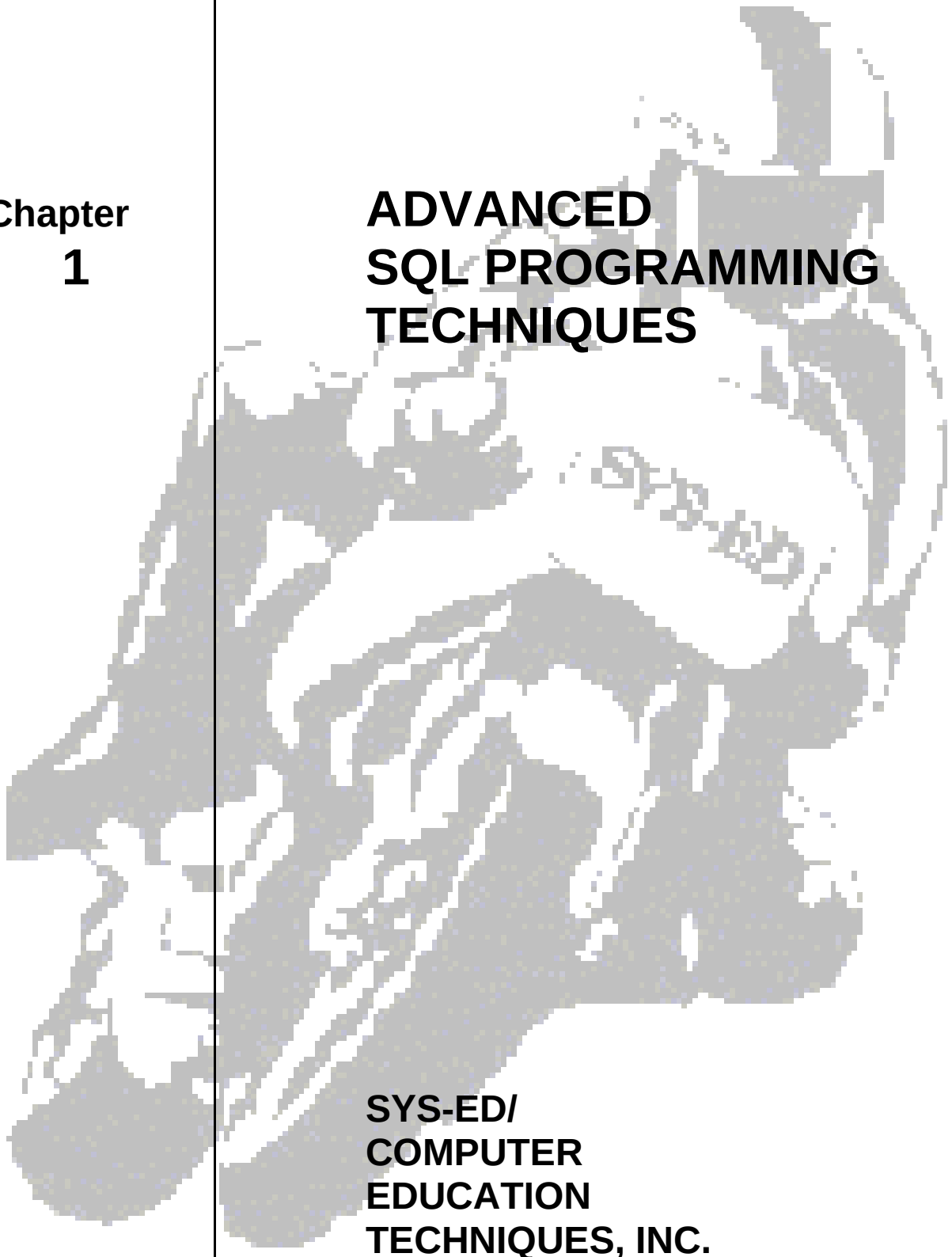
Chapter
1

ADVANCED SQL PROGRAMMING TECHNIQUES

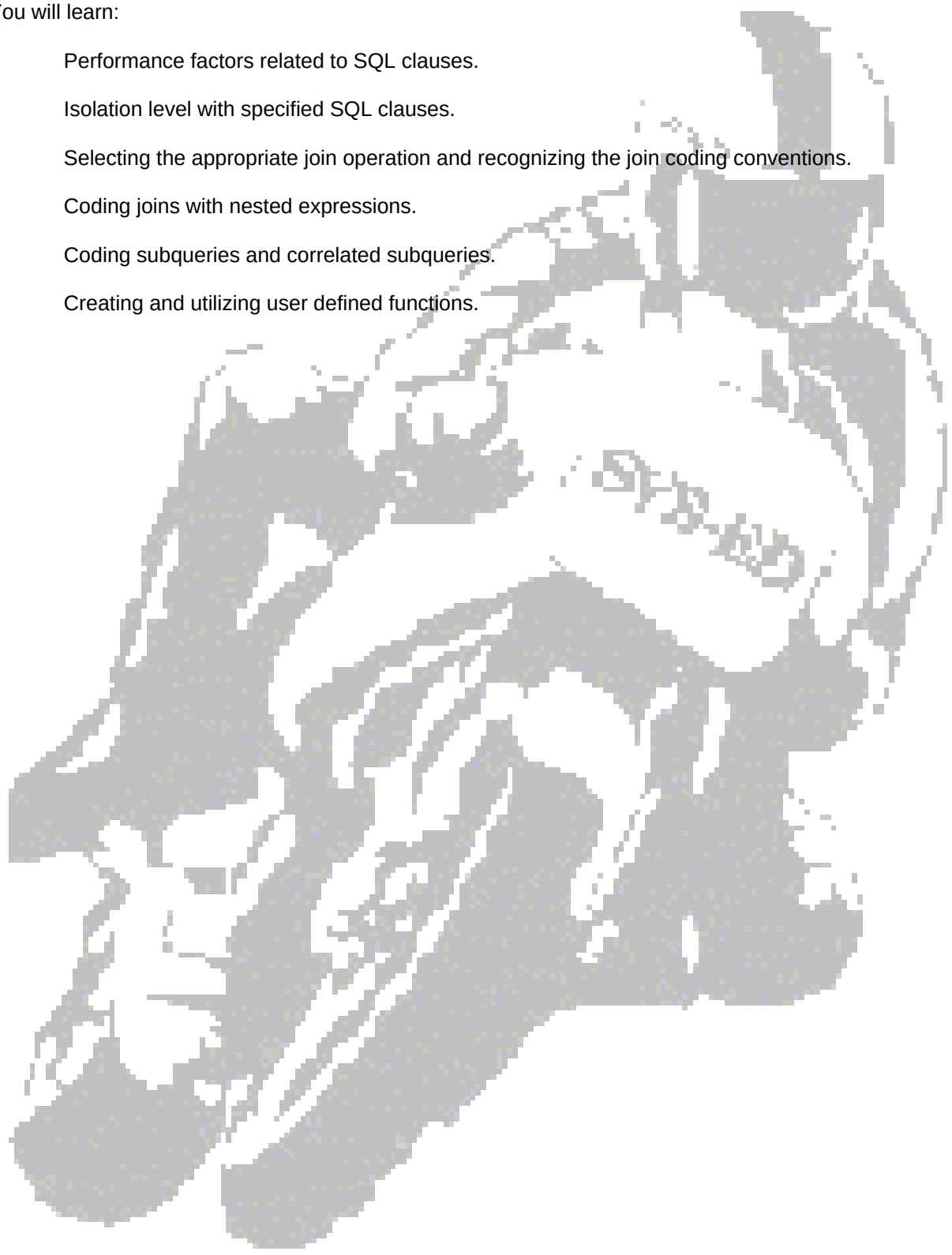


SYS-ED/
COMPUTER
EDUCATION
TECHNIQUES, INC.

Objectives

You will learn:

- Performance factors related to SQL clauses.
- Isolation level with specified SQL clauses.
- Selecting the appropriate join operation and recognizing the join coding conventions.
- Coding joins with nested expressions.
- Coding subqueries and correlated subqueries.
- Creating and utilizing user defined functions.



1 FOR FETCH ONLY

The FOR FETCH ONLY can be specified on the SELECT in a cursor.

