Design of LDV: A Multilevel Secure Relational Database Management System

PAUL D. STACHOUR AND BHAVANI THURAISINGHAM

Abstract—**In a multilevel secure database management system (MLS/DBMS) users cleared at different security levels access and share a database consisting of data at a variety of sensitivity levels. The system should ensure that the users only acquire the information to which they are authorized. This is difficult as users could pose multiple queries and deduce unauthorized information. In this paper, we describe the design of Lock Data Views (LDV), a MLS/DBMS which is hosted on the LOgical Coprocessing Kernel (LOCK) Trusted Computing Base (TCB). LDV's security policy builds on the security policy of LOCK. Its design is based on three assured pipelines for the query, update, and metadata management operations. We describe the security policy of LDV, its system architecture, the designs of the Query Processor, the Update processor, the Metadata Manager, and the operating system issues. LDV's solution to the inference and aggregation problems are also described.**

Index Terms—**Classification constraint, classification level, inference and aggregation, LDV, LOCK, multilevel secure relational database management system, pipelines, polyinstantiation, secure policy, type enforcement.**

I. Introduction

A. Problem Definition

Within the Department of Defense (DoD), the number of computerized databases containing classified or otherwise sensitive data is increasing rapidly. Access to these databases must be restricted and controlled to limit the unauthorized disclosure, or malicious modification, of data contained in them. Present database management systems (DBMS's) do not provide adequate mechanisms to support such control. Penetration studies have clearly shown that the mechanisms provided even by "security enhanced" database systems can be bypassed, often due to fundamental flaws in the systems which host the DBMS. This has led to a reliance on a number of techniques for isolating sensitive database information. These include physical protection, "system high" operations, and use of manual techniques for data sharing. These actions are very costly and detrimental to operational utility and flexibility.

Trusted Computing Bases (TCB's), such as Honeywell's LOCK [4], have been designed to provide this type of control in terms of abstract entities and operations which refect an operating system orientation. The LOCK security policy consists of a discretionary security policy and a mandatory security policy. The discretionary security policy enforces need to know structures, while the mandatory security policy provides a multilevel control policy. The multilevel control policy is a noninterference policy which addresses both access to data and the flow of information in the system.

A DBMS presents a more difficult problem than that dealt with the current TCB's with their operating system orientation. This results from the ability of the DBMS to preserve or even enhance the information value of the data it contains. This is possible because it captures information in addition to the raw data values themselves through the incorporation of knowledge about the types of data and relationships among the data elements. A DBMS also allows for the creation of new data and relationships through the application of complex functions to the data. Because of these capabilities, one is forced to consider a number of factors beyond those normally addressed when dealing with operating system security. These include the impact of data context, aggregation, and inference potential.

B. Design Approach

Honeywell's LOCK Data Views (LDV) system, which is a multilevel secure relational database system (MLS/RDBMS) hosted on LOCK, addresses the above problems by allowing individuals possessing a range of clearances to create, share, and manipulate relational databases [7] containing information spanning multiple sensitivity levels. In LDV, the relational query language, Structured Query language (SQL) [1], [26], is enhanced with constructs for formulating security assertions. These security assertions serve to imply sensitivity labels for all atomic values, contexts, and aggregations in a database. The labeled data are partitioned across security levels, assigned to containers with dominating security markings or levels, and may only flow upward in level unless authorized otherwise. The ability of LDV to perform in this manner is a function of its design, and the operating system upon which it is hosted.

This paper describes the complete design of LDV. Section II presents the security policy of LDV. An overview of the design is described in Section III. The LDV language is described in Section IV. Sections V, VI, and VII

Manuscript received November 28, 1988; revised August 15, 1989. This work was supported under Contract F30602-86-C-0003 from the Distributed Systems Section/COTD of the Rome Air Development Center.

P. D. Stachour is with Secure Computing Technology Corporation, St. Anthony, MN 55418.

B. Thuraisingham is with The MITRE Corporation, Bedford, MA 01730.

IEEE Log Number 9035102.

describe the query, update, and metadata processing operations, respectively, in LDV. Operating system support for LDV is addressed in Section VIII. The paper is concluded in Section IX.

II. LDV Security Policy Overview

To meet the DoD security policy requirement, as stated in DoD directives 5200.28 [10], 54200.28-M [11], and 5200.1-R [12], both the operating system (LOCK) and a secure application (LDV) must itself define a security policy that it enforces. In order to understand the LDV security policy, it is essential to understand the LOCK security policy. Therefore, we first describe the LOCK security policy and then describe the DBMS security policy requirements and extensions.

A. LOCK TCB Security Policy

The LOCK TCB satisfies the security policy requirements defined for the A1 level in the Trusted Computer Security Evaluation Criteria [13]. These include requirements regarding mandatory and discretionary access control, object reuse and maintenance, integrity, and export of sensitivity labels for subjects, objects, and devices. In addition, it supports the A1 requirements for accountability, audit, and assurance. The LOCK security policy at the highest level states that:

''*Data is labeled with a level and flows upward in level unless authorized to do so otherwise*.'' This captures the DoD notion of security, which focuses on the confinement and protection of information (data in a context) from compromise. The policy is interpreted in terms of increasingly detailed specifications of the security relevant mechanisms for the system. This provides the basis for the enforcement of the security policy within LOCK. Supporting mechanisms, such as user authentication and accountability, provide assurance that the security policy mechanisms act in a manner consistent with the security policy. In addition to the mandatory and discretionary security policies, LOCK provides labeling, integrity and authentication, and accountability mechanisms. These are described in [17].

The mechanisms which implement the LOCK security policy are defined in terms of abstract entities and operations. There are three principal entities in the LOCK security policy; subjects, objects, and the Effective Access Matrix (EAM). Subjects are the active process-like entities in the system and objects are the passive file-like entities. The EAM defines the permissible flows of information within the system. The EAM is computed based on the security relevant attributes associated with the subjects and objects. The LOCK policy describes these attributes and the allowed accesses based on the notion of potential interferences between subjects.

The security attributes associated with the subjects include 1) clearance level (subject_level(S)), 2) user on whose behalf it is executing, and 3) domain in which it is executing (subject_domain(S)). Objects have a set of corresponding security attributes which include 1) classification level (object_level(O)), 2) access control list (ACL(O)), and 3) type (object_type(O)).

The term ''level'' (subject_level(S) or object_level(O)) represents a sensitivity level that captures both the hierarchical classification levels and nonhierarchical categories which from a part of the DoD security policy. Within LOCK it is assumed that the sensitivity levels from a partially ordered set (POSET). Level $L1$ is said to dominate $L2$ if $L1 \geq L2$ in the POSET (e.g., Unclassified < Confidential < Secret < TopSecret). The term ''User'' refers to the human on whose behalf a subject is executing. The ACL(O) is a list of permissible access modes to an object, on a per user basis, which are maintained for all objects in the LOCK system. The subject domain (subject_domain(S)) and the object type (object_type(O)) are introduced to support the Type Enforcement mechanisms. The relationships between domains and types in terms of allowable access modes are captured in a Domain Definition Table (DDT). The DDT is a matrix which is indexed by domain and type, and has as entries those modes allowed to objects of the given type of subjects in the given domain.

The LOCK security consists of a policy discretionary security policy and a mandatory security policy. The discretionary security policy allows for users to specify and control sharing of objects. A subject's access to an object is restricted based on the ACL and the user on whose behalf it is executing. The mandatory security policy is based on controlling the potential interferences among subjects. It consists of a mandatory access control policy and a type enforcement policy.

The mandatory access control policy restricts the access of a subject to an object based on the sensitivity levels of the subject and object. The system enforces the simple security property and the *-property of the Bell and LaPadula security policy [3]. The simple security property states that a subject has read access to an object if the subject's sensitivity level dominates the sensitivity level of the object. The *-property states that a subject has write access to an object if the subject's sensitivity level is dominated by the sensitivity level of the object. The type enforcement policy deals with aspects of security policy that are inherently nonhierarchical in nature. It restricts accesses of subjects to objects based on the domain of the subject and type of the object.

B. DBMS Security Policy Requirements

The LOCK security policy is incomplete in dealing with DBMS security because of its operating system orientation. The most significant contributor to complexity within the DBMS environment is the information carrying potential of the database structure. The DBMS preserves or even enhances the information content of the database by incorporating knowledge of the types of data and relationship among the data. The data manipulation capabilities of the DBMS also allow the creation of new data relationships through the application of complex functions to the stored data.

Our approach to providing a complete and tractable DBMS security policy extends the basic LOCK security policy through the incorporation of an explicit classification policy. The classification policy must address those factors which are crucial to a correct determination of the sensitivity level of data within the DBMS context. In particular, the policy includes the following classifications:

Name Dependent Classification: rules that refer to data items by name. This provides classification at the granularity of relations and attributes (for example, all the values of the salary attribute in the relation Employee are Secret).

Content Dependent Classification: rules that refer to the content of data item occurrences. This provides classification at the granularity of tuples and elements (for example, the values of the name attribute in the relation Employee are Secret if the corresponding salary values exceed 100 K).

Context Dependent Classification: rules that refer to combinations of data items. This can be used to reflect sensitivity of specific fields when accessed together (for example, each name value, salary value pair in the relation Employee is Secret).

Aggregate Classification: rules that classify collections of data items (for example, more than 10 name values taken together has a Secret classification).

Inference Control: the determination of data sensitivity based on the potential inferences that can be made based on a sequence of access requests.

C. DBMS Policy Extensions

The additional concern for a DBMS in a multilevel secure environment beyond that of LOCK is the proper labeling of information. To provide for that concern, two extensions to the policy of the TCB are required. One extension summarizes the actions that happen when a database is updated and the other when a query is made to the database. These extensions are described briefly here. A detailed discussion on these extensions is given in [18].

Update Classification Extension: The update classification policy addresses the problem of proper classification of the database data. That is, when the database is updated, the classification level of the data is determined. The data are then inserted into an object whose level dominates the level of the data.

Formally stated, we have the following: For all security levels $L1$ and $L2$ ($L1 \leq 12$) and all base relations R in the database (where $L1$ is the basic_level of R), a tuple T being stored securely in a partition P (at level $L2$) of R implies that the basic_level of any data of T stored in P is $\leq L2$.

We define the BASIC_LEVEL(T) of a tuple (or portion of a tuple) as the lowest level of the set of levels at which T can be securely stored. The security level of the subject who attempts the update operation, the name-dependent, content-dependent, and context-dependent classification rules are used to determine the basic-level of a tuple.

Informally, this means that we partition the data in the database into file objects based on the basic_level classification level of the data. We use LOCK enforcement on objects to provide most of the security, with the database extension mechanisms only handling special cases such as classification by context.

