An Introduction to Database Systems

Chapter 5. Types

spark@dblab.sogang.ac.kr

5.1 Introduction (1/6)

- Relational model
  - data structure (objects) – Types, Relations
  - data manipulation (operation) – Relational Algebra, Relational Calculus
  - data integrity

spark@dblab.sogang.ac.kr

5.1 Introduction(2/6)

- **Structural Terminology**

Formal relation term	Informal equivalents
relation	table
tuple	row or record
cardinality	number of rows
attribute	column or fields
degree	number of columns
primary key	unique identifier
domain	pool of legal values

spark@dblab.sogang.ac.kr

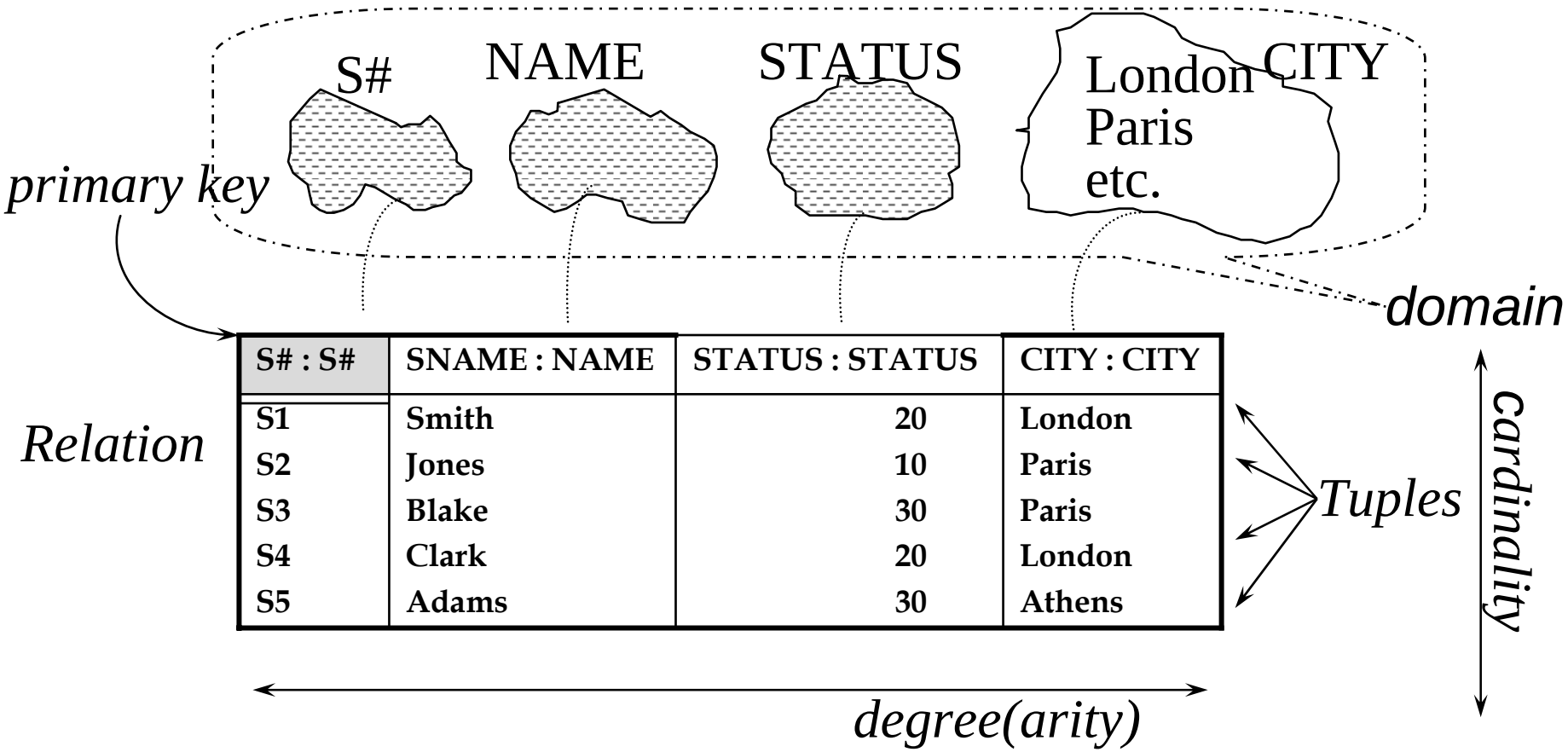
5. Introduction(3/6)

