Quick NonStop SQL/MX Database Management

UC404S A.00

HP Training
Student guide
# Table of Contents

## Preface

- Course Objectives ................................................................. Preface-2
- Course Schedule ................................................................. Preface-3

## Module 1 — Overview of SQL/MX Release 2.x Architecture

- Module Objectives ................................................................. 1-2
- SQL/MX Release 2.x Architecture ........................................ 1-3
- System Catalog and Schemas .................................................. 1-4
- DEFINITION_SCHEMA_VERSION_nnnn Schema .......................... 1-5
- SYSTEM_SCHEMA Schema ..................................................... 1-6
- SYSTEM_DEFAULTS_SCHEMA Schema .................................... 1-7
- MXCS_SCHEMA (MX Connectivity Services Schema) ............. 1-8
- SYSTEM_SQLJ_SCHEMA Schema .......................................... 1-9
- User Metadata (UMD) Tables .................................................. 1-10
- User Catalog and Schema ...................................................... 1-11
- SQL/MX User Tables ............................................................ 1-12
- SQL/MX Objects ................................................................ 1-13
- SQL/MX Tables .................................................................. 1-14
- SQL/MX Indexes ................................................................ 1-15
- SQL/MX Views .................................................................. 1-16
- SQL/MX Constraints ............................................................ 1-17
- SQL/MX Triggers ................................................................. 1-18
- SQL/MP Aliases, Stored Procedures, and DDL Locks ............. 1-19
- SQL/MX Object Namespaces — Object Type ......................... 1-20
- Underlying Physical Support for SQL/MX Objects ............... 1-21
- Physical File Structure ......................................................... 1-22/23
- SYSTEM_SCHEMA — Schema Subvolume and Files ............. 1-24
- DEFINITION_SCHEMA_VERSION_nnnn — Schema Subvolume and Files ......................................................... 1-25
- SYSTEM_DEFAULT_SCHEMA — Schema Subvolume and Files ......................................................... 1-26
- MXCS_SCHEMA — Schema Subvolume and Files ................. 1-27
- mxci SHOWLABEL Command .............................................. 1-28/29
- mxci SHOWDDL Command .................................................. 1-30
- FUP INFO DETAIL Display for an SQL/MX Table .................. 1-31/32
- SQL/MX Security Model ...................................................... 1-33
- SQL/MX Process Architecture ............................................. 1-34
- SQL/MX Components ......................................................... 1-35
- Catalog Manager ............................................................... 1-36
- SQL/MX DDL Operations ...................................................... 1-37
- SQL/MX Utilities .............................................................. 1-38
- Utilities ............................................................................ 1-39
- Syntax-Directed Utility Architecture .................................... 1-40
- NSM/Web Architecture ....................................................... 1-41
Module 2 — Referential Integrity Constraints

Module Objectives ................................................................................................. 2-2
Referential Integrity (RI) ....................................................................................... 2-3
Referential Integrity Terminology ......................................................................... 2-4
REFERENCES Column Constraint — Syntax ..................................................... 2-5
SYSTEM_DEFAULTS Table — Referential Action ........................................... 2-6
References Constraint — Example 1 ............................................................... 2-7
References Constraint — SHOWDDL for Example 1.......................................... 2-8
References Constraint — Example 2 ............................................................... 2-9
References Constraint — Example 3 ............................................................... 2-10
FOREIGN KEY Column or Table Constraint — Syntax ....................................... 2-11
Foreign Key Constraint — Example 1 ............................................................... 2-12
Foreign Key Constraint — Example 2 ............................................................... 2-13
Foreign Key Constraint — SHOWDDL ........................................................ 2-14/15
Referential Integrity Checks for Inserts .............................................................. 2-16
Referential Integrity Checks for Deletes ............................................................. 2-17
Referential Integrity Checks for Updates ............................................................ 2-18

Module 3 — Creating SQL/MX Partitioned Tables and Indexes

Module Objectives ................................................................................................. 3-2
Range Partitioning ................................................................................................. 3-3
Hash Partitioning ................................................................................................... 3-4
Range and Hash Partitioning Comparison ............................................................ 3-5
Range Partitioning — Advantages and Disadvantages ......................................... 3-6
Hash Partitioning — Advantages .......................................................................... 3-7
Hash Partitioning — Disadvantages ...................................................................... 3-8
CREATE TABLE — Syntax ................................................................................. 3-9
CREATE TABLE — Range Partitioning Syntax .............................................. 3-10/11
Creating a Range Partitioned Table ..................................................................... 3-12
CREATE TABLE — Hash Partitioning Syntax ................................................. 3-13
Creating a Hash Partitioned Table ....................................................................... 3-14
CREATE INDEX — Syntax .............................................................................. 3-15/16
Creating a Range-Partitioned Index .................................................................... 3-17
Creating a Hash-Partitioned Index ...................................................................... 3-18
Multiple Partitions per Disk Volume ................................................................... 3-19
Decoupled Keys ................................................................................................. 3-20
Terminology ...................................................................................................... 3-21/22
Co-Partitioning ................................................................................................. 3-23
Creating a Partitioned Table with a Decoupled Partitioning Key ....................... 3-24
Creating a Partitioned Table with a Decoupled Partitioning Key — Example .... 3-25
Creating a Partitioned Index with a Decoupled Partitioning Key ....................... 3-26
Creating a Decoupled and Co-partitioned Index ............................................... 3-27
Partitioning Key Columns ................................................................................... 3-28
Specifying a Clustering Key ............................................................................... 3-29/30
Key Columns for Indexes ................................................................................... 3-31
Co-partitioned and Co-located Table and Index ............................................... 3-32
Co-partitioned and Co-located Tables ............................................................... 3-33
Module 4 — Creating SQL/MX Triggers

Module Objectives........................................................................................................ 4-2
Trigger Definition....................................................................................................... 4-3
Trigger Terminology .................................................................................................. 4-4/5
CREATE TRIGGER Statement — Syntax .................................................................. 4-6/7
Trigger Temporary Table .......................................................................................... 4-8/10
TEMPORARY_TABLE_HASH_PARTITIONS — System Default .................................. 4-11
Referencing for a BEFORE Triggering Event ........................................................... 4-12
BEFORE Triggers — Syntax ..................................................................................... 4-13
BEFORE UPDATE ROW Trigger with SET Statement .............................................. 4-14
BEFORE UPDATE ROW Trigger with SIGNAL Statement ....................................... 4-15
BEFORE Triggers — Considerations ........................................................................ 4-16/17
Referencing for an AFTER Triggering Event ........................................................... 4-18
AFTER Trigger — Syntax ......................................................................................... 4-19
AFTER INSERT Trigger with an Insert Action ......................................................... 4-20
AFTER INSERT Trigger with an Update Action ....................................................... 4-21
AFTER UPDATE Trigger with an Update Action ..................................................... 4-22
AFTER DELETE Trigger with an Insert Action ....................................................... 4-23
AFTER Triggers — Considerations .......................................................................... 4-24
Cascading Triggers .................................................................................................... 4-25
Desired Characteristics of Triggers — Termination and Confluence ...................... 4-26/27
Conflicting Triggers and Access Sets ................................................................. 4-28/29
Conflicting Triggers — Example .............................................................................. 4-30
Trigger Thresholds .................................................................................................... 4-31
Error Handling with Triggers .................................................................................. 4-32
Dropping Triggers and Enabling or Disabling Triggers ........................................ 4-33
Trigger Metadata Tables ......................................................................................... 4-34/35
Triggers Versus Constraints .................................................................................... 4-36
Triggers Versus Publish and Subscribe ................................................................... 4-37
Restrictions on Triggers ............................................................................................ 4-38
Course Objectives

- Describe the SQL/MX Release 2.x logical and physical architecture.
- Describe and create referential integrity constraints.
- Describe and create SQL/MX range and hash partitioned tables and indexes.
- Describe and create SQL/MX triggers.

The intent of this short one day course is for the experienced NonStop SQL users to adapt the necessary information to operate in SQL/MX 2.x environment. There are full version training available in the related topic area.

Below a list of the full version course flow including lab topics.

**U8617S NonStop SQL/MX Database Management 5 days**

Module 1 — Overview of SQL/MX Release 2.0 Architecture
  - Module 1 Lab 1 — Overview of SQL/MX Release 2.0 Architecture Using mxci
  - Module 1 Lab 2 — Overview of SQL/MX Release 2.0 Architecture Using NSM/web

Module 2 — SQL/MX Release 2.0 Installations
  - Module 2 Lab — Installing SQL/MX

Module 3 — Metadata Migration from SQL/MX R1.8 to SQL/MX R2.0
  - Module 3 Lab — Migrating Metadata Demo

Module 4 — Creating SQL/MX Objects
  - Module 4 Lab 1 — Creating SQL/MX Database Objects Using mxci
  - Module 4 Lab 2 — Creating SQL/MX Database Objects Using NSM/web

Module 5 — Referential Integrity Constraints
  - Module 5 Lab 1 — Creating SQL/MX Referential Integrity Constraints Using mxci
  - Module 5 Lab 2 — Creating SQL/MX Referential Integrity Constraints Using NSM/web

Module 6 — Creating SQL/MX Partitioned Tables and Indexes
  - Module 6 Lab 1 — Creating a Range Partitioned Table and Index Using mxci
  - Module 6 Lab 2 — Creating a Hash Partitioned Table and Index Using mxci
  - Module 6 Lab 3 — Creating a SQL/MX Range Partitioned Table and Index Using NSM/web
  - Module 6 Lab 4 — Creating a SQL/MX Hash Partitioned Table and Index Using NSM/web

Module 7 — Creating SQL/MX Triggers
  - Module 7 Lab 1 — Creating SQL/MX Triggers
  - Module 7 Lab 2 — Creating SQL/MX Triggers (Optional)
Course Schedule

- Module 1 — Overview of SQL/MX Release 2.x,
- Module 2 — Referential Integrity,
- Module 3 — Partitioning,
- Module 4 — Triggers.

U8617S NonStop SQL/MX Database Management 5 days (Continued)

Module 8 — NonStop SQL/MX Security
  - Module 8 Lab 1 — Granting and Revoking Privileges with mxci
  - Module 8 Lab 2 — Granting and Revoking Privileges with NSM/web

Module 9 — Import Utility and Populate Index
  - Module 9 Lab 1 — Importing Data
  - Module 9 Lab 2 — Importing Data into Range and Hashed Partitioned Tables

Module 10 — Reorganizing SQL/MX Tables and Indexes
  - Module 10 Lab — Reorganizing Data

Module 11 — Updating Statistics
  - Module 11 Lab — Updating and Viewing Statistics

Module 12 — Database Protection and Recovery Using TMF
  - Module 12 Lab — Database Protection and Recovery Using TMF

Module 13 — Compiling SQL/MX Programs in the OSS Environment
  - Module 13 Lab 1 — Compiling Embedded SQL/MX COBOL Programs in the OSS Environment
  - Module 13 Lab 2 — Compiling Embedded SQL/MX C Programs in the OSS Environment

Module 14 — PURGEDATA Command and MODIFY Utility
  - Module 14 Lab — PURGEDATA Command and MODIFY Utility

Module 15 — SQL/MX Utilities
  - Module 15 Lab — SQL/MX Utilities

Module 16 — Managing SQL/MX Applications
  - Module 16 Lab 1 — Managing Embedded SQL/MX COBOL Applications
  - Module 16 Lab 2 — Managing Embedded SQL/MX C Applications

Module 17 — SQL/MX Distributed Database
  - Module 17 Lab — SQL/MX Distributed Database Demo

Module 18 — Adding, Altering, and Dropping SQL/MX Database Objects
  - Module 18 Lab — Adding, Altering and Dropping SQL/MX Database Objects
Module Objectives

- Describe the HP NonStop SQL/MX Release 2 architecture that uses ANSI catalogs and schemas.
- Describe the physical underlying files that support the SQL/MX Release 2 objects.
- Describe the SQL/MX Release 2 objects, processes, and utilities.

