong value could set a parameter in the audit trail interface that, in turn, would set the status flag to the "bad" data indication.
c. Audit trail entry superseded—This flag would indicate that there exists another audit trail entry later in time that logically supersedes this entry (i.e., an adjustment exists to this version). The cutoff period that contains the superseding entry could be in a dictionary.
d. Purge control parameters—Certain conditions may be indicated in status flags such that when a purge routine scans audit trail entries, it may automatically remove audit trail entries from system control. The simplest case of automatic purging may be the or-ing and/or and-ing of certain status flags followed by purging when the result is true. For example, when the value has been deleted or is older than three years and entered by user *x* it could automatically be purged.

Program identification and version. The program identification interacting with the data is recorded. If the program has multiple versions, another table is hypothesized, in addition to the audit trail, that contains the unique program identification in the audit trail and which version of the program the identification corresponds to.

Transaction identification. In a DB/DC environment, the unique transaction identification must be recorded so that the scope and unit of work is bounded and identifiable for reasons such as resource allocation, backout, etc. The sequence of transactions (e.g., one transaction generates three transactions each of which generate two, etc.) is also hypothesized to be recorded in a separate table for transaction history and later sequence reconstruction.

User identification. The identification of the user is required to be recorded for recovery and audit reasons. A recovery use of

the user identification might be as follows. A user has depended upon incorrect data and the data is corrected. The user must be notified and a compensation recovery process undertaken. Verification of the recovery procedure would, however, still be an audit function. It is especially important to record the user identification when the commitments are between humans and the computer discipline environment. (If a human used the erroneous data then to recover, the job cannot simply be rerun and the person not told.)

Terminal identification. The terminal the user was at when he entered the transaction is optionally recorded in the audit trail. The mapping from logical to physical terminals (if one exists) is a separate table that also must be maintained through time.

Time sequence linkages. As indicated in the example in the third section, at least two time sequences are recorded in the audit trail. The first one is the real-time sequence of operations against the data. By traversing this sequence from the latest time to the entry of data creation, every operation on every version of the data can be examined (assuming the event descriptors specify all operations on the data to be recorded). The second sequence is the correct logical value for each version of the data. This is *not* the real "time of happening" sequence since an earlier version value can be corrected later in time. We assume here that the "now" values will be used more frequently than prior values. Therefore, the time sequences will run backward through time between audit trail entries starting with the "now" value and running backward to the first creation operation. The user of prior values will incur more overhead than users of the "now" values.

audit trail format

We now examine some of the considerations and tradeoffs in designing an audit trail format.

Compatibility with current data set formats. As previously explained, the audit trail is considered another dimension to today's data sets with the "now" value the top (and only) entry in the conceptual stack. An audit trail format must be designed so that existing data set formats continue to be satisfactory for the "now" values, whereas future activity against the "now" value is recorded in a new format. If this can be done, *no* migration to a new format is required. Activity against the data set is recorded in a new format suitable for the audit trail requirements (i.e., multiple time sequences), whereas the new "now" value is preserved in existing data formats. Also, if no audit trail is maintained for a data set, then no changes (e.g., copy) are required for existing data sets.

Factoring by component of audit trail entry. Depending upon the mode of processing (e.g., batch versus on-line), various components of an audit trail entry may be factored out of each entry, thereby saving much storage space. As already assumed, the name of the data is factored out of each entry and placed into a dictionary; likewise, the representation, if it is constant through all versions, can be factored.

In a batch environment, user identification, transaction identification, and terminal identification can be factored to the job level, whereas program identification and representation can be factored only to the job-step level. In an on-line environment, no general factoring rules are readily apparent, assuming random arrival of different transaction types.

Hardware implications. The possibility of designing hardware to support audit trails optimally are enormous. Some of the major tradeoffs are listed below.

a. Sequential write-once only—As explained previously, the audit trail has no concept of update-in-place, but rather a sequential write-once-only characteristic.
b. Stageable from archives—As the data in the audit trail becomes older, there is a "decay" function on the usefulness of the audit trail contents. The audit trail may be automatically written onto an archive-type device according to some time function specified by the installation.
c. Audit trail content reconstruction—The audit trail must be physically reconstructable if damage occurs and is detected. Detection of damage to the audit trail is critical. Use of a damaged audit trail entry under the assumption that it is correct constitutes automatic error propagation. Worse yet is the fact that system recovery is invalid or impossible to perform since the vehicle (the audit trail) that would have permitted recovery is invalidated. This circumstance assumes that the information required for audit is a subset of that required for recovery, and both are recorded in the audit trail. Damage repair of the audit trail may be effected by duplication of the audit trail, Hamming codes, use of a checkpoint and journal, and other ways of recreating the original data.
d. Higher reliability in writing the audit trail—The audit trail must be written with a high degree of reliability so that its contents are safe in the event of system failure prior to completion of the write operation. The Advanced Administration System, for example, wrote unblocked records onto its journal so that if a power failure occurred and main memory was destroyed, the record was safe and preserved in the journal.[3] One way of meeting this requirement is to provide a highly reliable buffer not destroyed by power failure.

e. Need parallelism in writing the audit trail—An area that obviously needs investigation is what percentage of a system's "horsepower" would be directed to the audit trail under various activity mixes. Activity levels could rapidly be attained such that useful throughput was nil because of the volume of data queued to be written onto an audit trail.
f. One format (both machine and human processable)—An alternative to two different audit trail formats (i.e., an internal machine-readable format and a human-readable format, such as a report) is a common format. This would permit a human, such as the auditor, to browse through the audit trail contents using some off-line device such as a microfilm reader while simultaneously allowing the system to provide addressability to audit trail contents. The possible disadvantage to a single format is that the sheer bulk of data potentially in the audit trail may overwhelm a human trying to browse through the audit trail. Possibly application-level data interactions could be kept in a common form, and system-level audit trail entries kept in another form.