Response Classification Extension: The response classification policy addresses the problem of proper classification of response to queries. This is a problem because the response may be built based on the data in many base relations. In the process of manipulating and combining the data, it is entirely possible that the data will be used in a manner that reveals higher level information. The problem becomes more acute when one realizes that the response will be released into an environment in which many responses may be visible. Thus, the problem becomes one of aggregation and inference over time as well as across relations. In light of this, it seems fairly clear that a response can only be released if it is placed in an object whose level dominates the derived level of the response. This derived level is the maximum level of any information that can be deduced from the response by a user reading this response.

Formally stated, we have the following: For all responses R, and all objects O, a response R being written into object O implies that the security level of the object O is in the set of levels defined by Admissible_Derived_Level_Set(R). This set consists of all levels for which releasing the information in the response R at that level will not enable any user to infer any further information whose sensitivity level exceeds the user's level.

Informally, this means that the response is written into an ordinary object that can be shared in any arbitrary way, subject to operating system security policy. The appropriate security level of the object containing the response depends not only on the response, but upon what can be inferred by the response being released at that level.

D. Type Enforcement

Lock's type enforcement mechanism allows us to encapsulate applications such as DBMS in a protected subsystem, by declaring the DBMS objects to be of special types which are only accessible to subjects executing in the DBMS domain. We then carefully restrict the subjects which are allowed to execute in this domain. It is this approach that makes LDV a unique design.

The underlying LOCK security mechanisms are available within the DBMS domain. However, since only DBMS programs are allowed to execute in this domain, we can extend the underlying security policy to account for functional requirements of the MLS/DBMS without affecting other applications code. The principal concern is how to securely release data from the DBMS domain to the user domain. Fortunately, the underlying LOCK type enforcement mechanism supports the implementation of assured pipelines [5]. This provides a way to ensure that data passed between the DBMS and user domains are processed through appropriate trusted import and export filters. These pipelines can be proven to be both unbypassable and tamper-proof. Using this mechanism, in conjunction with some trusted (and we believe small) subset of the DBMS code, it is feasible to implement a classification policy which guarantees that any set of data released to the user domain has been properly labeled as to classification based on the set of static and dynamic dependencies known to the DBMS.

Such a classification policy can be as simple as merely classifying a set of data at the high water mark of the levels of the individual data elements in the set. It can also be extremely sophisticated, incorporating such features as history files for inference control or automatic downgrading of the results of certain numerical functions, such as averages or counts, of raw data. The type enforcement mechanism makes this possible. The integrity of the history files will be maintained by using type enforcement to restrict write access to a specific domain within the DBMS and then ensuring that such a write is always done correctly. Similarly, the automatic downgrade will be limited by domain, and we plan to ensure that subjects in this domain only perform the downgrade under very specific conditions.

III. Overview of the LDV Design

In this section, we provide an overview of the LDV design. In particular, data classification issues dealt with by LDV, the system architecture of LDV, and the processing of the three major operations; query, update, and metadata management, are described. Finally we revisit type enforcement upon which the design is built.

A. Data Classification

Due to the data manipulation potential of a DBMS, determining the proper classification of data is nontrivial. Factors which may affect the proper classification of a set of data include:

—data content
—context of the data
—functional manipulations of the data
—external dependencies
—potential for inference.

Inference is the most difficult factor to treat adequately. Determining the potential for inference from an arbitrary collection of information is generally very difficult. We believe that it is possible to arrive at partial, yet meaningful, solutions to this problem within the context of a specific application.

It is important to observe that these classification factors can be partitioned into two categories: static and dynamic. Static dependencies (e.g., content) can be applied to the data at any point between its creation and release from the DBMS. Dynamic dependencies (e.g., functional transforms), on the other hand, can only be applied at the time a particular set of data is instantiated for output. In addition, what has been released from the database before may partially determine the classification of the current item to be released.

DBMS security requirements are most naturally expressed in terms of the user visible output of the DBMS. However, one must also be concerned as to what constitutes a secure write to the DBMS. Writes to a database will be secure, provided that they are done in a manner consistent with secure queries and do not introduce covert channels.

Data content dependencies arise when certain values, within the simple context of being in the database, are considered sensitive. Context dependencies are typified by intelligence information. Typically intelligence data and sources are individually of significantly different sensitivity than their combination. Functional manipulations of data involve the application of deterministic functions such as counting the number of elements, average, and rounding. Such functions may significantly lower the sensitivity level of the response relative to the raw data that was processed. Alternatively, the joining of two items of data may raise the joint classification beyond that of any individual one. There are also external dependencies, such as time. An aircraft destination may be highly classified until after departure, at which point its flight plan is openly disseminated to air traffic control.

The factors affecting classification which we address reflect our belief that aggregation and inference represent a significant security problem in a DBMS. Instances where the sensitivity of data is enhanced due to inference and aggregation frequently, but are significant when they occur. We note that if inference and aggregation frequently raise the security level of the output, then it is much more economical and safer to simply overclassify the data. For example, if a database consisting of largely Unclassified data returns the bulk of its results at the confidential level, then it is operationally more effective to simply treat the data as Confidential, and manually downgrade the occasional Unclassified response. We believe that real-world databases contain data at a mixture of classifications. Thus, the approach of running everything at database-high and manually downgrading severely limits the flexibility and usability of the database.

Incorporation of an explicit classification policy into our design reflects a radically different view than that taken by simple access control policies, and most secure DBMS research (see for example [16]). These systems assume that its is statically possible to bind a sensitivity level to each piece (or grouping) of data. The appropriate classification for any collection of data is then determined through a simple "high water mark" calculation over the set of sensitivity levels. Such an approach can result in either overclassification or a breach of security.

B. LDV System Architecture

LDV is hosted on the LOCK TCB. The user interacts with LDV through a request importer and a request exporter as shown in Fig. 1. Access to data as well as the metadata is controlled by LOCK. Information in the da-

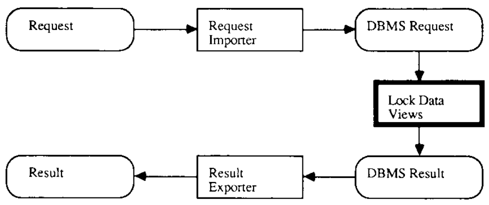

Fig. 1. LOCK to LDV interface.

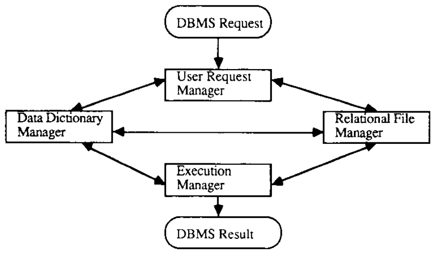

Fig. 2. LDV system data flow.

tabase as well as the metadatabase are stored in single level files, i.e., LOCK objects. LOCK ensures that these database files may be opened for read/write operations only by subjects executing at the appropriate levels, and in the appropriate database domains. LOCK enforces the operations provided that the LDV application defines properly in the LOCK DDT the allowed operations upon objects by subjects.

The LDV system architecture is illustrated in Fig. 2. It shows the major subsystems and the data flow between them. These subsystems are the Data Dictionary Manager (DDM), the User Request Manager (URM), the Relational Access Manager (RAM), and the Execution Manager (EM). Each of these subsystems is described.

DDM—this subsystem is responsible for maintaining all information about the multilevel database and acts as a server of this information to the other subsystems of LDV. That is, all of the metadata are handled by the DDM. The metadata include the database schemas, security constraints (both the discretionary security constraints and the mandatory constraints which are also referred to as classification constraints), integrity constraints, and history information.

URM—this subsystem provides an SQL interface to LDV that is consistent with the ANSI/SQL standards. It parses both SQL data definition and data manipulation requests, and translates them into an internal representation of SQL. Its function also includes discretionary access checks on views, enforcing semantic integrity constraints, and query modification. The modified request is passed to the RAM.

RAM—this subsystem takes the internal representation of a query or update request and performs optimization. Information on access paths is obtained from the DDM. The output of RAM is the execution strategy which is passed to the EM.

EM—this subsystem uses the services of two major modules; the Relational File Manager (RFM) and the Transaction Execution Manager (TEM). The EM carries out the execution strategy by making appropriate calls to the RFM and TEM. In the case of a query, it also builds the result. The RFM is responsible for managing the files and TEM is responsible for concurrency control, recovery, and integrity maintenance.

The LDV design described here assumes a single user updating environment. It allows multiple users if those users are only querying the database. Extending this design to a multiple user update environment is under current investigation.

C. LDV Pipeline Organization

The three major operations performed by LDV are query, update, and metadata management. Each of the operations is an interaction between a non-LDV subject representing a user, and the LDV subjects that manipulate the database. In our design of LDV, each of these operations is processed by an assured, enforced pipeline. Therefore, the LDV design consists of three pipelines which pass through a number of subjects in order to support encapsulation and the security and/or integrity policies. The three pipelines are 1) the Response Pipeline 2) the Update Pipeline, and 3) the Metadata Pipeline.

The Response Pipeline maps a query from the application domains to the DBMS, processes the query to produce a result relation, labels this result, and exports it to the user domain. This pipeline runs untrusted in the early stages (e.g., the SQL parser and the Query Modifier); the portion of the Pipeline which determines the classification level of the data to be released is an example of a trusted component.

The Update Pipeline allows subjects executing in special data input domain to prepare records for input to the DBMS, identify records to delete, and transforms them into a data type readable by the DBMS domain. This update pipeline also runs untrusted in the early stages; the portion which determines the data classification and where-to-write are trusted code.

The final pipeline provides the mechanisms for defining a database structure, specifying relations, views, attributes, classifications, and would normally be restricted to access by the database administrator (DBA) or the database systems security officer (DBSSO). As with the others, the Metadata Pipeline allows untrusted code in the early stages; an example of the trusted portion is that which actually stores the classification constraints.

We need to make a distinction between the trustworthiness of the code and the correctness of the code. If the trusted code is not correct, then there can be breach of security. If the untrusted code is not correct, then the worst thing that can happen is that the result is incorrect; it cannot cause a breach of security. This is why we can accept unverified design code in the untrusted portions of a pipeline. The categorization into untrusted portions of a pipeline is based on the security needs, and not on the effect on the correctness of results.

The LOCK TCB is responsible for spawning processes, for efficient context switching, and for preserving interference among processes at different levels. Even though the DBMS obtains these services from the LOCK TCB, the overall design of each pipeline must ensure that the security critical code is minimal, traceable, verifiable, and that the number of processes performing trusted write-downs is minimized. The pipelines are a collection of communicating processes each of which could be verified in isolation and all of which could be proven to communicate in ways that do not permit overt or covert disclosure of information. Since a DBMS is a large application, there are numerous possible configurations for the pipeline's processes. The choice of configuration is motivated by the goal to reduce the amount of design verification needed without compromising security. A module is security critical if an accidental or deliberate malfunction in that module could result in improper leakage of information to the outside world.