References

HP NonStop SQL/MX Installation and Management Guide.
SQL/MX Release 2.x Architecture

Uses ANSI catalogs and schemas, which provide a logical architecture for organizing the database objects.

- A catalog is a collection of schemas.
  - The system catalog is created when SQL/MX is initialized.
  - User catalogs are created with the CREATE CATALOG command.
- A schema is a collection of objects owned by the schema creator.
  - System schemas are created when SQL/MX is initialized; contains system metadata tables.
  - User Schemas are created with the CREATE SCHEMA command.
  - DEFINITION_SCHEMA_VERSION_nnnn schema — One per catalog and they are created when the first user schema is created in a catalog. They contain thirty-four system metadata tables, which hold the definitions of the SQL objects created in that catalog.
- SQL objects — Tables, indexes, views, constraints, triggers, stored procedures, and SQLMP ALIASes.
This slide illustrates the SQL/MX system catalog and schemas that are created when you initialize SQL/MX 2.0. The system catalog is named NONSTOP_SQLMX_<nodename> and contains the following system schemas and a DEFINITION_SCHEMA_VERSION_nnn schema.

**SYSTEM_SCHEMA** — Contains metadata tables used to resolve object names.

**SYSTEM_DEFAULTS_SCHEMA** — Contains the SYSTEM_DEFAULTS metadata table, which contains the system-level default settings that override some of the system-defined default settings.

**MXCS_SCHEMA** — The MX Connectivity Services schema contains metadata tables for the ODBC and JDBC configuration data.

**SYSTEM_SQLJ_SCHEMA** — Is used for Java stored procedure support.

**DEFINITION_SCHEMA_VERSION_nnnn schema** — Each catalog contains one DEFINITION_SCHEMA_VERSION_nnnn schema. This schema contains the metadata tables that describe all the objects created in the catalog. The nnnn value represents the version number for a particular software release. For example, the version number for SQL/MX Release 2 is 1200. This definition schema contains the definitions of all SQL/MX objects created in the system catalog, including its metadata tables because it is self-describing. This means all objects created in the MXCS_SCHEMA, SYSTEM_DEFAULTS_SCHEMA, SYSTEM_SCHEMA, and SYSTEM_SQLJ_SCHEMA.
This slide illustrates the system metadata (SMD) tables in the `DEFINITION_SCHEMA_VERSION_nnnn` schema. There is one `DEFINITION_SCHEMA_VERSION_nnnn` schema per catalog, which is created when the first schema is created in a catalog. These thirty-four metadata tables contain the definitions of all objects created in the catalog. The `nnnn` value represents the version number for a particular software release. For example, the version number for SQL/MX Release 2 is 1200. This definition schema in the system catalog contains the definitions of all SQL/MX objects created when SQL/MX is initialized on a system.

Users cannot modify data directly in the system metadata tables. However, they are secured for PUBLIC SELECT access so that you can query them. SQL/MX automatically creates and maintains the metadata as users create, alter, and drop SQL/MX objects.

| System Metadata (SMD) Tables                  | MVS                  | MVS_COLS             | TBL_PRIVILEGES
|----------------------------------------------|----------------------|----------------------|-----------------------
| ACCESS_PATHS                                 |                      |                      |                       |
| ACCESS_PATH_COLS                             |                      |                      |                       |
| CK_COL_USAGE                                 |                      |                      |                       |
| CK_TBL_USAGE                                 |                      |                      |                       |
| COLS                                         |                      |                      |                       |
| COL_PRIVILEGES                               |                      |                      |                       |
| DDL_LOCKS                                    |                      |                      |                       |
| DDL_PARTITION_LOCKS                          |                      |                      |                       |
| KEY_COL_USAGE                                |                      |                      |                       |
| MODULES                                      |                      |                      |                       |
| MP_PARTITIONS                                |                      |                      |                       |
| MV_GROUPS                                    |                      |                      |                       |
| SYSTEM                                           |                      |                      |                       |
| VW_COL_USAGE                                 |                      |                      |                       |
| VW_COL_TBL                                   |                      |                      |                       |
| VW_COL_TBL_COLS                              |                      |                      |                       |
| VW_TBL_USAGE                                 |                      |                      |                       |
| VW_TBL_USAGE                                 |                      |                      |                       |
This slide illustrates the system metadata tables (SMDs) in the SYSTEM_SCHEMA schema.

- **ALL_UIDS table** — Stores the object UIDS for all the objects in the system. A UID is a 64-bit unique identifier that is internally generated to represent an object in the metadata. It is a shorthand for the three part ANSI name of an object. The ALL_UIDS table prevents duplicate object UIDS by having the UID as the primary key. This unique identifier is a full or partial primary key for many of the metadata tables. There is also a UID for each catalog name. Catalog UIDs must be unique among all catalog UIDs on a node. Likewise the schema UID representing a schema must be unique on a node.

- **CATSYS table** — Records all of the catalogs that are created on the system. It has the catalog information, such as, catalog name, catalog UID, catalog owner, the volume where system metadata (SMD) and user metadata (UMD) tables in this catalog reside.

- **CAT_REFERENCES table** — Describes the location of catalog replicas (remote catalogs registered on this node).

- **SCHEMATA table** — Records all of the schemas in the system. It has information such as schema name, schema UIDs, schema owner, schema version, etc.

- **SCHEMA_REPLICAS table** — Lists locations (schema UID and Expand node name) of all replicas for all schemas that have definitions on the node.
SYSTEM_DEFAULTS_SCHEMA — Contains the SYSTEM_DEFAULTS table, a user metadata (UMD) table, that you use to store system-level default settings that override some of the system-defined default settings. SQL/MX uses system-defined default settings for attributes that are associated with compiling and executing queries. The system-defined default settings, which are hard-coded settings, are optimal under most circumstances. However, in some circumstances, you might want to override specific system-defined default settings.

To update the SYSTEM_DEFAULTS table, you must be the super ID or a user to whom the super ID has granted UPDATE privileges. All other users have SELECT privileges on this table.
This slide illustrates the MX Connectivity Services schema, which has system metadata tables that contain the ODBC and JDBC configuration data. This schema and its metadata tables are created when you INITIALIZE SQL. The schema contains the following SMD tables:

- ASSOC2DS — Associates a MXCS service to a data source.
- DATASOURCES — Contains data source definitions.
- ENVIRONMENTVALUES — Contains environment variable (SET statements, CONTROL statements, and DEFINES) definitions.
- NAME2ID — Associates a service or data source names to an ID.
- RESOURCEPOLICIES — Contains governing information.
SYSTEM_SQLJ_SCHEMA is used for Java stored procedure support. It has 1 stored procedure called VALIDATE_ROUTINE which is used by SQL/MX during the creation of a stored procedure to verify that the user’s Java method does exist. To do this, SQL/MX internally performs some Java work by calling this VALIDATE_ROUTINE through an internal CALL statement.
User Metadata (UMD) Tables

- All user schemas contain the following user metadata tables.
  - HISTOGRAMS table.
  - HISTOGRAM_INTERVALS table.
  - MVS_TABLE_INFO_UMD table.
  - MVS_U MD table.
  - MVS_USED_UMD table.
- The SYSTEM_DEFAULTS_SCHEMA only contains the SYSTEM_DEFAULTS user metadata table.

The user metadata (UMD) tables are SQL/MX metadata tables. You can modify data in user metadata tables and grant privileges on these tables but you cannot alter or drop these tables.

- **HISTOGRAMS table** — Describes columns, interval count, total number of rows and number of unique rows, and the low and high values of column distribution for a SQL/MX table.
- **HISTOGRAM_INTERVALS table** — Describes for each interval, the number of rows and number of unique rows in the interval and the value of the upper boundary for the interval.
- **MVS_TABLE_INFO_UMD table** — Reserved for future use.
- **MVS_U MD table** — Reserved for future use.
- **MVS_USED_UMD table** — Reserved for future use.
- **SYSTEM_DEFAULTS table** — Used to store system-level default settings that override some of the system-defined default settings.
This slide illustrates the SQL/MX sample database user catalog and schemas that you can create by executing the setmxdb script in the /usr/tandem/sqlmx/bin directory. When you create the first user schema, for example, the INVENT schema in the SAMDBCAT user catalog, the DEFINITION_SCHEMA_VERSION_nnnn schema is created first. The DEFINITION_SCHEMA_VERSION_nnnn schema contains the metadata definitions for all the objects created in these three schemas (INVENT, PERSNL, and SALES).

The SAMDBCAT user catalog is created with the CREATE CATALOG command. The sample database user schemas (INVENT, PERSNL, and SALES) are created with the CREATE SCHEMA command.
This slide illustrates some of the objects in the PERSNL user schema. Note that the PERSNL user schema also contains the user metadata tables (HISTOGRAM, HISTOGRAM_INTERVALS, MVS_TABLE_INFO_UMD, MVS_UMD, and MVS_USED_UMD). These user metadata tables are also created in the INVENT and SALES schemas when those schemas are created. Note the HISTOGRAMS and HISTOGRAM_INTERVALS tables are used to hold the histogram statistics for the SQL/MX tables in the PERSNL user schema when you update statistics for those tables. If you create SQL/MP aliases for SQL/MP tables in the SAMDBCAT.PERSNL schema, then HISTOGRAM and HISTOGRAM_INTERVALS table will also contain histogram statistics when for those SQL/MP tables, when you use the mxci UPDATE STATISTICS utility on those SQL/MP tables.
**SQL/MX Objects**

- Tables.
- Indexes.
- Views.
- Constraints.
- SQLMP aliases.
- Triggers.
- Stored procedures.
- DDL_LOCKS.

This slide lists the SQL/MX objects.
SQL/MX Tables

- Format 2 only.
- Key-sequenced only.
- New row format.
  - VARCHAR columns physically placed at the end of the row.
  - Provides constant access time to any column within a row.
- Audited only.
  - Temporarily un-audited during some utility and DDL operations.
- Data types same as SQL/MP except:
  - Standard DATE, TIME, TIMESTAMP columns only.
  - IEEE floating point instead of Tandem floating point.
  - ISO8859/1 and UCS2 character sets.
  - Default collation.

This slide illustrates some of the differences between SQL/MX and HP NonStop SQL/MP tables. SQL/MX uses format 2 tables only, which means the architectural limit is 1 terabyte. Basically the table size is limited by the size of the disk.

Key-sequenced tables only, unlike SQL/MP which also has relative and entry-sequenced tables. The row format in SQL/MX tables is different than in SQL/MP tables. In SQL/MX tables VARCHAR columns are always physically placed at the end of the row and all fixed-length columns are placed at the beginning of the row, even if the column has been added with an ALTER TABLE ADD COLUMN command. This provides a constant time to access any column within the row.