Integrated versus distributed audit trail. In an implementation of an audit trail, several criteria have been identified that would be options in determining whether one integrated audit trail or several physical audit trails should be provided. The criteria for determining the scope of entries in an audit trail are examined below.

a. An audit trail per data set—An audit trail per data set (as assumed in this paper) is motivated primarily by the concept of the "now" value being a special case of all versions of a unit of data. Another assumption is that the total history of activities across all users, programs, time intervals, etc. is important to be directly associated with the data. Finally, just as in an integrated data base environment (i.e., minimally redundant data), one data set is shared by many programs and users. Here, the audit trail per data set is the central repository for all historical activities (with no redundancy) against the data set.
b. User or class of users—An audit trail could be partitioned into disjoint exhaustive users or classes of users. For example, an audit trail of union members' activities might be desirable in the proper environment.
c. Program or application program system—Separate audit trails might be desirable for each program or family of programs such as all application programs using IMS, CICS, etc. Audit trail implementations and options could be tailored to each application program using this criterion.
d. Transaction type—A separate audit trail could be defined for one or more transaction types.

e. Time intervals—The cutoff period concept is the facility for separating the audit trail into discrete time intervals. If, in addition, the Data Base Administrator can change the audit trail options at each new cutoff period, the system would provide a general audit trail partitioning scheme as a function of time.
f. Logical data base—It may be desirable to keep one audit trail per logical data base. One audit trail may exist for all the information in the data base on a common subject. In this case, one audit trail would have a scope of one or more data sets.
g. Operations upon data—Separate audit trails may be kept per type of operation upon the data. For example, update operations may be recorded in one audit trail on a faster device for quicker data restore capability, whereas data references are in a separate audit trail for the auditor, security officer, etc.
h. Combinations of the above—Combinations of the above criteria will undoubtedly prove most useful for any given implementation. For example, specific operations upon a logical data base may have different cutoff periods in separate audit trails.

Method of synchronization between distributed audit trails. If related recorded data has been distributed between more than one physical audit trail or even within the same audit trail, a mechanism must exist that permits the distributed data to be recollected according to some criteria. This method of synchronizing entries in different audit trails is performed according to criteria such as common program identifications, the same time interval, or in general, equal values of components of audit trail entries. The unit of data to be synchronized is subject to the general requirement that its value must be unique with respect to the set of values over all audit trails that can contain that unit of data. In other words, if user identification is to be the unit of data for synchronization, each user must have a unique identification. Conversely, all entries in all audit trails within some scope that have the same user identification indeed refer to the same user. Each component of an audit trail entry is a potential synchronization unit of data.

Conclusions

Audit trail requirements were discussed as they apply to data bases. In summary, the major conclusions reached in this paper follow.

The concept of an audit trail—as a time-ordered history of activities against a unit of data—is anticipated to be a major integrity tool for shared data usage for the late 1970s and beyond.

Time domain addressing is introduced as the model upon which an audit trail is developed. Time domain addressing adds the dimension of time to current values of stored records and fields within stored records.

A single time invariant interface that addresses both current as well as former values of a field is required to provide data independence to historical versions. Without this single interface, approximately 90 percent of an enterprise's data (the former versions) does not have data independence since two interfaces would be required by an application program to address the current version and former versions.

The DB/DC system must understand the concept of version of a field so that the system can: (a) provide the common, but complex version management functions (not force the user to provide his own support), (b) synchronize recovery and backout of data base updates by version of records/fields, (c) permit changes for field attributes to be monitored transparently to the application program, and (d) prevent the user from supplying a false version number by having the system know how to construct a new version number.

The audit trail contents is defined that identifies what additional data must be recorded when a transaction operates upon data in the data base. Operations other than update are provided for in the contents. The act of recording the audit trail contents must be capable of being transparent to the transaction.

A cutoff period concept is introduced which is an installation-defined boundary for versions of data. A cutoff period permits data reduction and purging criteria to be introduced so that versions no longer of interest can be eliminated from system control.

Finally, feasibility of audit trail technology is investigated by showing an example of the types of time sequences required to support time domain addressing.

ACKNOWLEDGMENTS

Mr. R. C. Kendall has earlier developed the idea of a uniform, time-domain addressable interface.[4] Mr. G. H. Smith[5] and Mr. C. T. Davies, Jr.[6] have explored the idea of tracing read-only references as well as modifications to data to support post-process recovery. Special acknowledgment is given to Mr. K. R. Wright for many helpful comments and suggestions on the paper.

CITED REFERENCES

1. C. T. Davies, Jr., "Recovery semantics for a DB/DC system," *Proceedings, ACM '73* **28,** 136–141 (1973).
2. L. A. Bjork, Jr., "Recovery scenario for a DB/DC system," *Proceedings, ACM '73* **28,** 142–147 (1973).
3. J. H. Wimbrow, "A large-scale interactive administrative system," *IBM Systems Journal* **10,** No. 4, 260–282 (1971).
4. R. C. Kendall, personal communication, IBM Corporation, 1000 Westchester Ave., White Plains, New York (1972).
5. G. H. Smith, personal communication, IBM Corporation, Poughkeepsie, New York (1971).
6. C. T. Davies, Jr., *A Recovery/Integrity Architecture for a Data System,* Technical Report No. 02.528, IBM Corporation, San Jose, California (May 19, 1972).

Reprint Form No. G321-5012