SUMMARY DATABASES

THIRD EDITION

Andy Verkeyn

1996

PREFACE

Hi there, this summary is intended for students following the courses “Databases I” and “Databases II”, taughed by Prof. dr. R. De Caluwe. This is only a summary which means that you should read the book (and notes) at least once before studying this text. This summary covers all the material that is found in the main book and related notes. I cannot be held responsible for any missing or wrong information in this summary : using it is at your own risk !

Good luck,
Andy Verkeyn.
8 juli 1996.

Notes on the summary :

- Chapters that are only very briefly summarized :
  * Chapter II.5 : The SQL language
  * Chapter III.2 : Semantic modeling
- Subjects that are NOT summarized :
  * The Object Database Standard
  * An example of an ODBMS : ONTOS
  * Temporal databases
  * Fuzzy databases
  * Concepts from the panel conversation
  * Database machines

Sources for this summary :

- “An introduction to database systems”, C.J. Date, fifth edition
- “An introduction to database systems”, C.J. Date, sixth edition
- “Fundamentals of database systems”, Elmasri / Navathe

I. BASIC CONCEPTS

I.1. AN OVERVIEW OF DATABASE MANAGEMENT

I.1.1. DATABASE SYSTEM

I.1.1.1. DEFINITIONS

- Data and information :
  * Data : values stored in the database, f.i. 1995, Jan, 31/3/1975,...
  * Information : meaning of those values, f.i. year, name, birthday,...

- A database system is a computerized system whose purpose is to maintain information and to make that information available on demand (= a computerized record keeping system).

- A database consists of files that consists of records, and these records consists of fields.

I.1.1.2. COMPONENTS

- Data :
  * Data is integrated :
    The database can be thought of as a unification of several distinct data files, with any redundancy among those files wholly or partly eliminated (redundancy can be reduced, or at least, controlled).
  * Data can be shared :
    In a multi-user-system ($\leftrightarrow$ single-user-system), user can have access to the same data at the same time, for different purposes ($\rightarrow$ concurrency).
  * Data interchange is possible, because standards can be enforced.
  * Integrity can be maintained : impossible values cannot be entered.
  * Inconsistency can be avoided (different values for the same fact)
    $\rightarrow$ Propagating update : If one value is updated, the others change automatically.
  * Security restrictions can be applied
  * Conflicting requirements can be balanced :
    The fastest access is guaranteed for the most important application.
  * Data independence can be provided :
    Immunity of applications to change in storage structure and access technique.
    $\rightarrow$ a logical record doesn't have to correspond with an identical stored record
    Things that can change in storage :
    - representation of numeric data (binary, decimal), character data (ASCII, EBCDIC)
    - units for numeric data (inches, centimeters,...)
    - data encoding (coded values for colors)
    - data materialization :
      - direct : A logical field correspond with some stored field.
      - virtual :
        A logical field value is a computation performed on a collection of stored fields (but isn't stored itself) (f.i., a total quantity).
    - structure of stored records (adding or removing a field)
    - structure of stored files (indexes, number of storage devices,...)

- Hardware :
  * processor and main memory : support the execution of the database system software.
  * secondary storage device : hold the stored data.

- Software :
  * DataBase Management System (DBMS) : support the user operations
  * DataCommunication manager (DC manager) :
    Transmit messages (f.i., between two different rooms) from the users to the DBMS.

- Users :
  * application programmers :
    Responsible for writing application programs that uses the database (in COBOL, PL/I, C,...). All these functions are supported by the DBMS.
  * end users :
    They use the applications (written by an application programmer) or the interface (built-in applications) :
    - command-driven-interface : an interactive query language (f.i., SQL)
    - menu-driven-interface (= form-driven-interface)
  * Data Administrator (DA) :
    Decides what data should be stored in the database and establishes policies for maintaining the data (f.i., who may have access to what data).
  * DataBase Administrator (DBA) : creates the database and implements the policies

I.1.2. DATABASE

- Persistent :
  Database data differs in kind from data, such as input data, intermediate results, output data,... which are transient.
  * Input data : Becomes persistent after it is entered.
  * Output data :
    Is derived from persistent data but, theoretical, doesn't belong to the database anymore.

- Definition :
  A database is a collection of persistent data that is used by the application system of some given enterprise.

I.2. AN ARCHITECTURE FOR A DATABASE SYSTEM

I.2.1. EXTERNAL LEVEL

- The external level consists of many external views (= individual user views), which represents the part of the data seen by that particular user.

- Each user has a language :
  * application programmers : conventional language (PL/I, COBOL,...)
  * end users : query language (SQL) or other interface

- Each language consists of two components :
  * host language : functions for I/O, calculations,... (PL/I, COBOL, C,...)
  * DataSubLanguage (DSL) : functions for database access (SQL)
    - Data Definition Language (DDL) : to define/declare a database object
    - Data Manipulation Language (DML) : to transfer information to/from the database

- An external view (consisting of external records = logical records) is defined by an external schema.

I.2.2. CONCEPTUAL LEVEL

- The conceptual level consists of the conceptual view (= community user view), which represents the data as it really is (= the entire database).

- The conceptual view (consisting of conceptual records) is defined by the conceptual schema (which also contains the security rules, the integrity rules,...).

I.2.3. INTERNAL LEVEL

- The internal level consists of the internal view (= storage view = stored database), which represents a low-level representation of the entire database.

- The internal view (consisting of internal records = stored records) is defined by the internal schema (= storage structure definition) (which also specifies what indexes exist,...).

- NOT the physical records

I.2.4. MAPPINGS

They define the correspondence between the different levels (f.i., other field types, field and record names,...).

I.2.5. FUNCTIONS OF THE DBMS

- Data Definition Language (DDL) :
  To accept external schemas, the conceptual schema and the internal schema (These schemas and mappings are built and maintained by the DBA).

- Data Manipulation Language (DML) :
  To retrieve, update, delete,... existing data :
  * planned requests : from applications
  * unplanned requests : from interactive query language

- Data recovery and concurrency

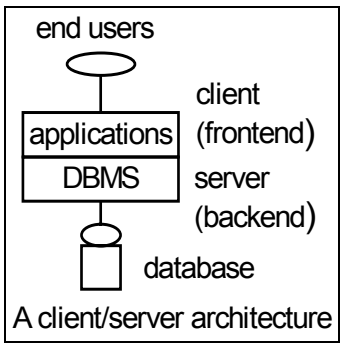
- Data security and integrity

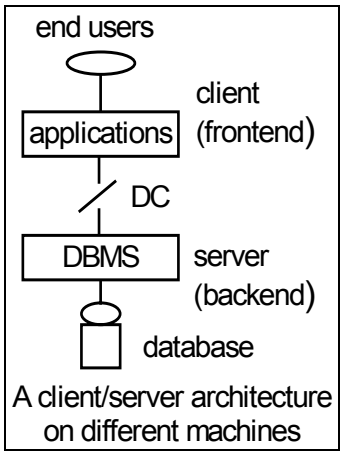
- Data dictionary : a system database with the schemas, mappings, files, records, fields,...

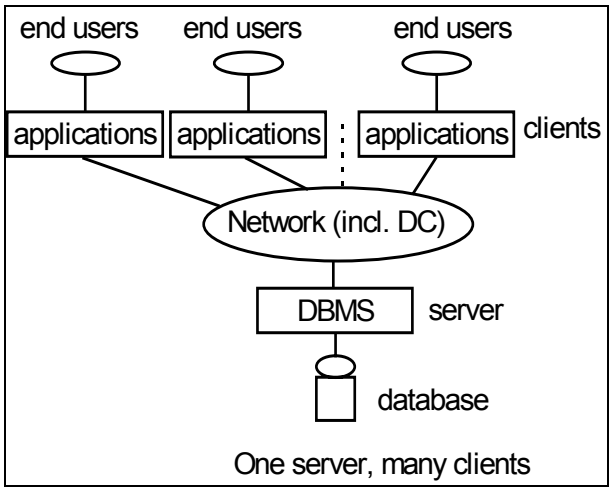
- Performance : all operations must be as efficiently as possible

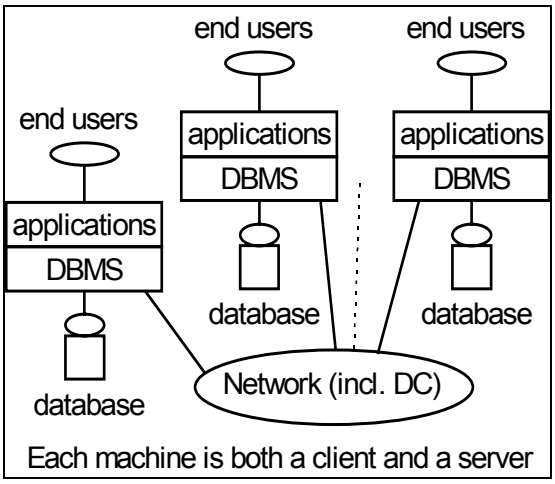
→ The DBMS must provide the user-interface at the external level.
→ Applications that operate directly on the internal level are called utilities and are best considered as a part of the DBMS (f.i. reorganization tools, analysis tools,...).

I.2.6. CLIENT/SERVER ARCHITECTURE









- Client/server : A client can have access to many servers, but only one at a time.
- Distributed database :
  A client can have access to many servers simultaneous (all the servers are seen as one by the client).

I.3. AN INTRODUCTION TO DATABASES

I.3.1. HIERARCHIC MODEL

- The data is represented to the user in the form of a set of tree structures.
- The entities have a typical parent-child relation, each parent can have many children, but a child can only have one parent.

I.3.2. NETWORK MODEL

This extension of the hierarchic model allows entities to have many parents also.

I.3.3. RELATIONAL MODEL

- The data is seen by the user as tables (= files), relations are made by common columns (= fields). Records are called rows in the relational model.

- The operators at the user's proposal generate new tables from old (property of closure).
  * The output from an operator can become input for another.
  * At least this is true at the external and conceptual level, not at the internal level, which means that some tables, f.i. intermediate results, are not materialized (storage level details are hidden from the user).

- The operators are set-at-a-time : the results are all entire tables, containing a set of rows. Non-relational operators are row-at-a-time.

- Relational languages are nonprocedural : the user specifies "what" he wants, not "how". In other words, the navigation through the database is done automatically (= automatic navigation system $\leftrightarrow$ manual navigation system). It's the job of the optimizer to perform the navigation as efficient as possible.

- The information content of the database is represented in only one way, namely as explicit data values (there are no pointers connecting a table to another, at least not at the external and conceptual level !).

- All data values are atomic (= scalar). Repeating groups, more than one value in one column, are not allowed.

- The catalog (= dictionary) is also a table (= system table) and can be interrogated in exactly the same way as an ordinary table.

- The relational model is concerned with three data aspects :
  * data structure (object)
  * data integrity
  * data manipulation (operators)

I.3.4. OBJECT-ORIENTED MODEL

Object oriented techniques (abstraction) are used in database systems (f.i. in the relational model).

II. THE RELATIONAL MODEL

II.1. RELATIONAL DATA OBJECTS

II.1.1. TERMINOLOGY

General	Relational Informal (= SQL)	Relational Formal
file	table	relation
record	rows	tuple
field	column	attribute

II.1.2. DOMAINS

- Scalars : the individual data values (= the smallest semantic unit of data)
  → Scalars are atomic, they have no internal structure so far as the relational model is concerned (f.i., a city name consisting of characters, which have no individual sense).

- Domain : a named set of scalar values, all of the same type.
  * Each attribute must be defined on exactly one domain from which actual values are drawn.
  * Comparisons (and other operations) involving two attributes makes only sense when they have the same underlying domain (= domain constrained comparisons).
  * Domains are also kept in the catalog (mostly not as a set of scalars).
  * Domain-based-query :
    F.i., "Which relations contain any information pertaining to suppliers ?", asks for a relation that includes an attribute that is defined on the supplier numbers domain (↔ attribute-based-query).
  * A domain is a data-type and an attribute is a variable of that data-type.

- A note on naming :
  * Domains have unique names in the database.
  * Named relations have unique names in the database.
  * Attributes have unique names in the containing relation.

II.1.3. RELATIONS

- Relation variable and values :
  * Relation variable : a named object whose value changes over time.
  * Relation value : the value of such a variable at any given time.

- A relation (value) R, on a collection of domains D_1, D_2,..., D_n (not necessarily all distinct) consists of two parts : a heading and a body.
  * The heading consists of a fixed set of attribute name and domain name pairs, ($<A_1, D_1>$, $<A_2, D_2>$,..., $<A_n, D_n>$), such that each attribute A_j corresponds to exactly one of the underlying domains D_j (j = 1,..., n). The attribute names A_j are all distinct.

* The body consists of a set of attribute name and attribute value pairs, ($<A_1, v_{i1}>$, $<A_2, v_{i2}>$,..., $<A_n, v_{in}>$) (i = 1,..., m). In each such tuple, there's one such pair for each attribute A_j in the heading. For each given pair $<A_j, v_{ij}>$, v_{ij} is a value from the unique domain D_j that is associated with the attribute A_j.
* m is called the cardinality and n is called the degree (= arity).

- Properties of relations :
  * There are no duplicate tuples.
    → The body of a relation is a mathematical set of tuples and sets in mathematics don't include duplicate elements.
  * Tuples are unordered (top to bottom)
    → The body of a relation is a mathematical set of tuples and sets in mathematics are unordered.
  * Attributes are unordered (left to right)
    → The heading of a relation is a mathematical set of attributes and sets in mathematics are unordered.
  * All attribute values are atomic.
    → An underlying domain contains atomic values only (a relation is normalized).
    → A mathematical relations doesn't have to be normalized. The reason why we demand it here is just simplicity (f.i. of all operators).

- A table is thus only a representation of a relation (f.i. on paper) and not a relation itself, f.i. the rows and columns of a table do have an order.

II.1.4. KINDS OF RELATIONS

- Named relation : a relation that has been defined to the DBMS (↔ unnamed relation).

- Real relation :
  A named relation that is represented by its own data (= stored), at least conceptually (↔ virtual relation).

- Base relation : a real named relation that has independent existence.

- Derived relation :
  A relation that is defined, by means of some relational expression, in terms of other named relations.
  * View : a virtual named derived relation
  * Snapshot : a real named derived relation
  * Query result : an unnamed derived relation that results from executing a query.
  * Intermediate result : an unnamed derived relation that results from the evaluation of some relational expression that is nested within some larger relational expression.

- Expressible relation :
  A relation that can be obtained from the set of named relations by means of some relational expression
  → expressible relations = all base relations + all derived relations
  Stored relation : an expressible relation that is supported in physical storage.

II.1.5. RELATIONAL DATABASE

A relational database is a database that is perceived by the user as a collection of normalized relations (= relation variables) of assorted degrees.

II.2. RELATIONAL DATA INTEGRITY

II.2.1. ATTRIBUTE INTEGRITY

Attribute integrity :

The values for each attribute must be drawn from the relevant domain (domain constraint).

II.2.2. ENTITY INTEGRITY

II.2.2.1. MISSING INFORMATION

- Missing information in a database (f.i., birthday not known), is represented by a NULL.
- NULLS are not real values, they are only markers for missing information.
- Each attribute must include a NULLS ALLOWED or NULLS NOT ALLOWED specification.

II.2.2.2. CANDIDATE KEYS

- A candidate key for a relation (variable) R, is a subset of the set of attributes of R, say K such that for all time :
  * Uniqueness property :
    No two distinct tuples in the current value of R have the same K value.
  * Irreducibility property :
    No proper subset of K has the uniqueness property.
- The irreducibility property is necessary, f.i., if we define {S#, CITY} instead of {S#} as a candidate key, it isn't ensured that S# is globally unique (only that S# is locally unique within equal cities).
- Because relations don't contain duplicate tuples, every relation must have at least one candidate key : the combination of all attributes.
- If a candidate key involves more than one attribute, it is called composite, otherwise it's called simple.
- Candidate keys provide the basic tuple level addressing mechanism (it's the only way to point to one specific tuple).
- From all candidate keys (a relation can have more than one candidate key), there's one candidate key elected to be the primary key (on a basis of simplicity). The other remaining candidate keys are then called alternate keys. If a relation has only one candidate key, then, of course, that candidate key is the primary key.