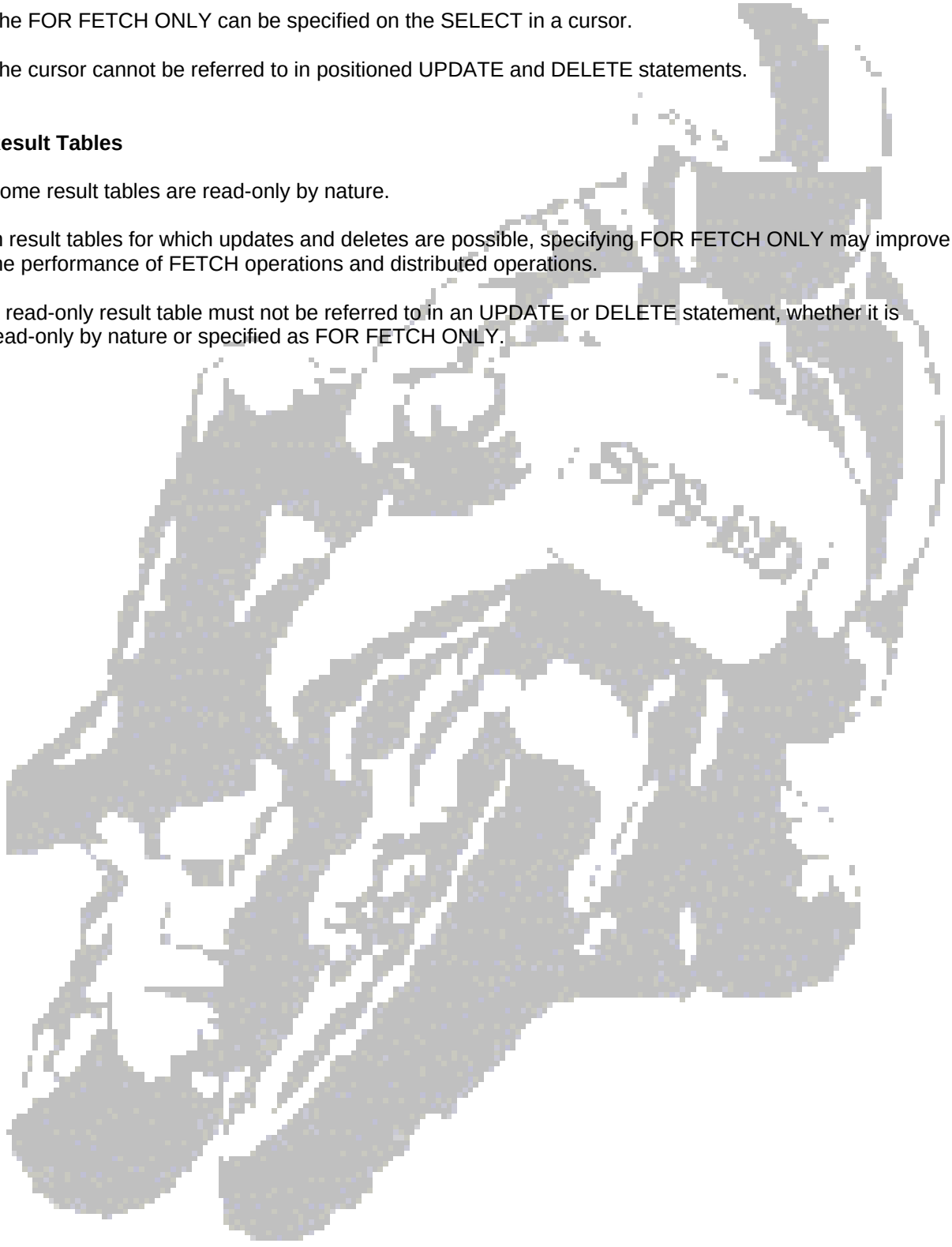
The cursor cannot be referred to in positioned UPDATE and DELETE statements.

Result Tables

Some result tables are read-only by nature.

In result tables for which updates and deletes are possible, specifying FOR FETCH ONLY may improve the performance of FETCH operations and distributed operations.

A read-only result table must not be referred to in an UPDATE or DELETE statement, whether it is read-only by nature or specified as FOR FETCH ONLY.



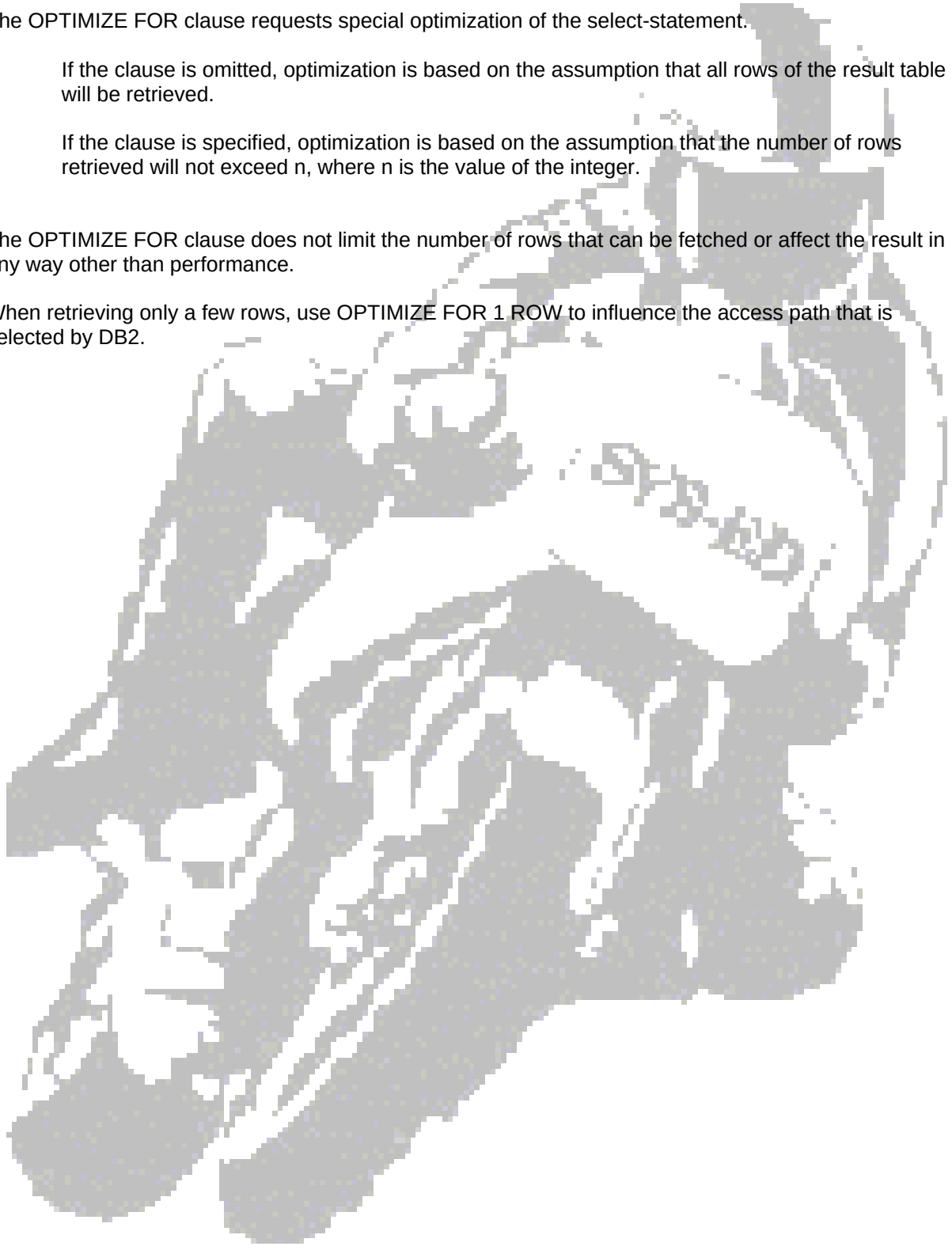
2 OPTIMIZE-FOR-CLAUSE

The OPTIMIZE FOR clause requests special optimization of the select-statement.

- If the clause is omitted, optimization is based on the assumption that all rows of the result table will be retrieved.
- If the clause is specified, optimization is based on the assumption that the number of rows retrieved will not exceed n, where n is the value of the integer.

The OPTIMIZE FOR clause does not limit the number of rows that can be fetched or affect the result in any way other than performance.

When retrieving only a few rows, use OPTIMIZE FOR 1 ROW to influence the access path that is selected by DB2.



3 WITH-CLAUSE

```
ª >>--WITH----CS-------------------------------------------------------><  ª
             +-UR-------------------------ª
             +-RR-------------------------ª
             ª      +-KEEP UPDATE LOCKS-+  ª
             +-RS-------------------------+
                    +-KEEP UPDATE LOCKS-+
```

The WITH clause specifies the isolation level at which the statement is executed.