D. LOCK Interactions, Dependencies and Benefits

We have taken a design approach towards protection which is significantly different from other approaches (see for example [16] and [9]) and yet retains their best features. The approach is attractive in that it retains the simplicity and assurance associated with the access control policies, yet by extending the model to include an explicit classification policy, one can support the requirements for an MLS/DBMS. It does raise some interesting implementation problems. Among these are

—how is DBMS information isolated from application domains?

—how is information (data and associated classification rules) imported to the DBMS?

—how is information exported from the DBMS to the application domains?

Our mechanism for solving these problems is the LOCK type enforcement mechanisms. Using type enforcement, it is possible to encapsulate the DBMS code in isolated domains over which subjects can read and write only specific types of data. This allows a layered system. The important features of this design are that

—DBMS queries can be forced to pass through an importer domain which implements filters to perform consistency checks and input canonicalization

—only subjects in the DBMS domain can both read and write DBMS data; even within the DBMS we have explicit limitations.

—All responses exported from the DBMS domains to the application domains must pass through an export domain.

Our design uses type enforcement to restrict the ability of portions of the DBMS code to write and/or read certain types of data. We note that the underlying LOCK security kernel is available in all domains. This allows us to take advantage of the LOCK mechanisms as needed to enforce the underlying extensions to the basic security policy to deal specifically with the needs of that domain. This is feasible since only code within that domain must exhibit special properties required for the "local" interpretation of security; thus, the domain's security proof can be separate from that of LOCK.

LOCK enforces types in the following way. Each DBMS subject has a domain attribute. Similarly, each object is of a particular type. Access to objects of certain types is restricted to subjects in designated domains. This access information is recorded in the Domain Definition Table. The observation underlying the role of type enforcement is that the format and organization of data are security relevant. Thus, the raw application data are different from other data such as mailboxes, engineering drawings, labeled text, database data, database metadata, and database history. Domains are essentially mechanisms for encapsulating managers of different data types and the transformation between data types. This provides a way to decompose the proof of security for the system into manageable pieces and to tailor the security policy for a system in an application dependent fashion. The basic idea is that as data are transformed from one type into another, they are moving along an assured pipeline. That is, the design code in each domain of the pipeline has been verified to assure that it possesses the appropriate security relevant functionality, and the type enforcement mechanism provides assurance that data cannot be transformed from one type to another except via the pipeline.

There are three theorems necessary to guarantee an assured pipeline.

—The verified transforms are unbypassable.

—The transforms in the assured pipeline are correct.

—Data of the transformed type cannot be tampered with.

The truth of the second of these theorems must be established by a separate analysis. The formal top level specification (FTLS) for LDV has performed this analysis [20]. The first and third are corollaries of the fact that any access allowed by LOCK is consistent with the DDT. Nothing more must be verified to assure the truth of these theorems. It is only necessary to configure the DDT appropriately.

So by using the type enforcement mechanisms to create a set of assured pipelines for the DBMS, it is possible to encapsulate the DBM's as a protected subsystem on a multilevel secure TCB. This gives us a basic structure whose bottom level consists of TCB hardware and software, whose next level consists of the encapsulated DBMS subsystem, and whose highest level consists of untrusted and presumed hostile applications programs which access the DBMS. Within the DBMS subsystem there will be both security critical and noncritical subsets of programs; the detailed security analysis of an entire DBMS is clearly impractical, and the alternative of a design for a MLS/DBMS without verification of the security critical portions of the design is not possible. The verified subset of the DBMS must enforce the DBMS security policy extensions which refine the basic TCB security policy to classify data appropriately in cases where the information-value based classification of data exceeds the inherent classification level of that data, and to limit the possibility of covert channels through the DBMS.

IV. LDV Language

The DBMS must provide a language for use in describing the various schemas and for retrieving and maintaining the data in the database. This language allows users to deal with the database in abstract terms; to be concerned with what must be done and not how it must be done. Three distinct user roles are considered; ordinary users, database administrators (DBA's), and the database system security officers (DBSSO's). Ordinary users perform query insert, delete, and update operations on the data. DBA's maintain the metadata, i.e., the data that describe the database data. DBSSO's maintain the fundamental classification rules.

The LDV language consists of two parts: a data definition language (DDL) and a data manipulation language (DML). The DDL is used by the DBA and the DBSSO to describe the data and the DML is used by the ordinary users to retrieve and maintain the data. The DDL and DML for this design [19] were originally based on the American National Standards Institute (ANSI) Database Language SQL [1]. In [26] the LDV DDL and DML have been made compatible with a later version of ANSI SQL [2]. We have extended the DDL to allow for the specification of classification constraints, primary keys, and the derivation of values of one tuple from those of other tuples. The DML has been extended with a time-oriented construct that refers to points in time, and a level-oriented construct that refers to the classification levels of data.

V. Response Pipeline

This section presents the design of the Response Pipeline of the LDV system. This pipeline is the query processor. The LDV data distribution scheme, reconstructing views at a given level, handling aggregate constraints and inference control, overview of the major modules, and the security critical components are described.

A. LDV Data Distribution Scheme

The basic scheme for data distribution across LOCK files is to assign a set of files per security level. There is no replication of data across levels. The Update pipeline determines the appropriate assignment of data to files by examining the name-dependent, content-dependent, and context-dependent classification constraints. The view at any particular level is reconstructed by the MERGE operation to be described later in this section. Since partial relations that are stored at each level may have numerous null values, these nulls can be squeezed out by padding each partial tuple with a tuple descriptor. A tuple descriptor is a bitstring whose length is the order of the relation. A "1" in a position indicates that a value exists for that attribute, and a 0 indicates that the field is null. A "*D*" in the first position indicates that the tuple has been logically deleted. In addition to the tuple descriptor, a timestamp and the level of the tuple are stored. The level of the tuple is the level at which the tuple was inserted. These three fields are not displayed to the user by default; they are manipulated internally by LDV. However, the user may request the retrieval of the timestamp and level fields. The tuple descriptor always precede the tuple, followed by the timestamp, level, and values for the attributes that have "1"s in their corresponding positions in the tuple descriptor.

The data distribution scheme used by LDV is described using the sample relation EMPLOYEE illustrated in Table I. The constraints on EMPLOYEE are the following:

SSN is the key
Default level for Name is (U)
Name is (TS) where Name = ON
Default level for Address is (S)
Default level for Salary is (S)
(Name, salary) is (TS) when taken together.

One way to distribute this relation across LOCK data files is to use the method of [16] and assign one file per attribute. This is also the method used in [9]. A discussion on this method is given in [15]. Since LDV handles security constraints, additional files have to be created for each additional level incurred by the content-based constraints. This method has security advantages as the attributes are strictly separated. However, it has performance disadvantages due to the large number of files that have to be maintained. LDV does not use this scheme because of the performance disadvantage [15]. Instead the LDV distribution scheme packs as many attributes into a file as possible, but at the same time ensuring security. The following distribution schemes were studied and compared with respect to security and performance.

Method 1—Each Level in a Separate File: If one file is assigned per security level, the EMPLOYEE relation can be distributed across files as shown in Tables II-IV. Each file contains the partial relation visible at the level of that file or higher. The partial relation for the view at any given level is computed from the data stored at that level and from lower level data using the MERGE. The tuple descriptor at the beginning of each partial tuple indicates the attribute values represented by the partial tuple.

With this method of storing multiple attributes at the same level in the message file, the enforcement of context based constraints is difficult. This problem is illustrated in the following examples.

1) No Trojan Horse in the System (A Trojan Horse is malicious or hostile code): Suppose the *S*-user (Secret user) issues the request: SELECT Name, Address, which is then transformed into the following operations:

$T1 \leftarrow$ MERGE F-U, F-S over SSN
Result $\leftarrow$ PROJECT $T1$ over Name, Address.

The result is shown in Table V. The File Manager determines that nothing is being released at a higher level, and releases the data.

TABLE I

EMPLOYEE			
SSN	Name	Address	Salary
1	PD	DC	100K
2	BT	NY	110K
3	EO	LA	110K
4	ON	SF	200K

TABLE II

F-U		
1100	1	PD
1100	2	BT
1100	3	EO

TABLE III

F-S			
1011	1	DC	100K
1011	2	NY	110K
1011	3	LA	110K

TABLE IV

F-TS				
1111	4	ON	SF	200K

TABLE V

Result	
Name	Address
PD	DC
BT	NY
EO	LA

2) Trojan Horse in MERGE or PROJECT operators: Suppose the same request is posed by the *S*-user. The request is transformed into the same operations described in the earlier example. The result is shown in Table VI. The address fields have values for salaries in them because the Tojan Horse has switched those fields. The File Manager must detect the Trojan horse in the buffer Manager that switched the salary with the address, thus violating the context-based constraint that Name and Salary cannot be seen together. Another alternative is to verify all of the design of the low level buffer management and file retrieval operations. Both alternatives seem difficult. The performance of this method is promising because there is a manageable set of files. Query processing and response times are likely to improve considerably over the attribute per file method.

Method 2: By Context Upgrade in an Upgraded File: One way to avoid the costly verification or complicated File Manager design implied by the file-per-level method is to do a "by context upgrade." In this method, the "by context" constraints are taken into account when inserting the values into files. That is, an Upgrader (in the

TABLE VI

Result	
Name	Address
PD	100K
BT	110K
EO	110K

Update Pipeline) upgrades one of the attributes involved in the "by context" constraint. For the example considered here, the files shown in Tables VII–IX are created.

A disadvantage with this method is that the Secret subject cannot read salary by itself (without name) as intended by the constraints. A Downgrader would be necessary to provide the required functionality. However, it is impossible for a Trojan Horse running at Secret level to switch the salary and address fields because the Salary data are stored in a TopSecret file. As with the previous method, there is a manageable set of files. Query processing and response times are likely to improve considerably over the attribute per file method.

Method 3: By Context Upgrade in Separate File: In this method, initially there is one file per security level as in method 1. In addition, each attribute involved in a "by context" constraint is placed in a separate file. For the example considered the files created are shown in tables X–XIII.

The advantage of this method is described with examples.

1) No Trojan Horse in the system: Suppose a *S*-user makes the same request described in the example of method 1. The request is transformed into the following operations:

*T*1 ← MERGE F-U, F1-S over SSN
Result ← PROJECT *T*1 over Name, Address

The result is shown in Table XIV. Here, the File Manager opens the files F-U and F1-S. It then gets the context constraints relevant to the files opened. In this case, there are none. So the response is assigned the level (S).

2) Trojan Horse in the File Manager: Suppose an *S*-user makes the same request. The request is transformed into the same operations as in case 1. The result is shown in Table XV. Here, the Trojan Horse has changed the MERGE so that it opens F2-S instead of F1-*S*. The File Manager gets the relevant context constraints based on the files opened. The constraint obtained is: *Name, Salary is TS located in F-U, F2-S.*

The File Manager upgrades the result, thus frustrating the Trojan Horse attempt. The level of the response is (TS).