II.2.2.3. ENTITY INTEGRITY

- Entity integrity :
  No component of the primary key in a base relation is allowed to accept nulls.
- Every tuple (representing an entity in the real world) must be identifiable.

- The rule only applies to base relations, not to derived relations. Suppose we have a part tuple with the attribute color NULL. The query “list all part colors” would have only one attribute, which is the only candidate key and is therefore also the primary key, and one of the tuples would have a null in that primary key.

- The rule only applies to primary keys, not to alternate keys. But why was an alternate key that has nulls allowed, then chosen as a candidate key if it can never be a primary key ?

II.2.3. REFERENTIAL INTEGRITY

II.2.3.1. FOREIGN KEYS

- A foreign key in a base relation (variable) R2, is a subset of the set of attributes of R2, say FK, such that :
  * there exists a base relation (variable) R1 (R1 and R2 not necessarily distinct) with a candidate key CK.
  * for all time, each value of FK in the current value of R2 is either NULL OR is identical to the value of CK in some tuple in the current value of R1.

- Each component attribute of a foreign key must be defined on the same domain as the corresponding component attribute of the matching candidate key.

- A foreign key doesn’t have to be a component of a candidate key of its own relation, f.i., the employees-department database where the EMP# is the candidate key in the employees relation and the DEPT# is the foreign key matching the candidate key in the department relation.

- A foreign key can be a NULL, f.i., an employee who isn’t assigned to a specific department. Thus, there must also be specified if a foreign key has NULLS ALLOWED or NULLS NOT ALLOWED.

- A foreign key represents a reference to the tuple containing the matching candidate key (= the referenced or target tuple). A relation that contains of foreign key is a referencing relation and the relation that contains the matching candidate key is the referenced or target relation.

II.2.3.2. REFERENTIAL INTEGRITY

- Referential integrity :
  The database must not contain any unmatched foreign keys (referential constraint).

- Referential diagrams :
  An arrow means there’s a foreign key from which the arrow emerges that refers to a candidate key of the relation to which the arrow points.
  F.i., S $\leftarrow$ SP $\rightarrow$ P

- Referential path : The chain of arrows from R_n to R_1, f.i., $R_n \rightarrow R_{n-1} \rightarrow ... \rightarrow R_2 \rightarrow R_1$

- A relation might include a foreign key that must match the value of some candidate key in that same relation (such a relation is called self-referencing), f.i., the employee database with an attribute indicating the manager of that employee, who is at his turn also an employee.

- Referential cycle :
  A referential path from some relation R_n to itself.
  F.i., $R_n \rightarrow R_{n-1} \rightarrow ... \rightarrow R_2 \rightarrow R_1 \rightarrow R_n$

- What should happen on an attempt to an operation (delete or update) that violates the referential integrity constraint ?
  * RESTRICTED : The operation is rejected.
  * CASCADES : Additional operations are executed to guarantee that the final state is legal.
  * NULLIFIES : The foreign key is set to NULL.
  * There should also be a possibility to invoke a user-defined database procedure (= a triggered procedure).

  Note : This problem occurs along the complete referential path.

II.3. RELATIONAL DATA OPERATORS (ALGEBRA)

II.3.1. BNF GRAMMAR

```
expression ::= monadic-expression | dyadic-expression
monadic-expression ::= renaming | restriction | projection |
                       extension | summarization
renaming ::= term RENAME rename-commalist
rename ::= attribute AS attribute
term ::= relation | (expression)
restriction ::= term WHERE condition
condition ::= boolean-comparison | attribute-comparison |
              relation-comparison
boolean-comparison ::= NOT condition | condition AND condition |
                       condition OR condition
attribute-comparison ::= attribute comparator attribute
comparator ::= = | ≠ | < | ≤ | > | ≥
relational-comparison ::= expression comparator expression (1)
projection ::= term | term [attribute-commalist]
extension ::= EXTEND term ADD extend-commalist
extend ::= scalar-expression AS attribute
summarization ::= SUMMARIZE term BY (attribute-commalist) ADD
                  summarize-commalist
summarize ::= aggregate-expression AS attribute
dyadic-expression ::= projection dyadic-operation expression
dyadic-operation ::= UNION | INTERSECT | MINUS | TIMES | JOIN |
                     DIVIDEBY

relation-assignment ::= target := source
insertion ::= INSERT source INTO target
updating ::= UPDATE expression assignment-commalist
assignment ::= attribute := scalar-expression
deletion ::= DELETE expression
empty-operator ::= IS_EMPTY (expression)
set-membership-operator ::= tuple IN expression
```

Notes :
* X-commalist : a sequence of zero or more X's, separated by a comma (and optionally one or more blanks).
* tuple : a reference to just one tuple.
* attribute : an identifier for an attribute (= attribute name)
* relation : an identifier for a relation (= relation name)

* source and target are expressions that evaluate to type compatible relations.
* aggregate functions are COUNT, SUM, AVG, MAX, MIN,....
* (1) : both expression must evaluate to type compatible relations.

II.3.2. PROPERTY OF CLOSURE

- The property of closure consists of two subproperties :
  * The output of a relational operator is another relation
    → It is possible to write nested expressions in which the operands are represented by expressions.
  * Attribute name inheritance rules : it must be possible to predict the attribute names of the output of an operator (see RENAME operation).

- We could go further and demand the candidate key inheritance rules so that the system can deduce the candidate keys for the result of an expression.

II.3.3. SET OPERATORS

- Type compatible operands means :
  * they each have the same set of attribute names
  * corresponding attributes are defined on the same domain.
  → They have identical headings.

- The union of two type compatible relations A and B (A UNION B) is a relation with the same heading as each of A and B and with the body consisting of the set of all tuples t belonging to A or B or both. Duplicate tuples are eliminated by definition.

- The intersection of two type compatible relations A and B (A INTERSECT B) is a relation with the same heading as each of A and B and with the body consisting of the set of all tuples t belonging to both A and B.

- The difference between two type compatible relations A and B (A MINUS B) is a relation with the same heading as each of A and B and with the body consisting of the set of all tuples t belonging to A and not to B.

- The Cartesian product of two relations A and B (A TIMES B) is a relation with a heading that is the coalescing of the heading of A and B and with the body consisting of the set of all tuples t such that t is the coalescing of a tuple a belonging to relation A and a tuple b belonging to relation B. A and B may not have no common attribute names.
  Notes :
  * cardinality = cardinality A * cardinality B
  * degree = degree A + degree B

- UNION, INTERSECT and TIMES are associative and commutative (MINUS is not).

II.3.4. RELATIONAL OPERATORS

- The θ-restriction of a relation A on the attribute X and Y (A WHERE X θ Y) is a relation with the same heading as A and with the body consisting of the set of all tuples t such that the condition X θ Y evaluates to true for that tuple t. The attributes X and Y must be defined on the same domain (→ domain comparison constraint) and the θ-operator must make sense for that domain.
  Notes :
  * θ stands for any simple scalar comparison operator (=, ≠, >, ≤,...)

* A scalar literal value can be specified instead of attribute X or Y (or both), f.i.,
  A WHERE X θ literal
* This definition can be extended to cover a condition that consists of an arbitrary Boolean combination of such simple comparisons.
  A WHERE c_1 AND c_2 ≡ (A WHERE c_1) INTERSECT (A WHERE c_2)
  A WHERE c_1 OR c_2 ≡ (A WHERE c_1) UNION (A WHERE c_2)
  A WHERE NOT c ≡ A MINUS (A WHERE c)
  Such a condition (that results in a true or false for a given tuple) is called a restriction condition.

- The projection of a relation A on the attributes X, Y,..., Z (A[X, Y,...,Z]) 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, Y-value y,... and Z-value z. Duplicate tuples are eliminated by definition.
  Notes :
  * If the attribute commalist is omitted, we specify the identity projection.
  * The projection with an empty attribute commalist is also legal : the nullary projection.

- The natural join of two relations A and B (A JOIN B) where A has the heading {X, Y}, B has the heading {X, Z} (X, Y and Z are composite attributes and corresponding attribute names are defined on the same domains), 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 {X:x, Y:y} appears in A and a tuple {X:x, Z:z} appears in B.
  Notes :
  * Join is associative and commutative.
  * If A and B have no attributes in common, then A JOIN B is equivalent to A TIMES B.

- The θ-join of relation A on attribute X with relation B on attribute Y is defined as :
  (A TIMES B) WHERE X θ Y
  Notes :
  * Attribute X and Y must be defined on the same domain and the θ-operation must make sense for that domain.
  * The θ-join is used where we need to join two relations together on the basis of some condition other than equality.
  * θ-join is not a primitive operator.
  * If θ is “equals” then the θ-join is called equijoin.
    → The result of an equijoin has two attributes with the property that the values in those two attributes are equal. If one of these two attributes is eliminated (with a projection), the result is a natural join.
    → A natural join is also not a primitive operation.

- The division of two relations A and B (A DIVIDEBY B), where A has the heading {X, Y}, B has the heading {Y} (X and Y are composite attributes and corresponding attributes are defined on the same domain), is a relation with heading {X} and body consisting of the set of all tuples {X:x} such that a tuple {X:x, Y:y} appears in A for all tuples {Y:y} appearing in B.

II.3.5. USE OF THE ALGEBRA

- The algebra is used to define :
  * a scope for retrieval (defining the data to be fetched)
  * a scope for update (defining the data to be inserted, modified, deleted,...)
  * a view
  * a snapshot
  * security rules (defining the data over which authorization is to be granted)
  * ....

- The expressions serve as a high-level and symbolic representation of the user's intent. They can be manipulated by means of symbolic transformation rules (f.i., by the optimizer to execute the expression as efficient as possible)

- Their are five primitive operators :
  * UNION
  * DIFFERENCE
  * CARTESIAN PRODUCT
  * RESTRICTION
  * PROJECTION

II.3.6. SOME MORE OPERATORS

- EXTEND A ADD exp AS Z :
  This operator returns a relation with the same heading as relation A extended with the new attribute Z and with the body consisting of the set of 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 scalar expression exp on that tuple of A. A must not include an attribute Z and exp must not refer to Z.
  Note : Rename can be expressed in terms of extend (so rename isn't a primitive operator), f.i.,
  S RENAME CITY AS SCITY ≡
  (EXTEND S ADD CITY AS SCITY)[S#, SNAME, STATUS, SCITY]

- SUMMARIZE A BY (A_1,..., A_n) ADD exp AS Z (with A_1,..., A_n distinct attributes of A) :
  This operator returns a relation with heading {A1,..., An, Z} and with body consisting of the set of all tuples t such that t is a tuple of the projection of A over A1,..., An extended with a value for the new attribute Z. That new Z-value is computed by evaluating the aggregate function exp on all tuples of A that have the same values for A1,..., An as tuple t. The attribute commalist must not include an attribute Z and exp must not refer to Z.
  Note : The relation is grouped into sets of tuples and then each group is used to generate one tuple in the result.

II.3.7. SHORTHAND

((A RENAME X AS Y) WHERE Y = X) ≡ (MATCHING A)
F.i.: EXTEND S ADD COUNT ((SP RENAME S# AS X) WHERE X = S#) AS NP
≡ EXTEND S ADD COUNT (MATCHING SP) AS NP

II.4. RELATIONAL DATA OPERATORS (CALCULUS)

II.4.1. INTRODUCTION

- Difference between the relational algebra and the relational calculus :
  * Relational algebra :
    - Based on the mathematical set theory.
    - Prescribes a procedure for solving a problem (procedural)
  * Relational calculus :
    - Based on mathematical logic (predicate calculus).
    - Describes the problem (non-procedural)
  * But : The algebra and calculus are precisely equivalent to one another.

- Kinds of relational calculus :
  * Tuple oriented relational calculus :
    - Defines tuple variables (= range variables) that ranges over a relation (a variable whose only permitted values are tuples of that relation).
    - Languages : ALPHA (never implemented), QUEL
  * Domain oriented relational calculus :
    - Defines domain variables that ranges over a domain
    - Languages : ILL, FQL, DEDUCE, QBE

II.4.2. TUPLE ORIENTED RELATIONAL CALCULUS

II.4.2.1. BNF GRAMMAR

```
range-variable-definition ::= RANGE OF variable
                              IS range-item-commalist  (1)
range-item ::= relation | expression
expression ::= (target-item-commalist) [WHERE wff]
attribute-reference ::= variable.attribute
wff ::= condition | (wff) | IF condition THEN wff | NOT wff |
        condition AND wff | condition OR wff |
        EXISTS variable (wff) | FORALL variable (wff)
condition ::= (wff) | comparand comparator comparand | comparand
comparand ::= literal | attribute-reference | scalar-expression
comparator ::= = | ≠ | < | ≤ | > | ≥
aggregate-function-reference ::= aggregate-function (expression
                                 [,attribute])
aggregate-function ::= COUNT | SUM | AVG | MAX | MIN
```

Notes :
* A scalar-expression can include literals, attribute-references and aggregate-function-references.
* (1) : the range-item-commalist must evaluate to type compatibles relations.

II.4.2.2. INTRODUCTION

- A tuple variable name can occur in a WFF (= Well Formed Formula) :
  * in the context of an attribute reference T.A (where A is an attribute of the relation over which T ranges).
  * as the variable immediately following one of the quantifiers EXISTS and FORALL.
  * Note : $T \equiv T.A_1, T.A_2, ..., T.A_n$

- A variable can only become bound if it occurs immediately after a quantifier.Once it is bound, it cannot become free anymore. A WFF with only bound variables is called a closed-WFF. An open-WFF is a WFF that is not closed (it contains at least one free variable).

- Quantifiers :
  * f is a WFF in which X is free :
    - EXISTS X (f) : there exists at least one value of X that makes f true.
    - FORALL X (f) : for all values of X, f is true.
  * R is a relation with tuples T_1,..., T_n, T is a tuple variable over R and f(T) is a WFF in which T is free :
    EXISTS T (f(T)) ≡ false OR (f(T_1)) OR ... OR (f(T_n))
    → iterated OR (false if R is empty)
    FORALL T (f(T)) ≡ true AND (f(T_1)) AND ... AND (f(T_n))
    → iterated AND (true if R is empty)
  * An aggregate function also acts like a quantifier, because a free variable also becomes bound.

- Expressions :
  * Example : (SX.S#, PX.P#) WHERE SX.CITY ≠ PX.CITY
  * Every free variable in the WFF must be mentioned in the target item commalist.
  * The evaluation of such an expression goes as follow, f.i., variables in the target item commalist T, U,..., V over the respective relations TR, UR,..., VR. The result names are specified (or inherited) as X_1,..., X_n
    → Cartesian product : TR TIMES UR TIMES ... TIMES VR
    → Tuples that don't satisfy the WFF are eliminated
    → The result is the projection over X_1,..., X_n

II.4.2.3. RELATIONAL ALGEBRA VERSUS RELATIONAL CALCULUS

- Codd's algorithm to convert a expression from the calculus into an expression from the algebra goes as follow, f.i.,

```
(SX.SNAME, SX.CITY) WHERE
EXISTS JX
FORALL PX
EXISTS SPJX (JX.CITY = 'Athens' AND JX.J# = SPJX.J#
                AND PX.P# = SPJX.P# AND SX.S# = SPJX.S#
                AND SPJX.QTY ≥ 50)
```

  1) For each tuple variable retrieve the range, restricted if possible (the restriction condition), f.i. :
     JX : CITY = 'Athens'
     SPJX : QTY ≥ 50
  2) Construct the Cartesian product (and rename the attributes if necessary)
  3) Restrict that intermediate result (the join condition), f.i. :
     JX.J# = SPJX.J# AND PX.P# = SPJX.P# AND SX.S# = SPJX.S#
  4) Apply the quantifiers from right to left (RX is a tuple variable over relation R) :
     * EXISTS RX : Project the intermediate result to eliminate all attributes of R.
     * FORALL RX :
       Divide the intermediate result by the restricted range of step one (this will also eliminate all the attributes of R).
  5) Project the intermediate result in accordance with the target item commalist.

