An Introduction to Database Systems

Koen Van Boxstael
BC2

2002-2003[1]

[1]Version January 6, 2003

Chapter 1

An Overview of Database Management

1.1 Introduction

A **database system** is a computerized record-keeping system. A **database** is a container for a collection of computerized data files.

1.2 What is a database system?

- Data
  - Integrated
    A database is a unification of otherwise distinct files, with any redundancy among those files at least partly eliminated.
  - Shared
    Different users can have access to the same piece of data.
- Hardware
- Software
  Between the physical database and the users is the **Database Management System (DBMS)**.
- Users
  - Application programmers
  - End users
  - Database administrator (DBA)

1.3 What is a database?

Data is **persistent**: once the data is in the database it can only be removed by an explicit request to the DBMS.

A database is a collection of persistent data that is used by the application systems of some given enterprise.

Entities are objects that are to be represented in the database.

Relationships link entities together, are a special kind of entity.

Properties of an entity: information we wish to record about them.

Data are true propositions.

A **data model** is an abstract, self-contained, logical definition for the objects and operators that constitute the abstract machine with which the users interact. The objects model the structure of data, operators model its behavior.

An **implementation** of a given data model is a physical realization on a real machine of the components of the abstract machine that together constitute that model.

1.4 Why database?

The **Data Administrator (DA)** decides what data should be stored in the database and establishes policies for maintaining and dealing with the data once it has been stored.

The **Database administrator (DBA)** creates the actual database and implements the controls needed to enforce the policies.

1.5 Data independence

Data independence is the immunity of applications to change in physical representation and access technique.

1.6 Relational systems and others

A relational system is one in which:

1. Data is perceived by the user as tables and nothing but tables.
2. The operators available to the user are operators that generate new tables from old.

Note: Relation = table.

Chapter 2

Database System Architecture

2.2 The three levels of the architecture

- **Internal level** (physical level): The way data is physically stored.
- **External level** (user logical level): The way data is seen by individual users (many external views).
- **Conceptual level** ((community) logical level): Indirection between internal and external level (only one conceptual view).

In a relational system the conceptual level is definitely relational, the external level is typically also relational, but the internal level is never relational.

2.3 The external level

Each user has a language at his disposal. A conventional programming language or a 4GL for the application programmer, and a query language or special purpose language for end users. A **data sublanguage (DSL)** (concerned with database objects and operations) is embedded in a **host language** (concerned with nondatabase facilities). They can be **tightly coupled** or **loosely coupled**.

A DSL is a combination of a **data definition language (DDL)** and a **data manipulation language (DML)**.

An **external view** is the content of the database as seen by some user. It consists of many **external records**.

An external view is defined by an **external schema**, written using an external DDL.

2.4 The conceptual level

The **conceptual view** is a representation of the entire information content of the database. It consists of many **conceptual records**.

The conceptual view is defined by the **conceptual schema**, written using a conceptual DDL. It includes security and integrity constraints. It has to be completely independent of physical representation.

2.5 The internal level

The **internal view** is a low-level representation of the entire database. It consists of many **internal records** or **stored records**. The internal view is also called storage structure or **stored database**.

An internal view is defined by an **internal schema** or **storage structure definition**, written using an internal DDL. It not only defines the stored record types but also the indexes, representation of stored records, physical sequence of stored records, . . .

2.6 Mappings

- The **conceptual/internal mapping** defines the correspondence between conceptual view and stored database.
- An **external/conceptual mapping** defines the correspondence between a particular external view and conceptual view.

They are the key to **physical** and **logical data independence**.

2.7 The database administrator

His tasks include:

- Defining the conceptual schema (**logical database design**) together with the data administrator
- Defining the internal schema (**physical database design**) and the associated mapping
- Liaising with users (write or help write external schemas and mappings)
- Defining security and integrity constraints
- Defining dump and reload policies
- Monitoring performance and responding to changing requirements

2.8 The database management system

The DBMS is the software that handles access to the database

Functions include:

- Data definition

  It must include a **DDL processor** or **DDL compiler**.
- Data manipulation

  It must include a **DML processor** or **DML compiler** for **planned** and **unplanned** requests.

- Optimization and execution

  DML request are processed by the **optimizer** and then executed by the **run-time manager**.
- Data security and integrity
- Data recovery and concurrency
- Data dictionary

  Contains data about the data (metadata).
- Performance

The DBMS provides a **user interface** to the database system at the external level.

2.10 Client/server architecture

The server is the DBMS, the clients are the applications running on top of the DBMS. Applications can be user-written or vendor-provided (**tools**).

2.11 Utilities

Programs to help the DBA. Some utilities operate directly on the internal level.

2.12 Distributed processing

Two possibilities:

1. Many clients connect to one server, clients and servers are on different machines.
2. Every client machine can connect to several different server machines (**distributed database system**).

Chapter 3

An Introduction to Relational Databases

3.2 An informal look at the relational model

The **relational model of data**:

- Structure: The data in the database is perceived by the user as tables and nothing but tables.
- Integrity: Those tables satisfy certain integrity constraints.
- Manipulation: The operators available to the user for manipulating those tables derive tables from tables.

Important operators are:

- **restrict**: extracts rows
- **project**: extracts columns
- **join**: joins two tables together

The **closure** property of relational systems: the output of any operation on a table is again a table. This makes nested expressions possible.

The Information Principle means that the entire information content of the database is represented in only one way, namely as explicit values in column positions in rows in tables. (There are no pointers.)

The **primary key** in a table is a column in which every value is unique.

A **foreign key** in a table is a column that references the primary key of another table.

3.3 Relations and relvars

Tables are called **relation variables** (**relvars**), which have **relation values** (the actual contents of the table at a particular time).

3.4 What relations mean

Every relation value has two parts:

- The **heading**, a set of column-name:type-name pairs
- The **body**, a set of rows that conform to that heading

The heading of a relation can be regarded as a **predicate**, and each row in the body denotes a **true proposition**.

3.5 Optimization

Relational systems perform **automatic navigation**. The user specifies *what* he wants to get, not *how* he wants to get it. Deciding how a request is implemented is the job of the optimizer, an important component of a DBMS.

3.6 The catalog

A **catalog** or a **dictionary** contains all the schemas and mappings. It contains metadata or descriptor information about various objects like base relvars, indexes, users, integrity and security constraints, ... In a relational system the catalog also consists of relvars.

3.7 Base relvars and views

The original relvars in a database are called **base relvars** ad their values are **base relations**. A relation that is obtained from those base relations by means of some relational expression is called a **derived relation**. A view is a relvar whose value is a derived relation, and is also called a **derived relvar**. A view is just a window to the underlying base relvar. Changes in the base relvar are automatically and instantaneously visible in the view.

3.8 Transactions

A transaction is a logical unit of work, involving several database operations. Transactions are supposed to be

- **Atomic**, execute entirely or not at all
- **Durable**, when the transaction successfully executes COMMIT, its updates must be applied to the database
- **Isolated**, updates from one transaction are not visible to another transaction until that transaction successfully executes COMMIT (COMMIT $\leftrightarrow$ ROLLBACK)
- Interleaved execution of concurrent transactions is **serializable**

Chapter 5

Domains, Relations, and Base Relvars

5.1 Introduction

If we think of a relation as a table, then we can compare formal terms with more informal terms:

Formal	*Informal*
relation	table
tuple	row
cardinality	number of rows
attribute	column or field
degree	number of columns
primary key	unique identifier
domain	pool of legal values

5.2 Domains

A **domain** or **data type** or **type** is a set of all possible values of the type along with the valid operators that can be applied to values of that type.

There is a difference between a type and its physical representation.