- primary key
  - unique identifiers
  - column or column combinations
  - no two rows of the table contain the same value
- domains
  - not all relational systems support all aspects of the relational model

spark@dblab.sogang.ac.kr

5. Introduction(4/6)

- the suppliers relation S



S# : S#	SNAME : NAME	STATUS : STATUS	CITY : CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

spark@dblab.sogang.ac.kr

5.1 Introduction(5/6)

- Types are called domains
  - System-defined type : INTEGER, CHAR
  - User-defined type : S#, P#, WEIGHT, QTY

- What is a type?
  - A set of values : all possible values of the type

    ex) S# : the set of all possible supplier numbers

spark@dblab.sogang.ac.kr

5.1 Introduction(6/6)

- Any given type has an associated set of *operators* that can validly be applied to values of the type. Values of a given type can be operated upon solely by means of the operators defined for that type

  ex) INTEGER (system-defined)
  - for comparing integers → '=', '<'
  - for performing arithmetic on integers → '+', '*'
  - for performing string operators on integers → '||' (concatenation) (X)

  ex) type S#
  - for comparing supplier number → '=', '<'
  - for performing arithmetic → '+', '*' (X)

spark@dblab.sogang.ac.kr

5.2 Values vs. Variales

- Logical difference between values and variables
  - A value is an individual constant. A value has no location in time and space. A value cannot be updated.
    - EX) complex, geometric point, polygon, X ray, XML document, finger print, …
  - A variable is a holder for an appearance of a value. A variable does have a location in time and space. Variables can be updated.
  - It is important to distinguish between a value per se and an appearance of that value in some particular context

    - There is a logical difference an appearance of a value and the internal encoding or physical representation of that appearance

spark@dblab.sogang.ac.kr

5.2 Values vs. Variables

- **Values and Variables are *typed***
  - **Every value has some type. – Any given value always has exactly one type.**
  - **Every variables is explicitly declared to be of some type.**
  - **Every attribute of every relvar is explicitly declared to be of some type.**
  - **Every operator that returns a result is explicitly declared to be of some type.**
  - **Every parameter of every operator is explicitly declared to be of some type.**
  - **Every expression is at least implicitly declared to be of some type.**

spark@dblab.sogang.ac.kr

5.2 Values vs. Variables

- **Values are typed**
  - **Every value has a type**

ex) If v is a value, then v can be thought of as carrying around with it a kind of flag that announces " I am integer" or "I am a supplier number" or "I am a geometric point"

  - **Any given value will always be of exactly one type and can never change its type (Distinct types are always disjoint)**

spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- Types are a model issue, while physical representations are implementation issue
- Distinguish between a type per se and the physical representation of values of that type inside the system
  - The operators for a given type will depend on the intended meaning or semantics of the type
  - ex) S# - physically represented as character strings
- $\rightarrow$ character string op (X)

  - The operations for a given type will depend on the intended meaning or semantics of the type
  - Physical representation should be hidden from the user

spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- **Scalar vs. Nonscalar Types**
  - **A nonscalar type is a type whose values are explicitly defined to have a set of user-visible, directly accessible components.**
    - **EX) relation types, tuple types**
  - **A scalar type is a type that is not nonscalar (encapsulated/atomic)**

spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- **Strong typing in PL**
  a. **Every value has a type**
  b. **Whenever we try to perform an operation, the system checks that the operands are of the right types fro the operation**

  ex) P.WEIGHT + SP.QTY /* part weight plus shipment quantity */
  P.WEIGHT * SP.QTY / * part weight times shipment quantity */
  → '* ' (O), '+'(X)

- **The nature of values – any kind at all**
  - **simple : numbers, strings, and so on**
  - **complex : domains of audio recordings, maps, video recordings, drawings, blueprints, geometric points, etc**
  - **The only requirement is that the values in the domain must be manipulable solely by means of the operators defined for the domain in question**

spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- **nonscalar type vs. scalar type**
  a. **A nonscalar type is a type that is explicitly defined to have user-visible components ex) relation type**
  b. **Scalar types are types that have no user-visible components**

  Points
  1) **All of the types will be scalar types**
  2) **The physical representation of a given scalar value can be arbitrarily complex**
  3) **Given that they have no user-visible components, scalar types are said to be encapsulated / atomic**
  4) **Scalar types do have possible representations**

spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- **Possible Representations**

```
TYPE    POINT       /* geometric points */
TYPE    POSSREP  CARTESIAN  ( X  RATIONAL,  Y  RATIONAL)
        POSSREP  POLAR  ( R  RATIONAL,  THETA  REATIONAL) ;
```