As with methods 1 and 2, there is a manageable set of files. Method 3 is basically Method 1 with attributes relevant to the context constraints isolated. The other attributes can still be grouped. This is the method used in LDV. It has the security advantages of the attribute per level method and the performance advantages of the methods 1 and 2 discussed here.

TABLE VII

F-U		
1100	1	PD
1100	2	BT
1100	3	EO

TABLE VIII

F-S		
1010	1	DC
1010	2	NY
1010	3	LA

TABLE IX

F-TS				
1001	1	100K		
1001	2	110K		
1001	3	110K		
1111	4	ON	SF	200K

TABLE X

F-U		
1100	1	PD
1100	2	BT
1100	3	EO

TABLE XI

F1-S		
1010	1	DC
1010	2	NY
1010	3	LA

TABLE XII

F2-S		
1001	1	100K
1001	2	110K
1001	3	110K

TABLE XIII

F-TS				
1111	4	ON	SF	200K

TABLE XIV

Result	
Name	Address
PD	DC
BT	NY
EO	LA

B. Reconstruction of a View for a Given Level

The query processor reconstructs a partial relation representing a given user view from the data distributed across files. There is one such partial relation corresponding to each base relation in the user's query. The remaining query processing (for example join) is performed using these partial relations. In order to reconstruct a partial relation at a particular level, the query processor must take into account context-dependent classification constraints, and merge tuples from different files with the same primary key. For the second step, an operator called the MERGE is presented. This operator works with a knowledge of the properties of the tuples in the different partitions of a relation. These properties are discussed first.

TABLE XV

Result	
Name	Address
PD	100K
BT	110K
EO	110K

Characterizing Multilevel Tuples of a 3NF Relation: There is a relationship between tuples in a multilevel view and the tuples in the partial relations that are distributed in single level files. The tuple in a multilevel view is formed by merging disjoint sets of attributes from tuples of partial relations in lower level files. In this case, we can say that the tuple at a higher level subsumes partial tuples from lower levels. Another way to look at it is that a tuple that was previously partitioned across levels is being reconstructed by concatenating attribute values that are at or below the level at which the reconstruction is being done. The partial tuples involved in this reconstruction have the following property.

$$\text{Key} = \text{Key}_{l1} = \text{Key}_{l2} = \cdots = \text{Key}_{ln} \quad (1)$$

where Key_h is the key of the higher level tuple being constructed and Key_{li} $(1 \leq i \leq n)$ is the key of a tuple (from the lower level partial relations) that is being used in the reconstruction. The keys are the same because one is simply reconstructing the same tuple whose values were dispersed during the distribution. The reconstruction of a multilevel 3NF relation is simply to sort each file on the primary key and then MERGE as in sort-merge. Since this is a common operation, one way to optimize it is to maintain multilevel clustered indexes on the primary key ([8], [30]).

The remaining group of tuples in a multilevel relation are those that are at a single level. For example, a *U*-user could enter a tuple that is not distributed across levels. After subsuming tuples that are distributed across levels, the MERGE then adds on the single level tuples to the resulting relation. These single level tuples can be considered special cases of the subsumed ones, except that they only subsume themselves. They can be characterized in terms of their keys as:

$$\text{Key}_{\text{single}} \neq \text{Key}_{\text{other}} \quad (2)$$

where $\text{Key}_{\text{single}}$ is the key of the single level tuple and $\text{Key}_{\text{other}}$ is any key in any file in the system.

The Polyinstantiation Problem for 3NF Relations: The major kind of integrity constraint on a 3NF relation is the PRIMARY KEY CONSTRAINT which stipulates that a set of attributes must uniquely identify each row in the relation. For example, if SSN must uniquely identify the rows in the EMPLOYEE relation, then the next version of the same relation in which a user has completed an update and inserted another tuple with SSN = 1, has violated the primary key constraint as shown in Table XVI.

In a multilevel environment, it is possible for users at different security levels to have different views of the same tuple. That is, there could be two different tuples with the same primary key at different security levels. This is known as polyinstantiation. When polyinstantiation is present, the primary key constraint is violated. LDV provides mechanisms for stipulating and enforcing primary key constraints. The assumption is that the relations are normalized up to 3NF. Suppose a TS-user adds the new tuple (SSN = 1, name = PS, Address = BO, Salary = 100 K), which violates the primary key constraint. The Update Pipeline stores the new tuple along with its timestamp and level. Following a TS-user request to retrieve all of the EMPLOYEE relation, LDV's Relational File Manager reconstructs EMPLOYEE for this particular user as shown in Table XVII.

The user has the ability to choose which tuples are displayed based on the timestamp and level. The default is to display all tuples. Suppose the TS-user's query was the following:

SELECT* FROM EMPLOYEE WHERE Level(*) = *S*

The result produced is shown in Table XVIII.

Option to Derive a Tuple: In addition to adding constructs in the retrieval language to allow user to specify time and to add a classification factor to a selection condition, the data definition language supports the derivation of data. As an example, the definitions of Address and Salary could contain the following derivation factor.

DERIVE LEVEL TS FROM LEVEL *S*.

The derivation factor says that, for this attribute, values are derived from the *S*-level tuple having the same primary key, because the primary key is being enforced (in the Update Pipeline) at each level. Even with polyinstantiation, there can be one tuple with each primary key value at the *S*-level.

Reconstruction Process: The multilevel 3NF relation is constructed as follows.

1) Open the files containing data required for computation of the result. Take into account context-dependent classification constraints and the history of opened files at this level. Using this history, context-dependent classification constraints are considered not only for current query, but for previous queries also.

2) Merge the files that were opened by performing the following steps:

TABLE XVI

EMPLOYEE			
SSN	Name	Address	Salary
1	PD	DC	100K
2	BT	NY	110K
3	EO	LA	110K
4	ON	SF	200K
1	PS	BO	100K

TABLE XVII

EMPLOYEE			
SSN	Name	Address	Salary
1	PD	DC	100K
2	BT	NY	110K
3	EO	LA	110K
4	ON	SF	200K
1	PS	BO	100K

TABLE XVIII

EMPLOYEE			
SSN	Name	Address	Salary
1	PD	DC	100K
2	BT	NY	110K
3	EO	LA	110K

2a) SUBSUME all the tuples that are partitioned at or below the level at which the reconstruction is being done.

2b) DERIVE all those tuples that have derived attributes.

2c) Tag on all the single level tuples

After the execution of the MERGE, the RFM then filters the remaining spurious tuples by enforcing the primary key constraint in the manner requested by the user using the time and level constructs. If the user has not requested any other filtering outside of what the MERGE has already performed, then the result of the MERGE is passed on.

The user may request that timestamps and levels be included in the result. This enables later updates and expedites manual declassification of part of the output. In addition, we distinguish between a "blank" and a "null" field. A "blank" can be used to prevent lower level data from being merged into higher level field, whereas a "null" always results in a merge.

The MERGE operation is illustrated using the following example. In this example, the Address and Salary at level TS are derived from level *S* if not specified by the user. The tuple descriptors, timestamps, and levels are shown and are carried along in each step of the MERGE for each attribute. The abbreviations used for tuple descriptors, timestamp, level, SSN, Name, Address and Salary are TD, *T*, *L*, *S*, *N*, *A*, and SA, respectively. Suppose that we need to build a TS view of the EMPLOYEE relation by merging the following files shown in Tables XIX–XXII. Note that F-TS contains a pointer (repre-

TABLE XIX

F-U				
TD	T	L	S	N
1100	1	U	1	PD
1100	2	U	2	BT
1100	3	U	3	EO

TABLE XX

F1-S				
TD	T	L	S	A
1010	1	S	1	DC
1010	2	S	2	NY
1010	3	S	3	LA

TABLE XXI

F2-S				
TD	T	L	S	SA
1001	1	S	1	100K
1001	2	S	2	110K
1001	3	S	3	110K

TABLE XXII

F-TS						
TD	T	L	S	N	A	SA
1111	4	TS	4	ON	SF	200K
1111	5	TS	1	PS	BO	F2-S(1)

TABLE XXIII

SUBSUME Result												
TD	T	L	S	T	L	N	T	L	A	T	L	SA
1111	1	S	1	1	U	PD	1	S	DC	1	S	100K
1111	2	S	2	2	U	BT	2	S	NY	2	S	110K
1111	3	S	3	3	U	EO	3	S	LA	3	S	110K

TABLE XXIV

DERIVE Result												
TD	T	L	S	T	L	N	T	L	A	T	L	SA
1111	5	TS	1	5	TS	PS	1	S	BO	1	S	100K

sented by F2-S(1)) to the appropriate file containing the values specified in the DERIVE construct. These are initialized in the Update Pipeline when the tuple is inserted.

The first step in merging these four files is to SUBSUME all of the tuples that are partitioned into files at or below TS. The result of the SUBSUME is shown in Table XXIII. The second step in merging these four files is to derive all of the tuples that have derived attributes. The result of the derive is shown in Table XXIV. The third step is to combine the results of the first two steps with the single level tuples producing the result shown in Table XXV. This result can now be passed on for further query processing or for display to the user.

TABLE XXV

Result												
TD	T	L	S	T	L	N	T	L	A	T	L	SA
1111	1	S	1	1	U	PD	1	S	DC	1	S	100K
1111	2	S	2	2	U	BT	2	S	NY	2	S	110K
1111	3	S	3	3	U	EO	3	S	LA	3	S	110K
1111	5	TS	1	5	TS	PS	1	S	BO	1	S	100K
1111	4	TS	1	4	TS	ON	4	TS	SF	4	TS	200K

C. Enforcement of Constraints on Aggregate and General Inference Control

Aggregation constraints are enforced by the EXPORTER after the result of the query has been built. Aggregation constraints are defined using the extended SQL, i.e., using the aggregate functions COUNT, AVG, MAX, MIN, SUM, and COUNT. The EXPORTER examines the aggregation constraints relevant to the query and may reclassify the result. Initially, the EXPORTER is an application specific downgrader. We believe that this is the correct place for general inference control as well.

As an alternative to the EXPORTER, we considered using the LDV data distribution scheme to partition data used in the computation of aggregates, and to enforce the aggregation constraints in the same way that context-dependent constraints were enforced by ensuring that certain files are never accessed together.

Consider the constraint: "COUNT (EMPLOYEE) > 10 is SECRET". One way to enforce this constraint is to split all UNCLASSIFIED EMPLOYEE files so that no single file ever has more than 10 tuples, and use the same data structure used for the context-based constraints to make sure that some files are never accessed together. The problem is that there are many possible aggregation constraints and many possible combinations of such constraints that could be defined per relation. This method could potentially lead to one partial tuple per file, or to constraints that would make only one file accessible to the first user that logs on, and none of anyone else. As a result, we used the first alternative, i.e., the EXPORTER.