Strong typing means that

- Every value has a type
- The operands of an operator have to be of the right type for that operation

A type can be **scalar** or **nonscalar**. A scalar type has no user-visible components.

Every scalar type has at least one possible representation. Every possible representation declaration causes definition of a selector operator to specify or select a value of the type, and one THE_ operator for each component of the possible representation, to access these components.

There are two types of operators. Read-only operators return a value, update operators don't return a value, but update their argument.

5.3 Relation values

Definition:

> Given a collection of n types or domains T_i ($i = 1, \ldots, n$), r is a **relation** on those types if it consists of two parts, a heading and a body, where:
>
> - The **heading** is a set of n **attributes** of the from A_i:T_i, where the A_i (all distinct) are the attribute names of r and the T_i are the corresponding type names
> - The **body** is a set of m **tuples** t, where t is a set of components of the form A_i:v_i in which v_i is a value of type T_i, the attribute value for attribute A_i of tuple t
>
> m is the **cardinality**, n is the **degree**.

A table is only a representation "on paper" of a relation if correctly interpreted, but strictly speaking a table is not exactly the same as a relation.

Properties of relations:

- There are no duplicate tuples
- Tuples are unordered, top to bottom
- Attributes are unordered, left to right
- Each tuple contains exactly one value for each attribute

The first three properties follow from the fact that sets are unordered and don't include duplicate elements. The fourth follows from the definition of a tuple. A relation that satisfies this property is called **normalized**, or in **first normal form**.

Tuples of a relation can be interpreted as true propositions. The **Closed World Assumption** (or **Closed World Interpretation**) says that the body of a relation contains all and only the tuples corresponding to true propositions.

5.4 Relation variables

The syntax for defining a base relvar is:

```
VAR <relvar name> BASE <relation type>
<candidate key definition list>
[ <foreign key definition list> ] ;
```

<relation type> is RELATION { <attribute commalist> }, and an <attribute> is an ordered pair <attribute name> <type name>.

The terms heading, body, attribute, tuple, cardinality and degree apply to relvars as well.

A relvar update is a **relational assignment**:

```
<relvar name> := <relational expression>
```

Chapter 6

Relational Algebra

6.1 Introduction

The *original algebra* (by Codd) consists of 8 operators:

1. Traditional set operators **union**, **intersection**, **difference** and **cartesian product**, modified to operate on relations instead of sets
2. Special relational operators **restrict**, **project**, **join** and **divide**.

6.2 Closure revisited

Closure means that the output of any relational operation is another relation. This enables us to write nested relational expressions, i.e. relational expressions in which the operands can be relational expressions in turn.

The output of a relational expression must be of a well-defined relation type. Therefore we need a set of type inference rules built into the algebra.

6.4 Semantics

Union

Given two relations A and B of the same type, the union of those two relations, `A UNION B`, is a relation of the same type, with body consisting of all tuples t such that t appears in A or in B or in both.

Intersect

Given two relations A and B of the same type, the intersection of those two relations, `A INTERSECT B`, is a relation of the same type, with body consisting of all tuples such that t appears in both A and B.

Difference

Given two relations A and B of the same type, the difference between those two relations, `A MINUS B` (in that order), is a relation of the same type, with body consisting of all tuples t such that t appears in A and not in B.

(Cartesian) product

The Cartesian product of two relations A and B, `A TIMES B`, where A and B have no common attribute names, is a relation with a heading that is the (set theory) union of the headings of A and B, and with a body consisting of all tuples t such that t is the (set theory) union of a tuple appearing in A and a tuple appearing in B.

Note: The cardinality of the result is the product of the cardinalities, the degree of the result is the sum of the degrees.

Note: If A and B have common attribute names, the RENAME operator has to be used first.

Restrict

Given a relation A with attributes X and Y (among others), and an operator Θ such that the condition X Θ Y is well defined and evaluates to a truth value, the Θ-restriction of relation A on attributes X and Y (in that order) — `A WHERE X Θ Y` — is a relation with the same heading as A and with a body consisting of all tuples t of A such that the condition X Θ Y evaluates to true for that tuple t.

Note: A literal can be used instead of X or Y, but this is actually a shorthand for an expression with the EXTEND operator.

Note: The WHERE clause can consist of multiple conditions with AND, OR and NOT, because these expressions can be written using INTERSECT, UNION and MINUS operators.

Project

Given a relation A with attributes X, Y, . . . , Z (among others), the projection of relation A on X, Y, . . . Z — `A { X, Y, ..., Z }` — is a relation with:

- A heading derived from the heading of A by removing all attributes not mentioned in the set {X, Y, . . . , Z}, and
- A body consisting of all tuples {X:x, Y:y, . . . , Z:z} such that a tuple appears in A with X value x, Y value y, . . . , and Z value Z.

Join

Let relations A and B have headings {X, Y} and {Y, Z} respectively (where X, Y and Z are composite attributes for simplicity), and the attributes Y are the only ones common to the two relations. Then the natural join of A and B — `A JOIN B` — is a relation with heading {X, Y, Z} and body consisting of the set of all tuples {X:x, Y:y, Z:z} such that a tuple appears in A with X value x and Y value Y and a tuple appears in B with Y value y and Z value z.

Note: A Θ-join is a join between two relations on the basis of some comparison other than equality. The Θ-join of relation A on attribute X with relation B on attribute Y is the result of `( A TIMES B ) WHERE X Θ Y`. It is a relation with the same heading as the Cartesian product of A and B and with a body consisting of the set of all tuples t such that t appears in that Cartesian product and the condition X Θ Y evaluates to true for that tuple t.

Note: If Θ is =, the Θ-join is called an equijoin.

Note: `A JOIN B` is equivalent to
`( ( A TIMES ( B RENAME Y AS T ) ) WHERE Y = T ) { ALL BUT T }`
(B's Y has to be renamed because it has the same name as A's Y).

Divide

Given a relation A with heading {X}, a relation B with heading {Y} (X and Y are composite attributes for simplicity), and a relation C with heading {X, Y}, then the division of A by B per C — `A DIVIDE B PER C` — is a relation with heading {X} and body consisting of all tuples {X:x} such that a tuple {X:x, Y:y} appears in C for all tuples {Y:y} appearing in B. A is the dividend, B is the divisor, C is the mediator.

6.6 What is the algebra for?

The mentioned 8 operators are not a minimal set. Join, intersect and divide can be defined in terms of the other five. These five are a **primitive** or minimal set.

The algebra is not only used for data retrieval. It allows the **writing of relational expressions**, intended to be used for a variety of purposes, such as

- Defining a scope for retrieval
- Defining a scope for update
- Defining integrity constraints
- Defining derived relvars
- Defining stability requirements
- Defining security constraints

It serves also as a convenient basis for **optimization** by transforming relational expressions.

6.7 Additional operators

Many other operators can be defined, such as SEMIJOIN, SEMIMINUS, EXTEND, SUMMARIZE, and TCLOSE.

Semijoin

`A SEMIJOIN B ≡ ( A JOIN B ) { X, Y }`

Semidifference

```
A SEMIMINUS B   ≡   A MINUS ( A SEMIJOIN B )
```

Extend

```
EXTEND A ADD exp AS Z
```

is a relation with heading equal to the heading of A extended with the new attribute Z, and with a body consisting off all tuples t such that t is a tuple of A extended with a value for the new attribute Z that is computed by evaluating the expression exp on that tuple of A.

Summarize

```
SUMMARIZE A PER B ADD summary AS Z
```

is defined as follows:

- B must be off the same type as some projection of A. Let the attributes be A1, . . . , An.
- The heading of the result consists of the attributes A1, . . . , An plus the new attribute Z.
- The body of the result consists of all tuples t such that t is a tuple of B extended with a value for the new attribute Z. That new Z value is computed by evaluating summary over all tuples of A that have the same values for A1, . . . , An as does tuple t.