CS	Cursor stability.
UR	Uncommitted read.
RR	Repeatable read.
RR KEEP UPDATE LOCKS	Repeatable read keep update locks.
RS	Read stability.
RS KEEP UPDATE LOCKS	Read stability keep update locks.

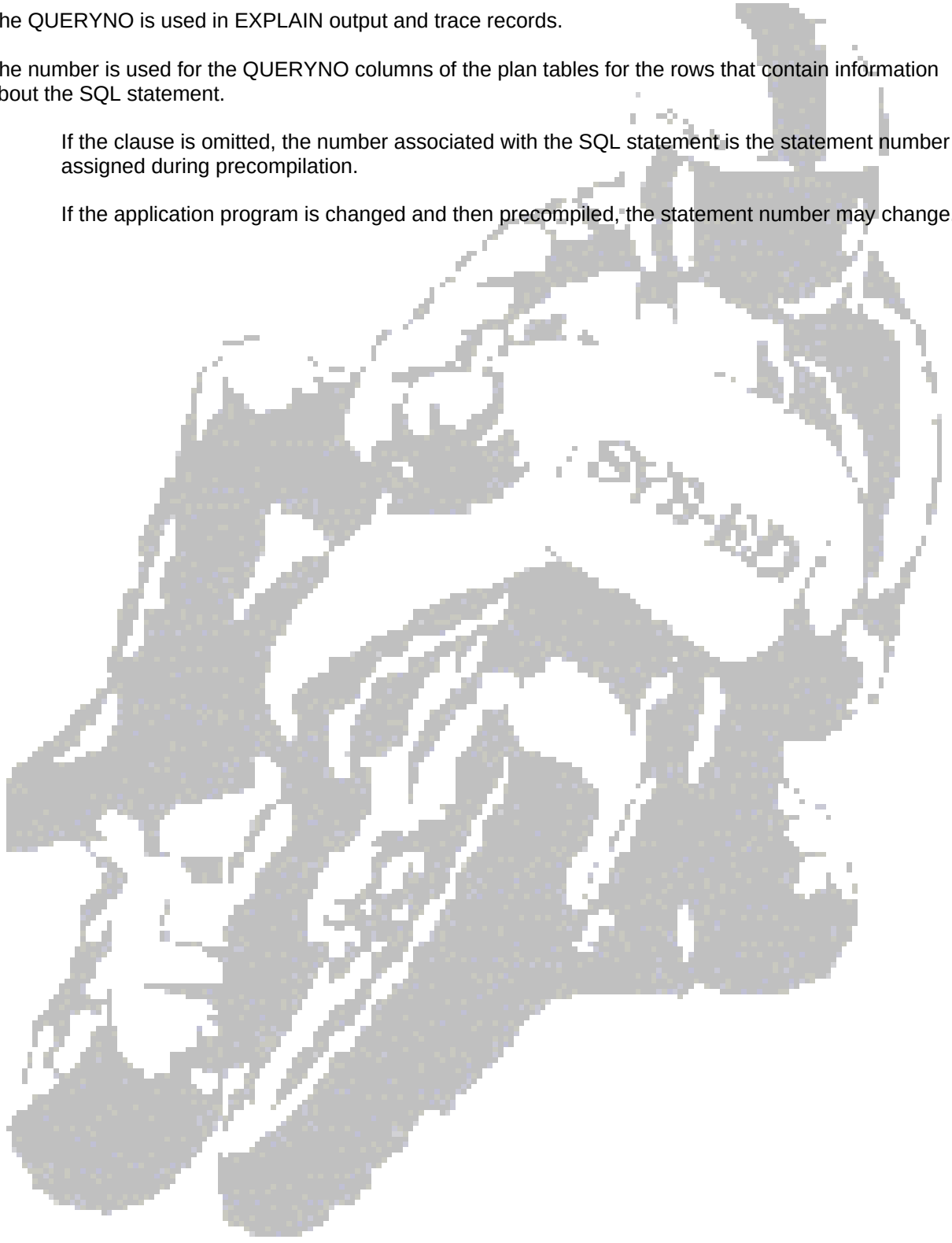
- WITH UR can be specified only if the result table is read-only.
- WITH RR KEEP UPDATE LOCKS or WITH RS KEEP UPDATE LOCKS can be specified only if the FOR UPDATE OF clause is also specified.
- KEEP UPDATE LOCKS tells DB2 to acquire and hold an X lock instead of an U or S lock on all qualified pages and rows. Although this option can reduce concurrency, it can prevent some types of deadlocks.

4 QUERYNO-CLAUSE

The QUERYNO is used in EXPLAIN output and trace records.

The number is used for the QUERYNO columns of the plan tables for the rows that contain information about the SQL statement.

- If the clause is omitted, the number associated with the SQL statement is the statement number assigned during precompilation.
- If the application program is changed and then precompiled, the statement number may change.



5 Joining Data from More than One Table

Sometimes the information may not be in a single table. To form a row of the result table, it may be necessary to retrieve some column values from one table and other column values from another table.

A SELECT statement can be used for retrieving and joining column values from two or more tables into a single row.

DB2 supports the following types of joins:

C inner join	C left outer join	C right outer join	C full outer join.

Joins can be specified in the FROM clause of a query.

5.1 Inner Join

When requesting an inner join, execute a SELECT statement and use the FROM clause to specify the tables to be joined. A WHERE clause or ON clause can then be used to indicate the join condition.

The join condition can be any simple or compound search condition that does not contain a subquery reference.

In the simplest type of inner join, the join condition is column1=column2.

Example 1:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
    FROM PARTS, PRODUCTS
    WHERE PARTS.PROD# = PRODUCTS.PROD#;
```

Example 2:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
    FROM PARTS INNER JOIN PRODUCTS
    ON PARTS.PROD# = PRODUCTS.PROD#;
```

Both examples provide the same result.

If a WHERE clause is not specified in the first form of the query, the result table contains all possible combinations of rows for the tables identified in the FROM clause.

5.2 Full Outer Join

The clause FULL OUTER JOIN includes unmatched rows from both tables. Missing values in a row of the result table contain nulls.

The join condition for a full outer join must be a simple search condition that compares two columns or cast functions that contain columns.

Example:

This code performs a full outer join of the PARTS and PRODUCTS tables:

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
        FROM PARTS FULL OUTER JOIN PRODUCTS
        ON PARTS.PROD# = PRODUCTS.PROD#;
```

5.3 Left Outer Join

The clause LEFT OUTER JOIN includes rows from the table that has been specified before LEFT OUTER JOIN and that have no matching values in the table that is specified after LEFT OUTER JOIN.

As with an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

Example:

Executing this query will include rows from the PARTS table that have no matching values in the PRODUCTS table and include only prices of greater than 10.00.