D. Overview of the Major Modules

The Response Pipeline is illustrated in Fig. 3. The major modules are the URM, the RAM, the EM and the RFM. Each of these modules are described here.

URM: This module provides an SQL interface to LDV for queries that is consistent with the ANSI SQL standard. It performs discretionary access control on views as defined by the ANSI standard (i.e., not LOCK's discretionary control policy). Modification of queries on views to form queries on base relations, and modification of queries using security constraints to ensure that the response is classified so that it is observable to the user [14]. All information needed for translation is provided by the DDM (Data Dictionary Manager).

RAM: This module takes the internal representation of

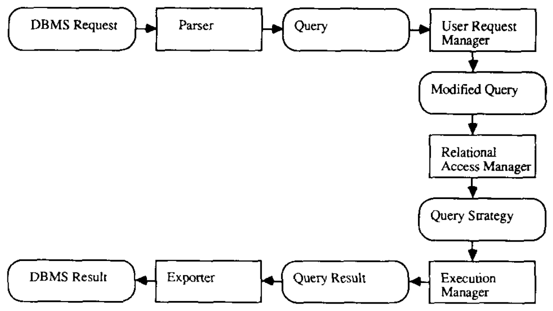

Fig. 3. Response pipeline.

the query built by the URM, produces a relational algebra representation of the query, and builds an optimal execution strategy. The information on access paths is obtained from the DDM.

EM: This module takes the execution strategy produced by the RAM and builds the result of the query by executing each operation in the relational algebra representation. It has a procedure called Build-Result which uses the service of RFM to build the response.

RFM: This module is used in the Response Pipeline by the EM to build a view of a single relation at a particular level as described earlier. The RFM is composed of the Relation Manager, the Record Manager, the Index Manager, and the File Manager. The Relation Manager manages the relations by using the services of the data storage and retrieval managers. The Record Manager manages collections of records stored as tuples by the Relation Manager. The Record Manager uses the services of the File Manager and Index Manager to store and retrieve records. The Index Manager is used by the Relation Manager to store keys in an index and manages them in sorted order. The Index Manager uses the services of the File Manager to store and retrieve keys. The File Manager manipulates the data management files.

E. Security Critical Modules

The content-based constraints are enforced by LOCK due to the distribution scheme that partitions data into appropriately labeled files. For context-based constraints, a data structure is maintained to indicate what files may or may not be accessed together (this data structure is created and maintained by transforming context-based constraints expressed in SQL into an equivalent internal form expressed as constraints on sets of files). Whenever a file is to be accessed, the File Manager must first read the history of previous accesses to determine if the file request can be granted. If so, the File Manager appends the new file request to the history file and then opens the requested file. Because it maintains this history file and restricts file access requests using context-based constraints, the File Manager is security critical. In addition, it is security critical that the File manager opens the right files and not switch file names (otherwise a file involved in a context-based constraint whose violation cannot be detected may be opened).

A module can be security critical for any one of the following reasons.

1) It controls the order-of-processing security critical modules.

2) It processes security critical data (potential misprocessing).

3) It transmits security critical data from one place to another (potential substitution).

4) It acts as a guard for security critical data (potential misauthorized deposit or withdrawal).

5) It would provide a covert channel (potential signaling on the data).

The security critical components of the Response Pipeline are portions of the EM and the EXPORTER.

VI. Update Pipeline

This section presents an overview of the Update Pipeline design. Processing an insert request, delete request, and a modify request, an overview of the major modules, and the security critical components are described.

A. Insert Request

An insert request must be processed so that data are inserted into the correct file at the correct level based upon the classification constraints and the inserting subject's level. Upgrades are determined by the values of the elements of the tuple to be inserted.

The insert request is first imported into the DBMS domain. The imported request is then sent to an Upgrader which computes the level of the insert operation as follows [23].

1) The level of each attribute specified in the insert request is set to the corresponding default level.

2) The relevant constraints visible at the processing subject's level are retrieved. Each relevant constraint satisfies the following condition:

It classifies an attribute which is specified in the insert request at the level which dominates the processing subject's level, and it has not been examined during a previous iteration of this algorithm.

3) For each relevant constraint, a new level is computed for each attribute that is classified by the constraint as follows:

new level = least upper bound (old level, level specified in the constraint).

4) Compute the least upper bound of the levels of the processing subject and all the attributes specified in the insert request.

5) If the new level dominates the level of the processing subject, then create a new subject at this new level and pass the parameters associated with the current processing subject to the new subject. Delete the current processing subject. The new subject becomes the current processing subject. Go back to step 2.

6) Otherwise, if the new level is equal to the current processing subject's level, then continue with the remain-

ing processing of the insert operation, i.e., this is the level of the insert operation.

We illustrate this algorithm with a simple example. Let $R(A1, A2, A3)$ be a relation with the following constraints:

$C1$: If $A2 = 5$, then $A1$ is TS
$C2$: If $A3 =$ ttt then $A2$ is S
$C3$: $C1$ is S
$C4$: $C2$ is U
$C5$: Default level of $A1$, $A2$, $A3$ is U.

A U (Unclassified) subject requests to insert (alpha, 5, ttt) into R. Initially the processing subject's level is U and the default levels of all three attributes are U. During the first pass of the Upgrader, the relevant constraint is $C2$. The level of the attribute $A2$ is computed to be S. Then the new level is set to the least upper bound of the levels of $A1$, $A2$, $A3$, and the processing subject's level. This new level is S. A new processing subject is created at the Secret level. During the second pass, the relevant constraint is $C1$. The level of $A1$ is computed to be TS. The new level is the least upper bound of the levels $A1$, $A2$, $A3$, and the processing subject's level. This new level is TS. A new processing subject is created at the TS level. During the third pass, no relevant constraints are retrieved. The levels of the attributes remain the same. The new level is computed to be TS. This new level is the same as the processing subject's level. Therefore, the insertion is performed at the TS level.

After the level of the insertion is computed, a view of the relation specified in the insert request is built using the MERGE operation of the Response Pipeline. Once the view is built, the request may be modified if necessary as follows:

If the primary key value specified in the request already exists and the tuple is visible at the level of the insert operation and not below this level, then the request is rejected as it is a duplicate tuple with the same primary key. If it is not a duplicate tuple at the level of the insert operation, then the tuple is inserted with a new timestamp and the level of insertion into a file at the level of the insert operation.

The modified request is passed to the RAM for optimization, and the EM for execution. RAM translates the request into requests on files. In the example considered here, the request is translated into operations on a TS file, say, $F1$, as follows:

OPEN $F1$
INSERT (alpha, 5, ttt) INTO $F1$
CLOSE $F1$.

Information about the file $F1$ is retrieved from the DDM. This is because the data dictionary includes the association between the file $F1$ and the relation R.

We illustrate the insert operation with some examples.

Consider the relation $R(A1, A2, A3)$ with the following constraints:

TABLE XXVI

F-S					
111	00	S	alpha	17	xxx
110	01	S	beta	34	
111	02	S	delta	20	uuu

TABLE XXVII

F-TS					
101	01	TS	beta	www	
111	03	TS	gamma	5	yyy

TABLE XXVIII

V-S									
TD	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
110	01	S	beta	01	S	34			
111	02	S	delta	02	S	20	02	S	uuu

TABLE XXIX

V-TS									
TD	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	01	TS	beta	01	S	34	01	TS	www
111	02	S	delta	02	S	20	02	S	uuu
111	03	TS	gamma	03	TS	5	03	TS	yyy

$A1$ is primary key
$A1$ is TS if $A2 = 5$
$A2$ is TS if $A2 = 5$
$A3$ is TS if $A3 =$ www or yyy
default value of $A1$, $A2$, $A3$ is S
level of the constraints is U.

The relation R is stored in a S file F-S and a TS file F-TS as shown in Tables XXVI and XXVII. Recall that the first field is the tuple descriptor, the second field is the timestamp, and the third field is the level. The views at levels S and TS are V-S and V-TS, respectively. These views are shown in Tables XXVIII and XXIX. They are computed using the MERGE operation in the Response Pipeline. Note that the timestamp and level precede each attribute.

In the examples to be given here, it is assumed that the level of the insert operation has already been computed.

Example 1: Suppose that a S subject requests to insert (gamma, 22, zzz).

This example illustrates the case where a subject attempts to insert new data where data already exist with the same primary key at a higher level. The solution is to insert the tuple at level S with a timestamp and level. The primary key and level uniquely identify the tuple. After the insertion, the file F-TS does not change. F-S, V-S, and V-TS are changed as shown in Tables XXX–XXXII.

Example 2: Let F-S, F-TS, V-S, and V-TS contain the values at the end of Example 1.

TABLE XXX

F-S					
111	00	S	alpha	17	xxx
D110	01	S	beta	34	
111	02	S	delta	20	uuu
111	04	S	gamma	22	zzz

TABLE XXXI

V-S									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
110	01	S	beta	01	S	34			
111	02	S	delta	02	S	20	02	S	uuu
111	04	S	gamma	04	S	22	04	S	zzz

TABLE XXXII

V-TS									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	01	TS	beta	01	S	34	01	TS	www
111	02	S	delta	02	S	20	02	S	uuu
111	03	TS	gamma	03	TS	5	03	TS	yyy
111	04	S	gamma	04	S	22	04	S	zzz

TABLE XXXIII

F-TS					
101	01	TS	beta	www	
111	03	TS	gamma	5	yyy
111	05	TS	alpha	18	aaa

TABLE XXXIV

V-TS									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	01	TS	beta	01	S	34	01	TS	www
111	02	S	delta	02	S	20	02	S	uuu
111	03	TS	gamma	03	TS	5	03	TS	yyy
111	04	S	gamma	04	S	22	04	S	zzz
111	05	TS	alpha	05	TS	18	05	TS	aaa

TABLE XXXV

F-TS					
101	01	TS	beta	www	
111	03	TS	gamma	5	yyy
111	05	TS	alpha	18	aaa
111	06	TS	pi	10	bbb

TABLE XXXVI

V-TS									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	01	TS	beta	01	S	34	01	TS	www
111	02	S	delta	02	S	20	02	S	uuu
111	03	TS	gamma	03	TS	5	03	TS	yyy
111	04	S	gamma	04	S	22	04	S	zzz
111	05	TS	alpha	05	TS	18	05	TS	aaa
111	06	TS	pi	06	TS	10	06	TS	bbb

Suppose that a TS subject requests to insert (alpha, 18, aaa).

This example illustrates the case where there is a tuple at the lower level with the same primary key. The solution is to insert the tuple at the higher level with a new time-stamp and level. After the insertion only F-TS and V-TS are changed. The new values are shown in Tables XXXIII and XXXIV.

