Handbook of Information Security Management (1994-95 Yearbook), Auerbach Publishers, 1994, pages 145-160.

RELATIONAL DATABASE ACCESS CONTROLS

Prof. Ravi S. Sandhu

Center for Secure Information Systems
&
Department of Information and Software Systems Engineering
George Mason University, Fairfax, VA 22030-4444
Telephone: 703-993-1659

1 INTRODUCTION

This chapter discusses access controls in relational database management systems. Access controls have been built into relational systems ever since the first products emerged. Over the years standards have developed, and these are continuing to evolve. In recent years products incorporating mandatory controls for multilevel security have started to appear.

The chapter begins with a review of the relational data model and the SQL language. Traditional discretionary access controls provided in various dialects of SQL are then discussed. Limitations of these controls, and the need for mandatory access controls, are illustrated by means of an example. Three architectures for building multilevel databases are presented. The chapter concludes with a brief discussion of role-based access control as an emerging technique for for providing better controls than traditional discretionary access controls, without going to the extreme rigidity of traditional mandatory access controls.

2 RELATIONAL DATABASES

A relational database stores data in relations which are expected to satisfy some simple mathematical properties. Roughly speaking, a *relation* can be thought of as a table, and is often shown as such. The columns of the table are called *attributes* and the rows are called *tuples*. There is no significance to the order of the columns or rows. Duplicate rows with identical values for all columns are not allowed. There is an important distinction between *relation schemes* and *relation instances*. The relation scheme gives us the names of the attributes as well as the permissible values for each attribute. The set of permissible values for an attribute is said to be the attribute's *domain*. The relation instance gives us the tuples of the relation at a given instant.

For example, consider the following relation scheme for the EMPLOYEE relation

EMPLOYEE(NAME, DEPT, RANK, OFFICE, SALARY, SUPERVISOR)

Let the domain of the NAME, DEPT, RANK, OFFICE, and SUPERVISOR attributes be character strings, and the domain of the SALARY attribute be integers. A particular instance of the EMPLOYEE relation, reflecting the employees who are currently employed, is shown below.

NAME	DEPT	RANK	OFFICE	SALARY	SUPERVISOR
Rao	Electrical_Engg	Professor	KH252	50,000	Jones
Kaplan	Computer_Sci	Researcher	ST125	35,000	Brown
Brown	Computer_Sci	Professor	ST257	55,000	Black
Jones	Electrical_Engg	Chairman	KH143	45,000	Black
Black	Administration	Dean	ST101	60,000	NULL

The relation instance of EMPLOYEE will change from moment to moment due to arrival of new employees, changes to data for existing employees and their departure.

The relation scheme, however, remains fixed. The NULL value in place of Black's supervisor signifies that Black's supervisor has not been defined.

2.1 Primary Key

A *candidate key* of a relation is a minimal set of attributes on which all other attributes are functionally dependent. In other words, it is forbidden to have two tuples with the same values of the candidate key in a relation instance. A candidate key is minimal, meaning that no attribute can be discarded without destroying this property. A candidate key always exists, since in the extreme case it consists of all the attributes.

In general, there can be more than one candidate key for a relation. In the EMPLOYEE relation above, for sake of example, let us assume that duplicate names can never occur. NAME is therefore a candidate key. Suppose also that there are no shared offices, so each employee has an unique office. In that case OFFICE is another candidate key. In the particular relation instance above there are no duplicate salary values. But that does not mean that salary is a candidate key. Identification of the candidate key is a property of the relation scheme and applies to every possible instance; not merely to the particular one that happens to exist at a given moment. SALARY would qualify as a candidate key only in the unlikely event that the organization forbids duplicate salaries.

The *primary key* of a relation is one of its candidate keys which has been specifically designated as such. In our example NAME is probably more appropriate than OFFICE as the primary key. Realistically one would use a truly unique identifier such as social security number or employee identity number, rather than NAME, as the primary key.

2.2 Entity and Referential Integrity

The primary key serves the purpose of uniquely identifying a specific tuple from a relation instance. It also serves the purpose of linking relations together. The relational model incorporates two application independent integrity rules, called *entity integrity* and *referential integrity* respectively, to ensure these purposes are properly served.

Entity integrity simply requires that no tuple in a relation instance can have NULL (undefined) values for any of the primary key attributes. This property guarantees that each tuple is uniquely identifiable by the value of the primary key.

Referential integrity is concerned with references from one relation to another. To understand this property in context of the EMPLOYEE relation above, let us suppose there is a second relation with the scheme

DEPARTMENT(DEPT, LOCATION, PHONE_NUMBER)

Let DEPT be the primary key of DEPARTMENT. The DEPT attribute of the EMPLOYEE relation is said to be a *foreign key* from EMPLOYEE to DEPARTMENT. In general a foreign key is an attribute (or set of attributes) in one relation R_1 whose values are required to match those of the primary key of a tuple in some other relation R_2. R_1 and R_2 need not be distinct. In fact, since supervisors are employees, the SUPERVISOR attribute in EMPLOYEE is a foreign key with $R_1 = R_2 =$ EMPLOYEE.