```
SELECT PART, SUPPLIER, PARTS.PROD#, PRODUCT
        FROM PARTS LEFT OUTER JOIN PRODUCTS
                ON PARTS.PROD#=PRODUCTS.PROD#
                AND PRODUCTS.PRICE>10.00;
```

Since the PARTS table can have nonmatching rows and the PRICE column is not in the PARTS table, rows in which PRICE is less than 10.00 are included in the result of the join, but PRICE will be to null.

5.4 Right Outer Join

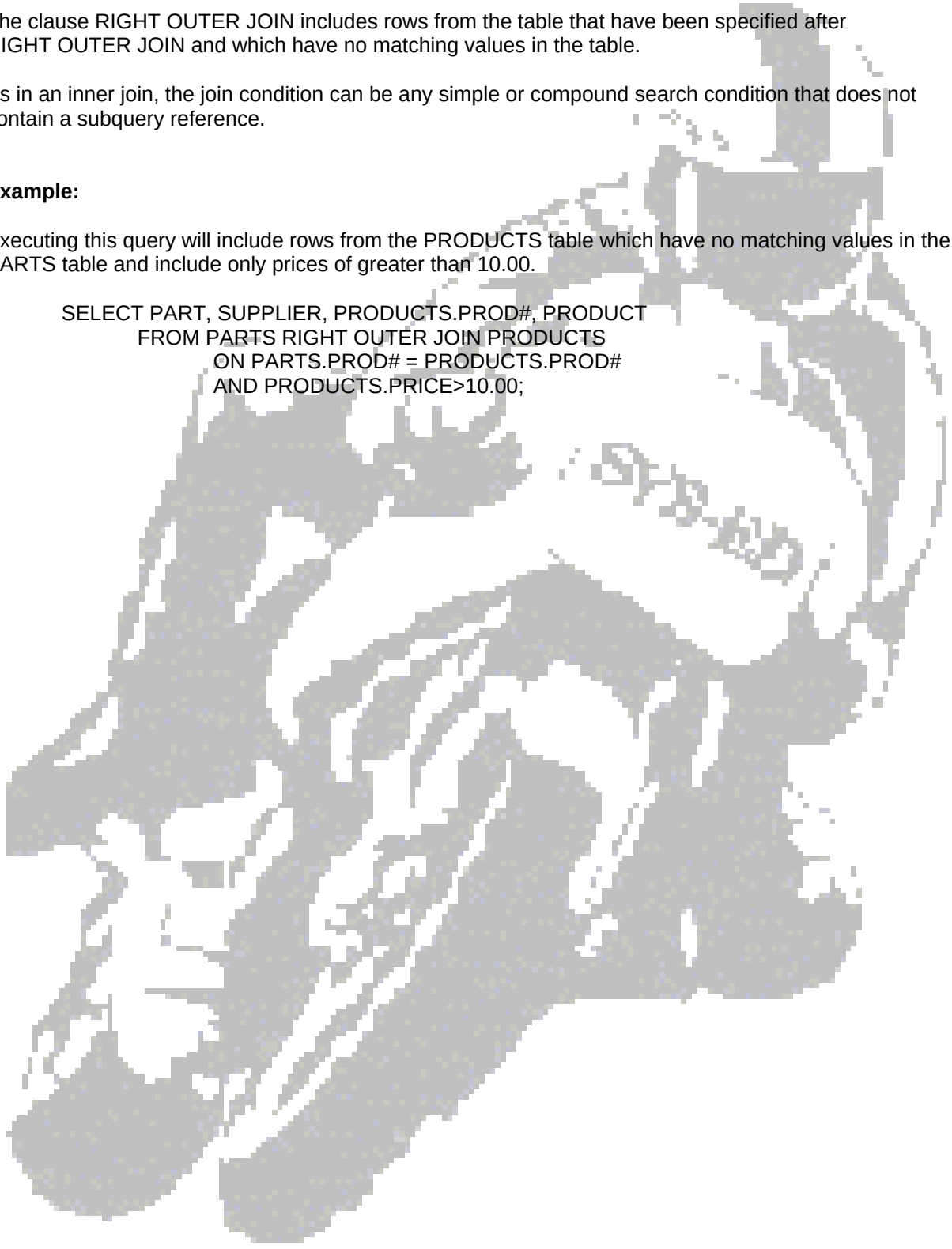
The clause RIGHT OUTER JOIN includes rows from the table that have been specified after RIGHT OUTER JOIN and which have no matching values in the table.

As in an inner join, the join condition can be any simple or compound search condition that does not contain a subquery reference.

Example:

Executing this query will include rows from the PRODUCTS table which have no matching values in the PARTS table and include only prices of greater than 10.00.

```
SELECT PART, SUPPLIER, PRODUCTS.PROD#, PRODUCT
        FROM PARTS RIGHT OUTER JOIN PRODUCTS
                ON PARTS.PROD# = PRODUCTS.PROD#
                AND PRODUCTS.PRICE>10.00;
```



6 SQL Rules for Statements Containing Join Operations

SQL rules dictate that the clauses in a SELECT statement be evaluated in the following order:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT

A join operation is part of a FROM clause. Therefore, when predicting which rows will be returned from a SELECT statement containing a join operation, assume that the join operation will be performed first.

When more than two tables need to be joined, more than one join type can be used in the FROM clause.

Example:

In order to generate a result table showing all the employees, their department names, and the projects they are responsible it will be necessary to join three tables..

```
SELECT EMPNO, LASTNAME, DEPTNAME, PROJNO
    FROM DSN8610.EMP INNER JOIN DSN8610.DEPT
        ON WORKDEPT = DSN8610.DEPT.DEPTNO
        LEFT OUTER JOIN DSN8610.PROJ
            ON EMPNO = RESPEMP
WHERE LASTNAME > 'S';
```

7 Nested Table Expressions and User-defined Table Functions in Joins

An operand of a join can be more complex than the name of a single table.

Nested table expression	Is a subselect enclosed in parentheses, followed by a correlation name.
User-defined table function	Is a user-defined function that returns a table.

Example:

This query contains a nested table expression:

```
SELECT PROJECT, COALESCE(PROJECTS.PROD#, PRODNUM) AS PRODNUM,
        PRODUCT, PART, UNITS
    FROM PROJECTS LEFT JOIN
        (SELECT PART,
            COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
            PRODUCTS.PRODUCT
            FROM PARTS FULL OUTER JOIN PRODUCTS
                ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
    ON PROJECTS.PROD# = PRODNUM;
```

Example:

Within the nested table expression, the correlation name is TEMP.

```
(SELECT PART,
    COALESCE(PARTS.PROD#, PRODUCTS.PROD#) AS PRODNUM,
    PRODUCTS.PRODUCT
    FROM PARTS FULL OUTER JOIN PRODUCTS
        ON PARTS.PROD# = PRODUCTS.PROD#) AS TEMP
```

Example:

This code demonstrates a simple nested table expression.

```
SELECT CHEAP_PARTS.PROD#, CHEAP_PARTS.PRODUCT
    FROM (SELECT PROD#, PRODUCT
        FROM PRODUCTS
            WHERE PRICE < 10) AS CHEAP_PARTS;
```

8 Subqueries

A subquery is used for to narrowing a search condition based on information in an interim table.

Example:

In order to list the employee numbers, names, and commissions, of all employees working on a particular project, the first part of the SELECT statement would be written as follows:

```
SELECT EMPNO, LASTNAME, COMM
        FROM DSN8610.EMP
        WHERE EMPNO
```

It will not be possible to further qualify the scope because the DSN8610.EMP table does not include project number data.

To ascertain which employees are working on project MA2111, another SELECT statement will need to be issued against the DSN8610.EMPPROJACT table.

A subselect can be used for solving this problem. A subselect in a WHERE clause is called a subquery. The SELECT statement surrounding the subquery is called the outer SELECT.

Example:

```
SELECT EMPNO, LASTNAME, COMM
        FROM DSN8610.EMP
                WHERE EMPNO IN
                        (SELECT EMPNO
                                FROM DSN8610.EMPPROJACT
                                WHERE PROJNO = 'MA2111');
```

The interim result table then serves as a list in the search condition of the outer SELECT.

8.1 Correlated and Uncorrelated Subqueries

Subqueries supply information needed to qualify a row in a WHERE clause or a group of rows in a HAVING clause.

- The subquery produces a result table used to qualify the row or group of rows selected.
- The subquery executes only once, if the subquery is the same for every row or group. This is a uncorrelated subquery.
- Subqueries that vary in content from row to row or group to group are correlated subqueries.

8.2 Subqueries and Predicates

A subquery is always part of a predicate.

The format of a predicate is:

operand operator (subquery)

The predicate can be part of a WHERE or HAVING clause. A WHERE or HAVING clause can include predicates that contain subqueries.

A predicate containing a subquery, as with any other search predicate, can:

- be enclosed in parentheses.
- be preceded by the keyword NOT.
- be linked to other predicates through the keywords AND and OR.

Example:

```
WHERE X IN (subquery1) AND (Y > SOME (subquery2) OR Z IS NULL)
```

Subqueries can also appear in the predicates of other subqueries. These types of subqueries will have with some level of nesting.

When a subselect is used in an UPDATE, DELETE, or INSERT statement, the subselect cannot use the same table as the UPDATE, DELETE, or INSERT statement.

9 Coding a Subquery

There are a 4 ways to specify a subquery in either a WHERE or HAVING clause.

C	Basic predicate	C	Quantified Predicates: ALL, ANY, and SOME.
C	IN keyword	C	EXISTS keyword

9.1 Basic Predicate

A subquery can be used immediately after any of the comparison operators. The subquery will then return a maximum of one value. DB2 compares that value with the value to the left of the comparison operator.

Example:

This SQL statement returns the employee numbers, names, and salaries for employees whose education level is higher than the average company-wide education level.

```
SELECT EMPNO, LASTNAME, SALARY
        FROM DSN8610.EMP
        WHERE EDLEVEL >
                (SELECT AVG(EDLEVEL)
                        FROM DSN8610.EMP);
```

9.2 Quantified Predicates: ALL, ANY, and SOME

A subquery can be used after a comparison operator followed by the keyword ALL, ANY, or SOME.

When used in this way, the subquery can return zero, one, or many values, including null values.

The ALL keyword should be used for indicating that the first operand of the comparison must compare in the same way with all the values the subquery returns.

Example:

Consider the greater-than comparison operator with ALL:

```
WHERE expression > ALL (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than all the values that the subquery returns. A subquery that returns an empty result table satisfies the predicate.