- A relational calculus expression is semantically equivalent to a nested expression in the relational algebra. A language which is at least as powerful as the relational algebra (f.i., the relational calculus) is called relationally complete. In fact, it can be proven that any algebraic

expression can be reduced to a calculus equivalent, which means that these two languages are logically equivalent.

- To prove that a language L is relationally complete, it's sufficient to prove that :
  * L has analogs for the five primitive algebraic operators.
  * the operands of any operation in L can be arbitrary L expressions.

II.4.3. DOMAIN ORIENTED RELATIONAL CALCULUS

```
membership-condition ::= relation (attribute-domain-commalist)
attribute-domain ::= attribute:domain-variable | attribute:literal
```

Notes :
* Scalar variables would be a better name for a domain varible because its value is a scalar, not a domain name.
* The additional membership condition becomes true if and only if there exists a tuple in R with the specified values for the specified attributes.

II.5. THE SQL LANGUAGE

II.5.1. INTERACTIVE SQL

II.5.1.1. SIMPLE SELECTION QUERIES

- Get full table :

```
SELECT *
FROM S;
```

- Order : default is ASC (ascending)

```
SELECT *
FROM P
ORDER BY WEIGHT DESC;
```

- Projection :

```
SELECT SNAME, CITY
FROM S;
```

Note : Creating a new column (with a name) : a new created column can't be used anywhere, except in the ORDER clause.

```
SELECT P#, WEIGHT * 454 AS GMWT
FROM P;
```

- Restriction :

```
SELECT *
FROM P
WHERE CITY <> 'Paris';
```

Note : Duplicate rows are not eliminated by default, if you want it you must use DISTINCT.

```
SELECT DISTINCT COLOR, CITY
FROM P
WHERE CITY <> 'Paris';
```

- Cartesion product :

```
SELECT *
FROM S, P;
```

Notes :

* Qualifying names : the columns in the ORDER-clause must be unqualified.

```
SELECT S.SNAME, S.CITY, P.PNAME, P.CITY
FROM S, P;
```

* Alias names : they can be used anywhere in the query.

```
SELECT FIRST.SNAME, FIRST.CITY, SECOND.PNAME, SECOND.CITY
FROM S AS FIRST, P AS SECOND;
```

- Join :

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY, P.P#, P.PNAME, P.COLOR,
P.WEIGHT
FROM S, P
WHERE S.CITY = P.CITY;
```

- Combining example :

  Get all pairs of supplier numbers such that the two suppliers concerned are colocated.

```
SELECT FIRST.S# AS SA, SECOND.S# AS SB
FROM S AS FIRST, S AS SECOND
WHERE FIRST.CITY = SECOND.CITY AND FIRST.S# < SECOND.S#;
```

II.5.1.2. AGGREGATE FUNCTIONS

- COUNT :

  * Get the total number of different cities of the suppliers :

```
SELECT COUNT (DISTINCT CITY) AS N
FROM S;
```

  * COUNT(*) : count all rows, including duplicate ones (DISTINCT is not allowed)
  * NULLS are eliminated before counting (except for COUNT(*)).

- MAX/MIN : Get the maximum and minimum quantity for P2 :

```
SELECT MAX(SP.QTY) AS MAXQ, MIN(SP.QTY) AS MINQ
FROM SP
WHERE SP.P# = 'P2';
```

II.5.1.3. GROUPING

- GROUP BY : Get the part number and the total shipment quantity for each supplied part.

```
SELECT P#, SUM(QTY) AS TOTQTY
FROM SP
GROUP BY P#;
```

  Note : The select must be single-valued per group. The referenced column must be grouped or must be an argument (or at least a part of an argument) to an aggregate function.

- HAVING : Get part numbers for all parts supplied by more than one supplier.

```
SELECT P#
FROM SP
GROUP BY P#
HAVING COUNT(S#) > 1;
```

  Notes :

  * The HAVING-clause if used to eliminate groups just as the WHERE-clause if used to eliminate rows.
  * Expressions in a HAVING-clause must be single-valued per group (see GROUP BY).

II.5.1.4. NESTED QUERIES

- Without extra keywords : Get supplier numbers with status less than the current maximum.

```
SELECT S#,
FROM S
WHERE STATUS <
        (SELECT MAX(STATUS)
        FROM S);
```

- IN :

  * Get the supplier names who supply part P2 :

```
SELECT DISTINCT SNAME
FROM S
WHERE S# IN
        (SELECT S#
        FROM S
        WHERE P# = 'P2');
```

  → The value S# is searched in the result table from the subquery (if found, the condition becomes true).

  * The query can be reversed by using NOT IN.
  * Subqueries can be nested to any depth, f.i.,
    Get the supplier names, who supply at least one red part :

```
SELECT DISTINCT SNAME
FROM S
WHERE S# IN
        (SELECT S#
        FROM SP
        WHERE P# IN
                (SELECT P#
                FROM P
                WHERE COLOR = 'Red'));
```

- EXISTS :

  * Get the supplier names who supply part P2 :

```
SELECT DISTINCT SNAME
FROM S
WHERE EXISTS
        (SELECT *
        FROM SP
        WHERE SP.P# = 'P2' AND SP.S# = S.S#);
```

  → The EXISTS expression becomes true if the result table isn't empty (there's no projection necessary).

  * The query can be reversed by using NOT EXISTS.
  * Forall can be simulated with NOT EXISTS, f.i.,
    Get the supplier names, who supply all parts :

```
SELECT DISTINCT SNAME
FROM S
WHERE NOT EXISTS
        (SELECT *
        FROM P
        WHERE NOT EXISTS
                (SELECT *
                FROM SP
                WHERE S.S# = SP.S# AND P.P# = SP.P#));
```

II.5.1.5. SET OPERATIONS

UNION :

Get the supplier numbers for parts that either weigh more than 16 pounds or are supplied by supplier S2, or both.