Referential integrity stipulates that if a foreign key FK of relation R_1 is the primary key PK of R_2, then for every tuple in R_1 the value of FK must either be NULL or equal to the value of PK of a tuple in R_2. In context of the above discussion,

referential integrity requires the following.

- Due to the DEPT foreign key, there should be tuples for the Electrical_Engg, Computer_Sci and Administration departments in the DEPARTMENT relation.
- Due to the SUPERVISOR foreign key, there should be tuples for Jones, Brown and Black in the EMPLOYEE relation.

The motivation of referential integrity is to prevent employees from being assigned to departments or supervisor who do not exist in the database. Note that it is all right for employee Black to have a NULL supervisor. It would similarly be acceptable for an employee to have a NULL department.

3 THE SQL LANGUAGE

Every Database Management System (DBMS) needs a language for defining, storing, retrieving, and manipulating data. SQL is the de facto standard, for this purpose, in relational DBMSs. SQL emerged from several projects at the IBM San Jose (now called Almaden) Research Center in the mid-1970s. The name SQL was originally an abbreviation for Structured Query Language. The official name now is Database Language SQL.

There is an official standard for SQL approved by the American National Standards Institute (ANSI), and accepted by the International Standards Organization (ISO) and the National Institute of Standards and Technology (NIST) as a Federal Information Processing Standard (FIPS). The standard has evolved and continues to do so. The base standard is generally known as SQL'89 and refers to the 1989

ANSI standard. SQL'92 is an enhancement of SQL'89 and refers to the 1992 ANSI standard. A third version of SQL, commonly known as SQL3, is being developed under ANSI and ISO aegis.

Most relational DBMSs support some dialect of SQL. It is important to understand that SQL-compliance does not guarantee portability of a data base from one DBMS to another. This is because DBMS vendors typically include enhancements not required by the SQL standard, but not ruled out by the standard either. Most products are also not completely compliant with the standard.

We will explain SQL in just enough detail to understand the examples and issues discussed in this chapter. Unless otherwise noted, our description is of SQL'89.

3.1 The CREATE Statement

Consider the EMPLOYEE relation discussed earlier. The relation scheme is defined in SQL by the following command.

```
CREATE  TABLE        EMPLOYEE
      (  NAME        CHARACTER     NOT NULL,
         DEPT        CHARACTER,
         RANK        CHARACTER,
         OFFICE      CHARACTER,
         SALARY      INTEGER,
         SUPERVISOR  CHARACTER,
         PRIMARY KEY (NAME),
         FOREIGN KEY (DEPT)        REFERENCES DEPARTMENT,
         FOREIGN KEY (SUPERVISOR)  REFERENCES EMPLOYEE )
```

This statement creates a table called EMPLOYEE with six columns. The NAME, DEPT, RANK, OFFICE and SUPERVISOR columns have character strings (of unspecified length) as values, whereas the SALARY column has integer values. NAME

is the primary key. DEPT is a foreign key which reference the primary key of table DEPARTMENT. SUPERVISOR is a foreign key which references the primary key (i.e., NAME) of the EMPLOYEE table itself.

3.2 INSERT and DELETE Statements

The EMPLOYEE table is initially empty. Tuples are inserted into it by means of the SQL INSERT statement. For example, the last tuple of the relation instance discussed above is inserted by the following statement.

```
INSERT
INTO      EMPLOYEE(NAME, DEPT, RANK, OFFICE, SALARY, SUPERVISOR)
VALUES    VALUES('Black, 'Administration', 'Dean', 'ST101', 60000, NULL)
```

The remaining tuples can be similarly inserted. Insertion of the tuples for Brown and Jones must respectively precede insertion of the tuples for Kaplan and Rao, so as to maintain referential integrity. Alternately, these tuples can be inserted in any order with NULL managers which are later updated to their actual values. There is a DELETE statement to delete tuples from a relation.

3.3 The SELECT Statement

Retrieval of data is effected in SQL by the SELECT statement. For example, the NAME, SALARY and SUPERVISOR data for employees in the Computer_Sci department is extracted as follows.

```
SELECT   NAME, SALARY, SUPERVISOR
FROM     EMPLOYEE
WHERE    DEPT = 'Computer_Sci'
```

This query applied to instance of EMPLOYEE given above returns the data shown below.

NAME	SALARY	SUPERVISOR
Kaplan	35,000	Brown
Brown	55,000	Black

The WHERE clause in a SELECT statement is optional. SQL also allows the retrieved records to be grouped together for statistical computations by means of built-in statistical functions. For example, the following query gives the average salary for employees in each department.

```
SELECT      DEPT, AVG(SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT
```

Data from two or more relations can be retrieved and linked together in a SELECT statement. For example, the location of employees can be retrieved by linking the data in EMPLOYEE with that in DEPARTMENT, as follows.

```
SELECT   NAME, LOCATION
FROM     EMPLOYEE, DEPARTMENT
WHERE    EMPLOYEE.DEPT = DEPARTMENT.DEPT
```

This query will attempt to match every tuple in EMPLOYEE with every tuple in DEPARTMENT, but will select only those pairs for which the DEPT attribute in the EMPLOYEE tuple matches the DEPT attribute in the DEPARTMENT tuple. Since DEPT is a common attribute to both relations, every use of it is explicitly identified as occurring with respect to one of the two relations. Queries involving two relations in this manner are known as *joins*.