  - **Every possible representation declaration cause automatic definition of the following operators**
    - **A selector operator which allows the user to specify or select a value of the type by supplying a value for each component of the possible representation**
    - **A set of THE_ operator which allow the user to access the corresponding possible representation components of values of the type**

ex)

```
CARTESIAN  ( 5.0,  2.5 )          /* denotes the point with x=5.0,  y=2.5  */
CARTESIAN  ( XXX,  YYY )          /* denotes the point with x=XXX,  y=YYY --  */
                                  /* XXX and YYY are variables of type RATIONAL  */
POLAR (2.7,  1.0 )                /* denotes the point with r=2.7,  theta=1.0  */
THE_X ( P )          /* returns the x coordinate of point P, P is a variable of type POINT  */
THE_R ( P )          /* returns the r coordinate of point P  */
```


spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- **Suppose the physical representation of points is in fact cartesian coordinates. Then the system will provide certain highly protected operators that effectively expose that physical representation, and the type definer will then use those operators to implement the necessary CARTESIAN and POLAR selectors**

ex) selectors :

```
OPERATOR      CARTESIAN ( X RATIONAL, Y RATIONAL )
                                          RETURNS ( POINT ) ;
    BEGIN ;
            VAR P POINT ;
            X component of physical representation of P := X ;
            Y component of physical representation of P := Y ;
            RETURN ( P ) ;
     END ;
END  OPERATOR;

OPERATOR  POLAR ( R RATIONAL, TEHTA RATIONAL )
                                                RETURNS ( POINT ) ;
            RETURN ( CARTESIAN ( R * COS ( TEHTA ), R * SIN ( THETA ) ) ) ;
END OPERATOR ;
```


spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

```
OPERATOR   POLAR ( R RATIONAL, TEHTA RATIONAL )
                                                    RETURNS ( POINT ) ;
     BEGIN ;
               VAR P POINT ;
               X component of physical representation of P := R * COS ( THETA ) ;
               Y component of physical representation of P := R * SIN ( THETA) ;
               RETURN ( P ) ;
      END ;
END OPERATOR ;
```

Ex) THE_ operators :

```
OPERATOR THE_X ( P POINT )   RETURNS ( RATIONAL ) ;
      RETURN ( X component of physical representation of P ) ;
END OPERATOR ;

OPERATOR THE_Y ( P POINT )   RETURNS ( RATIONAL ) ;
      RETURN ( Y component of physical representation of P ) ;
END OPERATOR ;
```

```
OPERATOR THE_R ( P POINT )   RETURNS ( RATIONAL ) ;
      RETURN ( SORT ( THE_X ( P ) ** 2 + THE_Y ( P ) ** 2 ) ) ;
END OPERATOR ;

OPERATOR THE_THETA ( P POINT )   RETURNS ( RATIONAL ) ;
      RETURN ( ARCTAN ( THE_Y ( P ) / THE_X ( P ) ) ) ;
END OPERATOR ;
```


spark@dblab.sogang.ac.kr

5.3 Types vs. Representations

- All of the concepts discussed apply to simpler types

ex) QTY

selector :
QTY (100)
QTY (N)
QTY ((N1 – N2)

THE_ operator :
THE_QTY (Q)
THE_QTY ((Q1 – Q2) * 2)

spark@dblab.sogang.ac.kr

5.4 Type Definition

- **Type Definition**

TYPE <type name> <possible representation> … ;

TYPE WEIGHT POSSREP (D DECIMAL (5,1)

CONSTRAINT D > 0,0 AND D < 5000.0) ;

ex)

```
TYPE    S#        POSSREP  (CHAR) ;
TPYE    NAME      POSSREP  (CHAR) ;
TYPE    P#        POSSREP  (CHAR) ;
TYPE    COLOR     POSSREP  (CHAR) ;
TYPE    WEIGHT    POSSREP  (RATIONAL) ;
TYPE    QTY       POSSREP  (INTEGER) ;
```

- **inherit / constraints**

DROP TYPE <type name> ;

spark@dblab.sogang.ac.kr

5.5 Operators

- Operator Definition
  - A user-defined operator for the builtin type RATIONAL: ABS, DIST

```
OPERATOR   ABS   ( Z RATIONAL ) RETURNS ( RATIONAL ) ;
    RETURN ( CASE
                        WHEN Z ≥ 0.0  TEHN  +Z
                        WHEN Z < 0.0  THEN   -Z
                END CASE ) ;
END OPERATOR ;

OPERATOR   DIST   ( P1  POINT,  P2  POINT ) RETURNS ( LENGTH ) ;
    RETURN ( WITH    THE_X ( P1 )  AS  X1,
                        THE_X ( P2 )  AS  X2,
                        THE_Y ( P1 )  AS  Y1,
                        THE_Y ( P2 )  AS  Y2,
                LENGTH  ( SQRT ( ) X1 – X2 ) * *  2 + ( Y1 – Y2 ) * *  2 ) ) ) ;
END OPERATOR ;
```