```
SELECT P#
FROM P
WHERE WEIGHT > 16
UNION
SELECT P#
FROM SP
WHERE S# = 'S2';
```

II.5.2. EMBEDDED SQL

- Dual mode principle :

  Any SQL statement that can be used interactively can also be used in an application program (embedded SQL).

  Notes :

  * All embedded SQL statements are prefixed by EXEC SQL and are terminated by a special terminator (f.i. ";"), to distinguish them from host language statements.
  * An executable SQL statement can appear wherever an executable host statement can appear.
  * SQL statements can include references to host variables. These variables must include a colon prefix and may appear wherever where a literal can appear (of course, these variables must have a compatible data type for which they are used). Every host variable used in an SQL statement must be defined within an embedded SQL declare section which is delimited by the BEGIN and END DECLARE SECTION.
  * Every embedded SQL program must include a host variable SQLSTATE (char(5)) which returns a status code (00000 means success, 02000 means no data was found to satisfy the request). To simplify the check for SQLSTATE after each executable statement, there's a special WHENEVER statement :

    EXEC SQL WHENEVER condition action;

    condition : NOT FOUND (SQLSTATE = 02000) or SQLERROR (an error occurred)

    action : CONTINUE (nothing happens) or GOTO label.

- Cursors :

  * In a host language, it's difficult to work with tables containing more than one row. Therefore the cursor concept is offered. A cursor is a kind of a pointer to just one row out of a set of rows.
  * To initialize a pointer, we must use a declare statement. Note that this statement is pure declarative, so nothing actually happens.

    EXEC SQL DECLARE cursor CURSOR FOR table-expression

    [ORDER BY order-item-commalist];

    order-item ≡ unqualified column name followed by ASC (default) or DESC.
  * To evaluate the table expression, we must open the cursor. The cursor is then placed just before the first row of the result table.

    EXEC SQL OPEN cursor;
  * To advance the cursor to the next row and to assign the values from that row to the host variables (SQLSTATE is set to 02000 if there isn't a next row) :

    EXEC SQL FETCH cursor INTO host-variables-commalist;
  * When the cursor reaches the end, we must close the cursor :

    EXEC SQL CLOSE cursor;

- Operations that can be performed without a cursor :
  * singleton select (a retrieval of at most one single row)
  * insert
  * update
  * delete

- Operations that do need a cursor :
  * select
  * current form of update (update the current position)
  * current form of delete (delete the current position)

- Dynamic SQL :
  * Consists of a set of embedded SQL facilities that are provided to allow the construction of generalized, online and possibly interactive applications.
  * Example :

    SQLSOURCE = ‘delete from SP where SP.QTY < 300’;
    EXEC SQL PREPARE SQLPREP FROM :SQLSOURCE;
    → Compiles the value of the host variable SQLSOURCE into the SQL variable SQLPREP (both names are arbitrary).
    EXEC SQL EXECUTE SQLPREP;
    → Executes the compiled statement.

III. DATABASE DESIGN

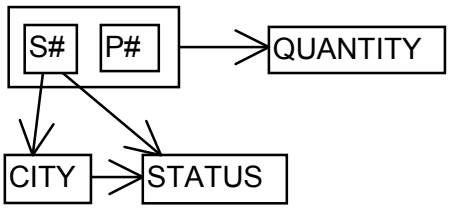
III.1. NORMALIZATIONS

III.1.1. INTRODUCTION

- Normalization theory is built around the concept of normal forms. A relation is said to be in a particular normal form if it satisfies a certain set of constraints.
- The database designer should aim for the highest normal form (= 5NF) but that isn't a law, only a recommendation.
- The normalization is done by decomposing the total set of attributes into several different relations. This process of decomposition must be reversible to ensure that no information is lost. Decomposition is really a process of projection, so the recomposition process is a natural join. Such a decomposition is called a nonloss-decomposition (there appear no "spurious" tuples).

III.1.2. FIRST NORMAL FORM

- A relation is in first normal form (= 1NF) if and only if all underlying simple domains contain atomic values only.
- Such a relation is called normalized.
- This is a normalization law as far as the relational model is concerned.
- Example :



Update anomalies :

We cannot insert information about a supplier until that supplier supplies at least one part (we must obbey the entity integrity rule, so P# may not be NULL). The same problem arises by deletion : if we delete the last shipment of a supplier, we also destroy the information about that supplier. Changing the city of a supplier, requires the update of all shipments by that supplier.

III.1.3. SECOND NORMAL FORM

- B is functionally dependent (f.d.) on A ($A \rightarrow B$), where A and B are subsets of the set of attributes of a relation (variable) R, if and only if, in every possible legal value of R (= for all time), each A-value has associated with it precisely one B-value.

Notes :
* If A is a candidate key of relation R then all attributes of relation R must necessarily be functionally dependent on A.
* A is called the determinant and B is called the dependent.

- A trivial f.d. is a f.d. that cannot possibly not be satisfied, f.i. {S#, P#} → S#.
  In general : All f.d. where the right-hand side is a subset of the left-hand side are trivial f.d.

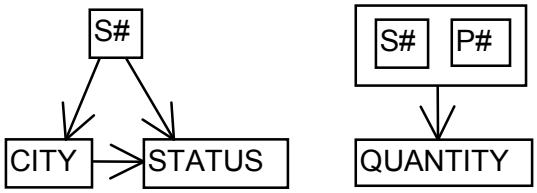
- B is full f.d. (f.f.d.) on A, where A and B are subsets of the set of attributes of a relation (variable) R, if and only if :
  * B is f.d. on A in R.
  * B is not f.d. on a proper subset of A in R.
  Notes :
  * A full functional dependency is also called a left-irreducible functional dependency.
  * Normally, the term f.d. stands for f.f.d.

- Heath's theorem :
  A relation R with A, B and C sets of attributes of R can be nonloss-decomposed into its two projections {A, B} and {A, C} if and only if the f.d. A → B holds in R.

- A nonkey attribute is an attribute that doesn't participate in the primary key.

- A relation is in second normal form (= 2NF) if and only if it is in 1NF and every nonkey attribute is f.d. on the primary key
  Note : This is always true if the primary key is simple (= non composite).

- Example :



This relation solves the previous update anomalies but not all :
We cannot insert the information that a city has some particular status, until there's one supplier from that city. Same problem by deletion : if we delete a supplier, we can lose the status value of a city. Changing the status of a city, requires the update of all suppliers from that city.

III.1.4. THIRD NORMAL FORM

III.1.4.1. DEFINITION

- A → C is called a transitive f.d. if A → B and B → C.

- A relation is in third normal form (= 3NF) if and only if it is in 2NF and every nonkey attribute is nontransitively f.d. on the primary key (except via another candidate key).
  Note :
  No transitive f.d. implies no mutual dependencies. Two or more attributes are called mutual independent if none of them is f.d. on any combination of the others.

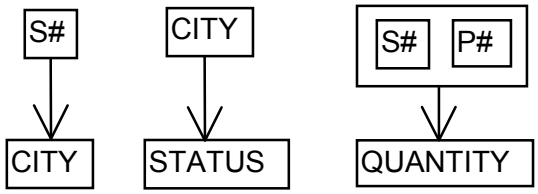
III.1.4.2. CHOOSING THE RIGHT DECOMPOSITION

- Rissanen's theorem :
  The projections R1 and R2 of a relation R are independent if and only if :
  * Every f.d. in R is a logical consequence of those in R1 and R2 (transitive f.d. in R can be deduced from the f.d. in R1 and R2).
  * The common attribute(s) of R1 and R2 form a candidate key for at least one the pair.

- A relation is atomic if it cannot be decomposed into independent projections.
  Notes :
  * A non-atomic relation doesn't necessarily need to be decomposed into atomic components.
  * Some atomic relations can be decomposed in a nonloss way (these projections are just not independent).

- Dependency preservation :
  If we can choose between two or more possibilities to decompose a relation, we should take the decomposition that produces independent projections.
  Note :
  If a relation R {A, B, C} satisfies A $\rightarrow$ B and B $\rightarrow$ C (and thus A $\rightarrow$ C), it is better decomposed into its projections {A, B} and {B, C} rather than {A, B} and {A, C}.

- Example :



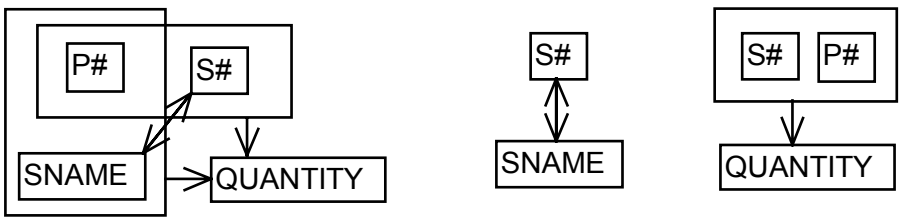
This relation solves the previous update anomalies.

III.1.5. BOYCE/CODD NORMAL FORM

- Our defined normal forms don't deal with two (or more) composite, overlapping (they have at least one attribute in common) candidate keys.

- A relation is in Boyce/Codd normal form (= BCNF) if and only if every f.d. is implied by a candidate key (thus every determinant is a candidate key)
  Notes :
  * Every binary relation is in BCNF.
  * BCNF implies 3NF.
  * This definition was actually first given by Heath, so Heath normal form might have been a more appropriate name.

- Example :
  * 3NF :
  * BCNF :



III.1.6. FOURTH NORMAL FORM

- B is multivalued dependent (= m.v.d.) on A (A $\rightarrow>$ B), where A, B and C are subsets of the set of attributes of a relation (variable) R, if and only if, in every possible legal value of R (= for all time), the set of B-values matching a given (A-value, C-value) pair in R depend only on the A-value and is independent of the C-value.
  Notes :
  * Every f.d. is a m.v.d. (the converse is not true).
  * M.v.d. comes always in pairs, f.i. if in a given relation R {A, B, C}, A $\rightarrow>$ B holds, then also A $\rightarrow>$ C holds. This can be represented by a single notation : A $\rightarrow>$ B $\mid$ C.

- Fagin's theorem (a stronger version of Heath's theorem) :
  A relation R with A, B and C sets of attributes of R, can be nonloss-decomposed into its two projections {A, B} and {A, C} if and only if the m.v.d. A $\rightarrow>$ B (and of course also A $\rightarrow>$ C) holds in R.

- A relation is in fourth normal form (= 4NF) if and only if whenever there exist subsets A and B of the set of attributes of R such that the (nontrivial) m.v.d. A $\rightarrow>$ B is satisfied, then all attributes of R are also f.d. on A.
  OR
  A relation is in fourth normal form (= 4NF) if and only if it is BCNF and all m.v.d. are in fact f.d. out of candidate keys.
  OR
  A relation is in fourth normal form (= 4NF) if and only if every m.v.d. is implied by a candidate key.
  Note :
  In fact, every m.v.d. implied by a candidate key is just a f.d. with that candidate key as a determinant (thus 4NF implies BCNF).

- Rissanen's theorem about independent projections is also true for m.v.d. instead of f.d., so if a relation R {A, B, C} satisfies A $\rightarrow>$ B and B $\rightarrow>$ C, it is better decomposed into its projections {A, B} and {B, C} rather than {A, B} and {A, C}.

III.1.7. FIFTH NORMAL FORM

- A relation is n-decomposable if it can be nonloss-decomposed into n projections but not into m projections for any m < n.
  Note :
  So far, we have assumed that the sole operation available in the decomposing process is the replacement of a relation (in a nonloss way) by two of its projections. However some relations can be nonloss decomposed into three projections but not into two projections.

- Time independent constraint :
  * Example :
    If (s_1, p_1) appears in SP and
    if (p_1, j_1) appears in PJ and
    if (j_1, s_1) appears in JS,
    then (s_1, p_1, j_1) appears in SPJ.
    $\equiv$
    If (s_1, p_1, j_2) and (s_2, p_1, j_1) and (s_1, p_2, j_1) appear in SPJ,
    then (s_1, p_1, j_1) also appears in SPJ.
  * If such a statement is true for all possible legal values of a relation, then it's called a time independent (cyclic) constraint.

- A relation will be n-decomposable, for some n > 2, if and only if there exists a time independent cyclic constraint.

- A relation R (with A, B,..., Z subsets of the set of attributes of R) satisfies the join dependency
  (= j.d.) *(A, B,..., Z) if and only if R is equal to the join of its projection on A, B,...., Z.
  Note : Every m.v.d. is a j.d. (the converse is not true).

- Fagin's theorem also holds for j.d. :
  R {A, B, C} satisfies the j.d. *(AB, AC) if and only if it satisfies the pair of m.v.d.
  A →> B | C.

- A j.d. can also be implied by a candidate key :
  * Examples (suppose S# and SNAME are both candidate keys) :
    - ((S#, SNAME, STATUS), (S#, CITY))
    - ((S#, SNAME), (S#, STATUS), (SNAME, CITY))
  * Fagin showed an algorithm to determine, given a j.d. and the set of candidate keys, if that j.d. is implied by these candidate keys.

- A relation is in fifth normal form (= 5NF = projection-join normal form = PJ/NF) if and only if every j.d. is implied by a candidate key.
  Notes :
  * 5NF implies 4NF, because a m.v.d. is a special case of a j.d.
  * It is still very difficult to find all j.d.

- The 5NF is the ultimate normal form with respect to projections and join. A relation in 5NF is guaranteed to be free of update anomalies that can be eliminated by taking projections.

III.2. SEMANTIC MODELING

III.2.1. INTRODUCTION

- Semantic modeling (that produces a semantic model, f.i. the E/R model) is an attempt to represent meaning. This is a necessary activity because a database system generally has only a limited understanding of what the data in the database means (they typically understand certain simple atomic data values, and perhaps certain simple integrity constraints that apply to those values, but very little else). Some of such semantic aspects do already exist in our relational data model, f.i. domains, candidate keys, foreign keys,... (→ a semantic model is a kind of an extension to the relational data model). However, capturing the meaning of data is a never ending task.

- Such a semantic model is very useful as an aid to database design (even if it isn't supported by the DBMS). Therefore, several design methodologies have been proposed that are based on one semantic modeling approach. Such methodologies are often referred to as top down methodologies because they start at a high level of abstraction with real world constructs and finish at the comparatively low level of abstraction represented by a specific concrete database design.

- Semantic modeling goes in four steps :
  * Identify a set of semantic concepts about the real world (these concepts are not formal, f.i., entities, relationships, properties,...)

* Devise a set of corresponding symbolic (= formal) objects that can be used to represent these semantic concepts.
* Devise a set of formal integrity rules to go along with those formal objects.
* Devise a set of formal operators for manipulating those formal objects.
  → These operators are not important from the point of view of database design.

III.2.2. THE ENHANCED ENTITY/RELATIONSHIP MODEL

III.2.2.1. THE ENTITY/RELATIONSHIP MODEL (CHEN) (= ER)

- This model also introduced an entity/relationship diagram (= ERD), a technique for representing the logical structure of a database in a particular manner.

- Entity :
  * A thing that can be distinctly identified.
  * Entities can be grouped into entity types (f.i., all individual employees are instances of the generic employee entity type).
  * A weak entity is an entity that is existence dependent on another entity (= the identifying owner). It cannot exist if that other entity doesn't also exist.
  * A strong entity is an entity that isn't weak.
  * ERD : A rectangle labeled with the entity type name. A weak entity type name has a double rectangle.

- Property :
  * Every entity has properties (= attributes in the relational model) and all entities of a given type have certain properties in common. Each kind of property draws its values from a corresponding value set (= domain in the relational model).
  * Properties can be :
    - simple / composite (f.i. the employee name can be made up by the simple properties "first name" and "last name").
    - key (= unique) : every entity has a special property that identifies that entity (an entity has an identity)
    - single / multi-valued (= repeating groups)
    - missing : unknown or not-applicable (→ NULLS)
    - base / derived (f.i. the total quantity might be derived from the individual shipment quantities)
  * ERD : An ellipse labeled with the property name and attached to the relevant entity with a continuous line. The ellipse is dotted if the property is derived and doubled if the property is multivalued. Key properties are underlined.

- Relationships :
  * An association among entities (they link the entities together).
  * The involved entities are called participants and the number of participants is called the degree (→ binary, ternary,...).
    Note : a binary relationship can also exist between one entity, f.i. a part (screw) can be a part of a bigger part (wheel).
  * Participation constraint :
    If every instance of an entity participates in a relation, then the relation is total else it's partial.
  * Cardinality ratio :
    Assuming that all relationship are binary (degree = 2), a relation can be one-to-one, many-to-one or many-to-many.
  * Structural constraint : the cardinality ratio and the participation constraint together.

* ERD : A diamond labeled with the relationship name and attached to the participants with a continuous line (each such line is labeled “1” or “M” indicating a 1-1, M-1 or M-M relationship).The diamond is doubled if that relationship has a weak entity as one of its participants (that relation is then called an identifying relationship).

III.2.2.2. THE ENHANCED ENTITY/RELATIONSHIP MODEL (= EER)

- Subtype and supertype (= subclass and superclass) :
  * An entity type Y (f.i. programmers) is a subtype of an entity type X (f.i. employees) if and only if every Y is necessarily an X (X is then called the supertype).
  * Properties and relationships that apply to the supertype are inherited by the subtype.
  * A subtype can also have subtypes of its own (→ this forms a type hierarchy).
  * EERD : A subclass is attached by lines to a circle, which is connected to the superclass. The subset symbol on the line connecting a subclass to the circle indicates the direction of the superclass/subclass relationship. When there’s only one subclass of a superclass, we do not use the circle notation.

- The type hierarchy can be viewed as a specialization (from top to bottom) or as a generalization (from bottom to top).

- Predicate-defined (= condition defined) subclass :
  * There exists a condition on the value of some attribute of the superclass that exactly determines the entities that will become members of each subclass.
    → The condition is called the defining predicate.
  * EERD : The defining predicate is written next to the line attaching the subclass to the circle.

- Kinds of specializations :
  * Attribute-defined : All subclasses are predicate-defined on the same attribute.
    → The attibute is called the defining attribute.
    → EERD : The defining attribute is written next to the line attaching the circle to the superclass.
  * User-defined :
    The membership to a certain subclass is specified individually for each entity by the user.

- Kinds of constraints :
  * Disjointness constraint :
    - The subclass of the specialization must be disjoint. An entity can be a member of at most one of the subclasses.
      → EERD : There is a ‘d’ written in the circle.
      → An attribute-defined specialization implies disjointness if the defining-attribute is single-valued.
    - If the subclasses are not disjoint, they are overlapping. The same entity may be a member of more than one subclass.
      → EERD : There is an ‘o’ written in the circle.
  * Completeness constraint :
    - Total : Every entity in the superclass must be a member of some subclass.
      → EERD : The line from the circle to the superclass is doubled.
    - Partial : An entity is allowed to belong not to any of the subclasses.

- Hierarchy versus lattice :
  * Hierarchy : Every subclass participates in exactly one super/sub-class relationship.
  * Lattice : At least one subclass participates in more than one super/sub-class relationship.
    → Such a subclass is called a shared subclass.

→ A shared subclass is a subset of the intersection of the superclasses (f.i. an ENGINEER_MANAGER must be an ENGINEER and a MANAGER).

- Category :
  A category is a subset of the union of the superclasses (f.i. an OWNER may be a COMPANY or a PERSON, an entity of OWNER must exist in at least one its superclasses).
  → EERD : A 'U' is written in the circle.
  → Selective inheritance :
  A category only inherits the attributes of one of its superclasses (not of all the superclasses like a shared subclass).
  → The completeness constraint also applies to categories (total or partial).

III.2.3. DATABASE DESIGN

III.2.3.1. THE ENTITY/RELATIONSHIP MODEL

- Each regular entity type maps into a base relation that includes all the simple attributes. Each of those base relations will have a primary key, corresponding to the key-properties in the ERD.

- For each weak entity type we create a relation that includes all simple attributes. The primary key of the owner entity type is also added as a foreign key.

- For each binary 1:1 relationship type, we identify the relations that correspond to the entity types participating in the relationship. We choose one of the relations and add the primary key of the other relation as a foreign key (it's better to choose an entity type with full participation the relationship). All the simple attributes of the relationship are also included as attributes.

- For each regular binary 1:N relationship type, we identify the relation that represents the participating entity type at the N-side and add the primary key of the other entity type that participates in the relationship as a foreign key.

- For each binary M:N relationship we create a new relation and include the primary keys of the two participating entity types as foreign keys. Their combination will form the primary key of the new relation. We also add the simple attributes of the relationship.

- For each multivalued attribute A we create a new relation R that includes an attribute corresponding to A and the primary key K of the relation that represents the entity type or relationship type that has A as an attribute. The primary key of R is then the combination of A and K.

- For each n-ary relationship type (n > 2) we create a new relation and include the primary keys of the involved entity types as foreign keys. Their combination will usually form the primary key of the new relation. We also add the simple attributes of the relationship.

III.2.3.2. THE ENHANCED ENTITY/RELATIONSHIP MODEL

Consider a superclass C with primary key attribute k, other attributes a_1,..., a_n and subclasses S_1,..., S_m. This can be mapped using the following options :

- Create a relation L for C with primary key k and the other attributes a_1,..., a_n. Also, create a relation L_i for each subclass S_i with primary key k and the attributes of S_i.
  → This option works for any constraint on the specialization.

- Create a relation L_i for each subclass S_i with primary key k, the attributes a_1,..., a_n and the attributes of S_i.