SQL/MX uses audited tables only; however, SQL/MX does make tables temporarily unaudited during some utility or DDL operation, for example, when you are doing partition management and splitting a partition. When the new partition is being loaded, the new partition is in an unaudited state but this is not visible to applications.

SQL/MX has the same data types as in SQL/MP but with some exceptions. SQL/MX only supports the standard DATE, TIME, and TIMESTAMP columns. SQL/MP has more DATETIME data types.

SQL/MX uses IEEE floating point instead of the Tandem floating. However, these floating-point data types can interoperate. You can insert data from a Tandem floating column from a SQL/MP table into an IEEE floating-point column in an SQL/MX table. SQL/MX will make the necessary conversions.

Character sets — Support the ISO8859/1 character set, which is a single-byte character set popular in Western Europe. Also support the UCS2 encoding of Unicode that is a 2 byte character set.

Collations — SQL/MX supports only the DEFAULT collation, which is based on binary ordering and is the default collating sequence for CHAR and NCHAR data types. You cannot access any SQL/MP tables defined with collations other than those tables defined with the default collation.
SQL/MX Indexes

- Provide an alternate access path to the data.
- No keytag column.
- Same file characteristics as tables, that is:
  - Key-sequenced, format 2, audited only, and new row format.
- Implicit versus explicit.
  - Explicit indexes are those created via CREATE INDEX.
  - Implicit indexes are created as needed to enforce constraints.

SQL/MX indexes are like SQL/MP indexes except for a few differences. The keytag column is not present in SQL/MX indexes. The keytag column was used in ENSCRIBE alternate-key files and carried over to SQL/MP indexes. The keytag column was not used by SQL/MP indexes and has been dropped from the SQL/MX index structure.

SQL/MX indexes have the same file characteristics as tables, that is, key-sequenced, format 2 structure, always audited except during certain utility and DDL operations, and having the new row format (VARCHAR columns placed at the end of the row).

SQL/MX indexes can be created explicitly or implicitly.

- An explicit index is one that you create using the CREATE index command.
- An implicit index is created by SQL/MX to enforce certain constraints. For example, if you create a primary-key constraint that is droppable, SQL/MX use an index to implement that. Foreign keys constraints also require an index. If no explicit index exists to enforce the constraint, then SQL/MX creates an implicit index to enforce the constraint. The implicit index names are internally generated, for example, EMPLOYEE_978583271_1679, which is a system-generated name of an implicit index created for a referential integrity constraint.
SQL/MX Views

- Comply with the SQL: 1999 standard.
  - Single table view (query expression on one underlying table).
  - Multi-table view (query expression on multiple underlying tables).
- View definition can contain columns from one or multiple SQL/MX tables. You cannot include columns from an SQL/MP table.
- View definition text length limit has been removed.
- View is updatable if the view definition is a simple select and project of a single table.
- View is insertable if updatable and all clustering key columns from the underlying table are included in the view definition.
- Views use GRANT and REVOKE security.

Views can be considered to be virtual tables; that is, they are formed by executing a query expression on underlying tables to create a result table (virtual table) for the view. SQL/MX supports single table views (query expression on one underlying table) or multi-table views (a query expression on multiple underlying tables).

Single table views are updatable:
  - If you update columns in a view and the columns in the underlying table are updated.
  - If you have UPDATE privileges on the underlying table’s columns.

You can also insert rows through a single table view if the view is insertable and if you have INSERT privileges on the underlying table. To create a view, you must have SELECT privileges on the columns of the underlying tables.
SQL/MX supports five kinds of constraints.

SQL/MX Constraints

- **CHECK** — Specifies a condition that must be satisfied for each row in the table.
- **NOT NULL** — Specifies that a column cannot contain nulls.
- **PRIMARY KEY** — Specifies a column or a set of columns as the primary key for the table.
- **UNIQUE** — Specifies that a column or a set of columns cannot contain more than one occurrence of the same value or set of values.
- **REFERENTIAL INTEGRITY** — Specifies that a column or set of columns in the referencing table can contain only values that match those in a column or set of columns in the referenced table.
SQL/MX Triggers

- An automated response to a database event, such as an INSERT, UPDATE, or DELETE operation on a subject table.
- Triggering-SQL-statement — The statement (event) that causes the trigger to fire.
- Trigger-action-time — Specifies when the trigger fires, AFTER or BEFORE the triggering SQL statement.
- Triggered-action or trigger type.
  - Row-level-trigger — A trigger whose triggered-SQL-statement is executed for every row modified by the triggering statement.
  - Statement-trigger — A trigger whose triggered-SQL-statement is executed once every time the triggering-SQL statement is executed.
- Trigger-condition — Determines whether trigger action should be executed.
- Triggered-SQL-statement — Specifies the SQL statement to be performed by the trigger.

Triggers are an automated response to a database event. For example, when a database event occurs, such as an INSERT, UPDATE, or DELETE operation on a table, you can create a trigger on the table to automatically respond to the event. When the event happens, an optional condition is evaluated, and if the condition is true the action is executed. The condition determines whether the trigger should be executed. The action is an SQL statement.

Triggers can be used:

- To perform logging and auditing, for example, tracking changes made to a table and by whom.
- To perform an operation on another table in response to an event occurring on the first table.
SQLMP Aliases, Stored Procedures, and DDL Locks

- SQL/MP aliases.
  - Defines a mapping of an ANSI name (catalog.schema.object) to a physical SQL/MP table or view.
    - The SQL/MX catalog and schema must exist.
  - You can map the same SQL/MP table or view to multiple different ANSI names in the same catalog and schema and in different catalogs and schemas.

- Stored procedures.
  - Java method that you register as a stored procedure in SQL/MX.

- DDL Locks.
  - Lock information for controlling concurrent DDL operations on SQL/MX objects.
**SQL/MX Object Namespaces — Object Type**

<table>
<thead>
<tr>
<th>Objects</th>
<th>Name Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>TA</td>
</tr>
<tr>
<td>View</td>
<td>TA</td>
</tr>
<tr>
<td>Store Procedure</td>
<td>TA</td>
</tr>
<tr>
<td>SQLMP ALIAS</td>
<td>TA</td>
</tr>
<tr>
<td>Index</td>
<td>IX</td>
</tr>
<tr>
<td>Constraints</td>
<td>CN</td>
</tr>
<tr>
<td>Trigger</td>
<td>TR</td>
</tr>
<tr>
<td>Trigger Temporary</td>
<td>TI</td>
</tr>
<tr>
<td>DDL Locks</td>
<td>LK</td>
</tr>
</tbody>
</table>

```sql
>>>SET SCHEMA <catalog-name>.DEFINITION_SCHEMA_VERSION_nnnn;
>>>SELECT OBJECT_NAME, OBJECT_TYPE
  FROM OBJECTS;
```

Within an SQL/MX schema each object’s name resides in a name space and must be unique within that name space. For example, tables, views, stored procedures, and SQLMP aliases share the same name space; therefore, their names must be unique within a schema. However, you could have an index with the same name as a table because the index name resides in a different name space.
Underlying Physical Support for SQL/MX Objects

- When you create an SQL/MX catalog, no underlying physical structure is created. However, the catalog is associated with a Guardian disk volume name, either by specifying the volume name in a LOCATION clause or using the default from the =_DEFAULT define.

- When you create an SQL/MX schema, it creates a Guardian subvolume, the schema subvolume. The schema subvolumes names begin with ZSD, which are system generated or you can specify the schema subvolume name with a LOCATION clause.

- When you create SQL/MX tables and indexes, they are supported by underlying Guardian files created in the schema subvolume.
This slide illustrates the underlying physical structures used to support SQL/MX tables and indexes.

The DP2 directory, which is common to Enscribe, SQL/MP, and SQL/MX, contains file labels. The file labels describe such things as the file itself, its format, extents, timestamps, and eof. There is also a table label that has information about the table, for example, the object type, ANSI name, and version and a security label that contains information that describes a user’s privileges (select, insert, update, delete, and reference privileges).

An SQL/MX table or index partition consists of two Guardian files.

- A file called the data fork that contains the user data, the rows inserted into the table.
- A file called the resource fork that contains run-time metadata. The resource fork is a key-sequenced file and its structure is the same for all SQL/MX objects. The resource fork contains additional run-time metadata, for example, the partition map array, the index map array, the record descriptor, the ANSI name, and constraint information. In SQL/MP the run-time metadata stored in the file labels has certain size limitations that limits, for example, the number of partitions, indexes, and so on. However, the resource fork is a separate key-sequenced file, which removes these architectural limits for SQL/MX.
Physical File Structure (2 of 2)

- A partition consists of:
  - File labels — Records in the DP2 directory.
  - Resource fork — A key-sequenced file containing additional run-time metadata (circumvents file label size limits).
  - Data fork — A key-sequenced file containing actual user data.
- Two Guardian files per partition.
  - Guardian names are “funny names,” usually system-generated
  - All files for objects in a given schema share the same subvolume name.
  - File names for data forks end in “00”.
  - File names for resource forks are the same as data forks, except that the end in “01”.

A physical partition consists of file labels which are records in the DP2 directory.
A resource fork can be thought of as an extension to the DP2 directory; it circumvents the file label size limits.
A data fork is where the actual user data is stored.
There are two files per partition instead of one as there is for SQL/MP objects.
The Guardian file names are funny names and are usually system-generated. You can specify the underlying filename when you create the object.
All of the files for objects in a given schema share the same subvolume name.
Data forks have filenames that end in the characters 00.
The corresponding resource fork will have the same file name as the data fork except that the last two characters end with 01.
This slide illustrates the underlying Guardian subvolume (a schema subvolume) for the SYSTEM_SCHEMA and the underlying Guardian files for the tables in this schema. The SYSTEM_SCHEMA subvolume name is ZSD0. Schema subvolume names for SQL/MX objects always begin with the characters ZSD. When you initialize SQL/MX, you specify the volume name for the system catalog, NONSTOP_SQLMX_<nodename> and all of the system schemas are created on that volume.

Note that each SQL/MX table has two underlying Guardian files. For example the ALL_UIDS table has the following underlying files in the ZSD0 subvolume: ALLUID00 and ALLUID01. ALLUID00 is the data fork and contains the data for the ALL_UIDS table. ALLUID01 is the resource fork and it contains run-time metadata. You can use the File Utility Program (FUP) INFO <filename>, DETAIL command to display detailed information about these files. The following files are created for the tables in the SYSTEM_SCHEMA when SQL/MX is initialized:

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Data Fork</th>
<th>Resource Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_UIDS</td>
<td>ALLUID00</td>
<td>ALLUID01</td>
</tr>
<tr>
<td>CATSYS</td>
<td>CATSYS00</td>
<td>CATSYS01</td>
</tr>
<tr>
<td>CAT_REFERENCES</td>
<td>CATREF00</td>
<td>CATREF01</td>
</tr>
<tr>
<td>SCHEMATA</td>
<td>SCHEMA00</td>
<td>SCHEMA01</td>
</tr>
<tr>
<td>SCHEMA_REPLICAS</td>
<td>SCHREP00</td>
<td>SCHREP01</td>
</tr>
</tbody>
</table>
This slide illustrates the underlying Guardian files for the metadata tables in a DEFINITION_SCHEMA_VERSION_nnnn schema. Note that each schema has an underlying schema subvolume, which begins with the letters ZSD. The fourth character in this schema subvolume name is always a digit for the DEFINITION_SCHEMA_VERSION_nnnn schema. File names ending in 00 are data fork files and file names ending in 01 are resource fork files.