3.4 The UPDATE Statement

Finally the UPDATE statement allows one or more attributes of existing tuples in a relation to be modified. For example, the following statement gives all employees in the ComputerSci department a raise of $1000.

```
UPDATE   EMPLOYEE
SET      SALARY = SALARY + 1000
WHERE    DEPT = 'Computer_Sci'
```

This statement selects those tuples in EMPLOYEE which have the value of Computer_Sci for the DEPT attribute. It then increases the value of the SALARY attribute for all these tuples by $1000 each.

4 BASE RELATIONS AND VIEWS

The concept of a view has important security application in relational systems. A *view* is a virtual relation which is derived by an SQL definition from base relations and other views. The database stores the view definitions and materializes the view as needed. In contrast, a *base relation* is actually stored in the database.

For example, consider the EMPLOYEE relation discussed earlier. This is a base relation. The following SQL statement defines a view called COMPUTER_SCI_DEPT.

```
CREATE   VIEW COMPUTER_SCI_DEPT
AS       SELECT   NAME, SALARY, SUPERVISOR
         FROM     EMPLOYEE
         WHERE    DEPT = 'Computer_Sci'
```

This defines the virtual relation shown below.

NAME	SALARY	SUPERVISOR
Kaplan	35,000	Brown
Brown	55,000	Black

A user who has permission to access COMPUTER_SCI_DEPT is thereby restricted to retrieving information about employees in the Computer Science Department. To illustrate the dynamic aspect of views suppose that a new employee Turing is inserted in base relation EMPLOYEE, modifying it as follows.

NAME	DEPT	RANK	OFFICE	SALARY	SUPERVISOR
Rao	Electrical_Engg	Professor	KH252	50,000	Jones
Kaplan	Computer_Sci	Researcher	ST125	35,000	Brown
Brown	Computer_Sci	Professor	ST257	55,000	Black
Jones	Electrical_Engg	Chairman	KH143	45,000	Black
Black	Administration	Dean	ST101	60,000	NULL
Turing	Computer_Sci	Genius	ST444	95,000	Black

The view COMPUTER_SCI_DEPT will be automatically modified to include Turing, as shown below.

NAME	SALARY	SUPERVISOR
Kaplan	35,000	Brown
Brown	55,000	Black
Turing	95,000	Black

In general views can be defined in terms of other base relations and views.

Views can also be used to provide access to statistical information. For example, the following view gives the average salary for each department.

```
CREATE   VIEW AVSAL(DEPT, AVG)
AS       SELECT    DEPT, AVG(SALARY)
         FROM      EMPLOYEE
         GROUP BY  DEPT
```

For retrieval purposes there is no distinction between views and base relations. Views, therefore, provide a very powerful mechanism for controlling what information can be retrieved. When updates are considered views and base relations must be treated quite differently. In general views cannot be updated directly by users, particularly when they are constructed by joining two or more relations. Instead the base relations must be updated, with views being updated indirectly through this means. This limits the usefulness of views for authorizing update operations.

5 DISCRETIONARY ACCESS CONTROLS

We now describe the access control facilities included in the SQL standard. The standard is incomplete and does not address several important issues. Some of these deficiencies are being addressed in the evolving standard. Different vendors have also provided more comprehensive facilities than called for by the standard.

5.1 SQL Privileges

The creator of a relation in an SQL database becomes its owner. The owner has the intrinsic ability to grant other users access to that relation. The *access privileges or modes* recognized in SQL correspond directly to the CREATE, INSERT, SELECT, DELETE and UPDATE SQL statements discussed earlier. There is also a REFERENCES privilege to control the establishment of foreign keys to a relation.

5.2 The CREATE Statement

SQL does not require explicit permission for a user to create a relation, unless the relation is defined to have a foreign key to another relation. In the latter case the user must have the REFERENCES privilege for appropriate columns of the referenced relation. To create a view a user must have the SELECT privilege on every relation mentioned in definition of the view. If a user has INSERT, DELETE or UPDATE privileges on these relations, corresponding privileges will be obtained on the view (if it is updatable).

5.3 The GRANT Statement

The owner of a relation can grant one or more access privileges to another user. This can be done with or without the GRANT OPTION. If the owner grants, say, SELECT with the GRANT OPTION the user receiving this grant can further grant SELECT to other users. The latter GRANT can be done in turn with or without the GRANT OPTION at the granting user's discretion.

The general format of a grant operation in SQL is as follows.

```
GRANT   privileges
[ON     relation]
TO      users
[WITH   GRANT OPTION]
```

The GRANT command applies to base relations as well as views. The brackets on the ON and WITH clauses denote that these are optional and may not be present in every GRANT command. Note that it is not possible to grant a user the grant option on a privilege, without allowing the grant option itself to be further granted.