Use ANY or SOME to indicate that the value that has supplied must compare in the indicated way to at least one of the values the subquery returns.

Example:

Consider an expression where the greater-than comparison operator is used with ANY:

```
WHERE expression > ANY (subquery)
```

To satisfy this WHERE clause, the value in the expression must be greater than at least one of the values (ie. greater than the lowest value) that the subquery returns. A subquery that returns an empty result table will not satisfy the predicate.

9.3 EXISTS Keyword

In the subqueries presented thus far, DB2 evaluates the subquery and uses the result as part of the WHERE clause of the outer SELECT.

In contrast, when the keyword EXISTS is used, DB2 checks whether the subquery returns one or more rows. Returning one or more rows satisfies the condition; returning no rows does not satisfy the condition.

Example:

```
SELECT EMPNO,LASTNAME
    FROM DSN8610.EMP
        WHERE EXISTS
            (SELECT *
                FROM DSN8610.PROJ
                WHERE PRSTDATE > '1999-01-01');
```

In this code, the search condition is true if any project represented in the DSN8610.PROJ table has an estimated start date which is later than 1 January 1999.

It does not show the full power of EXISTS, because the result is always the same for every row examined for the outer SELECT. As a consequence, either every row appears in the results, or none appear.

A correlated subquery is more powerful, because the subquery would change from row to row.

10 Correlated Subqueries

In the previous subqueries, DB2 executed the subquery once, substituted the result of the subquery in the right side of the search condition, and evaluated the outer-level SELECT based on the value of the search condition.

A subquery can be written to force DB2 to re-evaluate a new row in a WHERE clause or group of rows in a HAVING clause as it executes the outer SELECT. This is known as a correlated subquery.

Correlated versus Uncorrelated Subqueries

Consider a situation where it will be necessary to list all the employees whose education levels are higher than the average education levels in their respective departments.

To get this information, DB2 must search the DSN8610.EMP table. For each employee in the table, DB2 will need to compare the employee's education level to the average education level for the employee's department.

This is the point at which a correlated subquery differs from an uncorrelated subquery. The uncorrelated subqueries compares the education level to the average of the entire company, which requires looking at the entire table.

A correlated subquery evaluates only the department which corresponds to the particular employee.

Example:

In this subquery, DB2 computes the average education level for the department number in the current row.

```
SELECT EMPNO, LASTNAME, WORKDEPT, EDLEVEL
    FROM DSN8610.EMP X
    WHERE EDLEVEL >
        (SELECT AVG(EDLEVEL)
            FROM DSN8610.EMP
            WHERE WORKDEPT = X.WORKDEPT);
```

A correlated subquery looks like an uncorrelated one, except for the presence of one or more correlated references.

In the example code, the single correlated reference is the occurrence of X.WORKDEPT in the WHERE clause of the subselect. The qualifier X is the correlation name defined in the FROM clause of the outer SELECT statement.

10.1 Correlated Subqueries in an UPDATE Statement

When using a correlated subquery in an UPDATE statement, the correlation name refers to the rows that are being updated.

Example:

Consider a scenario where it is required that all activities of a project must be completed before September 1999.

The following SQL statement can be used for evaluating the projects in the DSN8610.PROJ table.

The code will write a 1 (a flag to indicate PRIORITY) in the PRIORITY column for each priority project:

```
UPDATE DSN8610.PROJ X
SET PRIORITY = 1
WHERE DATE('1997-09-01') >
        (SELECT MAX(ACENDATE)
        FROM DSN8610.PROJACT
        WHERE PROJNO = X.PROJNO);
```

10.2 Correlated Subqueries in a DELETE Statement

When a correlated subquery is used in a DELETE statement, the correlation name represents the row that has been deleted. DB2 evaluates the correlated subquery once for each row in the table named in the DELETE statement in order to decide whether or not to delete the row.

Example:

Consider a scenario in which a department considers a project to be complete when the combined amount of time currently spent on it is half a person's time or less. The department then deletes the rows for that project from the DSN8610.PROJ table.

In the following statements, PROJ and PROJACT are independent tables; they are separate tables with no referential constraints defined on them.

```
DELETE FROM DSN8610.PROJ X
        WHERE .5 >
                (SELECT SUM(ACSTAFF)
                        FROM DSN8610.PROJACT
                        WHERE PROJNO = X.PROJNO);
```

11 User Defined Functions

There are two types of user-defined functions:

Sourced user-defined	Based on existing built-in functions or user-defined functions.
External user-defined	Written by a programmer in a host language.

User-defined functions can be categorized as a user-defined scalar or user-defined table function:

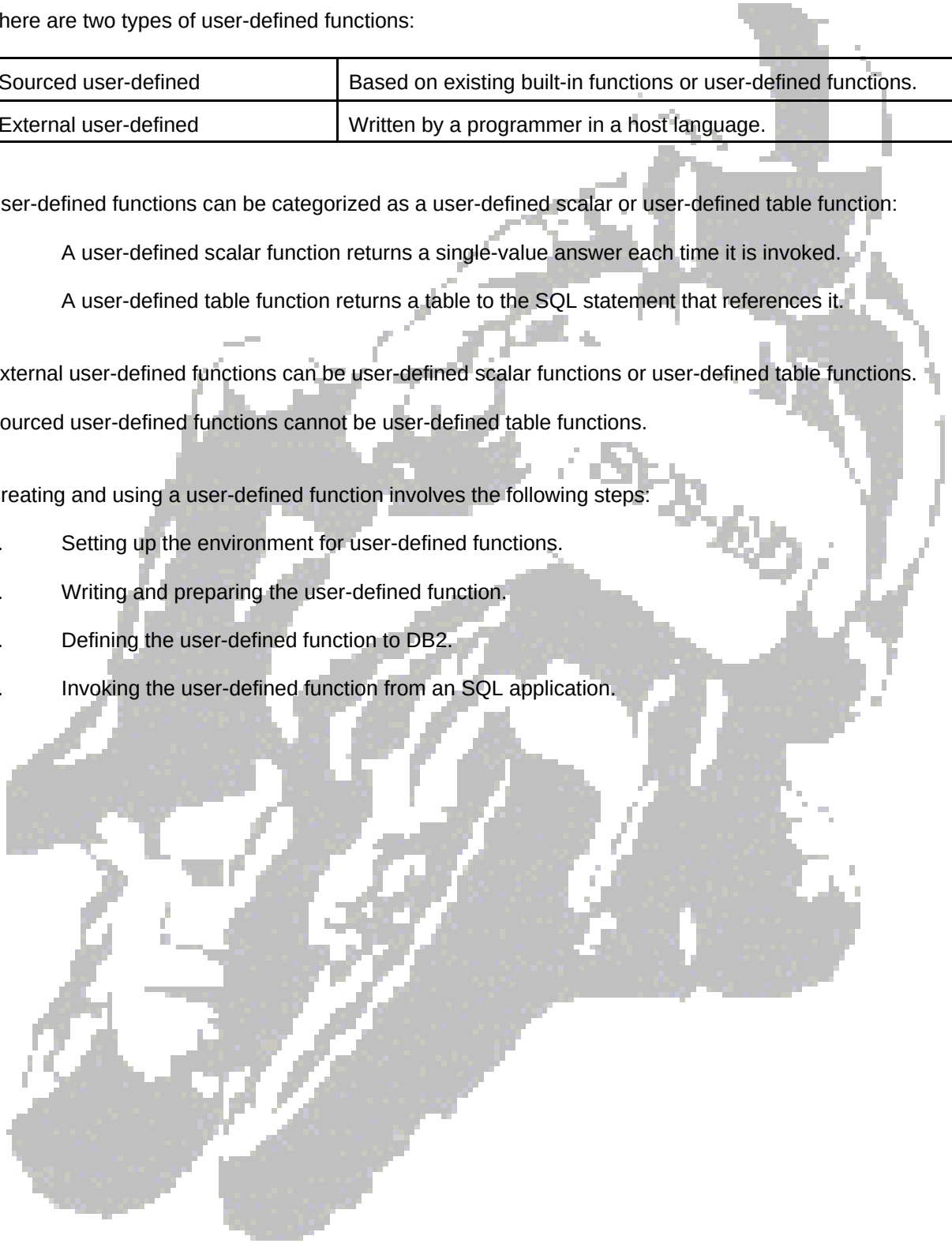
- A user-defined scalar function returns a single-value answer each time it is invoked.
- A user-defined table function returns a table to the SQL statement that references it.