Example 3: Let F-S, F-TS, V-S, and V-TS contain the values at the end of Example 2.

Suppose that a TS subject requests to insert (pi, 10, bbb).

This example illustrates the case where a subject attempts to build a tuple and no tuple exists in the database with the same primary key. The solution is to insert the tuple into F-TS. After the insertion, only F-TS and V-TS change. The new values are shown in Tables XXXV and XXXVI.

Example 4: Let F-S, F-TS, V-S, and V-TS contain the values at the end of Example 3.

Suppose that a TS subject requests to insert (pi, 10, kkk).

This example illustrates the case where a subject attempts to insert a tuple and there is already a tuple with the same primary key at the same level. The solution is to reject the insert request.

B. Delete Request

Processing of a delete request is less complex than that of an insert request. In this case, it is not necessary to compute the level of the delete operation as it is assumed to be that of the processing subject. This is because the *-property enforced by LOCK prevents higher level subjects from deleting information from lower level files. Therefore, upgrading the level of the delete operation does not make sense. A delete request is first imported. Then a request is made to the Response Pipeline to build a view of the relation specified in the delete request at the level of the processing subject. The delete request is modified according to the view just built as follows.

1) For each tuple being deleted, if any part of the tuple is visible at the lower level, then the delete request is rejected. This is because a higher level subject cannot write into a lower level file.

2) If the subject wants to delete the portion of the tuple visible at its level, then the values corresponding to this portion are changed to NULL.

3) If no part of the tuple to be deleted is visible at a lower level, then the tuple is marked as deleted in the file at the level of the delete operation. The tuple is not removed from the file immediately because it may be required by a higher level subject in reconstructing the higher level view using MERGE. An expunge daemon periodically reviews the files and remove the tuples that

are marked as deleted. Before removing the tuples, the daemon inserts them into the appropriate higher level files. We expect the expunge daemon to be a set of subjects running at various levels under the control of the DBSSO (database systems security officer). They would look at tuples logically deleted over some period and do the physical deletion.

Example 5: Let F-S, F-TS, V-S, and V-TS contain the values at the end of Example 4

Suppose that a *S* subject requests to delete the tuple where A1 = beta.

This example illustrates the case where a lower level subject deletes a tuple that is used in building a view at a higher level. The solution is not to remove the tuple but to mark it as deleted. After the delete operation, no changes are made to F-TS and V-TS. F-S and V-S are changed as shown in Tables XXXVII and XXXVIII.

TABLE XXXVII

F-S					
111	00	S	alpha	17	xxx
110	01	S	beta	34	
111	02	S	delta	20	uuu
111	04	S	gamma	22	zzz

TABLE XXXVIII

V-S									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	02	S	delta	02	S	20	02	S	uuu
111	04	S	gamma	04	S	22	04	S	zzz

C. Modify Request

The modify request is treated as a delete request followed by an insert request. Therefore, the details are not described here. We illustrate the modify request with an example.

Example 6: Let F-S, F-TS, V-S, and V-TS contain the values at the end of Example 5.

Suppose that a TS subject requests to modify $A2 = 81$ where $A1 =$ delta.

This example illustrates the case where a subject attempts to modify an element with a lower access class. The solution is to insert the tuple with a different timestamp and level at the higher level. (Note that the lower level information cannot be deleted due to the *-property.) F-S and V-S do not change as a result of the modify operation. F-TS and V-TS are changed as shown in Tables XXXIX and XL.

TABLE XXXIX

F-TS					
101	01	TS	beta	www	
111	03	TS	gamma	5	yyy
111	05	TS	alpha	18	aaa
111	06	TS	pi	10	bbb
111	07	TS	delta	81	uuu

TABLE XL

V-TS									
ID	T	L	A1	T	L	A2	T	L	A3
111	00	S	alpha	00	S	17	00	S	xxx
111	01	TS	beta	01	S	34	01	TS	www
111	02	S	delta	02	S	20	02	S	uuu
111	03	TS	gamma	03	TS	5	03	TS	yyy
111	04	S	gamma	04	S	22	04	S	zzz
111	05	TS	alpha	05	TS	18	05	TS	aaa
111	06	TS	pi	06	TS	10	06	TS	bbb
111	07	TS	delta	07	TS	81	07	TS	uuu

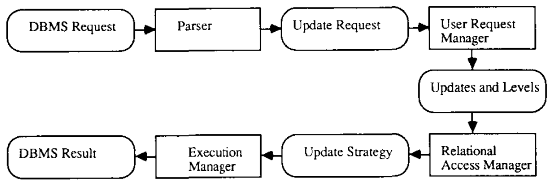

Fig. 4. Update pipeline.

D. Overview of the Major Modules

The major modules in the update Pipeline are the URM, RAM, and the EM. The relationship between these modules is shown in Fig. 4. Each of these modules is described here.

URM: This module provides an SQL interface to LDV for updates that is consistent with the ANSI standard. It performs discretionary access control on views as defined by the ANSI standard (i.e., not LOCK's discretionary access control policy), modification of the updates on views to form updates on base relations, integrity checking, and classification constraint enforcement. All information needed for the translation is provided by the DDM.

Among the functions of URM, of particular interest are update security modification and insert level calculation as these are peculiar to multilevel systems. The update security modification process modifies the update request using the classification constraints. For an insert request it computes the level of the insert using the Upgrader, builds the view of the relation being updated using MERGE, and checks for a tuple with the same primary key visible at the level of the insert and not below (it rejects the insert if it finds one). The Upgrader determines the level of an insert using classification constraints. If a predicate of a constraint evaluates to TRUE, it is used to assign a new level, otherwise, the constraint is ignored. The output is the modified insert request and its level.

For a delete request, it builds a view of the relation being updated using MERGE, builds a list of tuple identifiers (timestamp, level, primary key) being deleted by eliminating those that are visible at a lower level, and builds a delete request to delete those tuples. The output is the delete request and its level. For a modify request, it does the delete processing followed by the insert processing. The output is a delete request and its level, and an insert request and its level.

RAM: This module takes the internal representation of the update built by the URM, and builds an optimal execution strategy. The information on access paths required by the optimization process is obtained from the DDM.

EM: This module takes the execution strategy produced by the RAM and executes each operation in the strategy using the services of the Relational File Manager (RFM). The RFM is composed of the Relation Manager, the Record Manager, the Index Manager, and the File Manager. The RFM carries out the requests issued to it by the RAM by making use of the services of its component modules.

E. Security Critical Modules

As described earlier, in LDV, we restrict security critical code to a subset of the modules. Because of the organization of LDV, subjects performing designated controlled roles, and objects touchable only by certain role-players, only those modules executed by subjects that compute security critical information (such as the level of an insert) or that touch security-critical data (such as file containing the database data) are security critical. Therefore, in the Update Pipeline, the only security critical component is the Upgrader, when it computes the level of the insert. This is because once the level is computed, the remaining processing will continue at this level and LOCK will provide the necessary protection.

VII. Metadata Pipeline

This section describes the Metadata Pipeline. The major design issues, an overview of the major modules, and the security critical modules are described.

A. Design Issues

The Metadata Pipeline is used to create, delete, and maintain metadata. The major component of the Metadata Pipeline is the DDM that maintains all metadata, and acts as a server of the metadata to the Response and Update Pipelines. The metadata include information about all of the schemas (conceptual, external, and internal), relations, attributes, classification constraints, privileges, semantic integrity constraints, views, indexes, and database files. All of the metadata are stored in LOCK files. The only security critical functions of the Metadata Pipeline are those concerned with the classification constraints, i.e., it is security critical that the classification constraints be created, deleted, and maintained correctly. The incorrect modification of other metadata would not result in any security violation. Therefore, the design issues we addressed were the completeness and consistency of classification constraints and the translation of the classification constraints on relations into classification constraints on files. We discuss these issues here.

Completeness and Consistency of Classification Constraints: A set of classification constraints is complete if every piece of data is assigned a classification level via the classification constraints. We enforce completeness by ensuring that every piece of data has a default classification level.

Checking for consistency of a set of constraints was one of the more difficult tasks in the design. Our primary objective here is to ensure that the security constraints are defined in such a way that there is no security violation. Towards this direction, we have developed a set of rules that a set of classification constraints must satisfy. These rules include the following [24]:

Rule 1: The level of the value of the primary key in any tuple must be dominated by the levels of the values of all the other elements in the tuple.

Rule 2: The minimum level of any database entity must dominate the level of the existence of the entity.

Rule 3: The level of the existence of a database entity must be dominated by the level of any metadata that refers to that entity.

Rule 4: The classification level specified in the classification constraint must dominate the default levels of all the attributes which are included in the classification.

Rule 5: The level assigned to each classification constraint must be dominated by the level of at least one attribute referenced in the classification constraint.

Rule 6: The level assigned to a foreign key attribute must dominate the level assigned to the corresponding primary key attribute.

We justify some of the rules stated here. For example, consider rule 1. If the primary key of a tuple is not visible to a user, this user will not be able to uniquely identify a tuple. In the case of rule 2, if the level of the statement "the salary is 20 K" is Secret and the level of the statement "there exists an attribute salary" is TopSecret, then by getting the information "the salary is 20 K", a Secret user has acquired TopSecret information.

Ensuring the consistency of the classification constraints does not have to be done automatically by the system; it can be done off-line by the DBSSO. An automatic system for enforcing these rules would ensure that they are satisfied whenever a classification constraint is updated. Current logic programming systems such as those described in [6] and [22] can be used to develop such a tool.

Constraint Storage and Translation: The classification constraints as specified by the DBSSO at the conceptual level have to be stored with the metadata and also translated into classification constraints on the physical files. We use relations (system tables) to store both the constraints on relations and the constraints on the physical files. When a new classification constraint is inserted, deleted, or modified, the appropriate system tables which store the classification constraints at the conceptual level are first updated. If the classification constraints under consideration cause the security levels of the attributes or parts of the attributes to change, new files for the attributes are created at the new levels and the system tables are updated to reflect this change. The classification constraints may be specified when a relation is created or at a later time. The informal algorithm which translates the classification constraints specified at the conceptual level to classification constraints on files, and creates the appropriate files, is given below.

Assumption: For each attribute there is a default level. Therefore, a file at this default level exists which stores the attribute. The application of additional simple, content-based or context-based classification constraints can only result in upgrades.

Translation of Simple or Content-Based Classification Constraints: If the level specified in the classification constraint dominates the current level of the attribute specified in the constraint, then create a new file for this attribute at the level specified in the constraint. Translate the classification constraint to one on files, and update the appropriate system tables (note that the current level could be the default level or a level assigned to this attribute by another constraint).