INSERT, DELETE and SELECT privileges apply to the entire relation as a unit. INSERT and DELETE are operations on entire rows so this is appropriate. SELECT, however, implies the ability to select on all columns. Selection on a subset of the columns can be achieved by defining a suitable view, and granting SELECT on the view. This is somewhat awkward, and there have been proposals to allow SELECT to be granted on a subset of the columns of a relation. The UPDATE privilege in general applies to a subset of the columns. For example, a user could be granted the authority to update the OFFICE but not the SALARY of an EMPLOYEE. SQL'92 extends the INSERT privilege to apply to a subset of the columns. This can be used, for instance, to allow a clerical user to insert a tuple for a new employee with the NAME,

DEPARTMENT and RANK data. The OFFICE, SALARY and SUPERVISOR data can then be updated in this tuple by a suitably authorized supervisory user.

SQL'89 has several omissions in its access control facilities. These omissions have been addressed by different vendors in different ways. We will identify the major omissions here and illustrate how they have been addressed in products and in the evolving standard.

5.4 The REVOKE Statement

One major shortcoming of SQL'89 is the lack of a REVOKE statement to take away a privilege that has been granted by a GRANT. IBM's DB2 product provides a REVOKE statement for this purpose.

It is often required that revocation should cascade. In a cascading revoke, not only is the revoked privilege taken away, but also all GRANTs based on the revoked privilege are effectively revoked. For example say that user Tom grants Dick SELECT on relation R with the GRANT OPTION. Furthermore, Dick subsequently grants Harry SELECT on R. Now suppose Tom revokes SELECT on R from Dick. The SELECT on R privilege is taken away not only from Dick, but also from Harry. The precise mechanics of a cascading revoke is somewhat complicated. Suppose Dick had received the SELECT on R privilege (with GRANT OPTION) not only from Tom, but also from Jane before Dick granted the SELECT to Harry. In this case Tom's revocation of the SELECT from R privilege from Dick will not cause either Dick or Tom to loose this privilege. This is because the GRANT from Jane remains valid.

Cascading revocation is not always desirable. A user's privileges to a given table

are often revoked because the user's job functions and responsibilities have changed. Thus the Head of a Department, say Mary, may move on to a different assignment. Mary's privileges to that Department's data should be revoked. However, a cascading revoke could cause lots of employees of that Department to loose their privileges. These privileges would then need to be re-granted to keep the Department functioning.

SQL'92 allows revocation to be cascading or not cascading as specified by the revoker. The is a partial solution to the more general problem of how to reassign responsibility for managing access to data from one user to another as their job assignments change.

5.5 Other Privileges

Another major shortcoming of SQL'89 is the lack of control over who can create relations. In SQL'89 every user is authorized to create relations. The Oracle DBMS requires possession of a RESOURCE privilege in order to create new relations. SQL'89 also does not include a privilege to DROP a relation. Such a privilege is included in DB2.

SQL'89 does not address the issue of how new users are enrolled in a database. Several products take the approach that a database is always created with a single user, usually called the DBA (Data Base Administrator), to begin with. The DBA essentially has all privileges with respect to this database. The DBA is then responsible for enrolling users and creating relations. Some systems recognize a special privilege (called DBA in Oracle and DBADM in DB2) which can be granted to other users at the original DBA's discretion, and allows these users to effectively act as the DBA.

6 LIMITATIONS OF DISCRETIONARY CONTROLS

The standard access controls of SQL are said to be discretionary, because the granting of access is under user control. Discretionary controls have a fundamental weakness. Even if access to a relation is strictly controlled, it is possible for a user with SELECT access to create a copy of the relation thereby circumventing these controls. Furthermore, even if users are trusted not to deliberately engage in such mischief it is possible for Trojan Horse infected programs to do so.

To illustrate the basic limitation of discretionary access controls, consider the following grant operation.

TOM: GRANT SELECT ON EMPLOYEE TO DICK

Tom has not conferred the grant option on Dick. Tom's intention is that Dick should not be allowed to further grant SELECT access on EMPLOYEE to other users. However, this intent is easily subverted as follows. Dick creates a new relation, call it COPY-OF-EMPLOYEE, into which he copies all the rows of EMPLOYEE. As the creator of COPY-OF-EMPLOYEE, Dick has the authority to grant any privileges for it to any user. Dick can therefore grant Harry access to COPY-OF-EMPLOYEE as follows.

DICK: GRANT SELECT ON COPY-OF-EMPLOYEE TO HARRY

At this point Harry has access to all the information in the original EMPLOYEE relation. For all practical purposes Harry has SELECT access to EMPLOYEE, so

long as Dick keeps COPY-OF-EMPLOYEE reasonably up to date with respect to EMPLOYEE.

The situation is actually worse than the above scenario indicates. So far, we have portrayed Dick as a cooperative participant in this process. Now suppose that Dick is a trusted confidant of Tom and would not deliberately subvert Tom's intentions regarding the EMPLOYEE relation. However, Dick uses a fancy text editor supplied to him by Harry. This editor provides all the editing services that Dick needs. In addition Harry has also programmed it to create the COPY-OF-EMPLOYEE relation and execute the above grant operation. Such software is said to be a Trojan Horse, because in addition to the normal functions expected by its user it also engages in surreptitious actions to subvert security. Note that a Trojan Horse executed by Tom could actually grant Harry the privilege to SELECT on EMPLOYEE.