  → This option only works well with both the disjoint and total constraints on the specialization.

- Let the specialization be disjoint and t the attribute type that indicates the subclass to which each tuple belongs (the defining-attribute can serve the role of t). Create a relation L with primary key k, the attributes a_1,..., a_n, the attribute t and all the attributes of all the subclasses S_1,..., S_m.
  → This option is not recommended if many specific attributes are defined for the subclasses.

- Let the specialization be overlapping and t_1,..., t_m Boolean attribute types that indicate if a tuple belongs to subclass S_i. Create a relation L with primary key k, the attributes a_1,..., a_n, the attributes t_1,..., t_m and all the attributes of all the subclasses S_1,..., S_m.
  → This option is not recommended if many specific attributes are defined for the subclasses.

IV. DATABASE PROTECTION

IV.1. RECOVERY

IV.1.1. DEFINITIONS

- Recovery :
  Restoring the database to a state that is known to be correct after some failure has rendered the current state incorrect (or at least suspect).
  → It must be possible to construct any piece of information from some other information stored, redundantly (at the physical level, hidden form the user), somewhere else in the system.

- A transaction is a logical unit of work.

- A program is a sequence of several transactions.

- Kinds of failures
  * Local failure : affects only the transaction in which the failure has actually occurred.
  * Global failure : affects all of the transactions in progress at the time of the failure.
    - System failure (= soft crash, f.i. power failure) :
      affects all transactions currently in progress but doesn't physically damage the database.
    - Media failure (= hard crash, f.i. disk crash) :
      does damage to the database (or some portion) and affects the transactions currently using that portion.

IV.1.2. LOCAL FAILURES

IV.1.2.1. CONSISTENCY

A transaction transforms a consistent state of the database into another consistent state of the database, without necessarily preserving consistency at all intermediate points.

IV.1.2.2. ATOMICITY

- Transaction processing guarantees that if a transaction executes some updates and then a failure occurs before the transaction reaches its planned termination, then those updates will be undone (a sequence of operations is made atomic (= all-or-nothing) from an external point of view).

- This atomicity of transactions is provided by the transaction manager.
  * BEGIN TRANSACTION : begins a transaction.
  * COMMIT TRANSACTION :
    - A transaction has been successfully completed, all the updates can be made permanent
    - COMMIT establishes a commit point (= syncpoint).

* ROLLBACK TRANSACTION :
    - Something has gone wrong (the database might be inconsistent), all the updates so far, must be undone.
    - Rolls the database back to the previous commit point.

- A transaction can be rolled back, because the system maintains a log (= journal) on which details of all update operations are recorded (thus the log is used to undo the updates). In fact, there are two log portions :
  → The active (= online) log portion (normally, held on disk) :
  Used during normal system operation to record details of updates as they are performed.
  → The archive (= offline) log portion (usually held on tape) :
  When the active portion becomes full, its contents are transferred to the archive portion.

- Notes :
  * COMMIT and ROLLBACK terminate the transaction and when a transaction terminates, all database positioning (= tuple addressability, f.i. cursors) is lost.
  * Of course, the system must guarantee that individual statements are themselves atomic (this is very significantly in a relational system, where statements are set-level). Also some additional statements (f.i., foreign key cascade delete rule) must be atomic.
  * The system generates an implicit ROLLBACK for any program that fails to reach its planned termination (= RETURN statement).

IV.1.2.3. ISOLATION

Any given transaction's updates are concealed from all the rest, until that transaction commits.

IV.1.2.4. DURABILITY

- Once a transaction commits, its updates survive, even if there's a subsequent system crash.

- The write ahead log rule :
  The log must be physically written before commit processing can complete to ensure recovery from a system crash after a commit has been honored but before the updates have been physically written to the database (thus the log is also used to redo updates after a restart).

IV.1.3. GLOBAL FAILURES

IV.1.3.1. SYSTEM FAILURES

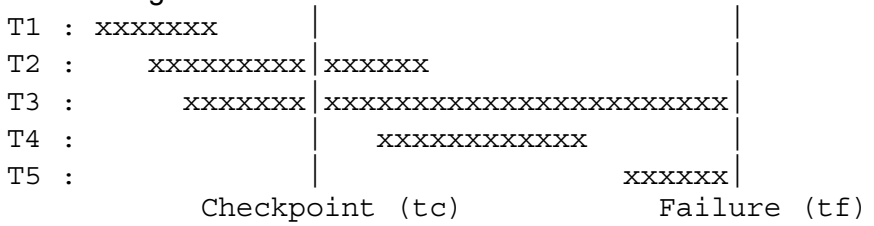
- Because the contents of main memory is lost, transactions that were in progress at the time of the crash can never be succesfully completed and so must be undone, when the system restarts.

- Also, transactions that did successfully complete prior to the crash but did not manage to get their updates transferred from the database buffer to the physical database, must be redone.

- To decide which transaction must be undone or redone at restart time, the system takes checkpoints after a prescribed number of entries has been written to the log.

- At a checkpoint :
  * the database buffers are written to the physical database.
  * a special checkpoint record is written in the physical log, containing a list of transaction in progress.

- Transaction catagories :



- Restart procedure :
  * Start with two lists, those of transactions to be UNDOne and those to be REDOne :
    - UNDO list = all transactions at last checkpoint (= tc)
    - REDO list = empty
  * Go through the log from the last checkpoint.
  * BEGIN TRANSACTION T : add T to the UNDO list.
  * COMMIT TRANSACTION T : move T from the UNDO list to the REDO list.

- Result : UNDO list : T3, T5; REDO list : T2, T4
  → T1 does not enter the restart procedure at all. Also transactions that completed unsuccessfully (= ROLLBACK) before the system failure (= tf) don't enter the restart procedure.

- Recovery :
  * Backward recovery :
    The system works backward through the log, undoing the UNDO-list transactions.
  * Forward recovery :
    The system works forward through the log, redoing the REDO-list transactions.

IV.1.3.2. MEDIA FAILURES

Recovery involves restoring the database from a backup copy (= dump) and to redo all transactions that completed since that backup was taken (using the log). Transactions that were still in progress at the failure time are lost (so they don't need to be undone).

IV.1.4. TWO PHASE COMMIT

- In a distributed database system one transaction can interact with two (or more) independent resource managers (each maintaining its own recovery log). Of course, the transaction must then commit both resources or none-at-all.

- The coordinator guarantees that both resource managers commit or rollback the update they are responsible for, even if the system fails in the middle of the process.

- Two phase commit protocol :
  * Each participant in the process must have force-write all log entries for local resources used by the transaction to its own physical log (they reply with "OK" or "NOT OK" to the coordinator).
  * After the coordinator has received all replies, it force-writes an entry to its own physical log with its decision (COMMIT if all replies were OK, else ROLLBACK). Each participant must then COMMIT or ROLLBACK the transaction locally as instructed.

- After a fail, the restart procedure will look for the decision record in the coordinator's log, so the process will be picked up where it was left-off (if that record can't be found, then the decision was assumable ROLLBACK).

- Note :
  After a participant replied the coordinator, then that participant becomes in wait mode till it receives the decision from the coordinator (which can take a long time if f.i. the coordinator fails). Any update by a transaction via that participant must be kept hidden (= locked) from other transactions.

IV.1.5. SQL

- SQL has a COMMIT and ROLLBACK statement, but not an explicit BEGIN TRANSACTION statement (this is done implicitly by a transaction-initiating statement (f.i. all data manipulating statements, cursor operations,...) if a transaction isn't already in progress.

- COMMIT and ROLLBACK force a CLOSE for every open cursor. However, some implementations (f.i. DB2) offer an option (WITH HOLD) to prevent this loss of positioning.

IV.2. CONCURRENCY

IV.2.1. DEFINITION

- Concurrency : many transactions have access to the same data at the same time.
  → A concurrency control mechanism is needed to ensure that concurrent transactions do not interfere with each other's operation.

- Concurrency problems (A1 : 40, A2 : 50, A3 : 30) :

Lost update		**Uncommitted dependency**		**Inconsistency analysis**	
A	B	A	B	A	B
retrieve p - update p -	- retrieve p - update p	- retrieve (*) p -	update p - rollback	retrieve A1 (sum = 40) retrieve A2 (sum = 90) - - - retrieve A3 (sum = 110)	- - update A3 (30 → 20) update A1 (40 → 50) commit -
LOST !		**LOST !**		**WRONG !**	

(*) Or : update p

IV.2.2. LOCKS

- Kinds of different locks :

	-	**Shared lock**	**Exclusive lock**
-	allowed	allowed	allowed
Shared lock	allowed	allowed	denied
Exclusive lock	allowed	denied	denied

Read lock = shared lock (= S)
Write lock = exclusive lock (= X)

- Data access protocol :
  * A transaction that wishes to retrieve a tuple must acquire a shared lock (this happens normally implicit).
  * A transaction that wishes to update a tuple must acquire an exclusive lock (this happens normally implicit).
  * If a lock request from transaction B is denied, then B turns in wait mode until A releases the lock. The system must guarantee that B doens't have to wait forever (= live lock), therefore lock requests are in first-come/first-served order.
  * A lock is normally held until the end of the transaction (= COMMIT or ROLLBACK).

- Concurrency problems (A1 : 40, A2 : 50, A3 : 30) :

Lost update		**Uncommitted dependency**		**Inconsistency analysis**	
A	B	A	B	A	B
retrieve p (acq.S)	-	-	update p (acq. X)	retrieve A1 (acq. S)	-
-	retrieve p (acq. S)	retrieve (*) p (req. S)	-	(sum = 40) retrieve A2	-
update p (req. X)	-	wait	rollback (rel. X)	(acq. S) (sum = 90)	
wait	update p (req. X)	retrieve (*) p (acq. S)	-	-	update A3 (acq. X)
wait	wait				(30 → 20)
				-	update A1 (req. X)
				retrieve A3 (req. S)	wait
				wait	wait
DEADLOCK		**SOLVED !**		**DEADLOCK**	

(*) Or : update p (request X-lock/acquire X-lock)

- Deadlock :
  * It must be detected and broken.
  * Detection involves detecting a cycle in the wait-for-graph (the graph of who's waiting for whom).
  * Breaking involves choosing one of the transactions as a victim, rolling it back and releasing its lock to allow the other transaction to proceed.

  Note :

  Some systems will automatically restart such a victim transaction, other systems just send a deadlock victim return code back to the application (it's always desirable to conceal the problem from the end-user).

IV.2.3. SERIALIZABILITY

- A schedule is any execution of a set of transactions.

- A serial schedule is a schedule where the transactions are executed one at a time.

- An interleaved schedule is a schedule that isn't serial.

- Two schedule are equivalent if they produce the same result independent of the initial state of the database.

- A schedule is correct (= serializable) if it's equivalent to some serial schedule (→ transactions are the unit of concurrency).
  Note :
  Two different serial schedules involving the same set of transactions might very well produce different results and yet both be considered correct !

- Two phase locking theorem :
  If all transactions obey the "two phase locking protocol", then all possible interleaved transactions are serializable

- Two phase locking protocol :
  * Before operating on any object, a transaction must acquire a lock on that object.
  * After releasing a lock, a transaction must never go on to acquire any more locks (→ locks are only released by a COMMIT or ROLLBACK instruction).

IV.2.4. ISOLATION LEVELS

An isolation level is the degree of interference that a transaction is prepared to tolerate on the part of concurrent transactions.
→ The higher the isolation level, the more the interference and the higher the concurrency.
→ An isolation level is generally regarded as a property of transactions.

IV.2.5. INTENT LOCKING

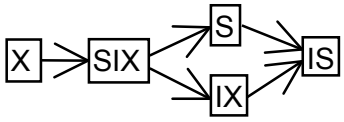
- The locking granularity is the unit for locking purposes (f.i. a tuple as previous used, but also an entire relation is a lock candidate).

- Kinds of relation locks :

	-	**Intent Shared**	**Shared**	**Intent Exclusive**	**Shared Intent Exclusive**	**Exclusive**
-	allowed	allowed	allowed	allowed	allowed	allowed
Intent Shared	allowed	allowed	allowed	allowed	allowed	denied
Shared	allowed	allowed	allowed	denied	denied	denied
Intent Exclusive	allowed	allowed	denied	allowed	denied	denied
Shared Intent Exclusive	allowed	allowed	denied	denied	denied	denied
Exclusive	allowed	denied	denied	denied	denied	denied

  * Intent Shared lock (= IS) : A transaction intends to set S-locks on tuples of a relation.
  * Shared lock (= S) : A transaction tolerates concurrent readers but not concurrent updaters in a relation.
  * Intent Exclusive lock (= IX) : A transaction intends to set X-locks on tuples of a relation.
  * Shared Intent Exclusive lock (= SIX) : S + IX-lock.
  * Exclusive lock (= X) : A transaction cannot tolerate any concurrent access to a relation.

- Precedence graph :
  *



  * The relative lock strength (f.i. is shared lock request fails, then a shared intent exclusive lock request will certainly fail).

- Intent locking protocol :
  * Before a transaction can acquire a shared lock on a tuple, it must first acquire an intend shared lock (or stronger) on the relation (this happens normally implicit).
  * Before a transaction can acquire an exclusive lock on a tuple, it must first acquire an intend exclusive lock (or stronger) on the relation (this happens normally implicit).

- Lock escalation is an attempt to balance the conflicting requirements of high concurrency and low lock management overhead.
  → When the system reaches some predefined threshold, it automatically replaces a set of fine granularity locks by a single coarse granularity check (f.i. many tuple level S locks and thus an IS relation lock is converted to one S lock on the relation).

IV.2.6. SQL

- Serializability can be violated in three ways :

Dirty read		**Nonrepeatable read**		**Phantoms**
A	B	A	B	
update p - rollback	- retrieve p -	retrieve p - retrieve p	- update p -	- A retrieves a set of tuples that satisfy a condition. - B inserts a new tuple that also satisfies that condition. - A repeats the same retrieval and gets a “phantom” tuple.
VALUE THAT NEVER EXISTED !		**TWO DIFFERENT VALUES !**		

- Isolation levels :

Isolation level	**Dirty read**	**Nonrepeatable read**	**Phantoms**
Read uncommitted	allowed	allowed	allowed
Read committed	not allowed	allowed	allowed
Repeatable read	not allowed	not allowed	allowed
Serializable	not allowed	not allowed	not allowed

  The system prevents phantoms occuring with a lock to the access path used to get to the data under consideration (f.i. an index entry).

- The isolation level for the next transaction to be initiated is set with :
  SET TRANSACTION option
  With option :
  * an access mode : READ ONLY or READ WRITE
  * or the isolation level : ISOLATION LEVEL isolation
    With isolation : READ UNCOMMITED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE

Note : SET TRANSACTION can only be executed when no transaction is in progress and is not transaction-initiating itself.

- SQL does not provide an explicit LOCK statement.

- Differences with DB2 :
  * Isolation levels :
    - REPEATABLE READ (= maximum level) : all schedules are serializable
      → Equal to the SERIALIZABLE level from SQL.
    - CURSOR STABILITY :
      If a transaction :
      - obtains addressability to a tuple p
      - (and thus) acquires a shared lock on p
      - releases its addressability to p WITHOUT updating it (there's no promotion to an exclusive lock)

      Then the lock can be released before the end of the transaction.
      → Equal to the READ COMMITTED level from SQL.
  * DB2 supports an explicit LOCK statement to acquire S, X or IX locks at the relation level.

IV.3. SECURITY

IV.3.1. DEFINITION

- Security : protection of data against unauthorized disclosure.
  → we'll consider only the database system aspects, not the legal, social or physical, hardware and operating system control ones.

- Two types of security control :
  * discretionary :
    Different users have different access right (= privileges = authorities) on different objects and different users will have different rights on the same object (→ flexible)
  * mandatory :
    Each data object is labeled with a classification level and each user is given a clearance level. A data object can only be accessed by users with an appropriate clearance level (→ rigid).

- Notes :
  * The policy decisions (who may have access to what) must be made known to (and remembered by) the system (statements in a definitional language are saved in the catalog in the form of security rules (= authorization rules).
  * The checking of a given access request (= requested operation, requested object and requested user) against the applicable security rules is done by the DBMS's security subsystem (= authorization subsystem).
  * The system must be able to recognize the requested user, so every user must log in with an user ID and a password to prove their identity.
  * Distinct users may be grouped together in a user group (they all share the same privileges). Which user is in which group is also kept in the catalog.

IV.3.2. DISCRETIONARY ACCESS CONTROL

- Each rule has :
  * a name : the rule will be kept in the catalog under this name.

* privileges :
  RETRIEVE [(attribute-commalist)], INSERT, DELETE,
  UPDATE [(attribute-commalist)], ALL
* a scope :
  A relational calculus (or algebra) expression that defines the scope of the rule (a subset of tuples of a single named relation).
  → Value independent rule : The scope is a complete table (vertical = attribute subset)
  → Value dependent rule :
    The scope contains a specific value for a specific attribute (horizontal = tupple subset)
  → Context dependent rule :
    A request will or not will violate the rule depending on the context (f.i. day or time specifications).
* user ID's : the users to which the rule applies (ALL can also be specified)
* an action :
  What should happen on a violation (f.i. REJECT by default or some other procedure of arbitrary complexity)

- Security syntax :
  * Creating a rule :
    CREATE SECURITY RULE rule
    GRANT privileges-commalist
    ON expression
    TO user-commalist
    [ON ATTEMPTED VIOLATION action];
  * Deleting a rule :
    DESTROY SECURITY RULE rule;
  * Example :
    CREATE SECURITY RULE SR
    GRANT RETRIEVE (S#, SNAME, CITY), DELETE
    ON S WHERE S.CITY ≠ 'London'
    TO Jim, Fred, Mary
    ON ATTEMPTED VIOLATION REJECT;

- Audit trail :
  A specific file (or database) in which the system automatically keeps track of all operations performed by users on the regular database.
  → request (source text), terminal, user, date and time, relation(s) / tuple(s) / attribute(s) affected, old / new values.
  Note : Sometimes the audit trail is integrated with the recovery log.

IV.3.3. MANDATORY ACCESS CONTROL

- Mainly applicable to databases with rather static and rigid classification structure (f.i. military or government environments).

- Each data object receives a classification level and each user receives a clearance level.

- Levels must have a strict ordening (f.i., top secret > secret > confidential)

- Rules :
  * User i can see object j, only if the clearance level of user i is greater than or equal to the classification level of object j.
  * User i can modify object j, only if the clearance level of user i is equal to the classification level of object j.

Note :
Everything written by user i, gets the classification level equal to the clearance level of user i.

- A DBMS that supports also mandatory access control is called a multilevel secure system or a trusted system.

- The U.S. Department of Defence (DoD), which requires any system it uses to support certain mandatory controls, has two books :
  * Orange book :
    Which defines a set of security requirements for any “Trusted Computing Base” (= TCB).
  * Lavender book :
    Which defines an interpretation of the TCB requirements for database systems.

IV.3.4. SECURITY CLASSES

- Discretionary protection (Class C) : discretionary access control
  * C1 : requires separation of data and users (users have shared but also private data).
  * C2 : C1 + accountability support (sign-on-procedures, auditing, resource isolation,...)

- Structured protection (Class B) : mandatory access control
  * B1 : requires a classification level for each data object and an informal statement of the security policy in effect.
  * B2 : B1 + formal statement of the same thing. It also requires that covert channels be identified and eliminated (covert channel : the answer to an illegal query is deduced from the answer of a legal query).
  * B3 : B2 + audit and recovery support plus a designated security administrator.

- Verified protection (Class A) :
  Requires a mathematical proof that the security system is consistent and adequate to support the specified security policy.

IV.3.5. DATA ENCRYPTION

IV.3.5.1. DEFINITIONS

- Of course, we must also offer security if an infiltrator bypasses the system and f.i. tries to copy a physical part of the database. This is done by data encryption (storing and transmitting sensitive data in an encrypted form).

- Plaintext : the original (unencrypted) data
  ↓ Encyption algorithm + encryption key
  Ciphertext : the encrypted data

- There are three basic ways to encrypt data :
  * substitution :
    For each character in the plaintext, a cipher character is substituted for that character.
  * permutation (= transposition) : The plaintext characters are simply rearranged.
  * algebraic : The plaintext is encrypted with some algebraic calculation method.

- Polyphase : the encryption algorithm is used more than once (↔ monophase).

- Monographic : each plaintext character is encrypted by another character
  (↔ polygraphic : two or more plaintext characters are encrypted by one character).

IV.3.5.2. DATA ENCRYPTION STANDARD (= DES)

A combination of the substitution and permutation methods (decryption algorithm is equal to the encryption algorithm, except for the order of the permutations).

IV.3.5.3. TRANSPOSITION METHODS

- Rail fence cipher system, example (+ stands for a space) :