Tclose

This stands for transitive closure.

Let A be a binary relation with attributes X and Y of the same type T. The transitive closure of A, `TCLOSE A`, is a relation A^+ with heading the same as that of A and body a superset of that of A, defined as follows: the tuple {X:x, Y:y} appears in A^+ if and only if it appears in A or there exists a sequence of values z1, . . . , zn, all of type T, such that the tuples {X:x, Y:z1}, {X:z1, Y:z2}, . . . , {X:zn, Y:y} all appear in A.

6.8 Grouping and ungrouping

Attributes of relations can have values that are in turn relations. GROUP and UNGROUP are operators to group attributes into one attribute.

Chapter 7

Relational Calculus

7.1 Introduction

The relational algebra is **prescriptive**, it tells the system how to construct a relation from given relations. The relational calculus is **descriptive**, it tells the system what the result must look like.

These distinctions are only superficial. In reality the algebra and the calculus are logically equivalent.

Range variables are a fundamental feature of the calculus. A **range variable** is a variable whose permitted values are tuples of a specified relation.
The original calculus is also known as the tuple calculus.

The domain calculus is a calculus in which the range variables range over domains instead of relations.

7.2 Tuple calculus

A range variable definition looks like

```
RANGEVAR <range variable name>
RANGES OVER <relational expression commalist> ;
```

A relational operation looks like

```
<relational operation> ::= <proto tuple> [ WHERE <boolean expression> ]
```

It evaluates to a relation containing every possible value of the `<proto tuple>` for which the `<boolean expression>` specified in the WHERE clause evaluates to true.

A boolean expression is called a well-formed formula or a WFF.

Every reference to a range variable is free or bound.
If p is a WFF in which range variable V is free, then `EXISTS V (p)` and `FORALL V (p)` are WFFs and V is bound in both of them.
In a WFF a bound variable is a dummy variable and can be replaced without changing the meaning.
A WFF in which all variable references are bound is called a closed WFF (a proposition). It always evaluates to true or false.

Every free variable reference in the WFF must also be mentioned in the proto tuple.

A WFF can contain quantifiers. There are two quantifiers, EXISTS and FORALL. The first means: there exists at least one value of V that makes p evaluate to true. The second means: for all values of V, p evaluates to true.
An EXISTS is an iterated OR.

`EXISTS V ( p ( V ) )` is equivalent to

`false OR p ( t1 ) OR ...  OR p ( tm)`

A FORALL is an iterated AND:

`FORALL V ( p ( V ) )` is equivalent to

`true AND p ( t1 ) AND ...  AND p ( tm)`

(If V ranges over a relation with tuples t1, . . . , tm.)

7.3 Examples

A WFF is in **prenex normal form** if all quantifiers appear at the front of the WFF.

7.4 Calculus vs. algebra

Any given expression of the calculus can be written as a semantically equivalent expression of the algebra. An informal algorithm:

1. For each range variable, retrieve the range, restricted if possible.
2. Construct the Cartesian product of these ranges.
3. Restrict this Cartesian product in accordance with the "join portion" of the WHERE clause.
4. Apply the quantifiers from right to left.
   - For EXISTS RX (where RX ranges over r), project the intermediate result to eliminate all the attributes of relation r.
   - For FORALL RX, divide the intermediate result by the restricted range relation associated with RX as retrieved in 1.
5. Project the result in accordance with the specifications in the proto tuple.

Chapter 8

Integrity

8.1 Introduction

Integrity refers to the correctness or accuracy of data in the database.

A **constraint** is a boolean expression that must not evaluate to false. When a new constraint is declared, it is only accepted if the database currently satisfies the constraint.

Categories of constraints are:

- Type constraint: specifies legal values for a given type
- Attribute constraint: legal values for a given attribute
- Relvar constraint: for a given relvar
- Database constraint: for a given database

8.2 Type constraints

A type constraint is an enumeration of the legal values of the type. It is included in the definition of the type.

Conceptually a type constraint is checked during the execution of a selector invocation. No relvar can ever acquire a value for any attribute in any tuple that is not of the appropriate type.

8.3 Attribute constraints

An attribute constraint is a declaration that an attribute is of a specified type. It is part of the definition of the attribute.

8.4 Relvar constraint

A relvar constraint is a constraint on an individual relvar. They are always checked immediately when the relvar is updated. Statements that try to assign a value to the relvar that violate the constraint are rejected.

8.5 Database constraints

Is a constraint that interrelates two or more relvars. They are checked at COMMIT-time, i.e. at the end of a transaction. If a database constraint is violated, the transaction is rolled back.

8.6 The Golden Rule

The Golden Rule states:

> *No update operation must ever be allowed to leave any relvar in a state that violates its own predicate.*

By predicate the internal predicate is meant. The **relvar predicate** is the logical AND of all relvar constraints that apply to that relvar.

A database also has a predicate, the **database predicate**, which is the logical AND of all database and relvar constraints that apply to the database. The Golden Rule can be extended to:

> *No update operation must ever be allowed to leave any relvar in a state that violates its own predicate. Likewise, no update transaction must ever be allowed to leave the database in a state that violates its own predicate.*

8.7 State vs. transition constraints

These are constraints on transitions from one correct state to another. Only relvar and database constraints exist.

8.8 Keys

A key is a set of attributes.

K is a **candidate key** for relvar R if and only if it possesses the following properties:

- Uniqueness: No legal value of R ever contains two distinct tuples with the same value for K.
- Irreducibility: No proper subset of K has the uniqueness property.

Every relvar has at least one candidate key. A candidate key can be simple or composite (consisting of more than one attribute).

A **superkey** is a superset of a candidate key. It has the uniqueness property but not necessarily the irreducibility property.

Exactly one candidate key must be chosen to be the **primary key**. Then the others are called **alternate keys**.

FK is a **foreign key** in relvar R2 if

- There exists a relvar R1 with a candidate key CK

- For all time, each value of FK in the current value of R2 is identical to the value of CK in some tuple in the current value of R1.

A foreign key value represents a **reference** to the tuple containing the matching candidate key (the **referenced tuple**). Ensuring that the database does not contain invalid values for foreign keys is known as **referential integrity**. The constraint that values of a given foreign key must match values of the corresponding candidate key is a **referential constraint**. The relvar that contains the foreign key is the **referencing relvar**, the relvar that contains the corresponding candidate key is the **referenced relvar**.

A relvar can be **self-referencing**. This is a special case of a **referential cycle**.

Referential integrity rule: The database must not contain any unmatched foreign key values.

Referential actions
When you delete a tuple from a referenced relvar, there are certain actions the system can perform.

- CASCADE: Deletes matching tuples in the referencing relvar as well.
- RESTRICT: Only delete when there are no matching tuples in the referencing relvar.
- NO ACTION: Nothing more is done.

The same actions apply to UPDATE operations. The user can also specify his own actions to be performed on DELETE or UPDATE.

Chapter 9

Views

9.1 Introduction

A **view** is just a named expression of the relational algebra, the **view-defining expression**. It is a **derived**, **virtual** relvar. Users can operate on a view is if it were a real relvar.

A view can be defined as follows:

```
VAR <relvar name> VIEW <relational expression>
<candidate key definition list> ;
```

9.2 What are views for?

- Automatic security for hidden data.
- A shorthand or "macro" capability.
  The system expands the name of a view to its definition.
- Views allow the same data to be seen by different users in different ways at the same time.
- Logical data independence.
  This is possible when the database is restructured but the new database is information-equivalent with the old one.