spark@dblab.sogang.ac.kr

5.5 Operators

- Operator Definition (cont.)
  - ‘ = ‘ comparison operator for type POINT
  - ‘ < ‘ operator for type QTY

```
OPERATOR   EQ   ( P1  POINT,  P2  POINT ) RETURNS ( BOOLEAN ) ;
    RETURN ( THE_X ( P1 )  = THE_X ( P2 ) AND
                  THE_Y ( P1 )  = THE_Y ( P2 ) ) ;
END OPERATOR ;

OPERATOR   LT  (Q1  QTY,  Q2  QTY ) RETURNS (BOOLEAN ) ;
    RETURN ( THE_QTY ( Q1 ) < THE_QTY ( Q2 ) ) ;
END OPERATOR ;
```


spark@dblab.sogang.ac.kr

5.5 Operators

- Operator Definition (cont.)
  - Update operator : REFLECT

```
OPERATOR  REFLECT  ( P POINT ) UPDATES ( P ) ;
   BEGIN ;
            THE_X ( P ) := - THE_X ( P ) ;
            THE_Y ( P ) := - THE_Y ( P ) ;
            RETURN ;
   END ;
END OPERATOR ;

 DROP      OPERATOR            REFLECT
```


spark@dblab.sogang.ac.kr

5.5 Operators

- Type Conversion

```
TYPE        S#          POSSREP  ( CHAR ) ;
```

- The possible representation has the inherit name S#, and hence the corresponding selector operator does.

```
S# (' S1 ' )
```

- S# selector might be regarded as a type conversion operator that converts character strings to supplier numbers.

```
... WHERE   P# = 'P2'              ← type error ( compile-time error )

... WHERE   P# = P# ( 'P2' )       ← coercion ( invoking a conversion operator implicitly)
```

- Coercion can lead to program bugs → No coercion are permitted – operands must be of the appropriate types.
  - The comparands for "=", "<", and ">" must be of the same types;
  - The left – and right – hand sides of an assignment (":=") must be of the same type;
- CAST operators to be defined and explicitly invoked for converting between types.

```
CAST_AS_RATIONAL ( 5 )
```


spark@dblab.sogang.ac.kr

5.5 Operators

- **Remarks**
  - **The system will know (a) exactly which expressions are legal, and (b) the type of the result for each such legal expression.**
  - **The total collection of types for a given database will be a closed set – that is, the type of the result of every legal expression will be a type that is known to the system.**
  - **The system knows which assignments are legal, and also which comparison.**
  - **Domains and object classes are the same thing.**

spark@dblab.sogang.ac.kr

5.6 Type Generators

- A type generator is just a special kind of generator because it returns a type instead of a simple scalar value.

spark@dblab.sogang.ac.kr

5.7 SQL Facilities

- **Built-in Types**

◆ BOOLEAN	NUMERIC (p,q)	DATE
◆ BIT [VARYING] (n)	DECIMAL	TIME
◆ BINARY LARGE OJECT (n)	INTEGER	TIMESTAMP
◆ CHARACTER [VARYING] (n)		SMALLINT
INTERVAL		
◆ CHARACTER LARGE OBJECT (n)		FLOAT (p)

- **(SQL : 8 basic types)**
  - ◆ boolean/bit string/binary/character string/numeric/date/time
  - ◆ timestamp
  - ◆ year/ month intervals
  - ◆ day/time intervals

spark@dblab.sogang.ac.kr

5.7 SQL Facilities

- SQL supports two kinds of user-defined types, DISTINCT types, and *structured* types.
  - a syntactic shorthand, not data types at all,
  - not full user-defined types
- DISTINCT Types
  - not of arbitrary internal complexity,
  - Limited to the complexity of the builtin types
  - CREATE TYPE <type name> AS <representation> FINAL;

    EX) CREATE TYPE WEIGHT AS DECIMAL (5,1) FINAL;
- Structured types
  - EX) CREATE TYPE POINT AS (X FLOAT, Y FLOAT) NOT FINAL;

spark@dblab.sogang.ac.kr

5.7 SQL Facilities

- **Type Generators**
  - **SQL supports three type generators (the SQL term is type constructors) : REF, ROW, and ARRAY.**
  - **EX) the use of ROW**
    - **CREATE TABLE CUST**
      - **(CUST# CHAR(3),**
      - **ADDR ROW (STREET CHAR(50),**

```
                    CITY      CHAR(25),,
                    STATE     CHAR(2),,
                    ZIP       CHAR(5),   )
          PRIMARY KEY ( CUST# ) ) ;
```


spark@dblab.sogang.ac.kr