```
PLAIN  : CANCEL+ALPI
         NE+CONTRACT
CIPHER : CN AE N+ CC EO LN +T AR LA PC IT
```

  → The matrix dimensions must be known

- Keyed column system, example :

```
KEY    : DIRECTON
ORDER  : 24731865 (alphabetic order of the characters in the key)
PLAIN  : CANCELAL
         PINECONT
         RACT????
CIPHER : EC? CPR CET AIA LT? AN? NNC LO?
```

- Also possible : adding dummy characters (f.i. ?) at regular places in the message.

IV.3.5.4. SUBSTITUTION METHODS

- Mono alphabetic system : simple substitution based on a cipher alphabet (without a key)
  * Example :

```
PLAIN  : A B C ... Q R S T U V W X Y Z
CIPHER : L K J ... C I H P A R G N O M
```

  Note : MON(O)GRAPHIC is written backwards at the end.
  * Example :

```
PLAIN  : A C D F G P U B L I S H E R ... X Y Z
CIPHER : L E X I C O G R A P H Y B D ... V W Z
```

  * Example :

```
   K M O Q S
   L N P R T
AB P U B L IJ
CD S H E R A
EF C D F G K
GH M N O Q T
IJ V W X Y Z

PLAIN  : GREAT
CIPHER : EFQR CDQR CDOP CDST GHST
```

- Poly alphabetic system : works with a Vigenere table (26 x 26) and a key, example :

```
  A B C D ... X Y Z
A A B C D ... X Y Z
B B C D E ... Y Z A
C C D E F ... Z A B
...
Z Z A B C ... W X Y

KEY    : STOPWATCHSTOPWATCHST
```

```
PLAIN  : CANCELALPINECONTRACT
CIPHER : UT...
```

- Digraph system (= Playfair cipher) :
  * Explanation :
    Two plaintext characters are encrypted with a character matrix. Call (r_1, c_1) and (r_2, c_2) the place of the plaintext characters, then these are replaced by the (r_1, c_2) and (r_2, c_1) characters from the matrix.
  * Example :

```
C O M P U
T E R A B
D F G H IJ
K L N Q S
V W X Y Z

PLAIN          : FR EN CH CO NN EC TI ON
MODIFIED PLAIN : FR EN CH CO NQ NE CT IO N
CIPHER         : GE RL PD OC QN LR CT FU N
```

  * Note :
    When two equal characters follow each other, then a dummy character is added to the plaintext (f.i. Q).

- Variation of the digraph system, example :