There are two important principles regarding base and derived relvars.
The Principle of Interchangeability states that there must be no arbitrary and unnecessary distinctions between base and derived relvars. This implies that all views *must* be updatable.
The Principle of Database Relativity states that the choice of which database is the "real" one is arbitrary, so long as all the choices are information-equivalent.

9.3 View retrievals

Conceptually, a retrieval is the result of **materializing** a copy of the relation that is the current value of the view and applying the retrieval operation to this copy. This technique is inefficient and can only be used for retrievals, not for updates. In practice a substitution is used. The view's name

is substituted by the view-defining expression and an equivalent retrieval operation is performed directly on the database.

9.4 View updates

The problem of view update is to find an equivalent update operation directly on the database.

The Golden Rule applies to views as well, so we need a set of predicate inference rules to find the predicate for a view in terms of the predicates of the underlying base relvars.

View updating principles:

- View updatability must not depend on the particular syntactic form of the view definition.
- View updating must also work when the view is in fact a base relvar.
- There must be no ambiguity in the way a given update operation is implemented.
- Updating rules must take into account triggered procedures and referential actions.
- UPDATE can be regarded as a shorthand for DELETE-INSERT, but there is no checking of predicates or execution of triggered procedures in the middle of the DELETE-INSERT.
- All updates on views must be implemented by the same kind of update on the underlying relvars.
- The updating rules must be capable of recursive application. This is necessary if the underlying relvars of a view are in turn views.
- The rules cannot assume that the database is well-designed.
- There should be no reason for permitting some updates but not others on a given view.
- INSERT and DELETE are each other's inverse.

Note: In what follows it is always assumed that a new tuple to be inserted or a tuple to be updated must satisfy the relvar predicate of the view, and that a new tuple must not already appear in one of the underlying relvars. If not, the operation fails and nothing happens.

Union

The relvar predicate of A UNION B is (PA) OR (PB).

- **INSERT**: If the new tuple satisfies PA, it is inserted into A. If it satisfies PB, it is inserted into B, if it wasn't already inserted in B as a side-effect of inserting it into A.
- **DELETE**: If the tuple to be deleted appears in A, it is deleted from A. If it appears in B and wasn't already deleted as a side-effect of deleting it in A, it is deleted from B.
- **UPDATE**: If the tuple to be updated appears in A, it is deleted from A (without executing triggered procedures or checking relvar predicates). If it appears in B and wasn't already deleted as a side-effect of deleting it in A, it is deleted from B (also without executing triggered procedures or checking relvar predicates). Next, if the updated version of the tuple satisfies PA, it is inserted into A. If it satisfies PB, it is also inserted into B, if it wasn't already inserted as a side-effect of inserting it in A.

Intersect

The relvar predicate of A INTERSECT B is (PA) AND (PB).

- **INSERT**: The new tuple is inserted into A. If it (still) doesn't appear in B, it is inserted into B.
- **DELETE**: The tuple to be deleted is deleted from A. If it (still) appears in B, it is deleted from B.
- **UPDATE**: The tuple to be updated is deleted from A (without executing triggered procedures or checking relvar predicates). If it (still) appears in B, it is deleted from B (also without executing triggered procedures or checking relvar predicates). Next, the updated version of the tuple is inserted into A. If it (still) doesn't appear in B, it is inserted into B.

Difference

The relvar predicate of A MINUS B is (PA) AND NOT (PB).

- **INSERT**: The new tuple is inserted into A.
- **DELETE**: The tuple to be deleted is deleted from A.
- **UPDATE**: The tuple to be updated is deleted from A (without executing triggered procedures or checking relvar predicates). Next, the updated version of the tuple is inserted into A.

Restrict

The relvar predicate of A WHERE p is (PA) AND (p).

- **INSERT**: The new tuple is inserted into A.
- **DELETE**: The tuple to be deleted is deleted from A.
- **UPDATE**: The tuple to be updated is deleted from A (without executing triggered procedures or checking relvar predicates). Next, the updated version of the tuple is inserted into A.

Project

If A has composite attributes X and Y and {X:x} is a tuple of the projection A{X}, then the relvar predicate of A{X} is "There exists some value y from the domain of Y values such that the tuple {X:x,Y:y} satisfies PA".

- **INSERT**: The new tuple is (x) and the default value of Y is y. Then the tuple (x,y) is inserted into A.
- **DELETE**: All tuples of A with the same X value as the tuple to be deleted are deleted from A.
- **UPDATE**: The tuple to be updated is x and the updated version is x'. Let a be a tuple of A with the same X value x, and let the value of Y in a be y. Then all such tuples a are deleted from A (without executing triggered procedures or checking relvar predicates). Next, for each such value y, the tuple (x',y) is inserted into A.

Extend

The relvar predicate of EXTEND A ADD exp AS X is PA(a) AND e.X = exp(a).
Here e is a tuple of the view and a is the tuple that remains when e's X component is removed.

- **INSERT**: The tuple that is derived from the tuple to be inserted by projecting away the X component is inserted into A.
- **DELETE**: The tuple that is derived from the tuple to be deleted by projecting away the X component is deleted from A.
- **UPDATE**: The tuple that is derived from the tuple to be updated by projecting away the X component is deleted from A (without executing triggered procedures or checking relvar predicates). Next, the tuple that is derived from the updated version of the tuple by projecting away the X component is inserted into A.

Join

If relvars A and B have headings {X,Y} and {Y,Z} and for a given tuple of the join of A and B, a is the A portion of that tuple and b is the B portion, then the relvar predicate of A JOIN B is PA(a) AND PB(b).

- **INSERT**: The A portion of the new tuple is inserted into A. The B portion of the new tuple is inserted into B.
- **DELETE**: The A portion of the new tuple is deleted from A. The B portion of the new tuple is deleted from B.
- **UPDATE**: The A portion of the tuple to be updated is deleted from A (without executing triggered procedures or checking relvar predicates) and the B portion is deleted from B. Next, the A portion of the updated version of the tuple is inserted into A, and the B portion is inserted into B.

There are three different cases:

- **One-to-one**: There is an integrity constraint in effect that guarantees that for each tuple of A there is at most one matching tuple in B and vice versa.
- **One-to-many**: There is an integrity constraint in effect that guarantees that for each tuple of B there is at most one matching tuple in A.
- **Many-to-many**: There is no integrity constraint in effect.

9.5 Snapshots (a digression)

Snapshots are derived relvars (like views), but they are real, not virtual. They are a materialized copy of the data.

Chapter 10

Functional Dependencies

10.2 Basic definitions

Definition of **functional dependence**:
Let R be a relation variable, X and Y arbitrary subsets of the set of attributes of R. Y is functionally dependent on X, `X → Y`, if and only if, in every possible legal value of R, each X value has associated with it precisely one Y value.
In other words: when two tuples have the same X value, they also have the same Y value.

The right-hand side of an FD is called the **dependent**, the left-hand side is called the **determinant**.

Consequences:

- If X is a candidate key of relvar R, all the attributes Y of relvar R must be functionally dependent on X.
- If relvar R satisfies A → B (not trivial) and A is not a candidate key, then R will involve some redundancy.

10.3 Trivial and nontrivial dependencies

A dependency is **trivial** if and only if it is impossible that that dependency is not satisfied. The FD is trivial if the rhs is a subset of the lhs.

10.4 Closure of a set of dependencies

The set of all FDs that are implied by a given set S of FDs is called the **closure** of S, S^+.

Armstrong's axioms:

1. **Reflexivity**: If A is a subset of B, then A → B
2. **Augmentation**: If A → B, then AC → BC
3. **Transitivity**: If A → B an B → C, then A → C

All FDs implied by a set S of FDs and no other can be derived from S using these rules.