In summary, even if the users are trusted not to deliberately breach security we have to contend with Trojan Horses which have been programmed to deliberately do so. We can require that all software that is run on the system is free of Trojan Horses. But this is generally not considered to be a practical option. The solution is to impose mandatory controls which cannot be violated, even by Trojan Horses.

7 MANDATORY ACCESS CONTROLS

Mandatory access controls are based on *security labels* associated with each data item and each user. A label on a data item is called a *security classification*, while a label on a user is called a *security clearance*. In a computer system every program run by a user inherits the user's security clearance.

Security labels in general form a lattice structure. For purpose of our discussion we will assume the simplest situation where there are only two labels: S for Secret and U for Unclassified. It is forbidden for S information to flow into U data items. There are two mandatory access controls rules to achieve this objective.

- **Simple Security Property:** A U-user cannot read S-data.
- **Star Property:** A S-user cannot write U-data.

There are some important points that should be clearly understood in this context. Firstly, the rules assume that a human being with Secret clearance can login to the system as a S-user or a U-user. Otherwise the star property will prevent top executives from writing publicly readable data. Secondly, these rules only prevent information flow due to overt reading and writing of data. It remains possible for Trojan Horses to leak Secret data using devious means of communication called covert channels.

Finally, mandatory access controls in relational databases usually enforce a stronger star property given below.

- **Strong Star Property:** A S-user cannot write U-data and a U-user cannot write S-data.

The strong star property limits each user to writing at their own level. It is motivated by integrity considerations. The (weak) star property allows a U-user to write S-data. This can result in overwriting, and therefore destruction, of S-data by U-users. In the remainder of this chapter we will require the strong star property.

7.1 Labeling Granularity

Security labels can be assigned to data at different levels of granularity in relational databases. Assigning labels to entire relations can be useful but is in general inconvenient. For example, if some salaries are secret but others are not, we will be forced these salaries in different relations. Assigning labels to entire column of a relation is similarly inconvenient in the general case.

The finest granularity of labeling is at the level of individual attributes of each tuple (row) or element-level labeling. This offers considerable flexibility. Most of the products emerging in this arena offer labeling at the level of a tuple. Although not so flexible as element-level labeling, this approach is more convenient than relation or column-level labels. It can be expected that products in the short term will offer tuple-level labeling.

8 MULTILEVEL DATABASE ARCHITECTURES

A multilevel system is one in which users and data with different security labels coexist. Multilevel systems are said to be *trusted* because they can keep data with different labels separated, and ensure that the simple security and (strong) star properties are enforced. Over the past fifteen years or so, considerable research and development has been devoted to the construction of multilevel databases. Three viable architectures have emerged as follows.

1. Integrated data architecture (also known as the trusted subject architecture).

2. Fragmented data architecture (also known as the kernelized architecture).

3. Replicated data architecture (also known as the distributed architecture).

The relational database products which are initially emerging in this arena are basically integrated data architectures. This approach requires considerable modification of an existing relational DBMS. It can be supported by DBMS vendors because they own the source code for their DBMS's, and are in a position to modify it in new products.

The fragmented and replicated architectures have been demonstrated in laboratory projects. They offer possibly greater assurance of security than the integrated data architecture. Moreover, they can be constructed by using commercial off-the-shelf (COTS) DBMS's as components. This allows non-DBMS vendors to build these by integrating COTS trusted operating systems and non-trusted DBMS's. We now describe these three architectures in turn.

8.1 Integrated Data Architecture

The integrated data architecture is illustrated in figure 1. In all of our diagrams the solid lines show flow of U-data, the dashed lines show flow of S-data and the dotted lines show flow of mixed U and S-data.

The bottom of figure 1 shows three kinds of data coexisting in the disk storage of the system, as follows.

- U-non-DBMS-data: unclassified data files managed directly by the trusted Operating System (OS).
- S-non-DBMS-data: secret data files managed directly by the trusted OS.

- U+S-DBMS-data: unclassified and secret data stored in files managed cooperatively by the trusted OS and the trusted DBMS.

At the top of the diagram, on the left hand side, there is a U-user and S-user interacting directly with the trusted OS. The trusted OS only allows these users to access non-DBMS data in this manner. As per the simple security and strong star properties, the U-user is allowed to read and write U-non-DBMS data, while the S-user is allowed to read U-non-DBMS data and read and write S-non-DBMS data.

The right hand side of the diagram shows a U-user and S-user interacting with the trusted DBMS. The trusted DBMS is responsible for enforcing the simple security and strong star properties with respect to the DBMS data. The trusted DBMS relies on the trusted OS to make sure that DBMS data cannot be accessed without intervention of the trusted DBMS.

8.2 Fragmented Data Architecture

The fragmented data architecture is shown in figure 2. In this architecture only the OS is multilevel and trusted. The DBMS is untrusted and interacts with users at a single level. The bottom of figure 2 shows two kinds of data coexisting in the disk storage of the system, as follows.

- U-data: unclassified data files managed directly by the trusted OS.

- S-data: secret data files managed directly by the trusted OS.