Translation of Context-Based Classification Constraints: If the level specified in the classification constraint dominates the current levels of the attributes, then for each attribute A classified by the classification constraint do the following. For each file which stores A, if this file stores other attributes as well, then remove attribute A from this file and create a new file at the same level which will store A in the future. Translate the classification constraint to one on files, and update the appropriate system tables (note that the current levels of the attributes could be their default levels or the levels assigned by other constraints).

Suppose a context constraint which classifies the names and salaries of employees taken together at the Secret level is enforced. Table XLI shows how this constraint is stored. The condition field is "null" because there is no condition attached to the constraint (for example, if the names and salaries taken together is Secret if the employee is not PD, then the condition field will specify "name ≠ PD"). Assuming that the names of employees are stored in file $F1$ and the salaries of employees are stored in file $F2$, Table XLII illustrates how this constraint is stored as a constraint on files.

B. Overview of the Major Modules

The major modules in the Metadata Pipeline are the URM and the DDM. The relationship between these modules is shown in Fig. 5. Each of these modules is described here.

URM: Creation, deletion, and modification of metadata are specified in the SQL DDL. The DDL provides a create statement, drop statement, alter statement, integrity assertion statement, index definition statement, privilege definition statement, privilege revocation statement, and security assertion statement. The URM processes these statements. It invokes the DDM to update the metadata.

DDM: The DDM has three major components:

1) the dictionary which consists of the conceptual, external, and internal schemas,

TABLE XLI

CONTEXT CONSTRAINTS				
constraint name	*table name*	*column name*	*condition*	*classification level*
C1	EMPLOYEE	Name	NULL	Secret
C1	EMPLOYEE	Salary	NULL	Secret

TABLE XLII

CONTEXT CONSTRAINTS ON FILES			
constraint name	file name	condition	classification level
C1	F1	NULL	Secret
C1	F2	NULL	Secret

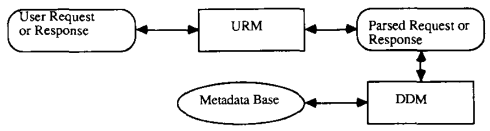

Fig. 5. Metadata pipeline.

2) the dictionary schema which is a description of the generic structure of the dictionary, and

3) the dictionary processing system which is the set of programs that interact with the dictionary and the dictionary schema to provide the functionality of the DDM.

The relational model is used to represent the dictionary component of the DDM. The dictionary consists of a set of tables which describe the conceptual, external, and internal schemas of the database. They are stored at the security level of the DBA or DBSSO who inserted the metadata. The conceptual schema consists of the description of the relations, attributes, classification constraints, privileges, and semantic integrity constraints. The external schemas consist of the description of views and view privileges. The internal schema describes the physical implementation of each relation. It includes a description of the files that correspond to each relation, the indexes defined by the user, and the classification constraints on files.

The dictionary processing system consists of modules that manipulate the dictionary tables. These modules can be grouped into two categories: 1) The modules used by the response and update pipelines to provide information of the metadata, and 2) the modules used by the metadata pipeline to update the metadata. Each SQL DDL statement results in updates to the dictionary table. These updates are processed at the level of the DBA or the DBSSO performing the update.

C. Security Critical Modules

The security critical components of the Metadata Pipeline are the components which are responsible for the maintenance of the classification constraints. The classification constraints are parsed by the URM and passed to the DDM. DDM first checks for the consistency of the classification constraints (this operation can be performed offline by the DBSSO). The classification constraint is then inserted into the appropriate dictionary tables at the level of the DBSSO inserting the classification constraint.

These classification constraints are later used by the Update Pipeline to determine the level at which the update request has to be processed. After the classification constraints are stored in the appropriate tables, they have to be translated into classification constraints on files. This is because the Response Pipeline needs these classification constraints on files to determine which files to open.

The security critical modules are the process assertion statement and process drop statement. The functions of the process assertion statement module include the insertion of security constraints while the functions of the process drop statement module include the deletion of security constraints.

VIII. Operating System Support for LDV

The operating system issues that have been identified are concerned with process management, file management, buffer management, and consistency control [25]. Process management issues affect all aspects of the LDV system design, file management issues affect the design of the Relational File Manager, and buffer management and consistency control issues affect the design of the Transaction Execution Manager. We describe the issues of process management, file management, and buffer management. Consistency control issues are only applicable to a multiple user updating environment. These issues are currently being investigated.

A. Process Management

There are two process management issues that arise in the system design of LDV: process isolation and process control.

Process Isolation: LDV is a set of processes running under the control of LOCK. Assured pipelines (Response, Update, and Metadata Pipelines) are used to provide process isolation. These pipelines consist of a series of subjects executing in special domains on special types of objects that ensure tamper-proof maintenance of sensitive data. Each subject and object have a associated set of characteristics.

The characteristics of a subject are:

Subject_Name = The name we use to identify this subject
Purpose = What this subject does
Role = The generalized role that this subject plays
Domain = The LOCK domain of objects this subject will read
Read_Datatype = Datatypes of objects this subject will read
Write_Datatype = Datatypes of objects this subject will write
Update_Datatype = datatypes of objects this subject will read and write
Parameter_Datatype = Datatypes of parameter object, not data-object.

The characteristics of an object are:

Object_Name = The namer we use to identify this object
Purpose = Why we have this object
Datatype = the LOCK datatype of this object
Contents = What it is that this object contains
Read_Domain = Domains of subjects that can read this object
Write_Domain = Domains of subjects that can write this object
Update_Domain = Domains of subjects that can update this object.

A sample subject and object belonging to the Response Pipeline are given here:

Subject_Name = Importer
Purpose = "Act as a first-gate into the database domains"
Role = Data_Importer
Domain = Importer
Read_Datatype = Vanilla
Write_DataType = DBMS_Request
Update_Datatype =
Parameter_Datatype =

Object_Name = DBMS_Request
Purpose = "Hold the input as typed by the user"
Datatype = DBMS_Request
Contents = "An SQL DML or DDL request as a string"
Read_Domain = URM
Write_Domain = Importer
Update_Domain =

Process Control: LDV relies on LOCK for process creation, scheduling, and synchronization. Various alternatives have been examined for the organization of the set of processes in the assured pipelines for LDV [25]. Our choice for a single user updating environment is to have a set of processes per user, with those processes running at appropriate levels for that user.

LDV uses the services provided by LOCK such as create subject, destroy subject, delete object, wait on signal, and signal subject in order to create subjects and objects in the proper domains and of the proper types as needed. Some objects exist within the database only for the lifetime of a single database transaction, and are deleted as its conclusion to prevent potential intertransaction covert channels. Which subjects are created at database startup, which at user entry to the database, and which with a particular transaction is an open question. One possibility would be to start up subjects as soon as possible, and use the LOCK signaling mechanism to pass the control action of "the next transaction is ready to process" down the pipelines.

B. File Management

A typical DBMS uses an Operating system file system to store the data, metadata, and log data for recovery. For performance reasons it is useful for a DBMS to be able to store the data in physically contiguous blocks if desired. This is useful for sequential file access to a single relation stored in a single file. In addition, the DBMS usually implements multilevel directories, hashing, and indexes on top of the file system.

LDV will be built on top of the LOCK TCB which provides services for the manipulation of objects. LOCK provides the following services that are used for file management: create object and delete object. Data values will be stored in the individual records of the database, rather than having pointers to the data values. These records will be manipulated as pointers in virtual storage. As a result, all of the data values of a record can be fetched by a single LOCK I/O operation, since they will be contiguous in a single LOCK object. Indexes will be implemented in the form of *B*-trees. Each relation may have anywhere from zero indexes up to an index on each column. It is also possible to create an index on a combination of columns, provided they are all at the same security level. Indexes make it possible to scan a relation in order by the indexed values, or to directly access the records which match a particular value or range of values. The relationship between a table and a set of files will be maintained in the data dictionary.

LDV subjects in the DDT are defined so that object of particular types can only be created and destroyed by particular subjects. This prevents object-substitution by a trojan horse elsewhere in the database. Since LOCK does not provide directory management, LDV will need to maintain its own internal directory that maps relation names to file names.

C. Buffer Management

A typical DBMS does not use the operating system buffer management for performance reasons [27]. In order to implement its own buffer management, LDV must be able to partially control the main storage replacement policy provided by LOCK.

LDV will implement its own buffer management in the form of a main memory cache for the stored files that normally reside in secondary storage. The buffer manager loads pages in main memory for manipulation and selects pages to be written back to disk when required. Since a physical access to a database page on disk is much more expensive than an access to a database page in the buffer, the main goal of the buffer manager is the minimization of physical I/O.

The interface to the buffer manager consists of a request to access a page and an optional statement of update intent. This is done by issuing a FIX operator with a virtual storage page identifier, e.g., page 8 of the object whose id is 438. As a result, the page is located and fixed in main memory to prevent replacement during use. The virtual address of the frame containing the page is known so that the data on the page may be directly manipulated by LDV. When the page is no longer being addressed by a transaction, an UNFIX operation on the page is issued and the page is made eligible for replacement.

LDV must implement its own buffer management in order to have control over prefetching of blocks, the replacement strategy, and the ability to do selected force. The buffer manager will provide three functions: prefetch, block management, and selected force. The prefetch function is used during query processing. The query processor can give advice to the buffer manager concerning what blocks to prefetch into the main memory cache. The block management function is concerned with the strategy for replacing blocks in the buffer pool. One such strategy is least recently used (LRU), in which blocks for which there is locality of reference will remain in the cache over repeated reads and writes. The block management function can accept advice from the query processor concerning alternative replacement strategies. The selected force function is necessary for recovery. The recovery manager must be able to force data to disk at certain times in the transaction execution, in order to recover from a system crash. Since LOCK is a virtual storage operating system, the lack of LDV buffer management, or the accidental reference to an item on a page that had not been fixed by the LDV buffer manager, would not result in an error, only poorer performance.

IX. Conclusion

Given the additional problems introduced by increasing granularity of items in a database over files, the possibilities of inference and aggregation, and the need to manage metadata as well as data in a secure way, the way in which to design and organize a secure database is not obvious. In this paper, we have described the design of a secure database system, LDV, that builds upon the classical security policies for operating systems. We have described our policy for LDV and shows how it builds on the policy for LOCK TCB. We have also discussed the design approach of LDV which is based on assured pipelines and described the design of the query, update, and metadata management operations. Finally we described the operating system issues involved. In addition to the security policy and the design, the design specification, the formal model, and the formal top-level specification (FTLS) for LDV are also completed [20].

Future work includes investigating the inference, aggregation, and consistency issues. The LDV design already addresses certain inference and aggregation problems. Furthermore, the design is such that logic-based inference controllers such as those described in [28], [29], and [21] can be implemented on top of LDV. At present, the LDV design team is extending the design to handle a multiple user updating environment. The issues being investigated include concurrency control and recovery when multilevel transactions operate concurrently.