Other rules:

4. **Self-determination**: $A \rightarrow A$
5. **Decomposition**: If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$
6. **Union**: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$
7. **Composition**: If $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$

General Unification Theorem:

8. If $A \rightarrow B$ and $C \rightarrow D$, then $A \cup (C - B) \rightarrow BD$

10.5 Closure of a set of attributes

Given a relvar R, a set of attributes Z of R, and a set S of FDs for R. The set of all attributes of R that are functionally dependent on Z is the **closure Z^+ of Z under S**.

An algorithm for determining this closure:

```
CLOSURE[Z,S] := Z;
do "forever" {
  for each FD X -> Y in S {
    if X subset of CLOSURE[Z,S] {
      CLOSURE[Z,S] := CLOSURE[Z,S] union Y;
    }
  }
  if CLOSURE[Z,S] did not change on this iteration
    break;
}
```

Corollary:

- Given a set S of FDs, $X \rightarrow Y$ follows from S if and only if Y is a subset of the closure X^+ of X under S.
- K is a superkey if and only if the closure K^+ of K under the given set of FDs is precisely the set of all attributes of R.

10.6 Irreducible sets of dependencies

If every FD implied by S1 is implied by S2, i.e. if $S1^+$ is a subset of $S2^+$, then S2 is a **cover** for S1.

If S2 is a cover for S1 and S1 is a cover for S2, i.e. $S1^+ = S2^+$, then S1 and S2 are **equivalent**.

A set S of FDs is **irreducible** if and only if it satisfies these three properties:

- The rhs (dependent) of every FD in S involves just one attribute.

- The lhs (determinant) of every FD in S is irreducible: no attribute can be discarded from the determinant without changing the closure S^+. Such an FD is left-irreducible.
- No FD in S can be discarded from S without changing the closure S^+.

A set I of FDs that is irreducible and equivalent to some other set S of FDs is an **irreducible equivalent** of S. Every set of FDs has at least one irreducible equivalent set of FDs.

An irreducible equivalent can be found without computing S^+.

- Use the decompostion rule to make sure that every rhs consists of only one attribute.
- For each FD f in S, examine each attribute A in the lhs of f. If deleting A from the lhs has no effect on the closure S^+, it can be deleted from the lhs of f.
- For each FD f remaining in S, if deleting f from S has no effect on the closure S^+, f can be deleted from S.

Chapter 11

Further Normalization I: 1NF, 2NF, 3NF, BCNF

11.1 Introduction

A relvar is in a particular **normal form** if it satisfies a prescribed set of conditions.

Every relvar at a given level of normalization is automatically at all lower levels also.

A procedure for replacing a relvar in a given normal form by a set of relvars in a higher normal form is called the **normalization procedure**. This procedure is reversible. This means that the normalization process is information-preserving.

11.2 Nonloss decomposition and functional dependencies

The normalization procedure decomposes relvars into other relvars and this decomposition is reversible. No information is lost in the process. This is called **nonloss (lossless) decomposition**.

The process of decomposition is actually a process of projection. Reversibility then means that the original relvar is equal to the join of its projections.

Heath's theorem:
Let R{A,B,C} be a relvar, where A, B, and C are sets of attributes. If R satisfies the FD A $\rightarrow$ B, then R is equal to the join of its projections on {A,B} and {A,C}.

11.3 First, second, and third normal forms

Note: In this section it is assumed that every relvar has only candidate key, which is the primary key.

First normal form

A relvar is in 1NF if and only if, in every legal value of that relvar, every tuple contains exactly one value for each attribute.

By definition relvars are always in 1NF.

If a relvar is only in 1NF and not in a higher NF, then there is **redundancy** in the relvar. This results in **update anomalies**. This is caused by the fact that too much information is bundled into one relvar. The solution is to place logically separate information into separate relvars.

Second normal form

A relvar is in 2NF if and only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key.

A relvar that is in 2NF but not in a higher NF will still have some update anomalies. Again this is caused by the bundling of too much information into one relvar.

Third normal form

A relvar is in 3NF if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key.

"No transitive dependencies" means that there are no mutual dependencies between nonkey attributes.

11.4 Dependency preservation

In a decomposition, two projections are independent if updates can be made to either one without regard for the other (except for referential constraints). In other words, no database constraints are necessary, only two simple relvar constraints involving the uniqueness of the primary keys.

A relvar that cannot be decomposed into independent projections is said to be **atomic**. This does not mean that every relvar *should* be decomposed into atomic relvars.

Rissanen has shown that projections R1 and R2 of a relvar R are independent if and only if

- Every FD in R is a logical consequence of the FDs in R1 and R2, and
- The common attributes of R1 and R2 form a candidate key for at least one of the pair.

The idea that the normalization procedure should decompose relvars into projections that are independent is known as **dependency preservation**.

11.5 Boyce/Codd normal form

This NF deals with relvars that have more than one candidate keys, such that they are composite and overlap. It is a stronger definition than that of the 3NF. For relvars where these mentioned conditions are not satisfied, BCNF is equivalent to 3NF.

Boyce/Codd normal form

A relvar is in BCNF if and only if every nontrivial, left-irreducible FD has a candidate key as its determinant.

Less formally: A relvar is in BCNF if and only if the only determinants are candidate keys.

Note: The two objectives, decomposing a relvar into BCNF components, and decomposing it into independent components, are sometimes in conflict.

Chapter 12

Further Normalization II: Higher Normal Forms

12.2 Multi-valued dependencies and fourth normal form

Multi-valued dependencies (MVDs) are a generalisation of FDs. A $\rightarrow\rightarrow$ B means: "B is multi-dependent on A" or "A multi-determines B".

The formal definition:
Let R be a relvar, and let A, B, and C be subsets of the attributes of R. Then we say that B is multi-dependent on A, A $\rightarrow\rightarrow$ B, if and only if, in every possible legal value of R, the set of B values matching a given (A value, C value) pair depends only on the A value and is independent of the C value.

If for a relvar R$\{$A,B,C$\}$ the MVD A $\rightarrow\rightarrow$ B holds, then also the MVD A $\rightarrow\rightarrow$ C holds. Notation: A $\rightarrow\rightarrow$ B $|$ C.

An FD is a MVD in which the set of dependent (rhs) values matching a given determinant (lhs) value is always a singleton set.

Stronger version of Heath's theorem (Fagin)
Let R$\{$A,B,C$\}$ be a relvar, where A, B, and C are sets of attributes. then R is equal to the join of its projections on $\{$A,B$\}$ and $\{$A,C$\}$ if and only if R satisfies the MVDs A $\rightarrow\rightarrow$ B $|$ C.

Fourth normal form

A relvar R is in 4NF if and only if, whenever there exists a subset A and B of the attributes of R such that the nontrivial MVD A $\rightarrow\rightarrow$ B is satisfied, then all attributes of R are also functionally dependent on A.

Two or more independent relation valued attributes in a relvar can be eliminated by separating them. The relvars are then in 4NF.

12.3 Join dependencies and fifth normal form

Some relvars cannot be nonloss decomposed into 2 projections. These relvars are called n-decomposable ($n > 2$).

A **join dependency** (JD) is the most general form of a dependency.

Let R be a relvar, and let A, B, ..., Z be subsets of the attributes of R. Then we say that R satisfies the JD $\star$ {A, B, ..., Z} if and only if every possible legal value of R is equal to the join of its projections on A, B, ..., Z.

An MVD is a special case of a JD: $A \rightarrow\rightarrow B \mid C \equiv \star$ {AB,AC}

Fifth normal form

A relvar R is in 5NF (projection-join normal form PJ/NF) if and only if every nontrivial join dependency that holds for R is implied by the candidate keys of R.

A given JD $\star$ {A,B,...,Z} is implied by candidate keys if and only if each of A, B, ..., Z is in fact a superkey for the relvar in question.