External user-defined functions can be user-defined scalar functions or user-defined table functions.

Sourced user-defined functions cannot be user-defined table functions.

Creating and using a user-defined function involves the following steps:

1. Setting up the environment for user-defined functions.
2. Writing and preparing the user-defined function.
3. Defining the user-defined function to DB2.
4. Invoking the user-defined function from an SQL application.



11.1 Example: User-defined Scalar Function

Consider a scenario where an organization requires a user-defined scalar function that calculates the bonus that each employee receives.

- All employee data, including salaries, commissions, and bonuses, is kept in the employee table, EMP.
- The input fields for the bonus calculation function are the values of the SALARY and COMM columns. The output from the function goes into the BONUS column.

Since this function gets its input from a DB2 table and puts the output in a DB2 table, a user-defined function would be a convenient way for manipulating the data.

The user-defined function's definer and invoker determine that this new user-defined function should have the following characteristics:

- The user-defined function name is CALC_BONUS.
- The two input fields are of type DECIMAL(9,2).
- The output field is of type DECIMAL(9,2).
- The program for the user-defined function is written in COBOL and has a load module name of CBONUS.

Since there is no existing built-in function or user-defined function on which to build a sourced user-defined function, the function implementer must code an external user-defined function.

The implementer performs the following steps:

1. Writes the user-defined function, which is a COBOL program.
2. Precompiles, compiles, and links the program.
3. Binds a package when the user-defined function contains SQL statements.
4. Tests the program.
5. Grants execute authority on the user-defined function package to the definer.

The user-defined function definer executes this CREATE FUNCTION statement to register CALC_BONUS to DB2:

```
CREATE FUNCTION CALC_BONUS(DECIMAL(9,2),DECIMAL(9,2))
        RETURNS DECIMAL(9,2)
        EXTERNAL NAME 'CBONUS'
        PARAMETER STYLE DB2SQL
LANGUAGE COBOL;
```

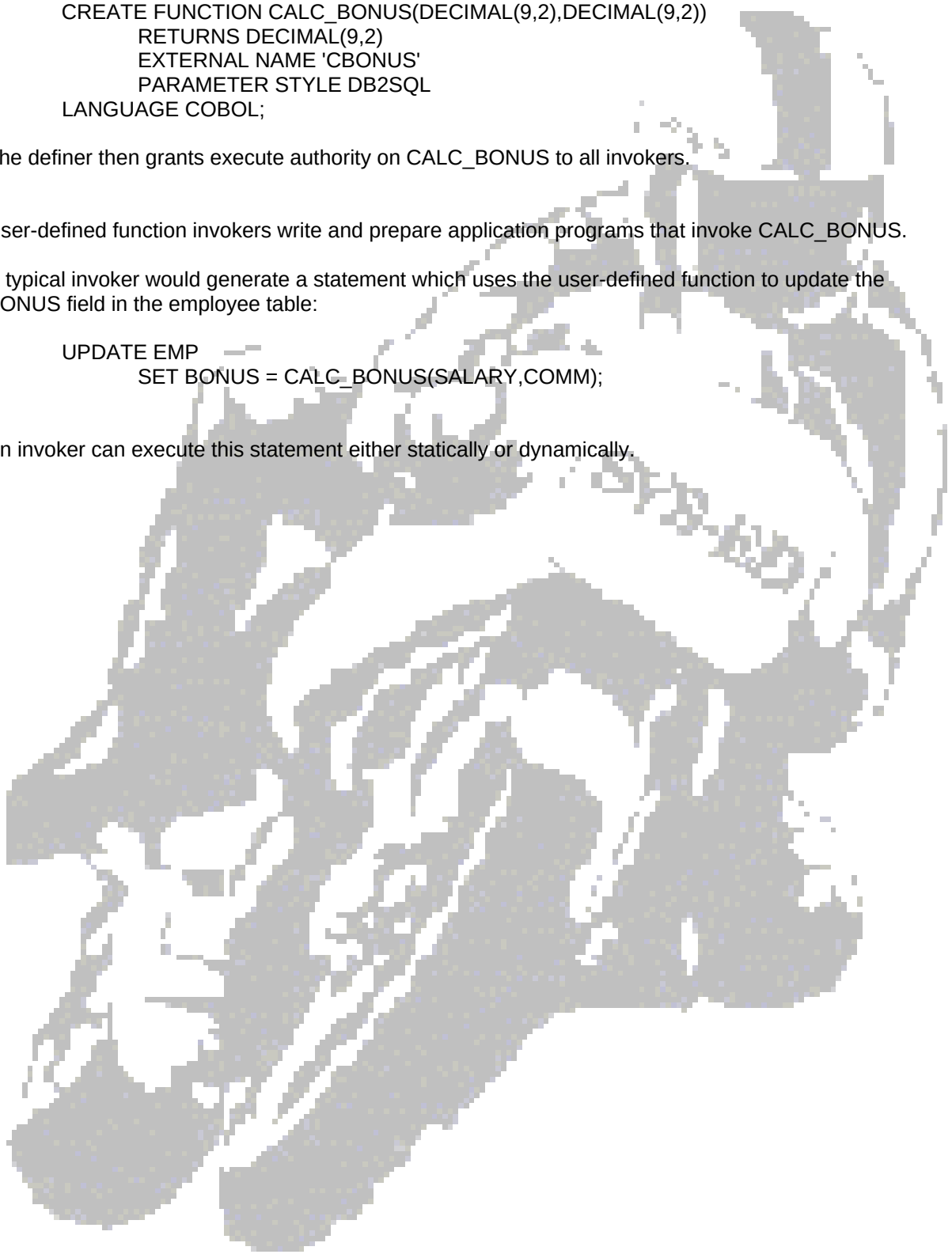
The definer then grants execute authority on CALC_BONUS to all invokers.

User-defined function invokers write and prepare application programs that invoke CALC_BONUS.

A typical invoker would generate a statement which uses the user-defined function to update the BONUS field in the employee table:

```
UPDATE EMP
        SET BONUS = CALC_BONUS(SALARY,COMM);
```

An invoker can execute this statement either statically or dynamically.



11.2 User-defined Function Definition: Components

The characteristics included in a CREATE FUNCTION or ALTER FUNCTION statement depend on whether the user-defined function is external or sourced.

Characteristic	CREATE FUNCTION or ALTER FUNCTION Parameter	Valid in Sourced Function?	Valid in External Function?
User-defined function name	FUNCTION	Yes	Yes
Input parameter types	FUNCTION	Yes	Yes
Output parameter types	RETURNS RETURNS TABLE(1)	Yes	Yes
Specific name	SPECIFIC	Yes	Yes
External name	EXTERNAL NAME	No	Yes
Language	LANGUAGE ASSEMBLE LANGUAGE C LANGUAGE COBOL LANGUAGE PLI	No	Yes
Deterministic or not deterministic	NOT DETERMINISTIC	No	Yes
Types of SQL statements in the function	NO SQL CONTAINS SQL READS SQL DATA MODIFIES SQL DATA	No	Yes(2)
Name of source function	SOURCE	Yes	No
Parameter style	PARAMETER STYLE DB2SQL	No	Yes
Address space for user-defined functions	FENCED	No	Yes
Call with null input	RETURNS NULL ON NULL INPUT CALLED ON NULL INPUT	No	Yes
External actions	EXTERNAL ACTION NO EXTERNAL ACTION	No	Yes
Scratchpad specification	NO SCRATCHPAD SCRATCHPAD length	No	Yes
Call function after SQL processing	NO FINAL CALL FINAL CALL	No	Yes

Characteristic	CREATE FUNCTION or ALTER FUNCTION Parameter	Valid in Sourced Function?	Valid in External Function?
Consider function for parallel processing	ALLOW PARALLEL DISALLOW PARALLEL	No	Yes(2)
Package collection	NO COLLID COLLID collection-id	No	Yes
WLM environment	WLM ENVIRONMENT name WLM ENVIRONMENT name,*	No	Yes
CPU time for a function invocation	ASUTIME NO LIMIT ASUTIME LIMIT integer	No	Yes
Load module stays in memory	STAY RESIDENT NO STAY RESIDENT YES	No	Yes
Program type	PROGRAM TYPE MAIN PROGRAM TYPE SUB	No	Yes
Security	SECURITY DB2 SECURITY USER SECURITY DEFINER	No	Yes
Run-time options	RUN OPTIONS options	No	Yes
Pass DB2 environment information	NO DBINFO DBINFO	No	Yes
Expected number of rows returned	CARDINALITY integer	No	Yes(1)