The trusted OS does not distinguish between DBMS and non-DBMS data in this architecture. It supports two copies of the DBMS, one which can interact only with

U-users and another which can interact only with S-users. These two copies run the same code but with different security labels. The U-DBMS is restricted by the trusted OS to reading and writing U-data. The S-DBMS, on the other hand, can read and write S-data as well as read (but not write) U-data.

This architecture has great promise, but its viability is dependent on availability of usable good-performance trusted Operating Systems. So far, there are few trusted OS's and these lack many of the facilities that users expect modern OS's to provide. Development of trusted OS's continues to be an active area, but one in which progress has been slow. Emergence of strong products in this arena could make the fragmented data architecture attractive in the future.

8.3 Replicated Data Architecture

The replicated data architecture is shown in figure 3. This architecture requires physical separation on backend database servers to separate U and S users of the database. The bottom half of the diagram shows two physically separated computers, each running a DBMS. The computer on the left hand side manages U data, whereas the computer on the right hand side manages a mix of U and S data. The U data on the left hand side is replicated on the right hand side.

The trusted OS serves as a front end. It has two objectives. Firstly, it is responsible for ensuring that a U-user can directly access only the U-backend (left hand side), and a S-user can directly access only the S-backend (right hand side). Secondly, the trusted OS is the sole means for communication from the U-backend to the S-backend. This communication is required so that updates to the U-data can be propagated to the U-data stored in the S-backend. Correct and secure propagation of

these updates has been a major obstacle to this architecture, but recent research has provided viable solutions to this problem. The replicated architecture is viable for a small number of security labels, perhaps a few dozen, but it does not scale gracefully to hundreds or thousands of labels.

9 ROLE-BASED ACCESS CONTROLS

It is generally agreed that traditional discretionary access controls are proving to be inadequate for the security needs of many organizations. At the same time, mandatory access controls based on security labels are also perceived as being inappropriate for many situations. In recent years the notion of role-based access control (RBAC) has emerged as a candidate for filling the gap between traditional DAC and MAC.

One of weaknesses of DAC in SQL is that it does not facilitate management of access rights. Each user must be explicitly granted every privilege that they need to accomplish their tasks. Often groups of users need similar or identical privileges. Thus all Supervisors in a Department might require identical privileges. Similarly all Clerks might require identical privileges, which are different from those of the Supervisors. RBAC allows the creation of roles for Supervisors and Clerks. Privileges appropriate to these roles are explicitly assigned to the role. Individual users are then enrolled in appropriate roles from where they inherit these privileges. This separates two concerns: (i) what privileges should a role get, and (ii) which user should be authorized to each role. With RBAC it becomes easier to reassign users from one role to another, or to alter the privileges for an existing role.

Current efforts at evolving SQL, commonly called SQL3, have included proposals

for RBAC based on vendor implementations such as in Oracle. In future consensus on a standard approach to RBAC in relational databases should emerge. However, this is a relatively new area, and a number of questions remain to be addressed before consensus on standards is obtained.

10 SUMMARY

Access controls have been an integral part of relational database management systems from the start. There are, however, major weaknesses in the traditional discretionary access controls built into the standards and products. SQL'89 is incomplete and omits revocation of privileges and control over creation of new relations and views. SQL'92 fixes some of these shortcomings. In the meantime vendors such as Oracle have developed new concepts such as role-based access control. Others such as Informix have started delivering products incorporating mandatory access controls for multilevel security. There is a recognition that SQL needs to be further evolved to take some of these developments into considerations. This activity will hopefully lead to stronger and better access controls in future products.