Note: Not all of the DEFINITION_SCHEMA_VERSION_nnnn schema tables are listed in this slide. However, all of the underlying files for those tables are listed.
This slide illustrates the schema subvolume and the underlying Guardian files for the tables in the SYSTEM_DEFAULTS_SCHEMA. Note that this schema subvolume name also starts with the letters ZSD and is exactly eight characters long. The last five characters of the schema subvolume name are generated when the schema is created during SQL/MX initialization. This schema contains the SYSTEM_DEFAULTS table and the five UMD tables. File names ending in 00 are data fork files and file names ending in 01 are resource fork files. You can use the FUP INFO command with the detail option to identify which files support the SQL/MX tables.
This slide illustrates the schema subvolume for the MXCS_SCHEMA and the underlying files for those tables. The schema subvolume name for the MXCS_SCHEMA is ZSD1, which is created when you initialize SQL/MX. This schema also contains the five UMD tables. File names ending in 00 are data fork files and file names ending in 01 are resource fork files. The following MXCS_SCHEMA tables are supported by the underlying files:

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Data Fork</th>
<th>Resource Fork</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOC2DS</td>
<td>ASSOC200</td>
<td>ASSOC201</td>
</tr>
<tr>
<td>DATASOURCES</td>
<td>DATASO00</td>
<td>DATASO01</td>
</tr>
<tr>
<td>ENVIRONMENTVALUES</td>
<td>ENVIRO00</td>
<td>ENVIRO01</td>
</tr>
<tr>
<td>NAME2ID</td>
<td>NAME2I00</td>
<td>NAME2I01</td>
</tr>
<tr>
<td>RESOURCEPOLICIES</td>
<td>RESOUR00</td>
<td>RESOUR01</td>
</tr>
</tbody>
</table>
This slide and the next illustrate the output of the mxci SHOWLABEL command. The first two lines show the mapping of the ANSI name to the underlying Guardian file name.
### mxci SHOWLABEL Command (2 of 2)

- **AuditCompress**: T
- **Audited**: T (If F, a Utility operation is in progress or has failed)
- **Broken**: F
- **Buffered**: T
- **ClearOnPurge**: F
- **Corrupt**: F (If T, a Utility operation is in progress or has failed)
- **CrashLabel**: F
- **CrashOpen**: F
- **IncompletePartBoundChg**: F (If T, a Utility operation is in progress or has failed)
- **RedoNeeded**: F
- **RollfwdNeeded**: F
- **UndoNeeded**: F
- **UnreclaimedSpace**: F (If T, a Utility operation is in progress or has failed)

- **Primary Extent Size**: 16 Pages
- **Secondary Extent Size**: 64 Pages
- **Max Extents**: 160
- **Extents Allocated**: 1
- **EOF**: 12288
- **Index Levels**: 2

This slide completes the listing of the SHOWLABEL command.
This slide illustrates an incomplete listing of the SHOWDDL command for PARTLOC table. But note that the first LOCATION clause specifies the location of the underlying Guardian file for the first partition of the table and the second location clause specifies the location of the underlying file for the second partition of the table. Also note that the subvolume name will always be the same for all the underlying files of an SQL/MX partitioned table or index. The SQL/MX schema subvolume names always start with ZSD and for user schemas the fourth character in the schema subvolume name is a character and not a digit.

The node name and volume name in the LOCATION clause can be different; the file names will be different.
This slide and the next illustrate a FUP INFO <filename>, DETAIL display for the underlying primary partition file for the PARTLOC table. The display illustrates that this is an SQL ANSI TABLE and that the ANSI NAME is SAMDBCAT.INVENT.PARTLOC; this display also illustrates the file name for the resource fork, the location of the system metadata, SYSTEM_SCHEMA tables, and the SQL/MX version of this table. The lower portion of the display specifies which columns make up the primary-key and the physical files for all of the partitions of the PARTLOC table.
This slide continues with the FUP INFO <filename> DETAIL display for the PARTLOC table. The owner of the table is -1, which is the super ID. The security attribute is set to *SQL. Guardian security is not used for SQL/MX objects. The security for SQL/MX objects is controlled by the owner of the object, who can grant privileges (SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges) to access the object. The SELECT, UPDATE, and REFERENCES privileges can be specified at the column level.
SQL/MX Security Model

- SQL/MX use the ANSI/ISO GRANT and REVOKE security. There is no support for Guardian or Safeguard security.
- All objects in a schema are owned by the schema creator, even if the super.super user ID creates them.
- In SQL/MX the schema owner grants or revokes object privileges to Guardian users to access tables, views, and stored procedures in the schema.
  - Access privileges are SELECT, INSERT, UPDATE, UPDATE on a column or set of columns, DELETE, REFERENCES, and EXECUTE.
- The schema owner can grant and revoke grant privileges to other Guardian users.
- Security is enforced at OPEN time, but unlike SQL/MP, REVOKE blows away opens (so revalidation occurs after a REVOKE).

SQL/MX uses Grant and Revoke security to grant or revoke access privileges to tables, views, and stored procedures to Guardian users. SQL/MX does not use Guardian or Safeguard security like SQL/MP.

The schema creator owns all objects in a schema and can create and drop objects in the schema. The super.super user ID can create objects in a schema; however, those objects are owned by the schema creator. The super.super user ID can also drop objects in a schema and the schema.

Security is enforced when the object is opened but unlike SQL/MP, when a privilege is revoked the open is blown away. Therefore, revalidation occurs after a revoke. In SQL/MP if you revoke security on an object, the programs that have that object open can continue to access it until they close it. But in SQL/MX if you revoke the security, SQL/MX will blow away the open and you need to re-open and get revalidated.
This slide illustrates the process architecture.

Application program — Has the application code and runs in a process that contains the SQL/MX executor and file system.

Module file — Is a binary file that contains the statically compiled query plans and is read in by the executor. If the executor needs to automatically recompile the query, it spawns an mxcmp process to recompile the query.

Executor — The executor reads and executes the query plans. It sends a request to DP2 to read and write data to tables and indexes.

Executor in DP2 — There is a copy of the executor code in DP2 so that certain relational operations (sorting, grouping, and aggregation) can be performed into DP2.

Executor server processes — SQL/MX also supports executor server processes, mxesp, which internally look like an application process. It has a main process loop, a copy of the executor and the file system code and it can send request to DP2 processes. The executor server processes are used to execute query plans in parallel.
### SQL/MX Components

<table>
<thead>
<tr>
<th>Releases 1.x:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler.</td>
<td></td>
</tr>
<tr>
<td>Executor.</td>
<td></td>
</tr>
<tr>
<td>FS.</td>
<td></td>
</tr>
<tr>
<td>DP2.</td>
<td></td>
</tr>
<tr>
<td>ODBC/JDBC.</td>
<td></td>
</tr>
<tr>
<td>NSM/web.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Release 2.x:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog Manager.</td>
<td></td>
</tr>
<tr>
<td>Utilities.</td>
<td></td>
</tr>
</tbody>
</table>

Release 2.x introduces the catalog manager and SQL/MX utilities.
Catalog Manager

- Implements DDL semantics.
  - Utilities use a DDL-like interface, CatAPIRequest, to get DDL-like services.
- Packaged in the SQL/MX compiler (mxcmp process) and invoked by the Executor at DDL execution time.
- Under normal conditions the catalog manager is the only application that writes to the metadata tables.
- Performs some specialized data management.
  - Populates indexes (optimized, for example, side tree insert).
  - Enforces constraints when created.

The catalog manager implements the DDL semantic. The catalog manager creates file labels, the data fork, the resource forks, and updates the metadata tables. The catalog manager code is packaged in the SQL compiler, mxcmp. When you execute a DDL statement, the executor spawns an mxcmp process and sends a request to the catalog manager code to execute DDL commands.

Under normal conditions the catalog manager code is the only application code that ever writes to metadata tables. SQL/MX utilities sometimes do DDL-like operations and use an internal interface called CATAPI request to get those services. There are a few utilities like MXTOOL and GOAWAY that are back doors that can write to metadata tables. These utilities can only be executed by the SUPER.SUPER user.

The catalog manager does some specialized data management, for example:

- When you create an index and you do not specify the no populate option, the index is created and populated in a single transaction. SQL/MX uses an optimized insert select to do this. It uses side-tree insert, a technique that SQL/MX can use on an unaudited partition. It build the indexes’ B-tree from front to back, which is a more efficient way to construct a B-tree, and then inserts the B-tree into the underlying key-sequenced file.

- The catalog manager also enforces constraints. When you create a constraint, the catalog manager code checks that the constraint is satisfied before it writes it to the metadata.
SQL/MX DDL Operations

- The DDL statement is compiled into a query tree.
- The query tree represents DDL operation.
- At run time, the executor spawns an MXCMP process to execute the DDL operation.
- Catalog manager code manages files, file labels, and resource forks and updates metadata tables.
- Completion code and diagnostics are returned via the query tree.

This slide and the next illustrate how DDL operations are performed in SQL/MX. The DDL statement is compiled into a very simple query tree consisting of a root node and a DDL node. Then, from an application perspective, it calls the CLI to execute a DDL statement in the same way it does a DML statement. When executed, the DDL node is used to spawn an mxcmp process, which functions as the catalog manager. SQL/MX uses the same mxcmp process that is used for automatic SQL/MX compilation, which also contains a copy of the catalog manager code. The DDL node in the query tree sends the text for the DDL statement in a message to that mxcmp process, which passes it to the catalog manager code. The catalog manager manages all of the files, file labels, and resource forks, and updates the metadata tables. The completion code and diagnostics are propagated back up the query tree back to the application.
This slide illustrates a partial list of the SQL/MX utilities.
Utilities

- Utilities can be accessed in three ways:
  - Accessed via syntax, that is, mxci commands.
  - Accessed via OSS prompt, for example, IMPORT and MXTOOLS.
  - Accessed graphically through NSM/web.

- DDL operations and utilities are different in SQL/MP.
  - DDL operations in SQL/MX are done in a single transaction.
  - SQL/MX implements utilities as the analogues of SQL/MP multi-transaction DDL operations.
  - For example: Partition management (ALTER TABLE) is a DDL command in SQL/MP and a utility (MODIFY) in SQL/MX.
Syntax-Directed Utility Architecture

- The utility statement is compiled into a query tree; the utility is modeled as an internal stored procedure.
- At run time, an MXCMP executes the operation.
- Additional MXCMP can be spawned for DDL-like suboperations.

This slide illustrates how syntax directed utilities are executed. The utilities statements are compiled into a very simple query tree that consists of a root node and a leaf node called an SP node. The SP node is an internal stored procedure. The input data consists of the utility command text, which is sent to a mxcmp process. The mxcmp process also contains the utility code to perform the utility operation. If the utility operation has to perform a DDL operation, it spans another mxcmp process and sends the DDL commands through the CATAPI request interface. Then the catalog manager code in that mxcmp process performs the DDL operations.
This slide illustrates the NSMweb architecture that enables you to use a Web browser running on your workstation to access SQL/MX, MXCS, and other subsystems.

The Web browser gets Web pages through http from the Web Server that is running on the HP NonStop server. If you want to run a utility, a request is sent to the ServerFactory through CORBA, which spawns different kinds of processes depending on what kind of operation is required.