11.3 User-defined Function Definitions: Examples

```
CREATE FUNCTION FINDSTRING (CLOB(500K), VARCHAR(200))
        RETURNS INTEGER
        CAST FROM FLOAT
        SPECIFIC FINDSTRINCLOB
        EXTERNAL NAME 'FINDSTR'
        LANGUAGE C
        PARAMETER STYLE DB2SQL
        NO SQL
        DETERMINISTIC
        NO EXTERNAL ACTION
        FENCED;
```

```
CREATE FUNCTION MATH."/" (INT, INT)
        RETURNS INTEGER
        SPECIFIC DIVIDE
        EXTERNAL NAME 'DIVIDE'
        LANGUAGE ASSEMBLE
        PARAMETER STYLE DB2SQL
        NO SQL
        DETERMINISTIC
        NO EXTERNAL ACTION
        FENCED;
```

```
CREATE FUNCTION FINDSTRING (BLOB(500K), VARCHAR(200))
        RETURNS INTEGER
        CAST FROM FLOAT
        SPECIFIC FINDSTRINBLOB
        EXTERNAL NAME 'FNDBLOB'
        LANGUAGE C
        PARAMETER STYLE DB2SQL
        NO SQL
        DETERMINISTIC
        NO EXTERNAL ACTION
        FENCED;
```

12 Writing a User-defined Function

A user-defined function is similar to any other SQL program.

When writing a user-defined function the following can be included:

- static or dynamic SQL statements.
- IFI calls.
- DB2 commands issued through IFI calls.

An external user-defined function can be written in assembler, C, C++, COBOL, or PL/I.

User-defined functions that are written in COBOL can include object-oriented extensions, just as other DB2 COBOL programs can.

12.1 Restrictions on User-defined Function Programs

When writing a user-defined function the following restrictions need to recognized:

- If the user-defined function is not defined with parameters SCRATCHPAD or EXTERNAL ACTION, the user-defined function is not guaranteed to execute under the same task each time it is invoked.
- COMMIT or ROLLBACK statements can not be executed in a user-defined function.
- All open cursors in a user-defined scalar function must be closed. DB2 returns an SQL error if a user-defined scalar function does not close all cursors before it completes.

12.2 User-defined Function as a Main Program or as a Subprogram

A user-defined function can be coded as either a main program or a subprogram.

The program must be coded in order to match the way it has been defined in the user-defined function either with:

- PROGRAM TYPE MAIN

  or

- PROGRAM TYPE SUB parameter.

The primary difference between the two is that when a main program starts, Language Environment allocates the application program storage that the external user-defined function uses. When a main program ends, Language Environment closes files and releases dynamically allocated storage.

If the user-defined function has been coded as a subprogram, better performance can be realized by managing the storage and files. The user-defined function should always free any allocated storage before it exits. A scratchpad can be used for keeping data between invocations of the user-defined function.

A user-defined table function which accesses external resources will need to be defined as a subprogram. It will also be necessary to ensure that the definer specifies the EXTERNAL ACTION parameter in the CREATE FUNCTION or ALTER FUNCTION statement.

Program variables for a subprogram persist between invocations of the user-defined function, and using the EXTERNAL ACTION parameter ensures that the user-defined function stays in the same address space from one invocation to another.

12.3 Passing Parameters in a User-defined Function

```
CBL APOST,RES,RENT
      IDENTIFICATION DIVISION.

 .
 .
 .
      DATA DIVISION.

 .
 .
 .
      LINKAGE SECTION.
     ************************************************************
     * Declare each of the parameters                           *
     ************************************************************
      01  UDFPARM1 PIC S9(9) USAGE COMP.
      01  UDFPARM2 PIC X(10).

 .
 .
 .
     ************************************************************
     * Declare these variables for result parameters            *
     ************************************************************
      01  UDFRESULT1 PIC X(10).
      01  UDFRESULT2 PIC X(10).

 .
 .
 .
     ************************************************************
     * Declare a null indicator for each parameter              *
     ************************************************************
      01  UDF-IND1 PIC S9(4) USAGE COMP.
      01  UDF-IND2 PIC S9(4) USAGE COMP.

 .
 .
 .
     ************************************************************
     * Declare a null indicator for result parameter            *
     ************************************************************
      01  UDF-RIND1 PIC S9(4) USAGE COMP.
      01  UDF-RIND2 PIC S9(4) USAGE COMP.

 .
 .
 .
```

```
*************************************************************
* Declare the SQLSTATE that can be set by the               *
* user-defined function                                     *
*************************************************************
 01  UDF-SQLSTATE PIC X(5).
*************************************************************
* Declare the qualified function name                       *
*************************************************************
 01  UDF-FUNC.
     49 UDF-FUNC-LEN PIC 9(4) USAGE BINARY.
     49 UDF-FUNC-TEXT PIC X(137).
*************************************************************
* Declare the specific function name                        *
*************************************************************
 01  UDF-SPEC.
     49 UDF-SPEC-LEN PIC 9(4) USAGE BINARY.
     49 UDF-SPEC-TEXT PIC X(128).

*************************************************************
* Declare SQL diagnostic message token                      *
*************************************************************
 01  UDF-DIAG.
     49 UDF-DIAG-LEN PIC 9(4) USAGE BINARY.
     49 UDF-DIAG-TEXT PIC X(70).
*************************************************************
* Declare the scratchpad                                    *
*************************************************************
 01  UDF-SCRATCHPAD.
     49 UDF-SPAD-LEN PIC 9(9) USAGE BINARY.
     49 UDF-SPAD-TEXT PIC X(100).
*************************************************************
* Declare the call type                                     *
*************************************************************
 01  UDF-CALL-TYPE PIC 9(9) USAGE BINARY.
*************************************************************
* Declare the DBINFO structure
*************************************************************
 01  UDF-DBINFO.
*       Location length and name
     02 UDF-DBINFO-LOCATION.
        49 UDF-DBINFO-LLEN PIC 9(4) USAGE BINARY.
        49 UDF-DBINFO-LOC  PIC X(128).
*       Authorization ID length and name
     02 UDF-DBINFO-AUTHORIZATION.
        49 UDF-DBINFO-ALEN PIC 9(4) USAGE BINARY.
        49 UDF-DBINFO-AUTH PIC X(128).
*       CCSIDs for DB2 for OS/390
     02 UDF-DBINFO-CCSID PIC X(48).
```

```
           02 UDF-DBINFO-CCSID-REDEFINE REDEFINES UDF-DBINFO-CCSID.
              03 UDF-DBINFO-ESBCS   PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-EMIXED  PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-EDBCS   PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-ASBCS   PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-AMIXED  PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-ADBCS   PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-ENCODE  PIC 9(9) USAGE BINARY.
              03 UDF-DBINFO-RESERV0 PIC X(20).
      *       Schema length and name
           02 UDF-DBINFO-SCHEMA0.
              49 UDF-DBINFO-SLEN PIC 9(4) USAGE BINARY.
              49 UDF-DBINFO-SCHEMA PIC X(128).
      *       Table length and name
           02 UDF-DBINFO-TABLE0.
              49 UDF-DBINFO-TLEN PIC 9(4) USAGE BINARY.
              49 UDF-DBINFO-TABLE  PIC X(128).
      *       Column length and name
           02 UDF-DBINFO-COLUMN0.
              49 UDF-DBINFO-CLEN PIC 9(4) USAGE BINARY.
              49 UDF-DBINFO-COLUMN  PIC X(128).
      *       DB2 release level
           02 UDF-DBINFO-VERREL PIC X(8).

      *       Unused
           02 FILLER            PIC X(2).
      *       Database Platform
           02 UDF-DBINFO-PLATFORM PIC 9(9) USAGE BINARY.
      *       # of entries in Table Function column list
           02 UDF-DBINFO-NUMTFCOL PIC 9(4) USAGE BINARY.
      *       reserved
           02 UDF-DBINFO-RESERV1 PIC X(24).
      *       Unused
           02 FILLER             PIC X(2).
      *       Pointer to Table Function column list
           02 UDF-DBINFO-TFCOLUMN PIC 9(9) USAGE BINARY.
      *       Pointer to Application ID
           02 UDF-DBINFO-APPLID  PIC 9(9) USAGE BINARY.
      *       reserved
           02 UDF-DBINFO-RESERV2 PIC X(20).
      *
       PROCEDURE DIVISION USING UDFPARM1, UDFPARM2, UDFRESULT1,
                                UDFRESULT2, UDF-IND1, UDF-IND2,
                                UDF-RIND1, UDF-RIND2,
                                UDF-SQLSTATE, UDF-FUNC, UDF-SPEC,
                                UDF-DIAG, UDF-SCRATCHPAD,
                                UDF-CALL-TYPE, UDF-DBINFO.
```

12.4 Scratchpad in a User-defined Function

A scratchpad can be used for saving information between invocations of a user-defined function. The function definer specifies the SCRATCHPAD parameter in the CREATE FUNCTION statement. It is used for indicating that a scratchpad should be allocated when the user-defined function executes.