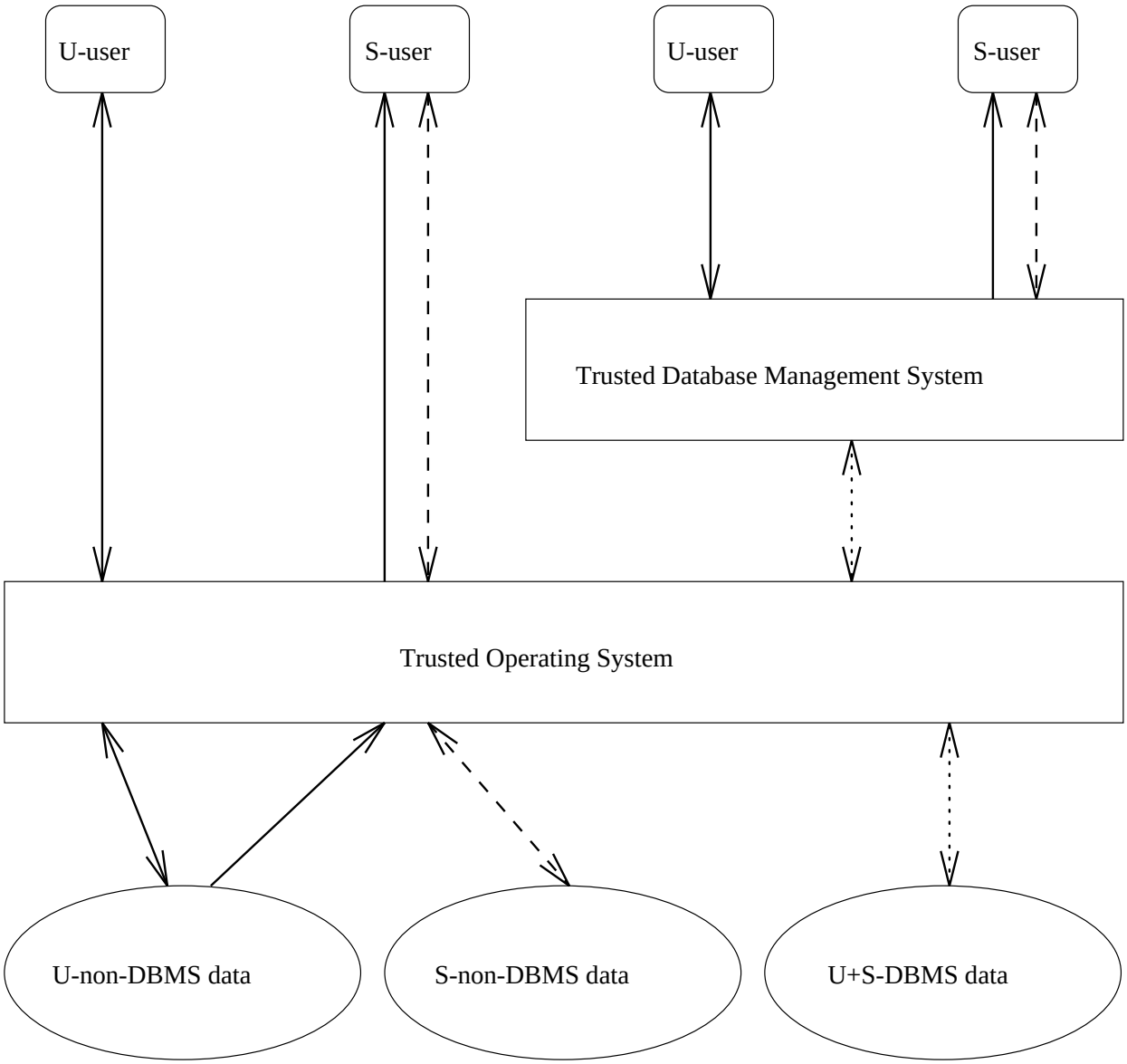

Figure 1: Integrated Data Architecture

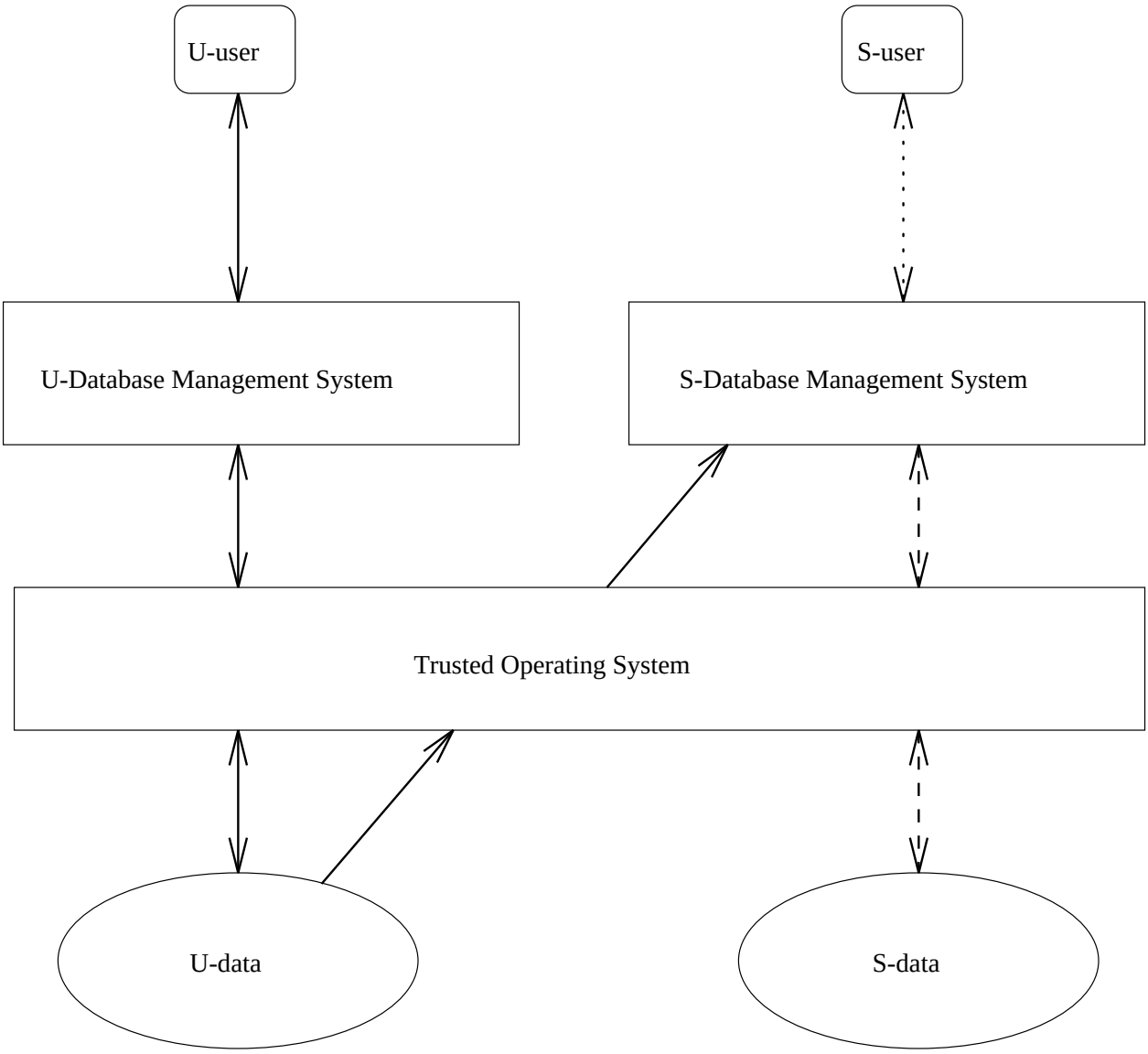