- **NSMweb NSKserver** provides access to Enscribe operations.
- **NSMweb MXCServer** provides a management and configuration interface to ODBC/MX.
- **NSMweb SQLMXServer** provides a management interface to SQL/MX. Note that the SQLMXServer has a copy of the DDOL layer bound into it; the same code is in the mxcmp program. You can use NSMweb to create a table; for example, it calls a routine in the DDOL layer that generates the create table request and sends it to a mxcmp process. The mxcmp process has a copy of the catalog manager code which executes the DDL request.
Module Objectives

- Describe referential integrity (RI).
- Implement referential integrity as a column or table constraint.

References

HP NonStop SQL/MX Installation and Management Guide.
This slide illustrates referential integrity (RI) between the EMPLOYEE and DEPT tables. Referential integrity is a mechanism that checks that a column or set of columns in one table (the referencing table) contains values that match those of a column or set of columns in another table (the referenced table).

In this illustration the DEPTNUM column in the DEPT table is defined as a primary-key constraint; however, the reference could reference a column or set of columns defined in a unique constraint. The EMPLOYEE.DEPTNUM column is a foreign key and its values are referencing the values in the DEPT.DEPTNUM column. To maintain this referential integrity, the values in the EMPLOYEE.DEPTNUM column must have a matching value in the DEPT.DEPTNUM column.
Referential Integrity Terminology

- Referencing table — The table with the RI constraint.
- Referenced table — The table referenced by the RI constraint.
- Referential integrity constraint:
  - Specifies a foreign key for the referencing table.
  - References a column or set of columns with a UNIQUE or PRIMARY KEY constraint in the referenced table.
  - Belongs to the referencing table.
- Foreign key:
  - The column or set of columns in the referencing table whose value must match the column or set of columns with a UNIQUE or PRIMARY KEY constraint in the referenced table.
  - Not required to have unique values.
  - An index is automatically created if one does not exist in advance.
REFERENCES Column Constraint — Syntax

REFERENCES referenced-table [(column-list)] [referential triggered action]
  referential triggered action is:
  [ ON UPDATE RESTRICT | NO ACTION ]
  [ ON DELETE  RESTRICT | NO ACTION ]

Example:
CREATE TABLE EMPLOYEE
{
  ..., 
  'DEPTNUM' INT NOT NULL NOT DROPPABLE
  REFERENCES DEPT(DEPTNUM) ON UPDATE RESTRICT
  ON DELETE RESTRICT 
  ..., 
};

This slide illustrates the syntax for the REFERENCES clause column constraint.

- REFERENCES referenced-table [(column-list)] [referential triggered action] — Specifies a references column constraint. The maximum combined length of the columns for a REFERENCES constraint is 255 bytes.

- referenced-table — Is the table referenced by the foreign key in the referencing table’s referential constraint. The referenced-table cannot be a view and the referenced-table cannot be the same as the referencing table.

- column-list — Specifies the column or set of columns in the referenced-table that correspond to the foreign key in the referencing table. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column-list associated with FOREIGN KEY. If the column-list is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

- ON UPDATE [RESTRICT | NO ACTION] — Specifies what referential action is taken when the column-list in the referenced-table is updated. If no ON UPDATE clause is specified, the default — ON UPDATE NO ACTION — is assumed.

- ON DELETE [RESTRICT | NO ACTION] — Specifies what referential action is taken when a row in the referenced-table is deleted. If no ON DELETE clause is specified, the default — ON DELETE NO ACTION — is assumed.
  - RESTRICT referential action means that the referential check is made for each row. An error is raised when the referential constraint is violated.
  - ANSI SQL-99 standard: NO ACTION referential action means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated. HP NonStop SQL/MX does not support the NO ACTION referential action as specified by the ANSI SQL-99 standard. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT.
SYSTEM_DEFAULTS Table — Referential Action

- The REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT system default determines how SQL/MX handles the NO ACTION referential action.

```
REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT [ 'OFF' | 'SYSTEM' | 'ON' ]
```

- OFF — SQL issues error 1301.
- SYSTEM — SQL issues error 1302, which means NO ACTION behaves like RESTRICT. This is the default.
- ON — Means that NO ACTION behaves like RESTRICT without issuing errors or warnings.

- From mxci issue the SHOWCONTROL * statement to display the current system default settings for the system.
This slide illustrates an example of an RI constraint on the EMPLOYEE.DEPTNUM column, which references the DEPT.DEPTNUM column. The NO ACTION referential action for ON UPDATE and ON DELETE is set to the system default 'SYSTEM'. Therefore, the warning 1302 is displayed to inform you how the NO ACTION referential action will be handled by the system.
This slide illustrates the SHOWDDL for the EMPLOYEE table that was created with the REFERENCE column constraint on the DEPTNUM column. SQL/MX created an implicit index EMPLOYEE_216214956_2787 to enforce the constraint because an existing explicit did not exist to enforce the RI constraint. Then the referential integrity constraint was added with the ALTER TABLE ... ADD CONSTRAINT statement.
This slide illustrates another example of an RI constraint on the EMPLOYEE.DEPTNUM column which references the DEPT.DEPTNUM column. The referential action for ON UPDATE and ON DELETE is set to RESTRICT.
This slide illustrates another example of an RI constraint on the EMPLOYEE.DEPTNUM column that references the DEPT table. Note that a column-list was not specified in the REFERENCES clause, only the name of the referenced table, DEPT. There is not an explicit reference to a column or set of columns that make up a primary key or unique constraint in the DEPT table. Therefore, the REFERENCE clause uses the primary key of the referenced table by default.
FOREIGN KEY Column or Table Constraint — Syntax

FOREIGN KEY (column-list) REFERENCES referenced-table [(column-list)]
[referential triggered action]

referential triggered action is:
[ ON UPDATE RESTRICT | NO ACTION ]
[ ON DELETE RESTRICT | NO ACTION ]

FOREIGN KEY (column-list) — Specifies a referential constraint for the referencing table. The column or set of columns (column-list) in the referencing table can contain only values that match those in a column or set of columns (column-list) in the referenced table specified in the REFERENCES clause. The columns in the two column-list must have the same characteristics (data type, length, scale, precision), and there must be a UNIQUE or PRIMARY KEY constraint on the column or set of columns specified in the REFERENCES clause (referenced table). Without the FOREIGN KEY clause, the foreign key in the referencing table is the column being defined; with the FOREIGN KEY clause, the foreign key is the column or set of columns specified in the FOREIGN KEY clause.

Referenced-table (column-list) — Specifies the column or set of columns in the referenced-table that corresponds to the foreign key in the referencing table. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If the column-list is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

[ON UPDATE RESTRICT | NO ACTION] — Specifies what referential action is taken when the column-list in the referenced-table is updated. If no ON UPDATE clause is specified, a default of ON UPDATE NO ACTION is assumed.

[NO DELETE RESTRICT | NO ACTION] — Specifies what referential action is taken when a row in the referenced-table is deleted. If no ON DELETE clause is specified, a default of ON DELETE NO ACTION is assumed.

- RESTRICT referential action — Means that the referential check is made for each row. An error is raised when the referential constraint is violated.

- NO ACTION — ANSI SQL-99 standard: NO ACTION referential action means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated. SQL/MX does not support NO ACTION referential action in the way it is specified by ANSI SQL-99. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT.
This slide illustrates creating a foreign key table constraint with RESTRICT ACTION on the EMPLOYEE.DEPTNUM column that references the DEPT.DEPTNUM column.
Foreign Key Constraint — Example 2

CREATE TABLE CAT1.SCH1.EMPLOYEE
   (EMPNUM NUMERIC(4, 0) UNSIGNED NO DEFAULT
    NOT NULL NOT DROPPABLE,
    ..., DEPTNUM NUMERIC(4, 0) UNSIGNED NO DEFAULT
    NOT NULL NOT DROPPABLE,
    FOREIGN KEY (DEPTNUM) REFERENCES DEPT(DEPTNUM)
    , CONSTRAINT CAT1.SCH1.EMPLOYEE_PK PRIMARY KEY (EMPNUM ASC)
    NOT DROPPABLE)
LOCATION $CLASS1
STORE BY (EMPNUM ASC);

*** WARNING[1302] NO ACTION referential action for ON UPDATE clause behaves like
RESTRICT referential action. To alter the behaviour, set the appropriate value
for the REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT default.

*** WARNING[1302] NO ACTION referential action for ON DELETE clause behaves like
RESTRICT referential action. To alter the behaviour, set the appropriate value
for the REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT default.

This slide illustrates creating a foreign key table constraint with NO ACTION on the
EMPLOYEE.DEPTNUM column that references the DEPT.DEPTNUM column.
This slide and the next illustrate the SHOWDDL statement for the foreign key constraint defined for the EMPLOYEE.DEPTNUM column.
This slide illustrates that an implicit index was created for the foreign key constraint on the EMPLOYEE.DEPTNUM column to enforce the constraint. Then, the foreign key constraint is added to the EMPLOYEE table using the ALTER TABLE .. ADD CONSTRAINT statement.
Referential Integrity Checks for Inserts

- Checks carried out during insert, update, and delete operations on the referenced and referencing tables of a referential integrity constraint. The tables mentioned are from the previous examples.
- Inserting a new row into the EMPLOYEE table, the referencing table.

```
INSERT INTO EMPLOYEE(empnum, _,deptnum, ...) VALUES (2600, _, 6000, ...);
```

- A referential check is done to make sure that value 6000 exists in the DEPTNUM column of the DEPT table.
- Inserting a new row into the DEPT table, the referenced table.

```
INSERT INTO DEPT(DEPTNUM, ...) VALUES (6001, ...);
```

- No referential check is needed.

When you specify a referential integrity constraint on a table and later do inserts or updates on the referencing table or deletes on the referenced table, SQL/MX performs referential integrity checks to make sure the referential integrity constraint is not violated.

What happens when you insert a new row into the EMPLOYEE table?

- Again the EMPLOYEE table is referencing the DEPT table. When you insert a new row into the EMPLOYEE table (the referencing table), you want to make sure that the foreign key value you are inserting is referencing a valid primary-key value in the DEPT table, the referenced table. In this example, SQL/MX performs a referential integrity check to make sure that the foreign key value, 6001, actually exists in the DEPTNUM column of the DEPT table. If the foreign key value does not exist, SQL/MX generates a constraint violation.

- What happens when you insert a new row into the DEPT table? When you insert a new row into the DEPT table, no checks are required because the DEPT table does not reference the EMPLOYEE table. Therefore, you can insert any valid values into the DEPT table.
Referential Integrity Checks for Deletes

- Deleting a row from the EMPLOYEE table, the referencing table.
  ```sql
  DELETE FROM EMPLOYEE WHERE EMPNUM = 2600;
  ```
  - No referential check is needed.

- Deleting a row from the DEPT table, the referenced table.
  ```sql
  DELETE FROM DEPT WHERE DEPTNUM = 6001;
  ```
  - A referential check is done to make sure that value 6001 is not referenced by any rows from the EMPLOYEE table.

What happens when you delete a row from the EMPLOYEE table (the referencing table)?

- When you delete a row from the EMPLOYEE table, no referential integrity check is required because you are deleting a referencing value and not the referenced value.

What happens when you delete a row from the DEPT table?