The scratchpad consists of a 4-byte length field, followed by the scratchpad area. The definer can specify the length of the scratchpad area in the CREATE FUNCTION statement. The specified length does not include the length field. The default size is 100 bytes.

DB2 initializes the scratchpad for each function to binary zeros at the beginning of execution for each subquery of an SQL statement. It will not examine or change the content thereafter. On each invocation of the user-defined function, DB2 passes the scratchpad to the user-defined function.

The scratchpad can be used for preserving information between invocations of a reentrant user-defined function.

Example:

```
CREATE FUNCTION COUNTER()
        RETURNS INT
        SCRATCHPAD
        FENCED
        NOT DETERMINISTIC
        NO SQL
        NO EXTERNAL ACTION
        LANGUAGE C
        PARAMETER STYLE DB2SQL
        EXTERNAL NAME 'UDFCTR';
```

13 Distinct Types

A distinct type is a data type that is defined using the CREATE DISTINCT TYPE statement. Each distinct type has the same internal representation as a built-in data type.

Distinct types can not be used in the same as built-in data types are used. The exception is use with a DB2 private protocol application.

Example:

Consider a scenario where audio and video data needs to be defined in a DB2 table. Columns can be defined for both types of data as BLOB; however, it might be useful to use a data type that more specifically describes the data. In order to do this, it will be necessary to define distinct types. These types can then be used when defining columns in a table or manipulating the data in those columns.

Distinct types for the audio and video data can be defined as follow:

```
CREATE DISTINCT TYPE AUDIO AS BLOB (1M);
CREATE DISTINCT TYPE VIDEO AS BLOB (1M);
```

The CREATE TABLE statement be:

```
CREATE TABLE VIDEO_CATALOG;
        (VIDEO_NUMBER       CHAR(6)  NOT NULL,
        VIDEO_SOUND         AUDIO,
        VIDEO_PICS          VIDEO,
        ROW_ID              ROWID GENERATED ALWAYS);
```

A column of type ROWID must be defined in the table because tables with any type of LOB columns require a ROWID column, and internally, the VIDEO_CATALOG table contains two LOB columns.

After defining distinct types and columns of those types, these data types can be used in the same way as built-in types are used. The data types can be used in assignments, comparisons, function invocations, and stored procedure calls. However, when assigning one column value to another or comparing two column values, the values must be of the same distinct type.

A column value of type VIDEO must be assigned to a column of type VIDEO, and a column value of type AUDIO can only be assigned to a column of type AUDIO. When assigning a host variable value to a column with a distinct type, any host data type can be used which is compatible with the source data type of the distinct type.

Example:

To receive an AUDIO or VIDEO value, a host variable can be defined.

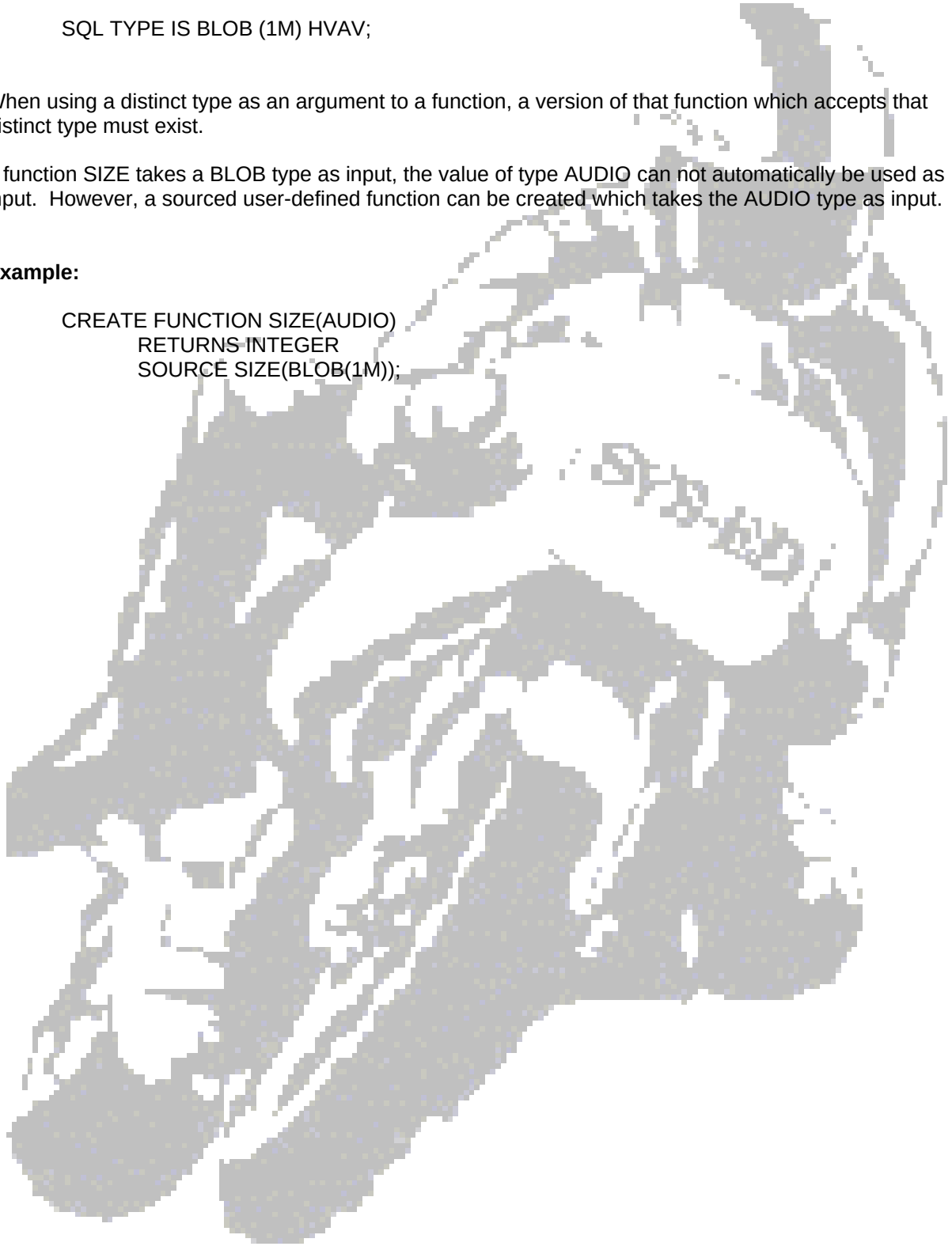
```
SQL TYPE IS BLOB (1M) HVAV;
```

When using a distinct type as an argument to a function, a version of that function which accepts that distinct type must exist.

If function SIZE takes a BLOB type as input, the value of type AUDIO can not automatically be used as input. However, a sourced user-defined function can be created which takes the AUDIO type as input.

Example:

```
CREATE FUNCTION SIZE(AUDIO)
        RETURNS INTEGER
        SOURCE SIZE(BLOB(1M));
```



14 Distinct Types in Application Programs

Using distinct types will result in DB2 enforcing strong typing for distinct types.

Strong typing ensures that only functions, procedures, comparisons, and assignments which have been defined for a data type can be used.

Example:

If a user-defined function has been assigned to convert U.S. dollars to euro currency, it will be necessary to prevent anyone else from using this same user-defined function to convert Japanese Yen to euros. This is because the U.S. dollars to euros function returns the wrong amount.

Consider a scenario where the following three distinct types have been defined:

```
CREATE DISTINCT TYPE US_DOLLAR AS DECIMAL(9,2) WITH COMPARISONS;
CREATE DISTINCT TYPE EURO AS DECIMAL(9,2) WITH COMPARISONS;
CREATE DISTINCT TYPE JAPANESE_YEN AS DECIMAL(9,2) WITH COMPARISONS;
```

If a conversion function is defined that takes an input parameter of type US_DOLLAR as input, DB2 returns an error when this function is executed with an input parameter of type JAPANESE_YEN.