5NF is the ultimate normal form (when only using projection and join as operators), and is guaranteed to be free of anomalies that can be eliminated by taking projections.

12.4 The normalization procedure summarized

1. Take projections of the original 1NF relvar to eliminate any FDs that are not irreducible. This step will produce a collection of 2NF relvars.
2. Take projections of those 2NF relvars to eliminate any transitive FDs. This step will produce a collection of 3NF relvars.
3. Take projections of those 3NF relvars to eliminate any remaining FDs in which the determinant is not a candidate key. This step will produce a collection of BCNF relvars.
4. Take projections of those BCNF relvars to eliminate any MVDs that are not also FDs. This step will produce a collection of 4NF relvars. By separating independent RVAs such MVDs can be eliminated before steps 1–3.
5. Take projections of those 4NF relvars to eliminate any remaining JDs that are not implied by the candidate keys—if you can find them. This step will produce a collection of 5NF relvars.

(These projections must of course be nonloss, and preferably dependency-preserving.)

Chapter 14

Recovery

14.1 Introduction

Recovery means restoring the database to a state that is known to be consistent after a failure has rendered the current state inconsistent or at least suspect. This can be achieved by adding redundancy at the physical level.

14.2 Transactions

A **transaction** is a logical unit of work. It's a sequence of database operations that transforms the database from one consistent state to another, without necessarily preserving consistency in intermediate points.

A **transaction manager** makes sure that a transaction is executed entirely or not at all. In case of a failure in the middle of a transaction, the already executed updates have to be undone. There are two important operations: **COMMIT** and **ROLLBACK**.

- COMMIT signals a successful end-of-transaction. The updates can be made permanent.
- ROLLBACK signals an unsuccessful end-of-transaction. All updates made so far have to be undone.

For all this to be possible, it is necessary that the DBMS maintains a **log** or **journal** which records all the updates, in particular before- and after-images of the updated object. There is an active log and an archive log.

The system must also guarantee that all individual statements are atomic.

14.3 Transaction recovery

A transaction begins with a BEGIN TRANSACTION statement and ends with a COMMIT or a ROLLBACK statement. A COMMIT establishes a **commit point**, a point at which the database is in a consistent state. ROLLBACK rolls the database back to the state of the previous commit point. (Here the term "database" means just that portion of the database accessed by the transaction.)

This is what happens when a commit point is established:

1. All updates since the previous commit point are made permanent.
2. All database positioning is lost and all tuple locks are released. (This also happens after ROLLBACK.)

Transactions are not only the unit of work, but also the **unit of recovery**, the unit of concurrency, and the unit of integrity.

By the time a transaction commits, the log must already be physically written. Thus, when the system crashes before the updates are written to the physical database, the restart procedure can determine which updates must be written to the database first. This is called the **write-ahead log rule**.

The ACID properties

- **Atomicity**: All or nothing
- **Consistency**: A transaction transforms a consistent state of the database into another consistent state.
- **Isolation**: A transaction's updates are concealed from all the other transactions currently running, until that transaction commits.
- **Durability**: Once a transaction commits, its updates survive in the database, even if there is a subsequent system crash.

14.4 System recovery

- **Local failures**: affect only an individual transaction.
- **Global failures**:
  - **System failures**: affect all transaction in progress, but does not physically damage the database (soft crash).
  - **Media failures**: cause physical damage to the database (hard crash).

System failures

The problem is that the contents of main memory are lost. The solution is taking checkpoints at prescribed intervals. When the system takes a **checkpoint**, it writes the contents of the database buffers to the physical database and writes a special **checkpoint record** to the physical log. The checkpoint record contains a list of all transactions that were in progress when the checkpoint was taken.

When the system fails, every transaction that was not finished at the time of the failure has to be **undone**, and every transaction that committed between the last checkpoint and the system failure has to be **redone**.

At restart time the system performs the following procedure:

1. Start with two lists of transactions, the UNDO list and the REDO list. UNDO equals the list of all transactions in the most recent checkpoint record. REDO is initially empty.

2. Search forward through the log, starting from the checkpoint record.
3. If a BEGIN TRANSACTION log entry is found for a transaction T, add T to the UNDO list.
4. If a COMMIT log entry is found for transaction T, move T from UNDO to REDO.
5. When the end of the log is reached, move backwards through the log and undo the transactions in the UNDO list. Then move forward again and redo the transactions in the REDO list.

14.5 Media recovery

The database is restored from a **backup** or **dump**. The log is then used to redo all the transactions that completed since the backup was taken.

14.6 Two-phase commit

This is used when a given transaction can interact with several independent resource managers. The transaction issues a system-wide COMMIT or ROLLBACK, which is handled by the **coordinator**. When there is a system-wide COMMIT, the coordinator goes through two phases:

1. The coordinator tells all the participants to write all its log entries to its physical log. The resource manager replies with “OK” or “Not OK”.
2. When the coordinator has received a reply from all participants, it writes its own decision to the physical log. If all replies were “OK” then the decision is “commit”, otherwise the decision is “rollback”. The coordinator then sends this decision to the participants and these have to do what they are instructed.

Chapter 15

Concurrency

15.1 Introduction

The term **concurrency** means that many transactions are allowed to access the same database at the same time.

15.2 Three concurrency problems

These problems are caused by the uncontrolled interleaving of operations from two different transactions, even if these transactions in itself are correct.

- **The lost update problem**
  Two transactions first retrieve a tuple and, in the same order, update that tuple based on the read value. The first update is lost.
- **The uncommitted dependency problem**
  Occurs when one transaction retrieves an uncommitted tuple, updated by a transaction that is afterwards rolled back. When a transaction updates an uncommitted tuple and the other transaction is rolled back, the update is lost.
- **The inconsistent analysis problem**
  One transaction is making an analysis while another transaction changes tuples that the first transaction is analyzing.

15.3 Locking

When a transaction needs assurance that an object (a tuple) that it is interested in will not change, it acquires a **lock** on that object.

There are two kinds of locks: **exclusive locks (X locks)** and **shared locks (S locks)**.

- If transaction A holds an X lock on tuple t, then a request from some distinct transaction B for an S or an X lock on t will be denied.
- If transaction A holds an S lock on tuple t, then a request from some distinct transaction B for an X lock on t will be denied, and for an S lock on t will be granted.

When a transaction's request is denied, it goes into a **wait state**.

Data access protocol or **locking protocol**:

1. A transaction that wishes to retrieve a tuple must first acquire an S lock on that tuple.
2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple. If it already holds an S lock on that tuple, it must promote that S lock to an X lock.
3. If a lock request from transaction B is denied because it conflicts with a lock already held by transaction A, B will go into a wait state. B will wait until A's lock is released. (B must not wait forever: a **livelock**).
4. X and S locks are held until end-of-transaction (COMMIT or ROLLBACK).

Compatibility matrix

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

15.4 The three concurrency problems revisited

Now all problems are solved, but in the lost update problem and inconsistent analysis problem a new problem arises: **deadlock**.

15.5 Deadlock

Deadlock is a situation in which two or more transactions are in a simultaneous wait state, each of them waiting for one of the others to release a lock before it can proceed.

Systems can detect deadlocks by finding cycles in the **Wait-For graph**. Some systems work with a timeout mechanism to detect deadlocks.

A deadlock can be broken by choosing one of the deadlocked transactions as the **victim** and rolling it back. Some systems then automatically restart the transaction. Other systems just send an exception code back to the application.

15.6 Serializability

A given execution of a set of transactions is considered to be correct if it is serializable, i.e. if it produces the same result as some serial execution of the same transactions, running them one at a time. Why?

- Individual transactions are assumed to be correct (they transform a correct state of the database into another correct state).
- Running the transactions one at a time in any serial order is therefore also correct.