- When you delete a row from the DEPT table, you are removing a primary-key value or a unique value from the table. There might be a foreign key value in the EMPLOYEE table referencing that primary key or unique value in the DEPT table. Note that the foreign key value does not have to be unique. Therefore, there might be multiple rows in the EMPLOYEE table with the same foreign key value referencing the row that is being deleted from the DEPT table. To speed up this check, when you created the referential integrity constraint for the DEPTNUM column in the EMPLOYEE table, SQL/MX created an index on that column. SQL/MX uses this index to determine if the value to be deleted exists in the EMPLOYEE table. If so, SQL/MX restricts the delete operation to prevent violation of the referential integrity constraint. In this example, the referential integrity check is performed to make sure that the primary-key value, 6001, is not referenced by any row in the EMPLOYEE table.
Referential Checks for Updates

• Updating a row in the EMPLOYEE table, the referencing table.

```
UPDATE EMPLOYEE SET DEPTNUM = 6010
WHERE EMPNUM = 2600;
```

- The old value is 6000 and the new value is 6010.
- A referential check is done to verify that the new value of 6010 exists as DEPTNUM in the DEPT table.

• Updating a row in the DEPT table, the referenced table.

- Update the DEPT table, set DEPTNUM = 6010.
- The old value is 6000 and the new value is 6010.
- A referential check is done to verify if DEPTNUM in the EMPLOYEE table is not referencing the old value of 6000.

What happens if you update an employee record?

- If you are updating the DEPTNUM value in an employee record, then SQL/MX must perform a referential integrity check to verify that the new DEPTNUM value exists in the DEPT table. If the value does not, then the update violates the referential integrity constraint and SQL/MX prevents (restricts) the update. If you are not updating the DEPTNUM value in the employee record, no referential integrity check is required.

What happens if you update a department record in the DEPT table?

- If you are not updating the DEPTNUM value in the department record, no referential integrity check is required.

- If you are updating the DEPTNUM COLUMN, which is defined as the primary key, the update fails because you cannot update a primary-key value. To update the primary key of a record, you must delete the record and then re-insert a new record with the new primary-key value. In this case, SQL/MX would perform the same referential integrity check as for a delete.

- Another scenario is if a referential constraint is referencing a column defined as a unique constraint and not as a primary-key constraint; then you could possibly update the referenced value, if the new value is unique. In this case, you are basically deleting the old unique value and replacing it with a new unique value. SQL/MX would perform the same referential integrity check as for a delete.
Module Objectives

- Describe referential integrity (RI).
- Implement referential integrity as a column or table constraint.

References

HP NonStop SQL/MX Installation and Management Guide.
Referential Integrity (RI)

This slide illustrates referential integrity (RI) between the EMPLOYEE and DEPT tables. Referential integrity is a mechanism that checks that a column or set of columns in one table (the referencing table) contains values that match those of a column or set of columns in another table (the referenced table).

In this illustration the DEPTNUM column in the DEPT table is defined as a primary-key constraint; however, the reference could reference a column or set of columns defined in a unique constraint. The EMPLOYEE.DEPTNUM column is a foreign key and its values are referencing the values in the DEPT.DEPTNUM column. To maintain this referential integrity, the values in the EMPLOYEE.DEPTNUM column must have a matching value in the DEPT.DEPTNUM column.

<table>
<thead>
<tr>
<th>EMPLOYEE</th>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMPNUM</td>
<td>DEPTNUM</td>
</tr>
<tr>
<td>1</td>
<td>9000</td>
</tr>
<tr>
<td>23</td>
<td>2000</td>
</tr>
<tr>
<td>87</td>
<td>3000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT</th>
<th>DEPTNAME</th>
<th>RPTDEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Referential Integrity Terminology

- Referencing table — The table with the RI constraint.
- Referenced table — The table referenced by the RI constraint.
- Referential integrity constraint:
  - Specifies a foreign key for the referencing table.
  - References a column or set of columns with a UNIQUE or PRIMARY KEY constraint in the referenced table.
  - Belongs to the referencing table.
- Foreign key:
  - The column or set of columns in the referencing table whose value must match the column or set of columns with a UNIQUE or PRIMARY KEY constraint in the referenced table.
  - Not required to have unique values.
  - An index is automatically created if one does not exist in advance.
REFERENCES Column Constraint — Syntax

REFERENCES referenced-table [(column-list)] [referential triggered action]
  referential triggered action is:
    [ ON UPDATE RESTRICT | NO ACTION ]
    [ ON DELETE  RESTRICT | NO ACTION ]

Example:
CREATE TABLE EMPLOYEE
{
  ...‘
  DEPTNUM INT NOT NULL NOT DROPPABLE
  REFERENCES DEPT(DEPTNUM) ON UPDATE RESTRICT
  ON DELETE RESTRICT
  , ...
};

This slide illustrates the syntax for the REFERENCES clause column constraint.

- REFERENCES referenced-table [(column-list)] [referential triggered action] — Specifies a references column constraint. The maximum combined length of the columns for a REFERENCES constraint is 255 bytes.

- referenced-table — Is the table referenced by the foreign key in the referencing table’s referential constraint. The referenced-table cannot be a view and the referenced-table cannot be the same as the referencing table.

- column-list — Specifies the column or set of columns in the referenced-table that correspond to the foreign key in the referencing table. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column-list associated with FOREIGN KEY. If the column-list is omitted, the referenced table’s PRIMARY KEY columns are the referenced columns.

- ON UPDATE [RESTRICT | NO ACTION] — Specifies what referential action is taken when the column-list in the referenced-table is updated. If no ON UPDATE clause is specified, the default — ON UPDATE NO ACTION — is assumed.

- ON DELETE [RESTRICT | NO ACTION] — Specifies what referential action is taken when a row in the referenced-table is deleted. If no ON DELETE clause is specified, the default — ON DELETE NO ACTION — is assumed.
  - RESTRICT referential action means that the referential check is made for each row. An error is raised when the referential constraint is violated.
  - ANSI SQL-99 standard: NO ACTION referential action means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated. HP NonStop SQL/MX does not support the NO ACTION referential action as specified by the ANSI SQL-99 standard. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT.
### SYSTEM_DEFAULTS Table — Referential Action

- The REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT system default determines how SQL/MX handles the NO ACTION referential action.

  ```
  REF_CONSTRAINT_NO_ACTION_LIKE_RESTRICT [ 'OFF' | 'SYSTEM' | 'ON' ]
  ```

- OFF — SQL issues error 1301.
- SYSTEM — SQL issues error 1302, which means NO ACTION behaves like RESTRICT. This is the default.
- ON — Means that NO ACTION behaves like RESTRICT without issuing errors or warnings.

- From mxci issue the SHOWCONTROL * statement to display the current system default settings for the system.
This slide illustrates an example of an RI constraint on the EMPLOYEE.DEPTNUM column, which references the DEPT.DEPTNUM column. The NO ACTION referential action for ON UPDATE and ON DELETE is set to the system default 'SYSTEM'. Therefore, the warning 1302 is displayed to inform you how the NO ACTION referential action will be handled by the system.
This slide illustrates the SHOWDDL for the EMPLOYEE table that was created with the REFERENCE column constraint on the DEPTNUM column. SQL/MX created an implicit index EMPLOYEE_216214956_2787 to enforce the constraint because an existing explicit did not exist to enforce the RI constraint. Then the referential integrity constraint was added with the ALTER TABLE ... ADD CONSTRAINT statement.
This slide illustrates another example of an RI constraint on the EMPLOYEE.DEPTNUM column which references the DEPT.DEPTNUM column. The referential action for ON UPDATE and ON DELETE is set to RESTRICT.
This slide illustrates another example of an RI constraint on the EMPLOYEE.DEPTNUM column that references the DEPT table. Note that a column-list was not specified in the REFERENCES clause, only the name of the referenced table, DEPT. There is not an explicit reference to a column or set of columns that make up a primary key or unique constraint in the DEPT table. Therefore, the REFERENCE clause uses the primary key of the referenced table by default.
FOREIGN KEY (column-list) — Specifies a referential constraint for the referencing table. The column or set of columns (column-list) in the referencing table can contain only values that match those in a column or set of columns (column-list) in the referenced table specified in the REFERENCES clause. The columns in the two column-list must have the same characteristics (data type, length, scale, precision), and there must be a UNIQUE or PRIMARY KEY constraint on the column or set of columns specified in the REFERENCES clause (referenced table). Without the FOREIGN KEY clause, the foreign key in the referencing table is the column being defined; with the FOREIGN KEY clause, the foreign key is the column or set of columns specified in the FOREIGN KEY clause.

Referenced-table (column-list) — Specifies the column or set of columns in the referenced-table that corresponds to the foreign key in the referencing table. The columns in the column list associated with REFERENCES must be in the same order as the columns in the column list associated with FOREIGN KEY. If the column-list is omitted, the referenced table's PRIMARY KEY columns are the referenced columns.

[ON UPDATE RESTRICT | NO ACTION] — Specifies what referential action is taken when the column-list in the referenced-table is updated. If no ON UPDATE clause is specified, a default of ON UPDATE NO ACTION is assumed.

[NO DELETE RESTRICT | NO ACTION] — Specifies what referential action is taken when a row in the referenced-table is deleted. If no ON DELETE clause is specified, a default of ON DELETE NO ACTION is assumed.

- **RESTRICT** referential action — Means that the referential check is made for each row. An error is raised when the referential constraint is violated.
- **NO ACTION** — ANSI SQL-99 standard: NO ACTION referential action means that the referential check is made at the end of the SQL statement. An error is raised when the referential constraint is violated. SQL/MX does not support NO ACTION referential action in the way it is specified by ANSI SQL-99. However, you can change NO ACTION's behavior to be the same as RESTRICT by setting an appropriate value for the Control Query Default REFCONSTRAINT_NO_ACTION_LIKE_RESTRICT.
This slide illustrates creating a foreign key table constraint with RESTRICT ACTION on the EMPLOYEE.DEPTNUM column that references the DEPT.DEPTNUM column.
This slide illustrates creating a foreign key table constraint with NO ACTION on the EMPLOYEE.DEPTNUM column that references the DEPT.DEPTNUM column.
Foreign Key Constraint — SHOWDDL
(1 of 2)

```sql
>>SHOWDDL EMPLOYEE;
CREATE TABLE CAT1.SCH1.EMPLOYEE
(
    EMPNUM NUMERIC(4, 0) UNSIGNED NO DEFAULT
        -- NOT NULL NOT DROPPABLE
    , ...
    , DEPTNUM NUMERIC(4, 0) UNSIGNED NO DEFAULT
        -- NOT NULL NOT DROPPABLE
    , ...
    , CONSTRAINT CAT1.SCH1.EMPLOYEE_PK PRIMARY KEY (EMPNUM ASC)
        NOT DROPPABLE
    , CONSTRAINT CAT1.SCH1.EMPLOYEE_923548971_9787 CHECK
        (CAT1.SCH1.EMPLOYEE.EMPNUM IS NOT NULL AND
        CAT1.SCH1.EMPLOYEE.FIRST_NAME IS NOT NULL AND
        CAT1.SCH1.EMPLOYEE.LAST_NAME IS NOT NULL AND
        CAT1.SCH1.EMPLOYEE.DEPTNUM IS NOT NULL) NOT DROPPABLE
)
    LOCATION 'SUNNY.$CLASS1.ZSDP643D.CQ6GBT00'
    NAME 'SUNNY_CLASS1_ZSDP643D_CQ6GBT00'
    STORE BY (EMPNUM ASC)
    ;
```

This slide and the next illustrate the SHOWDDL statement for the foreign key constraint defined for the EMPLOYEE.DEPTNUM column.
This slide illustrates that an implicit index was created for the foreign key constraint on the EMPLOYEE.DEPTNUM column to enforce the constraint. Then, the foreign key constraint is added to the EMPLOYEE table using the ALTER TABLE .. ADD CONSTRAINT statement.
Referential Integrity Checks for Inserts

- Checks carried out during insert, update, and delete operations on the referenced and referencing tables of a referential integrity constraint. The tables mentioned are from the previous examples.
- Inserting a new row into the EMPLOYEE table, the referencing table.
  ```sql
  INSERT INTO EMPLOYEE(empnum, deptnum, ...)
  VALUES (2600, ..., 6000, ...);
  ```
  * A referential check is done to make sure that value 6000 exists in the DEPTNUM column of the DEPT table.
- Inserting a new row into the DEPT table, the referenced table.
  ```sql
  INSERT INTO DEPT(DEPTNUM, ...)
  VALUES (6001, ...);
  ```
  * No referential check is needed.