The implementation issues concerning a multilevel secure (MLS) application like LDV include the application needs specific to MLS applications, module organization, and reusability considerations. The design team has also examined these issues in order to obtain an implementation strategy for LDV. We look forward to describing our implementation at a future date.

Acknowledgment

RADC's support of the LDV application and their interactions with the National Computer Security Center LOCK program is gratefully acknowledged. We thank J. Giordano of RADC for his encouragement and support throughout the LDV project. We thank P. Dwyer, T. Haigh, and E. Onuegbe for their contributions to the LDV design.

References

[1] "ANSI SQL," American National Standards Institute, ANSI X3H2-86-2, Jan. 1986.

[2] "ISO-ANSI Database Language SQL2," International Organization for Standardization and American National Standards Institute, ANSI X3H2-88-259, July 1988.

[3] D. E. Bell and L. J. LaPadula, "Secure computer system: Unified exposition and multics interpretation," Tech. Rep., MTR-2997, The MITRE Corp., July 1975.

[4] W. E. Boebert, W. D. Young, R. Y. Kain, and S. A. Hansohn, "Secure Ada target: Issues, system design and verification," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, 1985, pp. 176-184.

[5] W. E. Boebert and R. Y. Kain, "A practical alternative to hierarchical integrity policies," in *Proc. 8th Nat. Comput. Security Conf.*, Gaithersburg, MD, Sept. 1985, pp. 18-27.

[6] F. Bry and R. Manthes, "Checking consistency of database constraints: A logical approach," in *Proc. Very Large Data Bases Conf.*, Kyoto, Japan, 1986, pp. 13-20.

[7] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377-387, June 1970.

[8] C. J. Date, *An Introduction to Database Systems*, 2nd ed. London, England: Addison-Wesley, 1983.

[9] D. E. Denning *et al.*, "A multilevel relational data model," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, Apr. 1987, pp. 220-234.

[10] "Security Requirements for Automatic Data Processing (ADP) Systems," Department of Defense Number 5200.28, May 6, 1977.

[11] "ADP Security Manual," Department of Defense Number 5200.28M, June 25, 1979.

[12] "Information Security Program Regulations," Dep. of Defense Number 5200.1R, Oct. 2, 1984.

[13] "Trusted Computer Systems Evaluation Criteria," Department of Defense Standard 5200.28-STD, Dec. 26, 1985.

[14] P. A. Dwyer, G. Jelatis, and M. B. Thuraisingham, "Multilevel security in database management systems," *Comput. Security*, vol. 6, no. 3, pp. 252-260, June 1987.

[15] P. A. Dwyer, E. Onuegbe, P. Stachour, and M. B. Thuraisingham, "Query processing in LDV: A multilevel secure relational database management system," in *Proc. 4th Aerospace Comput. Security Conf.*, IEEE, Orlando, FL, Dec. 1988, pp. 118-124.

[16] T. Hinke and M. Schaefer, "Secure data management system," Tech. Rep. RADC-75-266, Systems Development Corp., Nov. 1975.

[17] "B-level design specification for the LOCK operating system," CDRL A009, Contract MDA 904-87-C-6011, Honeywell Inc., June 1987.

[18] "Secure distributed data views—Security policy extensions," Interim Rep. A002, RADC Contract F30602-86-C-0003, Honeywell Inc., Apr. 1987.

[19] "Secure distributed data views—Implementation specifications," Interim Rep. A003, RADC Contract F30602-86-C-0003, Honeywell Inc., May 1988.

[20] "Secure distributed data views," Final Rep., Vols. 1-6, RADC Contract F30602-86-C-0003, Honeywell Inc., May 1989.

[21] T. F. Keefe, M. B. Thuraisingham, and W. T. Tsai, "Secure query processing strategies," *IEEE Comput. Mag.*, vol. 22, no. 3, pp. 63-70, Mar. 1989.

[22] F. Sadri and R. A. Kowalski, "Theorem proving approach to database integrity," in *Foundations of Deductive Databases*, J. Minker, Ed. Morgan Kaufmann, 1988.

[23] P. Stachour, M. B. Thuraisingham, and P. A. Dwyer, "Update processing in LDV: A multilevel secure relational database management system," presented at the 11th National Comput. Security Conf., Baltimore, MD, Oct. 1988.

[24] P. Stachour and M. B. Thuraisingham, "Metadata management in LDV: A multilevel secure relational database management system," Tech. Note, Honeywell Inc., Dec. 1988.

[25] —, "Operating system support for LDV," Tech. Note, Honeywell Inc., Dec. 1988.

[26] —, "SQL extensions for security assertions," *Comput. Standards Interfaces J.*, to be published.

[27] M. Stonebraker, "Operating systems support for database management," *Commun. ACM*, vol. 24, no. 7, pp. 412-418, July 1981.

[28] M. B. Thuraisingham, "Security checking in relational database management systems augmented with inferences engines," *Comput. Security*, vol. 6, no. 6, pp. 479-492, Dec. 1987.

[29] —, "Towards the design of secure data/knowledge base management system," *Data Knowledge Eng. J.*, to be published.

[30] J. Ullman, *Principles of Database Systems*. Rockville MD: Computer Science Press, 1982.

Paul D. Stachour received the B.S. degree in mathematics from Iowa State University, Ames, the M.S. degree in computer science from Ohio State University, Columbus, and the Ph.D. degree in computer science from the University of Waterloo, Waterloo, Ont., Canada.

He is a Principal Research Scientist at Secure Computing Technology Corporation. His primary research areas are secure multilevel applications, software component design, program design methodologies, and software development styles using Ada. He currently works two roles with Honeywell SCTC. He is the technical director of the Secure Distributed Database Views project. The objective of SDDV is to design a secure relational database system. His second role involves his interests at Honeywell in technology, tools, processes, and practices that enable software engineering. This theme has driven his Ada-related activities. He previously worked at Honeywell's Corporate Systems Development Division and Honeywell's Computer Sciences Center. Prior to that he worked on the PL/S Compiler at IBM's Systems Development Division and on a U.S. Army Curriculum Management system at the Army Chemical School. He is a coauthor of the book *Ada, A Programmer's Guide, with Microcomputer Examples* and ten refereed papers in the computer security and software methodology areas. He is an Adjunct Professor in Computer Science at both the University of Minnesota and the College of St. Thomas.

Dr. Stachour is a member of the Association for Computing Machinery.

Bhavani Thuraisingham received the B.Sc. degree in mathematics and physics from the University of Sri-Lanka, the M.S. degree in computer science from the University of Minnesota, the M.Sc. degree in mathematical logic from the University of Bristol, U.K., and the Ph.D. degree in recursive functions and computability theory from the University of Wales, Swansea, U.K.

She is a lead engineer at the MITRE Corporation. Her research interests include database security and the applications of mathematical logic in computer science. Previously, she was at Honeywell, Inc., where she was involved in the design of lock data views, and at Control Data Corporation. She was also an Adjunct Professor and member of the graduate faculty in the Department of Computer Science at the University of Minnesota. She has published more than 25 journal papers in database security, computability theory, AI, and distributed processing.

Dr. Thuraisingham is a member of the IEEE Computer Society, the Association for Computing Machinery, and Sigma Xi.

Design of an Integrated Information Retrieval/ Database Management System

LAWRENCE V. SAXTON AND VIJAY V. RAGHAVAN, MEMBER, IEEE

Abstract—**The problem of increasing the semantics available in a database management system has received considerable attention in recent years. Information retrieval systems provide well-studied and well-understood models that lead to meaningfully ranked responses to queries. A number of approaches for the integration of these systems have been considered and each has been found to be restricted in some way. A new, unified architecture is presented in this paper. This architecture provides the flexibility of integrating any information retrieval model with any type of database management system. More importantly, the approach provides for the ability to use "aggregation" and "generalization" operations automatically to provide more meaningful responses. As well, this framework enables the systematic investigation of the potential of employing IR models as a general tool for supporting management decisions.**

Index Terms—**Database design, database management systems, decision support systems, information retrieval systems, integration.**

I. Introduction

DATABASE management systems (DBMS) are designed to represent and manipulate objects or events of the real world as well as associations between these entities. The major objective of a database system is to provide an enterprise with a facility for the centralized control of its operational data in order to reduce the amount of redundancy in the stored data, avoid inconsistency in the stored data, share the stored data among users, enforce standards, apply security restrictions, and maintain data integrity [3].

DBMS's are widely used nowadays in many application systems needed in businesses, government agencies, and professional offices. A DBMS has become one of the major components of today's information systems. Most of today's DBMS's are very sophisticated and provide many and varied data management facilities. However, the facilities provided by DBMS's are not catching up with the increase in requirements of today's information systems. For example, there is increasing pressure from management to have information systems that support unstructured decision making and weighted evaluations, and provide ranking facilities in cases where there are overwhelming amounts of information. Many researchers have attempted to use artificial intelligence (AI) for achieving certain kinds of improvement. Another direction for achieving such improvement is to adopt proven methods from information retrieval.

The discipline of information retrieval deals with the organizing, structuring, retrieving, and displaying of bibliographic information [13], [14]. In this context, documents or document surrogates (e.g., abstracts) are analyzed and a representation for each document is generated. The simplest, and the most common, representation consists of describing each document by a set of keywords or a vector in which each element corresponds to the importance of a particular keyword to the document. These document representations are very much like the records in a DBMS environment and they have to be organized on storage devices to enable efficient search and retrieval. However, research on IRS's places more emphasis on the handling of imprecise concepts and the efficiency of the search rather than on the controls which are obtained from the sophisticated architectural design of the DBMS. The documents should or should not be retrieved for any given user query on the basis of their relevance to the query. A document is relevant if, with respect to a specific query, the user judges that it has the information that is desired by the user; it is nonrelevant otherwise.

A document may or may not be relevant to a user query depending on many variables of the document (style of writing, comprehensiveness, quality, etc.) as well as numerous user characteristics (previous knowledge, the reason for the search, etc.). The influence of all these factors on the decision of whether a certain document is relevant to a given request is quite involved and cannot be exactly represented in the choice of keywords or descriptors. In this sense, the descriptors used to represent the content of a document or the user need are not precise [6]. Therefore, it is recognized that an IRS cannot precisely select only and all relevant documents. Rather, it is suggested that the system should adopt a method that facilitates the ranking of documents in the order of their estimated relevance to a user query. More details of an IRS can be seen in [13] and [14].

The abilities of IRS's in handling imprecisely described objects and relationships, handling imprecise user queries, and ranking of responses from overwhelming amounts of information according to their relevance should add an interesting dimension to the database management system. It is believed that the integration of a DBMS and an

Manuscript received November 28, 1988; revised August 12, 1989.

L. V. Saxton is with the Department of Computer Science, University of Regina, Regina, Sask., S4S 0A2 Canada.

V. V. Raghavan is with The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504.

IEEE Log Number 9025105.