```
  1 2 3 4 5
1 P U B L IJ
2 S H E R A
3 C D F G K
4 M N O Q T
5 V W X Y Z

PLAIN     : SELL S   HORT B   EFORE    THURS   DAY
            2244 2
            1311 1
REORDERED : 2423 1
            2411 1
CIPHER :    HQPP P
```

IV.3.5.5. PUBLIC KEY ENCRYPTION

- There are two keys : an encryption and a decryption key (the person who encrypts data cannot perform the corresponding decryption if he's not authorized to do so).

- The RSA (= Rivest, Shamir and Adleman) scheme is based upon :
  * there's a fast algorithm to determine if a given number is prime.
  * there's no fast algorithm to find the prime factors of a given composite number.

- The RSA works as follow :
  * choose p and q, two distinct large primes
  * r = p * q
  * choose a large integer e (= encryption key) that is relatively prime to (p - 1) * (q - 1) (f.i., any prime greater than both p and q).
  * d (= decryption key) is the multiplicative inverse of e modulo (p - 1) * (q - 1)
    $\rightarrow$ d * e = 1 modulo (p - 1) * (q - 1)
  * publish r and e (not d)
  * $C = P^e \bmod r$
  * $P = C^d \bmod r$

- Public key encryption schemes also permit encrypted messages to be signed, example :
  * A : ECB (Encryption algorithm for data to B), DCA (Decryption algorithm of A)
  * B : ECA, DCB
  * Cipher message : C = ECB (DCA(P))
  * Plain message : P = ECA(DCB(C)) = ECA(DCB(ECB(DCA(P)))) = ECA(DCA(P)) = P
  → B knows that C comes from A.

IV.3.6. SQL

- SQL supports only discretionary access controls

- Views :
  * They provide an important measure of security “for free” (views are included in the system for other purposes anyway).
  * However, this approach does suffer from some awkwardness on some occasions (f.i. if a user needs different privileges over different subsets of the same table at the same time).

- GRANT and REVOKE :
  * However, the view mechanism, doesn’t allow the specification of operations that authorized users are allowed to execute against that data.
  * The creator of any object is automatically granted all possible privileges and also has grant authority for these objects (he can grant privileges to other users).
  * General syntax :

    GRANT privileges-commalist
    ON object
    TO user-commalist [WITH GRANT OPTION];

    REVOKE [GRANT OPTION FOR] privileges-commalist
    ON object
    FROM user-commalist
    option;
  * privileges :

    USAGE, REFERENCES, SELECT, INSERT [(attribute_commalist)],
    DELETE [(attribute_commalist)], UPDATE [(attribute_commalist)]

    → USAGE is needed to refer to a specific domain in a constraint
    → REFERENCES is needed to refer to a specific named table in a constraint
  * object :

    DOMAIN domain | [TABLE] table

    Table can also be a view name (→ the view mechanism does also require the use of a grant instruction).
  * user : PUBLIC may also be specified
  * WITH GRANT OPTION : also grants grant authority
  * GRANT OPTION FOR : revokes only the grant authority
  * option :

    RESTRICT : the revoke fails if that user gave these privileges to another user.
    CASCADE : revokes these privileges from all users along the way
  * Note :

    Dropping a domain, attribute or table (or view) automatically revokes all privileges on the dropped object from all users.

- The security system of QUEL :
  * Any given QUEL request is automatically modified before execution in such a way that it cannot possibly violate any security rule.
  * Example :

- Suppose the following security rule :
  DEFINE PERMIT RETRIEVE ON P TO U WHERE P.CITY = “London”;
- Suppose user U has the following request :
  RETRIEVE (P.P#, P.WEIGHT) WHERE P.COLOR = “Red”;
- Then, the request will be executed as follows :
  RETRIEVE (P.P#, P.WEIGHT)
  WHERE P.COLOR = “Red”
  AND P.CITY = “London”:

* Advantages :
  - User U is not informed that his request is modified (he might not even be allowed to know that there are any parts not stores in London).
  - It is very easy to implement because the same technique is used for views (so much of the necessary code already exists).
  - It’s comparatively efficient because the overhead occurs at compilation time (not at execution time).
  - Some of the awkwardness of the SQL view approach can not arise.
* Disadvantages :
  Not all security rules can be handled in this way (f.i., if U isn’t allowed to access relation P at all).

IV.4. INTEGRITY

IV.4.1. DEFINITIONS

- Integrity :
  The correctness of the data in the database (→ the database is correct : it doesn’t violate any known integrity rules, the database is in a state of integrity).

- Integrity rules are not user specific (as security rules are).

- Integrity rules are also kept in the system catalog and are enforced by the DBMS’s integrity subsystem.

- Kinds of integrity rules :
  * state rules : concerned with the correct state of the database
  * transition rules : concerned with transitions from one state to another

- Each integrity rule has :
  * a name : the rule will be kept in the catalog under this name.
  * a constraint, specified by a truth valued expression (from the relational calculus). The rule is satisfied if the constraint evaluates to true.
  * an action : What should happen on a violation, f.i. REJECT by default or some other procedure of arbitrary complexity. Such a procedure can perform a compensating update somewhere else in the database and must be considered as a part of the update operation. The constraint must also be performed again after that procedure has executed, to ensure that the procedure doesn’t leave the database in a state that violates the constraint.

- Integrity syntax :
  * Creating a rule :
    CREATE INTEGRITY RULE rule
    constraint
    [ON ATTEMPTED VIOLATION action];

* Deleting a rule :
  DESTROY INTEGRITY RULE rule;
* Example :
  CREATE INTEGRITY RULE IR1
  FORALL PX (PX.WEIGHT > 0)
  ON ATTEMPTED VIOLATION REJECT;

- When a create integrity rule statement is executed, the system must first check to see if the current state of the database satisfies the new constraint. If it does not, the rule must be rejected.

- The constraint of an integrity rule can almost always start with an universal quantifier so we adopt the convention that any variable that is not explicitly quantified is assumed to be universal quantified.

IV.4.2. STATE INTEGRITY RULES

IV.4.2.1. DOMAIN INTEGRITY RULES

- A domain integrity rule specifies the legal values for a domain. Because this is the definition of a domain, a separated CREATE DOMAIN INTEGRITY RULE isn't needed.

- A domain rule has the same name of the domain.

- The domain rule constraint (= domain constraint) must :
  * be universally quantified
  * range over the domain in question.

- Because a domain can never be updated (attributes are updated instead), a domain rule is never checked (so a violation response is not needed).

- Many shorthands can be provided, f.i.: VALUES (literal-commalist), example :
  CREATE DOMAIN COLOR CHAR(6) VALUES ('Red', 'Green', 'Blue');
  ≡ CREATE DOMAIN COLOR CHAR(6)
  FORALL COLOR (COLOR = 'Red' OR COLOR = 'Green' OR COLOR = 'Blue');

- A domain rule can only be destroyed by destroying the domain.

IV.4.2.2. ATTRIBUTE INTEGRITY RULES

- An attribute integrity rule specifies the legal values for an attribute. Because this is the definition of the domain from which that attribute draws its values, a separated CREATE ATTRIBUTE INTEGRITY RULE isn't needed.

- An attribute rule has the same name of the corresponding domain integrity rule (= the name of the domain).

- The attribute rule constraint (= attribute constraint) is derived from the relevant domain constraint.

- An attribute rule is always checked immediately and the violation response is REJECT.

- An attribute rule can only be destroyed by destroying the attribute itself.

IV.4.2.3. RELATION INTEGRITY RULES

- A relation integrity rule specifies the legal values for a relation, f.i.,
  ```
  CREATE INTEGRITY RULE IR2
  FORALL S (IF S.CITY = 'London' THEN S.STATUS = 20)
  ON ATTEMPTED VIOLATION REJECT;
  ```

- A relation rule has a new name.

- The relation rule constraint (= relation constraint) is a closed WFF of the relational calculus (all the variables must range over the same relation).

- A relation rule is always checked immediately (every update operation includes the checking of all relation rules)

- Relation predicate :
  The logical AND of all relation constraints and all attribute constraints that apply to that relation ($\rightarrow$ the criterion for update acceptability).

- Example :
  ```
  CREATE INTEGRITY RULE IR3
  IF SX.S# = SY.S# THEN
      SX.SNAME = SY.SNAME
      AND SX.STATUS = SY.STATUS
      AND SX.CITY = SY.CITY;
  ```
  This can also be specified as a part of the base relation S by : CANDIDATE KEY (S#)

IV.4.2.4. DATABASE INTEGRITY RULE

- A database integrity rule specifies the legal values for a database, f.i.,
  ```
  CREATE INTEGRITY RULE IR4
  FORALL SX (FORALL SPX
      (IF SX.STATUS < 20 AND SX.S# = SPX.S# THEN SPX.QTY < 500));
  ```

- A database rule has a new name.

- The database rule constraint (= database constraint) must include at least one join condition involving two distinct bound variables, ranging over two distinct relations.

- A database rule is not checked immediately, it is checked at the end of the transaction (= COMMIT) because, possibly, several update operations will be needed to update all the relations in a consistent way ($\rightarrow$ transactions are the unit of integrity)

- The default action is ROLLBACK, and no other action will make much sense.

- Database predicate :
  The logical AND of all database constraints and all relation predicates that apply to that database.

- Example :
  ```
  CREATE INTEGRITY RULE IR4
  FORALL SP (EXISTS S (S.S# = SP.S#));
  ```
  This can also be specified as a part of the base relation SP by :
  ```
  FOREIGN KEY (S#) REFERENCES S
  ```

IV.4.3. TRANSITION INTEGRITY RULES

- We introduce the convention that a primed variable (f.i. : S') is understood to range over the corresponding relation as it was prior to the update operation under consideration.

- Example of a transition relation rule :
  CREATE INTEGRITY RULE IR5
  FORALL S' FORALL S (IF S'.S# = S.S# THEN S'.STATUS ≤ S.STATUS);
  → No supplier's status must ever decrease.

- Example of a transition database rule :
  CREATE INTEGRITY RULE IR6
  FORALL S
  (SUM (SP WHERE SP.S# = S.S#, QTY) ≥
  SUM (SP' WHERE SP'.S# = S.S#, QTY));
  → The total quantity of any given part, taken over all suppliers can only increase.

- Transition rules have no meaning for attribute and domain rules.

IV.4.4. SQL

- SQL has only three categories :
  * domain rules : they can involve a constraint of arbitrary complexity.
  * base table rules :
    - they can involve a constraint of arbitrary complexity.
    - is always satisfied if the table is empty (even if the rule says : "this table must not be empty").
  * general rules (= assertions)

- General rules :
  CREATE ASSERTION rule CHECK (conditional_expression);
  DROP ASSERTION rule;

- Any SQL integrity rule can be declared to be DEFERRABLE or NOT DEFERRABLE. If it's deferrable its state at the beginning of a transaction can be declared INITIALLY DEFERRED or INITIALLY IMMEDIATE. Deferrable rules can be switched on and off dynamically :
  SET CONSTRAINTS rule option
  Where option can be IMMEDIATE or DEFERRED.

V. THE INVERTED LIST MODEL

V.1. BACKGROUND

- Many relational systems can be thought of as inverted list systems at the internal level. Those functions are not exposed to the user but are used by other high-level components that provide the true human interface (f.i. SQL).

- There is in fact no formal, abstract defined inverted list model.

- Example : CA-DATACOM/DB

V.2. THE INVERTED LIST MODEL

V.2.1. DATA STRUCTURE

- A database is a collection of tables (files) with rows (records) and columns (fields).

- There is a database sequence.
  → Rows are ordered in a physical sequence (f.i. the rows of two tables can be interleaved).

- Search keys can be defined :
  * a field (or combination of fields) over which an index is to be built.
  * They permit :
    - direct access
    - sequential access (other than the physical sequence)

- The stored tables and their access paths (= indexes) are visible to the user (in case of the inverted list model, a user means an application programmer).

V.2.2. DATA MANIPULATION

- Data manipulation is record-at-a-time.

- Search (= locate, find) operators : provide record addressing :
  * direct search operators :
    → Put the address of the record at a given position (f.i. first) and access path (f.i. in physical sequence) in a database address variable (the used access path is also stored).
  * relative search operators :
    → Move to the next record given a variable (that contains an access path) and a condition (f.i. equal search key).

- Manipulation operators : operate on a given record address (f.i. delete, retrieve, update,...).

V.2.3. DATA INTEGRITY

Not provided : it's the user's responsability.

V.3. DATACOM/DB

V.3.1. OVERVIEW

- DATACOM/DB provides a CALL-level interface for application programs written in COBOL, FORTRAN, PL/I,... The interface for COBOL is called COBOL/DL which consists of a set of extensions to COBOL that are expanded via a preprocessor into conventional CALL's.

- Each database (up to 999) is defined by adding a set of descriptors to the DATADICTIONARY (this process has a form-driven interface). The descriptors specify more or less their version of the ANSI/SPARC internal, conceptual and external level.

- The unit of access is an element.
  → An element is a collection of contiguous fields (= a subrecord).

- The system does provide a set of recovery, concurrency and security controls at the element, table and database level.

- Integrity is the responsability of the user (except for the master key, a certain search key, which can have duplicates not allowed).

V.3.2. DATA DEFINITION

- A DATACOM/DB database can be thought of as a collection of stored records, each stored record belonging to exactly one table, together with a single B-tree index over all of the records in the database (which supports all indexes).

- Each table has :
  * TABLEID : A unique internal identifier to the table.
  * FIELDS : A definition of the field names and their data types.
  * ELEMENTS : A combination of contiguous fields (which can be retrieved separately).
  * SEARCH KEYS : Defined by a field name and a KEYID.
  * MASTER KEY :
    Exactly one of the search keys which can have DUPLICATES NOT ALLOWED and UPDATES NOT ALLOWED (→ there must be at least one search key).
  * NATIVE KEY :
    Exactly one of the search keys which controls the physical clustering (the native key can be optionally equal to the master key).

- The B-tree index : search key ID + search key value + table ID
  → The B-tree finally contains pointers to the records.
  → When two tables specify an equal search key ID, they will have adjacent index entries.
  → Every record has at least one index entry because there must be at least one search key (because every table must have both a master and a native key).
  → Records can have multiple index entries.

V.3.3. DATA MANIPULATION

- Direct search operators :
  * GSETP : Locate (and read) first record in physical sequence.
  * LOCKX : Locate first record with specified search key equal to specified value.

* LOCKY :
  Locate first record with specified search key greater than or equal to the specified value.
* REDKY/RDUKY : LOCKX + REDLE/RDULE.

- Relative search operators :
  * GETPS : Locate (and read) next record in physical sequence.
  * LOCNX : Locate next record.
  * LOCBR : Locate previous record (BackwaRd search).
  * LOCNE : Locate next record with the same search key value.
  * LOCNK : Locate first record with greater search key value.
  * LOCKL : Locate first record with less than or equal search key value.
  * REDNX/RDUNX/REDNE/RDUNE : LOCNX/LOCNE + REDLE/RDULE.

- Manipulation operators (operate on previously located record) :
  * REDLE : Read located record.
  * RDULE : Read located record for update (set exclusive lock).
  * DELET/UPDAT : Delete/update located record.
  * RELES : Release exclusive lock.
  * ADDIT : Store new record (does not establish addressability).

- Operands :
  * Request area :
    The database address variable (position and access path) and also a return code (blank for success).
  * I/O area : Required for manipulation operators.
  * Element list : Required for manipulation operators (the elements to be manipulated).

V.3.4. COMPOUND BOOLEAN SELECTION FEATURE (CBS)

- Offers increased flexibility and data independence at the application programming interface (a kind of embedded SQL).

- Operations :
  * SELFR : DECLARE CURSOR, OPEN, FETCH
    → The logical access path must not be predefined, it can be specified dynamically.
    → These logical access paths can be defined in terms of arbitrary restriction conditions, involving any fields (not even restricted to predefined search keys).
    → The ordering can also be specified dynamically.
    → But : the condition is restricted to one table, it cannot involve a join.
    → Optional specifications :
      - UNIQUE : Skip records with same ordering field value.
      - COUNT
      - n : Retrieve the first-n-records.
      - FOR UPDATE : Applies an exclusive lock.
      - Interrupt limit : maximum number of retrieved records.
        - SELPR : Cancel SELFR.
        - SELST : Accept partially built set.
        - SELCN : Continue search.
  * SELNR : FETCH (with optionally a skip count : +n or -n).
  * SELSM : Select the same record again.
  * UPDAT/DELET : UPDATE/DELETE CURRENT
  * SELPR : CLOSE CURSOR

VI. THE HIERARCHIC MODEL

VI.1. BACKGROUND

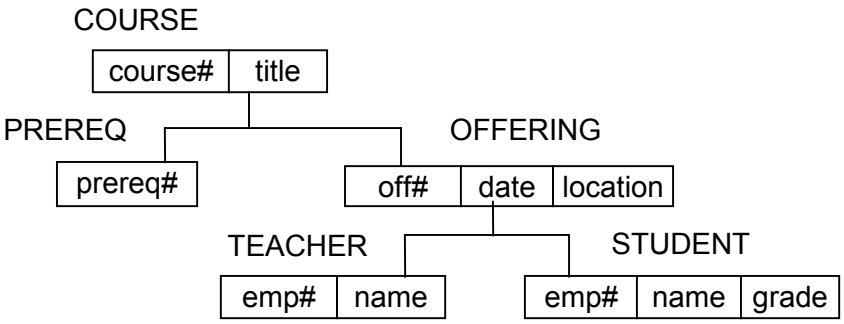
- The hierarchic data model was actually constructed after the implementation of hierarchic database systems.

- Example : IBM-IMS/VS : Information Management System/Virtual Storage (1968).

VI.2. THE HIERARCHIC MODEL

VI.2.1. DATA STRUCTURE

- Tree type, f.i. :



  → Certain information that is represented by foreign keys in a relational database is represented by a parent-child link in a hierarchic database.

- Tree occurrence :
  A single root record occurrence with an ordered set of zero or more occurrences of the subtree types immediatly dependent on the root record type.
  → Twins are occurrences of a given child type that share a common parent occurrence.

- Each individual tree can be regarded as a subtree of a hypothetical 'system' root record.
  → A hierarchic database consists of an ordered set consisting of multiple occurrences of a single type of tree.
  → The database is logically stored in depth-first order.

- A user of a hierarchic database means an application programmer.

VI.2.2. DATA MANIPULATION

- Data manipulation is record-at-a-time.

- Tree traversal operators :
  Locate a specific tree, move to the next tree, move from record to record (up and down in the tree), move in the hierarchic sequence,...

- Manipulation operators : Insert, delete, update,...

VI.2.3. DATA INTEGRITY

- Certain referential integrity support.

- No child is allowed to exist without its parent.

VI.3. IMS/VS

VI.3.1. OVERVIEW

- IMS is invoked via a CALL interface, called DL/I (Data Language/I), from application programs written in COBOL, FORTRAN, PL/I,...

- Recovery : Very extensive and sophisticated support.

- Concurrency : Based on record (called segments in IMS) locking.
  → No automatically support for two-phase locking (more like cursor stability from DB2).

- Security : A kind of views is used to hide information (and to specify the allowed operations).

- Integrity :
  * Certain field uniqueness constraints.
  * Certain referential integrity constraints.
  * Certain explicit insert/update/delete rules.

- IMS/VS is extremely complicated, not only internally but also externally (the user interface).

VI.3.2. DATA DEFINITION

- Database description (DBD) : Specifies the hierarchic structure of the database.
  * physical database : The DBD describes the physical storage.
  * logical database :
    The DBD describes the representation of that database in terms of one or more other (physical) databases.

- Program Communication Block (PCB) :
  * Specifies the hierarchic structure of a view to a database, a subhierarchy with possibly, rearranged segments and fields (= a secondary data structure).
  * Derivation rules :
    - Any field type can be omitted.
    - Any segment type can be omitted.
    - If a given segment type is omitted, then all of its children must be omitted too.
    → The PCB and DBD hierarchy must have the same root.

VI.3.3. DATA MANIPULATION

- A PCB can be regarded as the IMS equivalent of a cursor and a corresponding feedback area (= request area = communication area) in combination (together with a return code).

→ This cursor/feedback area is an internalized form of the corresponding external PCB (also referred as PCB).

- Operators :
  * GU : Get Unique (direct retrieval).
  * GN : Get Next (sequential retrieval).
  * GNP : Get Next within Parent (sequential retrieval under the same parent).
  * ISRT : InSeRT
  * DLET : DeLETe
  * REPL : REPLace (update)
  * GHU, GHN, GHNP : Get Holds (gets allowing subsequent DLET/REPL).

- Segment Search Arguments (= SSA) :
  A segment name with a command code and a restriction predicate (both optional).

VI.3.4. STORAGE STRUCTURE

- A physical database is represented by a stored database. Each segment of a physical database is represented by a stored segment and a stored prefix (pointers, flags, control information,...).

	Hierarchic Sequential (HS)		Hierarchic Direct (HD)	
	Uses physical contiguity		Uses pointers	
	Mainly sequential access		Mainly direct access	
	Hierarchic Sequential Access Method (HSAM)	Hierarchic Indexed Sequential Access Method (HISAM)	Hierarchic Direct Access Method (HDAM)	Hierarchic Indexed Direct Access Method (HIDAM)
Root segments	Sequential	Indexed	Hashed	Indexed
Dependent segments	Sequential	Sequential	Pointers	Pointers

- Both HD's can use two types of pointers :
  * Hierarchic pointers :
    Each segment points to the next in hierarchic sequence (better sequential access).
  * Child/twin pointers :
    Each parent points to its first child of each type and each child points to its next twin (better direct access).

- Other storage structures : simple HSAM, simple HISAM, Data Entry Databases (DEDB),...

VI.3.5. LOGICAL DATABASES

- A logical database is a hierarchic arrangement of segments that really belong to one or more physical databases.
  * The arrangement of the logical database can differ from the physical ones.
  * A logical database is defined by a logical DBD.

- Differences between a logical DBD and a PCB :
  * A logical DBD can be defined over multiple physical DBD's. A PCB must be defined over a single physical or logical DBD.
  * A view defined by a logical DBD can differ much more from the physical structure than a PCB can.
  * A view defined by a logical DBD is directly supported by its own physical pointers.

- A segment in a logical database can have :
  * no parents.
  * one physical parent.
  * one physical parent (through the tree type) and one logical parent (through a pointer).
  → Two child segment occurrences can be physical or logical twins.

- Two child segment types that point to each others parent segment type are called paired segments (f.i. a symmetric version of two physical databases).

- For every segment that participates in a logical relationship (a logical parent, a logical child or a physical parent with a physical child that is also a logical child), the physical DBD for that segment must specify an insert, update, delete and replace rule. Each rule can be :
  * physical
  * logical
  * virtual

VI.3.6. SECONDARY INDEXES

- A secondary index is an index on a field other than the primary key.

- A secondary index is not transparant to the user. The user must explicitly instruct IMS to use it. The definition of the index must be included in the DBD and specified in the DL/I statement.

- A secondary index allows the database to be processed in a secondary processing sequence.

- There are 4 possible indexes :
  * Indexing the root on a field other than the sequence one.
  * Indexing the root on a field in a dependent.
    → The index can be much larger because segments can appear multiple times.
  * Indexing a dependent on a field in that dependent.
    → The hierarchy is restructed as follow :
      - The indexed segment becomes the root.
      - Ancestors of that segment become the leftmost dependents of the root, in reverse order.
      - Dependents of the indexed segments appear as they were before (right of the former ancestors).
      - No other segments are included (f.i. other children from the former parent).
  * Indexing a dependent on a field in a lower-level dependent.

VI.4. CONCLUDING REMARKS

VI.4.1. CRITICAL COMMENTS ON HIERARCHIC SYSTEMS

- A hierarchic structure has a built-in bias.
  → It is good for some applications but bad for others.
  → A logical restructuring mechanism is desirable.
  → This is offered by :
    * secondary indexes
    * logical databases
    * PCB's

→ But : these solutions are not flexible :
* There are to many possible hierarchies to support them all.
* For some applications no hierarchy is suitable.

- There's no clear distinction between the logical and physical level (f.i. secondary indexes are both at the physical, performance-oriented, and logical, datastructuring, level).

VI.4.2. CRITICAL COMMENTS ON IMS

- The restructuring mechanism is cumbersome and not very dynamic (it must be performed by an IMS specialist).

- The choice of using an index is at the hands of the user, not the system.

- The insert/delete and replace rules are very complex and do not provide all the functions needed.

VII. THE NETWORK MODEL

VII.1. BACKGROUND

- There is in fact no formal, abstract defined network model.

- Example : CA-Integrated Database Management System (IDMS), based on the Database Task Group (DBTG) report (1971) by the CODASYL organisation (Conference On DAtabase SYstems Languages).

- The DBTG proposed three database languages :
  * Schema Data Description Language (= schema DDL) :
    → Describes the network structured database (this schema corresponds with the ANSI/SPARC internal and conceptual level).
  * Subschema Data Description Language (= subschema DDL) :
    → Defines an external view (= ANSI/SPARC external level).
  * Data Manipulation Language (= DML)

VII.2. THE NETWORK MODEL

VII.2.1. DATA STRUCTURE

- A network database consists of 2 sets :
  * A set of multiple occurrences of each of several types of records.
  * A set of multiple occurrences of each of several types of links (each involving 2 record types).

  Each occurrence of a given link type consists of a single occurrence of the parent record type, together with an ordered set of multiple occurrences of the child record type.

- Bachman diagrams : A technique for drawing link types :
  * A link is labeled with its name.
  * The link is directed from the parent to the children.

- There's no restriction on how record types can be combined into link types.

- Sometimes, all root records are regarded as children from a 'system' record.

VII.2.2. DATA MANIPULATION

- Data manipulation is record-at-a-time.

- Traversal operators :
  Locate a record, move from a parent to its first child, move from a child to the next child, move from a child to its parent within some link,...

- Manipulation operators :
  * Create, destroy and update a record.
  * Connect and disconnect 2 records.

VII.2.3. DATA INTEGRITY

Certain referential integrity rules are builtin (by virtue of links).

VII.3. IDMS

VII.3.1. OVERVIEW

- An IDMS database is defined by a schema (written using the schema DDL) that defines the records in the database, the elements (= fields) they contain and the sets (= links) in which they participate as either owner (= parent) or member (= child).

- Users interact with a view defined by a subschema (written using the subschema DDL), which is a subset of the schema.

- Both schemas are compiled and stored in the IDMS dictionary.

- IDMS provides an interface together with a preprocessor that translates those DML statements into the appropriate host language calling sequence (f.i. COBOL, FORTRAN, PL/I,...)

- IDMS provides full recovery and concurrency controls.

- Security constraints are enforced through the subschema mechanism which can be used to hide information and to restrict the range of operations the user is allowed to perform.

- Integrity :
  * Certain field uniqueness constraints.
  * Certain referential integrity constraints.

VII.3.2. DATA DEFINITION

- Location mode : Tell IDMS how to choosea storage location.
  * CALC key : Uses hashing (optionally duplicates not allowed).
  * VIA : Store it near its owner in the specified set.

- Order : Defines the sequence of member record occurrences.
  * NEXT : Specified procedurally at creation time.
  * FIRST : A new member appears in front of all existing members.
  * LAST : A new member appears behind all existing members.
  * PRIOR
  * SORTED

- Connection option :
  * MANUAL (= NULLS ALLOWED)
  * AUTOMATIC (= NULLS NOT ALLOWED)

- Disconnection option :
  * OPTIONAL (= NULLIFIES)
  * MANDATORY (= CASCADES)

- Subschema derivation rules :
  * Any segment type can be omitted.
  * Any record type can be omitted.
  * Any set type can be omitted.
  * If a given record type is omitted, then all set types in which that record types participates must be omitted also.

VII.3.3. DATA MANIPULATION

- A program issuing DML operations must contain a record description for each subschema record type it intends to process.
  → The name of that storage area (and its fields) must be identical to those for the record type.
  → All these record descriptions are called the User Work Area (UWA).

- ICM (= IDMS Communication Block) :
  → The communication area with a return code (= cursor/feedback area in the hierarchic model).

- Run unit : An operating program under the control of IDMS.

- Currency indicator :
  An object whose value at any given time is either null or the address of a record in the database (= a database key).
  → The cursor concept.

- A run unit keeps a table of currency indicators for several of the most recently accessed records :
  * Each type of record : “The current record of type R”.
  * Each type of set : “The current record of set type S” (owner of member record).
  * Any type of record : “The current record of the run unit”
    → The last accessed record no matter what its type is.

- Statements :
  * FIND
  * GET (current of run unit)
  * OBTAIN : FIND + GET
  * MODIFY (current of run unit)
  * CONNECT : Cnnects current run unit into current occurrences of the specified set.
  * DISCONNECT : Disconnects current run unit from the specified set.
  * ERASE (current of run unit)
  * STORE

VII.3.4. STORAGE STRUCTURE

- Every record has a hidden prefix.

- Mode is chain (also specified in the schema) :
  → For each set occurrence the owner is linked with the first member, the first member to the second,... and the last member back to the owner.
  → Optionally :
    * Prior linkage : There are also pointers in reverse order (= double linked)
    * Owner linkage : Each member also includes a pointer to its owner.

- Mode is index :
  → Each occurrence of a set uses an index (the index records are chained by means of next, prior and owner).

- The CALC hashing location mode can access any record type (it is not restricted to root records only).

VII.3.5. THE LOGICAL RECORD FACILITY (LRF)

- Provides a flat record view : the application programmer doesn't need to know the detailed underlying datastructure.
  → User DML (U-DML) : Used by the application programmer to operate on the view.
  → DBA DML (D-DML) : Used by the DBA to define the view in the relevant subschema.

- The U-DML can include a WHERE-clause (of arbitrary complexity).

- For each view the DBA must write a procedure for each U-DML operation that is allowed. Each U-DML statement then issues one of these procedures (expressed in D-DML).

- LRF can provide a relational view to a network database (of course not including all relational operators and still record-at-a-time).

- Each view requires a new subschema (a 'static' operation).

- It is not possible to construct a view on views.

- The DBA must tell the user exactly what operations can be performed on which view and also what they mean.

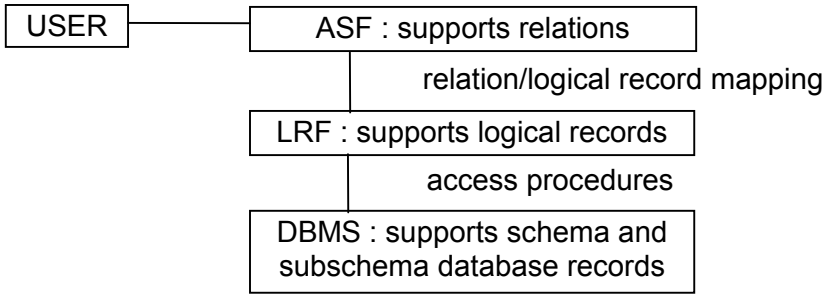
- The DBA procedures are hand-optimized and must be rewritten if the physical structure is changed.

- It's difficult to mix LRF and normal DML statements because the currency indicators (that control the LRF procedures) are affected.

VII.3.6. THE AUTOMATIC SYSTEM FACILITY (ASF)

- Purpose :
  * Support the definition of, and subsequent access to, new relational databases.
  * Provide a relational view of, and relational processing of, existing network databases.

- ASF is implemented on top of LRF (the access procedures are generated automatically by ASF).



- Creating a stored table :
  * The user must specify :
    - Table and field names.
    - Field data types (TEXT, NUMERIC, CURRENCY,...).
    - Integrity constraints (simple restriction conditions).
    - Keys (fields to be indexed, optionally UNIQUE).
  * ASF does :
    - Add the definition to the relational schema.
      → Definitions of records, CODASYL indexed sets,... that appear in the stored database.
      → An indexed set for each key.
    - Create a subschema defining the table as an LRF record (with access procedures that make appropriate use of the keys).
    - Comple the subschema.
    - Generate code to support form-based access.

- Creating a view :
  * The user must specify :
    - The view restriction (basically restrict, project and join up to 5 tables, no two can be the same).
    - Optionally different field names.
    - Updates allowed or not and also their effect if they are allowed (f.i. insertion in both, either or neither of two joined tables).
  * ASF does :
    - Create a subschema defining the view as an LRF record (with corresponding access procedures and update specifications).
    - Compile the subschema.
    - Generate code to support form-based access.

- ASF also allows :
  * Tables can be deleted.
  * Columns can be added and removed.
  * Keys can be added and removed.
  * Field names, lengths and data types can be changed.
  * Integrity constraints can be changed.

- Creating an ASF view of a network database (done by the DBA) :
  * Copy all the network definitions into the relational schema (and keep them synchronized !).
  * Write a subschema for each desired relation.
  * Construct an ASF view definition for each desired relation.

VII.4. CONCLUDING REMARKS

VII.4.1. CRITICAL COMMENTS ON NETWORK SYSTEMS

- They are very complicated.

- They do provide good performance, if the hand-optimization is done well.

VII.4.2. CRITICAL COMMENTS ON CODASYL AND IDMS

- The currency concept is complex (and a source of errors).

- The manual connect process is complex.

- Application programs are not data independent, f.i. :
  * FIND CALC : record must have CALC location mode.
  * FIND LAST : set must have prior linkage.

- ASF is not a report-writing or command-driven query language.

- ASF does not support :
  * duplicate elimination
  * union or difference operators
  * aggregate functions
  * set level update operations

- Referential integrity support is bundled in with the view mechanism :
  * Multiple views must be created to force the same rules (f.i. one for an S-SP join and one for an S-SP-P join).
  * Rules might be defined differently for different views.
  * What if the user updates the stored tables directly, without using the view ?

- The optimizations performed by ASF at generate time is bound by the index choice of LRF (which is done by a simple sequential scan and therefore not intelligent enough to take automatic advantage of the CODASYL links).

VIII. OBJECT-ORIENTED DATABASES

VIII.1. BASIC OO CONCEPTS

VIII.1.1. INTRODUCTION

- There is no 'object-oriented model'. Object-oriented technology is not formally described, it is a kind of a collection of interconnected ideas that are applied to many different areas in computer science (f.i., analysis and design, programming languages, databases,...).

- The fundamental idea : 'raise the level of abstraction'.

VIII.1.2. TERMINOLOGY

Programming term	**OO term**
value	immutable object (instance)
variable	mutable object (instance)
data type	(object) class
function	method
call	message

- Objects are encapsulated : their internal structure is not visible to the user. User's know only the methods an object understands. This guarantees data independence (changes to the internal representation doesn't imply changes to application programs).

- Every object has :
  * private memory : instance variables (members, attributes)
    → In a non-pure OO system, an instance variable can be public, private or protected. Often, instance variables are made public to avoid the need of accessor methods.
  * public interface : the definitions of the methods (also hidden from the user)
    → In fact, these methods are a part of the Class-Defining Object (CDO).
  * a unique identity (Object ID = OID) :
    - immutable objects (f.i. the integer 3) are self-identifying (they are their own ID).
    - mutable objects have (conceptual) addresses, which can be used as (conceptual) pointers.

    → These OID don't avoid the need for user-defined keys (they are needed for the interaction with the outside world), but all internal referencing can now be done through OID.
    → What's the OID of a derived object (f.i. a join of two objects) ?
    → OID's tend to lead to a low-level pointer chase programming style (just like in the prerelational databasesystems !).
    → Some relational systems do have system-defined keys (= surrogate keys).
    - A surrogate key is visible to the user (an OID is not).
    - A surrogate key is a value (an OID is an address).

- Every class understands the NEW message, which creates a new instance of the class (this is called the constructor function).

- Objects are the natural unit of security, recovery and concurrency.

VIII.1.3. HIERARCHIES

- Containment hierarchy :
  An object of a class includes an instance variable whose value is a reference (actually an OID) to an object of another class (or a collection of objects of another class) (see diagrams page 639 and 641).
  → It's difficult to represent a many-to-many relationship in hierarchies, because of the lack of symmetry.
  → Because of the use of OID's, one object can belong to many objects (or even collections) at the same time.

- Class hierarchy :
  * A class Y is a subclass of class X if and only if every object of class Y is necessarily an object of class X (is-a relation). Equivalently, X is called the superclass of Y.
  * Objects of class Y inherit the instance variables (= structural inheritance) and methods (= behavioral inheritance) that apply to class X.
    → Code reusability.
  * Substitutability : The user can always use a Y-object wherever an X-object is permitted.
  * The ability to apply different methods with the same name to different classes is called polymorphism.
  * Some systems also support multiple inheritance : a class can be a subclass of several superclasses simultaneously (the class 'hierarchy' becomes a class 'lattice').

VIII.2. ADDITIONAL OO CONCEPTS

VIII.2.1. DATABASE PROGRAMMING LANGUAGES

- OO systems use an integrated language (not an embedded language) for both database operations and nondatabase operations. The languages are tightly coupled.

- Advantages :
  * This approach allows improved type checking.
  * There's no impedance mismatch. This is the difference in level between record-at-a-time and set-at-a-time languages (f.i. SQL), which rises practical problems.
    → The solution should be to introduce set-at-a-time facilities into programming languages and not to bring the level of the database languages down to record-at-a-time level.

- Most OO systems support Smalltalk (or a dialect) and C++ (which is becoming the standard).

- They don't support the dual-mode principle : the programming language is different from the interactive query language (some form of Object SQL).

VIII.2.2. VERSIONING

- Many applications need the concept of multiple versions of a given object.

- Built-in versioning support includes :
  * The ability to create a new version of an object, by checking out a copy from the database to the user's private workstation.

* The ability to establish a given object version at the current database version, by checking it in from the user's workstation to the database (which may require a merging between two distinct versions).
* The ability to delete (or archive) a version.
* The ability to interrogate the version history of an object.

- Version histories aren't necessarily linear, they can branch into different versions and merged back together.

- Configuration : a collection of mutually consistent versions of interrelated objects.

- Configuration support includes the ability to copy and move an object version from one configuration to another.

VIII.2.3. TRANSACTION MANAGEMENT

- OO application often involve complex transaction that might last for hours or days
  * A rollback might cause the loss of an unacceptably large amount of work.
    → partial rollbacks
  * The use of conventional locking might cause unacceptably long delays.
    → nonlocking concurrency mechanism

- Long transactions :
  * savepoints :
    Established while executing a transaction and used to rollback to a previously established savepoint
  * sagas :
    A sequence of short transactions with the property that all the transactions execute succesfully or certain compensating transactions are executed to cancel the effects of a successfully completed transaction in an incomplete execution of the saga.
  * multiversion concurrency control (especially attractive if the system supports versioning) :
    - Reads are never delayed.
    - Reads never delay updates.
    - It's never necessary to rollback a read-only transaction.
    - Deadlock is only possible between two update transactions.

- Nested transactions :
  Nesting allows a transaction to be organized (recursively) as a hierarchy of actions (a kind of a generalization of savepoints).
  * BEGIN TRANSACTION is extended to support subtransactions (if issued when a transaction is running, it starts a child transaction).
  * COMMIT TRANSACTION commits within the parent scope (if the transaction is a child).
  * ROLLBACK TRANSACTION undoes work to the start of the subtransaction.

VIII.2.4. SCHEMA EVOLUTION

Object-oriented databases often offer more sophisticated schema change support :

- Changes to an object class :
  * instance variable changes (add, delete, rename, change default value, change data type,...)
  * method changes (add, delete, rename, change internal code,...)

- Changes to the class hierarchy (add or delete a class from its superclass)

- Changes to the overall schema (add, delete, rename or coaslesce classes)

VIII.2.5. PERFORMANCE CONSIDERATIONS

- Clustering :
  OO systems use the logical information from the schema as a hint as to how the data should be physically clustered (the DBA should be given explicit and direct control).

- Caching :
  Because OO databases are often used in a client/server environment, caching logically related data at the client side makes sense.

- Swizzling :
  This process replaces OID-style pointers to main memory addresses when the objects are read into memory (and changed back when they are written back to the database).

VIII.2.6. INVERSE VARIABLES

- Instead of a containment hierarchy representation, we can define two separated object classes and include instance variables that reference to each other. These are called inverses of each other.

- These variables can be used to represent one-to-many and many-to-many relationships.

- Example...