Figure 2: Fragmented Data Architecture

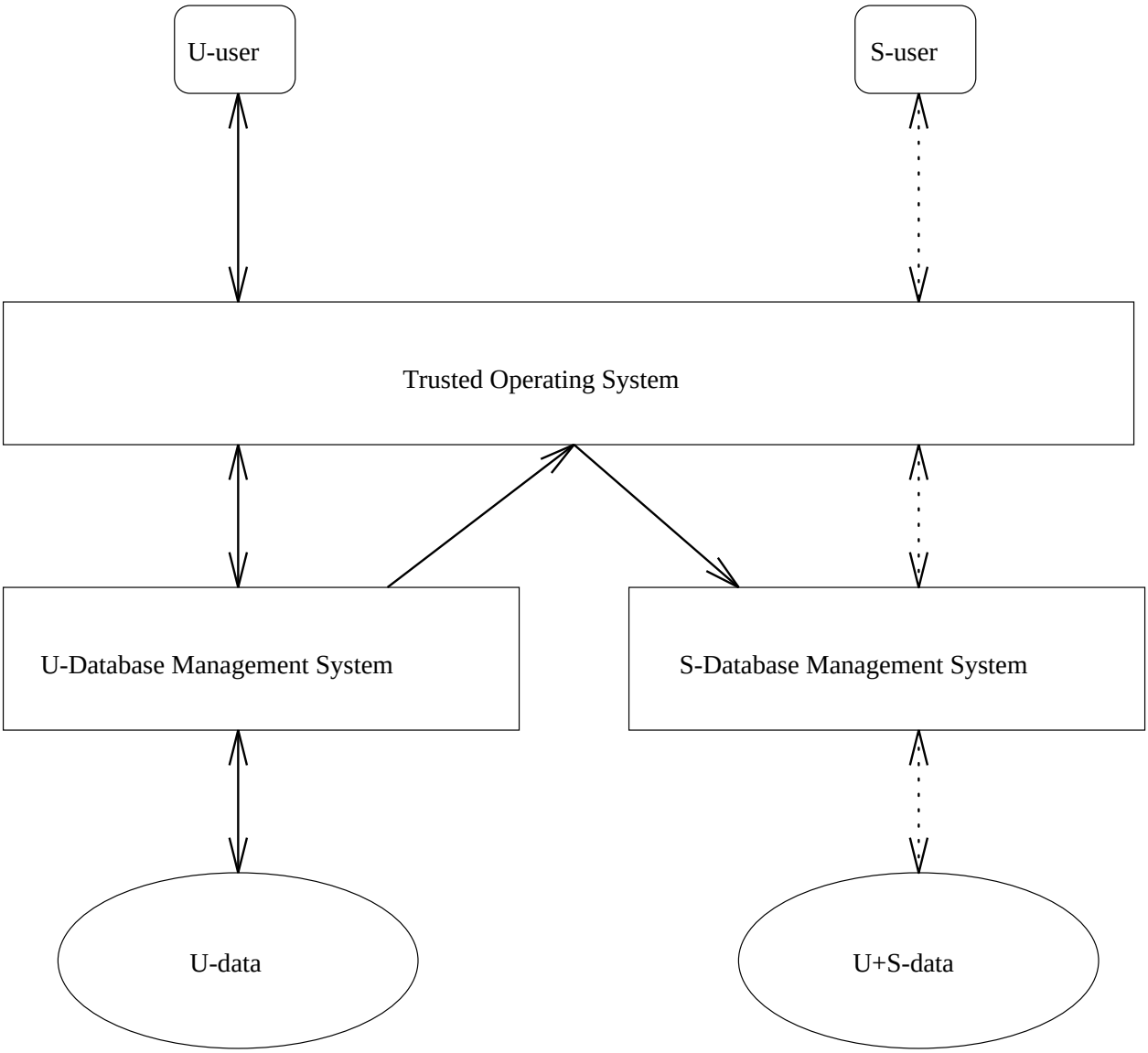

Figure 3: Replicated Data Architecture