- An interleaved execution is therefore correct if it is equivalent to some serial execution.

Given a set of transactions, any execution of these transactions is called a **schedule**. Executing the transactions one at a time is a **serial schedule**. A schedule that is not serial is an **interleaved schedule**. Two schedules are said to be equivalent if they are guaranteed to produce the same result, independent of the initial state of the database.

The **two-phase locking theorem**

> If all transactions obey the two-phase locking protocol, then all possible interleaved schedules are serializable.

The **two-phase locking protocol**

1. Before operating on an object, a transaction must acquire a lock on that object.
2. After releasing a lock, a transaction must never go on to acquire any more locks.

If some transaction A does not obey this rule, it is always possible to construct some other transaction B that can run interleaved with A in such a way as to produce a schedule that is not serializable and not correct.

15.7 Isolation levels

The **isolation level** is the degree of interference that a transaction is prepared to tolerate. To guarantee serializability, there must be **no** interference at all.

15.8 Intent locking

Not only tuples can be locked, but also relvars or even an entire database. We speak of **locking granularity**: The finer the granularity, the greater the concurrency. The coarser, the fewer locks need to be set and tested (lower overhead).

The **intent locking protocol** says that no transaction is allowed to acquire a lock on a tuple before first acquiring a lock on the relvar that contains it. Conflict detection becomes easier. Conflicts can now be seen at the relvar level.

We introduce three new locks that only apply to relvars: **intent locks**. They are **intent shared (IS)**, **intent exclusive (IX)** and **shared intent exclusive (SIX)**.

- **IS**: T intends to set S locks on tuples in R.
- **IX**: Same as IS, but T might update individual tuples in R and will therefore set X locks on those tuples.
- **S**: T can tolerate concurrent readers, but not concurrent updaters in R. T will not update any tuples in R.
- **SIX**: Combines S and IX: T can tolerate concurrent readers, but not concurrent updaters in R, but T might update tuples in R and will therefore set X locks on those tuples.

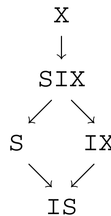
- **X**: T cannot tolerate any concurrent readers or writers in R. T might or might not update tuples in R.

1. Before a transaction can acquire an S lock on a tuple, it must first acquire an IS or stronger lock on the relvar containing that tuple.
2. Before a transaction can acquire an X lock on a tuple, it must first acquire an IX or stronger lock on the relvar containing that tuple.

Compatibility matrix

	X	SIX	IX	S	IS	-
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
-	Y	Y	Y	Y	Y	Y

Precedence graph



Lock escalation is implemented in many systems to balance high concurrency with low lock management overhead. When the system reaches a predefined threshold, it automatically replaces a set of fine-granularity locks by a single coarse-granularity lock.

Chapter 16

Security

16.1 Introduction

Security refers to the protection of data against unauthorized disclosure, alteration, or destruction.

- Security means protecting data against unauthorized users.
- Integrity means protecting data against authorized users.

There are two approaches:

- **Discretionary** control: the user has different access rights (**privileges**) on different objects.
- **Mandatory** control: each data object has a **classification** level and each user is given a certain **clearance** level.

The result of policy decisions must be made known to the system and must be remembered by the system (in the catalog). The checking of access requests is done by the DBMS's **security subsystem**. The system must be able to identify the requesting user (**user ID** and **password**).

16.2 Discretionary access control

Authorities define what is allowed. The general syntax is:

```
AUTHORITY <authority name>
GRANT <privilege commalist>
ON <relvar name>
TO <user ID commalist>
```

When a user makes an unauthorized request, the attempt is simply rejected. Sometimes an entry is added to a log or some other action may be necessary.

Authorities can be **value-independent** (the user can see a vertical subset of a base relvar), **value-dependent** (the authority is defined on a view), a **statistical summary** (the user cannot see individual data), and **context-dependent** (the authority has a WHEN clause).

An **audit trail** is a special file or database in which the system automatically keeps track of all operations performed by users on the regular data.

16.3 Mandatory access control

The rules (Bell and La Padula):

- User i can retrieve object j only if the clearance level of i is greater than or equal to the classification level of j.
- User i can update object j only if the clearance level of i is equal to the classification level of j.

Multi-level security
When the unit of data is a tuple, then every tuple has to be labeled with its classification level (in a hidden attribute CLASS). Every request `Q` is modified to `Q AND CLASS` $\leq$ `user clearance`. The hidden attribute is also part of the primary key. This kind of relvar is called a **multi-level relvar**. The fact that "the same" data looks different to different users is called **polyinstantiation**.

16.4 Statistical databases

A **statistical database** only allows queries that derive aggregated information but not queries that derive individual information. Sometimes it is possible to infer individual information from aggregated information. A solution to this problem is that the system must refuse to respond to queries for which the cardinality of the set to be summarized is less than some lower bound b or more than $N - b$. But this does not entirely solve the problem.

An **individual tracker** is a boolean expression that enables the user to track down information concerning an individual tuple. If the user knows a boolean expression BE that identifies an individual tuple I, and BE can be expressed in the form BE1 AND BE2, then the boolean expression BE1 AND NOT BE2 is a tracker for I.

For almost any statistical database a **general tracker** can always be found. A general tracker is a boolean expression that can be used to find the answer to any inadmissible query. Any expression with result set cardinality c in the range $2b \leq c \leq N - 2b$ is a general tracker.

16.5 Data encryption

This is a countermeasure against people who attempt to bypass the system.

Chapter 18

Missing Information

18.1 Introduction

Information is often missing in the real world. A lot of commercial products use **nulls** and **three-valued logic (3VL)**.

18.2 An overview of the 3VL approach

Boolean expressions

Any scalar expression in which either of the operands is UNK evaluates to the 'unknown' truth value unk. The usual operators are redefined, and a new operator is defined.

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

MAYBE	
t	f
u	t
f	f

EXISTS and FORALL

These operators can now also evaluate to unk.

Computational expressions

Any numeric expression evaluates to UNK if any of the operands are UNK.

UNK is not unk

unk is a truth value, UNK is not even a value at all.

Can a domain contain an UNK?

A relation that includes UNKs is no relation at all.

Relational expressions

Product is unaffected.

Restriction returns a relation whose body contains only those tuples for which the restriction condition evaluates to true, not false and not unk.

Projection: we need to have a new definition for **duplicates**:
Two tuples are duplicates of one another if and only if (a) they have the same attributes, and (b) for each such attribute, either the two tuples both have the same value, or they both have an UNK, in that attribute position.

The definition of **union** remains the same. Duplicate tuples in the result are eliminated according to the new definition of duplicates.

Difference: analogous to union.

Update operations

If attribute A of relvar R permits UNKs, and if a tuple is inserted into R and no value is provided for A, the system will place an UNK in the A position. If no UNK is allowed for A, it will lead to an error.

An attempt to create a duplicate tuple (with new definition) in R is as usual an error.

Integrity constraints

unk is regarded as false for the WHERE clause, but regarded as true for integrity constraints, because an integrity constraint is a boolean expression that must not evaluate to false.

18.3 Some consequences of the foregoing scheme

Expressions that always evaluate to true in 2VL, don't necessarily do so in 3VL.

Examples:

- x = x
- p OR NOT(p)
- r JOIN r does not necessarily give r
- INTERSECT is no longer a special case of JOIN
- A = B and B = C no longer imply A = C

18.4 Nulls and keys

Primary keys

Entity integrity: No component of the primary key of any base relvar is allowed to accept nulls.

Other relvars are allowed to have a primary key for which nulls are allowed. This violates the Principle of Interchangeability!

Foreign keys

Referential integrity: The database must not contain any unmatched keys.