```
...DEPT...(...EMPS REF(SET(REF(DEPT))) INVERSE EMP.EDEPT)...
...EMP...(...EDEPT REF(DEPT) INVERSE DEPT.EMPS)...
```

VIII.2.7. REFERENTIAL INTEGRITY SUPPORT

- No system support : it's the responsibility of the user (or user-written methods).

- Reference validation :
  The system checks that all references are to objects of the correct type.
  → DELETE CASCADES for nonshared subobjects within a containment hierarchy.
  → DELETE RESTRICTED for other objects.

- System maintenance :
  The system keeps all references up to date automatically (references to deleted objects are set to nil).
  → DELETE NULLIFIES

- Custom semantics :
  F.i. DELETE CASCADES (outside the containment hierarchy) must be handled with user-written methods.

VIII.2.8. GENERAL INTEGRITY CONSTRAINTS

- They have to be enforced with procedural code in various places (f.i., in the methods that creates an object, changes involved variables)

- Conclusions :
  * The system cannot determine when to do the integrity checking.
  * How do we ensure that all necessary methods include all necessary enforcements ?
  * How can we prevent the user from bypassing the creation method (and thus the integrity check) ?
  * If the constraint changes, how do we find all methods that need to be rewritten ?
  * How do we query the system to find all constraints that apply to a given object ?

- OO systems provide no way of declaring functional dependencies (a special kind of integrity constraints), which are important for database design (the normalization process). Complex data can be represented in an OO system without normalization but they don't solve the problems that unnormalized data causes.

VIII.2.9. METHODS THAT SPAN CLASSES

- The idea of a method being bundled in with an object class works fine so long as the method in question takes just one argument.

- However, a method must be bound with a class because it needs privileged access to the internal representation. Wouldn't it be better to unbundle methods from classes and use the security system to control which methods are allowed to access the internals of which objects ?
  → Methods could be permitted to access the internals of any number of objects.
  → The asymmetry and awkwardness inherent to the bundling scheme would be eliminated.
  → The (ad hoc) distinction between private and protected instance variables would be unnecessary.

VIII.2.10. AD HOC QUERIES

To permit ad hoc queries we must do the following two things :

- Define and maintain a collection that is the set of all currently existing objects of that class.

- Define the public interface (methods) for that class to expose one candidate representation of the objects to the user.

VIII.2.11. MISCELLANEOUS

- Some OO system do support derived instance variables (f.i. AGE might be derived from BIRTHDAY), but it is far from being a full view mechanism.

- A relational DBMS comes ready to use, while an OO DBMS can best be thought of as a construction kit for database professionals to construct an application specific DBMS.These will have to construct an appropriate catalog for their objects.

VIII.3. TOWARD AN OO/R RAPPROCHEMENT

VIII.3.1. GOOD FEATURES OF THE OO MODEL

- Objects : Essential (both mutable and immutable)

- Object classes : Essential, associated with a constructor function to create literals.

- Instance variables :
  Private and protected variables are implementation matters and public instance variables don't exist in a pure OO system. Instance variables, thus, can be ignored.

- Methods :
  Essential, except the bundling method definition (controlled by the security system). The conventional term 'function' is preferred instead of method (also, the essential message concept is replaced by the conventional term 'call').

- Class hierarchy and polymorphism :
  Essential. All inheritance is behavioral (since we have no instance variables). Multiple inheritance is not studied well enough to include it in a formal model.

VIII.3.2. FEATURES NOT SUPPORTED BY THE OO MODEL

- Ad hoc queries
- Closure : Generic operators are not supported.
- Foreign keys
- Declarative integrity support
- Views
- Catalog

VIII.3.3. DOMAINS VERSUS OBJECTS

- An object class and a domain is in fact the same thing : a user-defined, encapsulated data type of arbitrary internal complexity.
- An object instance is an element of domain.
- A collection is the collection of values of any attribute defined on the domain.
- The effect of object sharing can be achieved by means of foreign keys.

VIII.3.4. RELATIONS VERSUS OBJECTS

- Some products state that an object class is equal to a relation, f.i. UniSQL (that supports an extended form of SQL, named SQL/X).

- SQL/X offers some extensions :
  * tables can have composite columns
  * tables can have table-valued columns
  * tables are permitted to have associated methods
  * tables can be defined as subclasses

- Comments :
  * SQL/X objects are rows in tables, the instance variables are columns in tables.
  * An SQL/X class has public instance variables and not necessarily methods.

* SQL/X objects are containment hierarchies (subobjects are shared via OID's).
  → A subobject must exist before we can add an object
  → How about DELETE RESTRICED and DELETE CASCADES ? Procedural code ??
  → The table contains values and pointers (so it isn't relational anymore).
* Operators (f.i. join) on hierarchies need very careful definition and the result is still very complex (also, to what class does the result belong ?, which methods apply ?).
  → The property of closure doesn't apply to object classes.
* A projection over all columns results in the same relation. If we think of it referring it as a base relation, then it is an object class with methodes but referring it as a projection of a base relation over all its columns, then it is not an object class and it has no methods !

VIII.3.5. BENEFITS OF THE RAPPROCHEMENT

- Open architecture (extendability) : users are no longer limited to the builtin datatypes.
- User-defined data types and methods, with inheritance
- Strong typing : proper domain support provides a basis for catching type violations.
- Improved performance :
  Because methods are performed at the server, only the desired information is transmitted to the client (actually stored procedures achieve the same).
- Object level security, recovery and concurrency.
- All of the OO criticisms no longer apply (ad hoc queries, dual mode access, methods that span classes, pointer chasing is totally hidden from the user,...).

IX. EXAM QUESTIONS 1995-1996

- Discuss transaction processing and their use.
- Discuss the representation of missing information and their influence on operations.
- Is the ANSI/SPARC architecture only applicable to the relational database model ? Discuss that architecture.
- Discuss the different tasks of the Data Administrator and the DataBase Administrator.
- What's the relational algebra used for ? Give some examples.
- Discuss two phase commit.
- Describe the network model.
- What's the difference between OID and keys ?
- Give the critics from Date on object-oriented databases.
- What are paired segments ? What are Search Segment Arguments ?
- Explain class hierarchy and containment hierarchy.
- Discuss the impedance mismatch.

X. CONTENTS