When you specify a referential integrity constraint on a table and later do inserts or updates on the referencing table or deletes on the referenced table, SQL/MX performs referential integrity checks to make sure the referential integrity constraint is not violated.

What happens when you insert a new row into the EMPLOYEE table?

- Again the EMPLOYEE table is referencing the DEPT table. When you insert a new row into the EMPLOYEE table (the referencing table), you want to make sure that the foreign key value you are inserting is referencing a valid primary-key value in the DEPT table, the referenced table. In this example, SQL/MX performs a referential integrity check to make sure that the foreign key value, 6001, actually exists in the DEPTNUM column of the DEPT table. If the foreign key value does not exist, SQL/MX generates a constraint violation.
- What happens when you insert a new row into the DEPT table? When you insert a new row into the DEPT table, no checks are required because the DEPT table does not reference the EMPLOYEE table. Therefore, you can insert any valid values into the DEPT table.
Referential Integrity Checks for Deletes

- Deleting a row from the EMPLOYEE table, the referencing table.
  
  ```sql
  DELETE FROM EMPLOYEE WHERE EMPNUM = 2600;
  ```
  
  - No referential check is needed.

- Deleting a row from the DEPT table, the referenced table.
  
  ```sql
  DELETE FROM DEPT WHERE DEPTNUM = 6001;
  ```
  
  - A referential check is done to make sure that value 6001 is not referenced by any rows from the EMPLOYEE table.

What happens when you delete a row from the EMPLOYEE table (the referencing table)?

- When you delete a row from the EMPLOYEE table, no referential integrity check is required because you are deleting a referencing value and not the referenced value.

What happens when you delete a row from the DEPT table?

- When you delete a row from the DEPT table, you are removing a primary-key value or a unique value from the table. There might be a foreign key value in the EMPLOYEE table referencing that primary key or unique value in the DEPT table. Note that the foreign key value does not have to be unique. Therefore, there might be multiple rows in the EMPLOYEE table with the same foreign key value referencing the row that is being deleted from the DEPT table. To speed up this check, when you created the referential integrity constraint for the DEPTNUM column in the EMPLOYEE table, SQL/MX created an index on that column. SQL/MX uses this index to determine if the value to be deleted exists in the EMPLOYEE table. If so, SQL/MX restricts the delete operation to prevent violation of the referential integrity constraint. In this example, the referential integrity check is performed to make sure that the primary-key value, 6001, is not referenced by any row in the EMPLOYEE table.
**Referential Checks for Updates**

- Updating a row in the EMPLOYEE table, the referencing table.
  
  ```sql
  UPDATE EMPLOYEE SET DEPTNUM = 6010
  WHERE EMPNUM = 2600;
  ```
  
  - The old value is 6000 and the new value is 6010.
  - A referential check is done to verify that the new value of 6010 exists as DEPTNUM in the DEPT table.

- Updating a row in the DEPT table, the referenced table.
  
  - Update the DEPT table, set DEPTNUM = 6010.
  - The old value is 6000 and the new value is 6010.
  - A referential check is done to verify if DEPTNUM in the EMPLOYEE table is not referencing the old value of 6000.

What happens if you update an employee record?

- If you are updating the DEPTNUM value in an employee record, then SQL/MX must perform a referential integrity check to verify that the new DEPTNUM value exists in the DEPT table. If the value does not, then the update violates the referential integrity constraint and SQL/MX prevents (restricts) the update. If you are not updating the DEPTNUM value in the employee record, no referential integrity check is required.

What happens if you update a department record in the DEPT table?

- If you are not updating the DEPTNUM value in the department record, no referential integrity check is required.

- If you are updating the DEPTNUM COLUMN, which is defined as the primary key, the update fails because you cannot update a primary-key value. To update the primary key of a record, you must delete the record and then re-insert a new record with the new primary-key value. In this case, SQL/MX would perform the same referential integrity check as for a delete.

- Another scenario is if a referential constraint is referencing a column defined as a unique constraint and not as a primary-key constraint; then you could possibly update the referenced value, if the new value is unique. In this case, you are basically deleting the old unique value and replacing it with a new unique value. SQL/MX would perform the same referential integrity check as for a delete.
Creating SQL/MX Partitioned Tables and Indexes

Module 3

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice.
Module Objectives

- Describe the range and hash partitioning concepts.
- Describe the advantages and disadvantages of range and hash partitioning.
- Create range and hash partitioned tables.
- Describe partially decoupled keys (partitioning and clustering keys) support in HP NonStop SQL/MX.
- Describe and create co-partitioned tables.
- Describe and create co-located partitions.

References

HP NonStop SQL/MX Installation and Management Guide.
Range Partitioning

- Table is divided into horizontal partitions with each partition representing a range of rows.
- The range for each partition is set by specifying a FIRST KEY value for the partitioning key columns of the table.
- Range partitioning uses the partitioning column values to map a row to a table partition.
- Each partition is supported by a underlying Guardian physical file.

This slide illustrates range partitioning of an SQL/MX table. The table is divided into horizontal partitions, each partition represents a range of rows. The range for a partition is set by specifying a FIRST KEY value for the partitioning key columns of the table. The FIRST KEY value determines the lower boundary of a partition and the FIRST KEY value of the next partition specifies the upper boundary of the previous partition and the lower boundary for the next partition. Each partition is supported by a underlying Guardian physical file.
Hash Partitioning

- Uses a hash function and distribution function to evenly distribute the rows of a table over a number of partitions.
- The function used to assign rows to partitions is composed of two steps:
  - The partitioning key values are hashed by the hash function.
  - The result of the hash function is passed to the distribution function which maps the hashed value to a partition number.

This slide illustrates hash partitioning of an SQL/MX table. The partitioning function hashes the partitioning key of each row and then passes the row to the distribution function, which maps the row to a particular partition of the table. The distribution function provides the property of minimal record movement when partitions are added or dropped from the table. Another property of the distribution function is that the history of the insertions does not influence the distribution of the rows.
Range and Hash Partitioning Comparison

- Range partitioning:
  - Use range partitioning when your data has logical ranges and boundaries into which it can be subdivided and distributed.
  - Optimal when the data is evenly distributed across the ranges.
- Hash partitioning:
  - Use hash partitioning if your data is not easily distributed among ranges or to evenly distribute data across a specified number of partitions.
  - Use hash partitioning in a 'decoupled' fashion to cluster data on a key other than the partitioning key. Useful when you want to co-locate hash partitions of related tables and indexes.

This slide illustrates the comparison between range and hash partitioning.
Range Partitioning — Advantages and Disadvantages

• Advantages:
  – Data is located in a specific partition and is not randomly spread across the database.
  – With partition management, you can add, drop, and split partitions, plus move the partition boundaries of the partitions.
  – Superior performance for sequential access, point queries on the partitioning attributes, and range queries on the partitioning attributes.

• Disadvantage:
  – The range of each partition must be defined by the user.
  – Keeping the partition data balanced and equal in size might require considerable time and effort.
  – The size of the partitions can differ substantially because of hot spots, which might cause suboptimal performance for some queries.

This slide illustrates the advantages and disadvantages of range partitioning.
Quick NonStop SQL/MX Database Management

Creating SQL/MX Partitioned Tables and Indexes

Hash Partitioning — Advantages

Advantages:
- Automatically provides balanced and even distribution of data across the available disks.
- Record placement is not dependent on the history of insertions or partition management (ADD or DROP).
- Minimal record movement when adding new partitions.
- Suited for queries involving exact matches on key values.

This slide illustrates the advantages of hash partitioning.
Hash Partitioning — Disadvantages

Disadvantages:
• More difficult to remove specific data from a hash-partitioned table because the data is randomly distributed across the entire database.
• Range queries are inefficient when compared to range-partitioned tables.
• Partitioning management operations can be expensive for very large hash-partitioned tables. The data in the table has to be redistributed evenly across all partitions of the table.
• You can only add or drop partitions.

This slide illustrates the disadvantages of hash partitioning.
CREATE TABLE — Syntax

CREATE TABLE <table-name>
  (  <column-definitions>
      <constraint definitions>
  )
[STORE BY [PRIMARY KEY | (<key-column-list>) ] ]
LOCATION [node.$]volume.[subvolume.file-name]
[NAME partition-name]
partn-file-option
ATTRIBUTE[S] attribute [, attribute] …
  attributes
    ALLOCATE num-extents
    | {AUDITCOMPRESS | NO AUDITCOMPRESS }
    | BLOCKSIZE number-bytes
    | {CLEARONPURGE | NO CLEARONPURGE}
    | EXTENT ext-size | (pri-ext-size [, sec-ext-size] )
    | MAXEXTENTS num-extents

This slide illustrates the syntax for defining the primary partition of a table. The partn-file-option syntax is on the next slide.
CREATE TABLE — Range Partitioning Syntax

```sql
[RANGE] PARTITION
[BY (partitioning-column [, partitioning-column] ...) ]
[[ADD range-partn-defn [, ADD range-partn-defn] ...]]

ADD range-partn-defn
  FIRST KEY {col-value | (col-value [, col-value] ...)} \
  LOCATION [node.$volume[.subvolume.file-name] 
  NAME partition-name [attribute [, attribute] ...]
  attributes:
    [EXTENT ext-size | (pri-ext-size [, sec-ext-size] ) ]
    [MAXEXTENTS num-extents]
```

This slide illustrates the syntax for range partitioning a table.

- **RANGE** — Specifies the table is range partitioned, which is the default partitioning scheme.
- **[BY (partitioning-column [, partitioning-column] ...)]** — Specifies the partitioning columns. The default is the default partitioning key created by the STORE BY clause, the clustering key. Partitioning character columns must derive from the ISO88591 character set. Partitioning columns may not be floating-point data columns.
- **ADD FIRST KEY** — Defines a single secondary partition and includes FIRST KEY and partition options.
- **FIRST KEY {col-value | (col-value [,col-value]...)}** — Specifies the beginning of the range for a range-partitioned table. The FIRST KEY clause specifies the lowest values in the partition for columns stored in ascending order and the highest values in the partition for columns stored in descending order. These column values are referred to as the partitioning key. col-value is a literal that specifies the first value allowed in the associated partition for that column of the partitioning key. If there are more storage key columns than col-value items, the first key value for each remaining key column is the lowest or highest value for the data type of the column (the lowest value for an ascending column and the highest value for a descending column). col-value must contain characters only from the ISO88591 character set. If the table has a system-generated SYSKEY, its column list may not consist only of column SYSKEY. The SYSKEY must be the last column of the column list, and you may not specify a FIRST KEY value for the SYSKEY column. This does not apply to a user-created SYSKEY column.
LOCATION [\node.]$volume[.subvolume.file-name] — Specifies a volume and optionally the node, subvolume, and file name for the secondary partition of the table.