An unmatched foreign key value now means a nonnull foreign key value in some referencing relvar for which there does not exist a matching value of the relevant candidate key in the relevant referenced relvar.

When foreign keys might except nulls, there can be another referential action: SET NULL.

18.5 Outer join

The **outer join** is an extended form of the inner join and differs from it in that tuples in one relation having no counterpart in the other appear in the result with nulls in the other attribute positions, instead of being ignored.

It is not primitive. An example:

```
 ( S JOIN SP )
UNION
( EXTEND ( ( S { S# } MINUS SP { S# } ) JOIN S ) ADD NULL AS P#, NULL AS QTY
)
```

There are varieties of outer join: the left, right, and full outer join. Left preserves information from the first operand, right from the second, and full from both operands.

18.6 Special values

Don't use nulls, but special values. A special value is a value from the applicable domain that is different from the regular values of that domain.

Chapter 20

Distributed Databases

20.2 Some preliminaries

A **distributed database system** consists of a collection of **sites**, connected together via a communications network, in which

- Each site is a full database system site in its own right
- The sites have agreed to work together so that a user at any site can access data anywhere in the network exactly is if the data were all stored at the user's own site.

Each site has its own DBMS and has a software component that provides the partnership functionality. These two together are called a **distributed database management system**.

Advantages of a distributed system:

- It connects *islands of information*, it mirrors the structure of an enterprise.
- Efficiency of processing: the data is kept close to where it is most frequently used.
- Increased accessibility: it is possible to access remote sites.

Disadvantage: complexity!!

The fundamental principle of distributed database:

> To the user, a distributed system should look exactly like a nondistributed system.

(Users refer to users who are performing data manipulation operations.)

There are 12 objectives:

1. Local autonomy
2. No reliance on a central site
3. Continuous operation
4. Location independence

5. Fragmentation independence
6. Replication independence
7. Distributed query processing
8. Distributed transaction management
9. Hardware independence
10. Operating system independence
11. Network independence
12. DBMS independence

20.3 The twelve objectives

Local autonomy

Operations at a given site are controlled by that site. No site should depend on some other site for its successful operation. Local data is locally owned and managed. Complete autonomy is impossible: sites should be autonomous **to the maximum extent possible**.

No reliance on a central site

All sites are treated as equals. A central site might be a bottleneck, and would be the single point of failure.

Continuous operation

- Greater **reliability** (the probability that the site is up and running at any given moment): the system can can continue to operate (at a reduced level) even if one of the sites is down.
- Greater **availability** (the probability that the site is up and running continuously throughout a specified period) because of data replication.

This applies only to unplanned shutdowns because planned shutdowns should never be required!

Location independence

Users should not have to know where data is physically stored. This simplifies user programs and terminal activities.

Note: Every objective in this section with *independence* in its name is a form of **data independence**.

Fragmentation independence

Data fragmentation: a relvar can be divided up into fragments for physical storage purposes. Data can then be stored at the location where it is most frequently used to improve performance.

There are two kinds of fragmentation:

- Horizontal: corresponds to restriction and union (must be an orthogonal decomposition).
- Vertical: corresponds to projection and join (must be a nonloss decomposition).

For vertical fragmentation a hidden attribute 'tuple ID' or TID may be required.

Users should be able to behave as if the data is not fragmented at all. This simplifies user programs and terminal activities. Fragmentation can happen at any time to improve performance. The system optimizer has to determine which fragments need to be physically accessed to satisfy a user request.

Updating operations on fragments are the same as on join and union views: tuples can migrate from one fragment to another.

Replication independence

Data replication: a relvar (or fragment) can be represented by many distinct copies (**replicas**) stored at many distinct sites. Replication improves performance and availability. The disadvantage is the problem of update propagation.

Users should be able to behave as if the data is not replicated at all. This simplifies user programs and terminal activities. Replicas can be made and destroyed any time to meet performance requirements. The system optimizer has to determine which replicas need to be physically accessed to satisfy a user request.

Distributed query processing

- Relational systems (set-level retrieval) are better than nonrelational ones (record-at-a-time retrieval) because they significantly reduce the number of messages sent through the network.
- Optimization is very important to reduce the network traffic. Optimization is only possible with relational requests.

Distributed transaction management

An **agent** is the process performed on behalf of a given transaction at a given site. Each transaction consists of several agents.

Transactions use the **two-phase commit** protocol to ensure atomicity. Concurrency control is based on **locking**.

Hardware independence

Operating system independence

Network independence

DBMS independence

Strict homogeneity is not really necessary, but the different DBMSs all have to support the same interface.

20.4 Problems of distributed systems

Main problem: **networks are slow**
Main objective: **minimize network utilization**

Query processing

First the optimizer of the site where the request originated performs a **global optimization**. Then every participating site performs a **local optimization**.

Catalog management

Where should the catalog be stored?

- Centralized: violates objective 2.
- Fully replicated: violates objective 1.
- Partitioned: non-local requests are very expensive.
- Combination of centralized and partitioned: violates objective 1.

Real database systems use none of these approaches. Another approach uses **object naming**. An object has a **printname** that users use, and a unique unchangeable **system-wide name** that consists of the creator ID, the creator site ID, the local name and the birth site ID. It is also possible to define synonyms for objects in a synonym table.

Each site maintains a catalog entry for every object born at that site and a catalog entry for every object currently stored at that site. If an object was born at a site but moved to another, then the place it has moved to is added to the catalog of the site where it was born.

Update propagation

One copy is designated as the **primary copy**. The others are secondary copies. Primary copies of different objects are at different sites. Update operations are logically complete as soon as the primary copy is updated. The site holding that copy is responsible for propagating the update to the secondary copies, and this must happen before COMMIT.

This is a violation of the local autonomy objective. A lot of commercial products, on the other hand, don't propagate updates immediately, but this means that the database is not always in a consistent state.

Recovery control

Two-phase commit is used. Ever site must be able to act as the coordinator and as a participant. A participating site **must** do what it is told by the coordinator. This is a minor loss of local autonomy.

Some enhancements to the protocol can be made to reduce the number of messages that need to be sent.

- If the agent at a particular site is read-only, that participant can reply "ignore me" in phase one, and the coordinator can ignore it in phase two.
- If all participants reply "ignore me", phase two can be skipped.
- Presumed commit: participants must acknowledge "undo it" but not "do it" messages.
- Presumed rollback: participants must acknowledge "do it" but not "undo it" messages. (This is better than presumed commit.)

Concurrency control

For every update operation on a particular replica, 5 messages are necessary: a lock request, a lock grant, the update message, the acknowledgments, and an unlock request.

The number of messages can be reduced:

- Piggybacking
- A primary copy scheme: the site holding the primary copy will handle all locking operations.

Some problems arise:

- The primary copy scheme means a severe loss of autonomy.
- Global deadlocks: they are not detectable by the individual sites.

20.5 Client/server systems

A **client/server system** is distributed system in which

- Some sites are clients and some sites are servers.
- All data resides at the server sites.
- All applications execute at the client sites.
- Full location independence is not provided.

The client/server approach has implications for application programming. As much functionality as possible should be bundled in **set-level** requests to reduce the number of messages between client and server. The reduction is even greater when using **stored procedures** at the server site and invoking them using a remote procedure call (**RPC**). The advantages are:

- Improved performance
- Data independence
- One stored procedure can be shared by many clients
- Optimization at creation time
- Better security

Disadvantage: DBMS dependence.

20.6 DBMS independence

Gateways can be used to make two different DBMSs "understand" each other, to make one DBMS "look like" the other.

Data access middleware is used to enable a client to make requests to many different DBMSs through the data access middleware software. Such a system is not a full distributed system. It is called a **federated system**, or a **multi-database system**.