- node — Is the name of a node on the Expand network. For Guardian files representing a table or index partition or a view label, the node can be any node from which the object’s catalog is visible.
- volume — Is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian system named in the =_DEFAULTS define if none is specified). If you do not specify a LOCATION clause, SQL/MX uses the default volume named in the =_DEFAULTS define. You can locate more than one partition of a table on a single disk volume.
- subvolume — Is the name of the schema subvolume for the schema in which the table is being created. Follow these guidelines when using SQL/MX subvolume names: The name must begin with the letters ZSD, followed by a letter, not a digit (for example, ZSDa, not ZSD2). The name must be exactly eight characters long.
- file-name — Is a Guardian file name. file-name names must be eight characters in length and must end with the digits “00” (zero zero.) Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, you will receive an error.

NAME partition-name — Is an SQL identifier for a partition and must be unique. If you omit the NAME clause, SQL/MX generates a partition name.
Creating a Range Partitioned Table

```
CREATE TABLE QDPCAT.QDPDB.ORDERS
(   O_ORDERKEY INT NOT NULL NOT DROPPABLE
    ,O_CUSTKEY INT NOT NULL NOT DROPPABLE
    ...
    CONSTRAINT PK_ORDERS PRIMARY KEY (O_ORDERKEY)  NOT NULL
        NOT DROPPABLE
)
STORE BY PRIMARY KEY
LOCATION \SUNNY.$CLASS1
ATTRIBUTE
    EXTENT (126, 126) , MAXEXTENTS 210
RANGE PARTITION -- Range partitioned by default
(   ADD FIRST KEY (149989)
        LOCATION \SUNNY.DSMSCM
        EXTENT (126, 126) , MAXEXTENTS 210
        , ADD FIRST KEY (300001)
        LOCATION \SUNNY.OSS1
        EXTENT (126, 126) , MAXEXTENTS 210
        ...
);```

This slide illustrates an example of a range-partitioned table.

The keyword RANGE in the partition clause is optional because tables are range partitioned by default.

The first key values are specified for each secondary partition.

The LOCATION clause specifies the node, volume, subvolume, and file name. The node name, subvolume, and file name are optional. The subvolume name must match the schema subvolume name for the table, if specified. The file name must end with "00" because this is the name of the data fork.

The NAME clause specifies a logical name for the partition and is optional. If not specified, SQL/MX generates the name.
This slide illustrates the syntax for hash partitioning a table.

**HASH** — Specifies the table is hash partitioned.

- **[BY (partitioning-column [, partitioning-column] ...)]** — Specifies the columns that make up the partitioning key. If you do not specify this clause, the partitioning key is the same as the clustering key of the table. Partitioning columns may not be floating-point data columns.
- **ADD partn-defn** — Defines a single secondary partition and includes the LOCATION of the partition.
- **LOCATION [\node.]$volume[.subvolume.file-name]** — Specifies a volume and optionally the node, subvolume, and filename for the secondary partition of the table.
  - **node** — Is the name of a node on the Expand network. For Guardian files representing a table or index partition or a view label, the node can be any node from which the object's catalog is visible.
  - **volume** — Is the name of an audited, non-SMF DAM volume on the specified node (or the Guardian system named in the =_DEFAULTS define if none is specified). If you do not specify a LOCATION clause, SQL/MX uses the default volume named in the =_DEFAULTS define. You can locate more than one partition of a table on a single disk volume.
  - **subvolume** — Is the name of the schema subvolume for the schema in which the table is being created. Follow these guidelines when using SQL/MX subvolume names: The name must begin with the letters ZSD, followed by a letter, not a digit (for example, ZSDa, not ZSD2). The name must be exactly eight characters long.
  - **file-name** — Is a Guardian file name. file-name names must be eight characters in length and must end with the digits “00” (zero zero.) Any Guardian file name you specify must match the designated schema subvolume name for the schema in which the object is being created. Otherwise, you will receive an error.
- **NAME partition-name** — Is an SQL identifier for a partition and must be unique. If you omit the NAME clause, SQL/MX generates a partition name.
Creating a Hash Partitioned Table

```
CREATE TABLE QDPCAT.QDPDB.ORDERS
(
  O_ORDERKEY    INT NOT NULL NOT DROPPABLE,
  O_CUSTKEY    INT NOT NULL NOT DROPPABLE,
  ...,
  CONSTRAINT PK_ORDERS PRIMARY KEY (O_ORDERKEY) NOT NULL NOT DROPPABLE
)
STORE BY PRIMARY KEY
LOCATION 'SUNNY.$CLASS1.
  ATTRIBUTE
  EXTENT (126, 126),
  MAXEXTENTS 210
  HASH PARTITION
  ADD LOCATION 'SUNNY.$DSMSCM
  EXTENT (126, 126),
  MAXEXTENTS 210,
  ADD LOCATION 'SUNNY.$OSS1
  EXTENT (126, 126)
  MAXEXTENTS 210,
  ...
);
```

This slide illustrates an example of a hash-partitioned table.
The LOCATION clause specifies the node, volume, subvolume, and file name. The node name, subvolume and file name are optional. The subvolume name must match the schema subvolume name for the table. The file name must end with "00" because this is the name of the data fork. The NAME clause specifies a logical name for the partition.
This slide and the next illustrate the syntax for creating an index.

**UNIQUE —** Specifies that the values (including NULL values) in the column or set of columns (as a group) that make up the index field cannot contain more than one occurrence of the same value or set of values. If you omit UNIQUE, duplicate values are allowed. The columns you specify for the index need not be declared NOT NULL (note that this is unlike CREATE TABLE and ALTER TABLE, which do require all columns of a specified unique constraint to be NOT NULL).

**INDEX —** An SQL identifier that specifies the simple name of the new index. You cannot qualify the index with its catalog and schema names. Indexes have their own namespace within a schema, so an index name might be the same as a table or constraint name. However, no two indexes in a schema can have the same name.

**TABLE —** The name of the table for which to create the index.

**Column-name —** Specifies the columns in the table to include in the index. The order of the columns in the index need not correspond to the order of the columns in the table. ASCENDING or DESCENDING specifies the storage and retrieval order for rows in the index. The default is ASCENDING. Rows are ordered by values in the first column specified for the index. If multiple index rows share the same value for the first column, the values in the second column are used to order the rows, and so forth. If duplicate index rows occur in a nonunique index, their order is based on the sequence specified for the columns of the key of the underlying table.

**NO POPULATE —** Specifies that the index is not to be populated when it is created. The index’s partitions are created, but no data is written to the index; it is marked “offline”. DML statements have no effect on offline indexes. If an index is created with the intention of using it for a constraint, it must be populated before creating the constraint. You can populate an offline index and remove its offline designation by using the POPULATE INDEX statement.

**POPULATE —** Specifies that the index is to be created and populated. The default is POPULATE.

LOCATION, NAME, and ATTRIBUTES have the same description as for a table.
CREATE INDEX — Syntax (2 of 2)

```
partn-file-option is:

[RANGE] PARTITION [BY (partitioning-column [, partitioning-column] ...) ]
[(ADD range-partn-defn [ADD range-partn-defn] ... ) ]
range-partn-defn is:
  FIRST KEY { col-value | (col-value [, col-value] ...) }
  LOCATION [inode.$volume/schema-subvolume.file-name]
  [NAME partition-name]
  [EXTENT ext-size | (pri-ext-size [, sec-ext-size])]
  [MAXEXTENTS num-extnets]

| HASH PARTITION [BY (partitioning-column [, partitioning-column] ...) ]
| [(ADD partn-defn [, ADD partn-defn] ...) ]
| partn-defn is:
|  LOCATION [inode.$volume/schema-subvolume.file-name]
|  [EXTENT ext-size | (pri-ext-size [, sec-ext-size] ) ]
|  [MAXEXTENTS num-extents]
|  [NAME partition-name] [attribute [, attribute] ...]
```

This slide illustrates the partitioning options for the CREATE INDEX statement.
Creating a Range-Partitioned Index

CREATE INDEX RP_ORDIX ON QDPCAT.QDPDB.ORDERS
(
  O_CUSTKEY,
  O_ORDERKEY,
  O_ORDERSTATUS
)
NO POPULATE
LOCATION ISUNNY.$CLASS1
ATTRIBUTE
EXTENT (126, 126)
, MAXEXTENTS 210
RANGE PARTITION
(
  ADD FIRST KEY (3751)
  LOCATION ISUNNY.$DSMSCM.
  EXTENT (126, 126)
  , MAXEXTENTS 210
  , ADD FIRST KEY (7501)
  LOCATION ISUNNY.$OSS1
  EXTENT (126, 126)
  , MAXEXTENTS 210
  ....
);

Primary Partition

Secondary Partitions

This slide illustrates an example of creating a range-partitioned index.
Creating a Hash-Partitioned Index

CREATE INDEX HP_ORDIX ON QDPCAT.QDPDB.ORDERS
  (O_CUSTKEY
   , O_ORDERKEY
   , O_ORDERSTATUS
  )
NO POPULATE
LOCATION \SUNNY.$CLASS1
ATTRIBUTE
  EXTENT (126, 126)
  , MAXEXTENTS 210
HASH PARTITION
  (ADD LOCATION \SUNNY.$DSMSCM
    EXTENT (126, 126)
    , MAXEXTENTS 210
    , ADD LOCATION \SUNNY.$OSS1
    EXTENT (126, 126)
    , MAXEXTENTS 210
    , ...
  );

This slide illustrates an example of creating a hash-partitioned index.
This slide illustrates creating a table with multiple partitions per disk volume. Each partition has individual data and resource fork files in the same schema subvolume.
Decoupled Keys

- The term “decoupled keys” refers to the ability to decouple the definition of the partitioning key from the clustering key.
- Decoupled keys provide the ability to:
  - Co-partition or co-locate an index with its table.
  - Co-partition or co-locate tables that have overlapping keys.
  - Provide more flexible partitioning options when the ideal partitioning key is not also the ideal clustering key.
  - Potentially improve performance by allowing the SQL/MX optimizer to generate parallel plans taking advantage of co-partitioned objects.
Terminology (1 of 2)

- Primary key — The set of columns defined as the primary key in the CREATE TABLE statement.
- Clustering key — The complete set of columns specified in the STORE BY clause of the CREATE TABLE statement. These column values make up the b-tree index for a given access path (table or index).
- Partitioning key — The set of columns specified in the BY clause, RANGE PARTITION BY (col1, col2, ...), or HASH PARTITION BY (col1, col2, ...) of the CREATE TABLE or CREATE INDEX statements.
- Co-partitioned — Two objects (tables and indexes) are co-partitioned if they are partitioned "identically," but not necessarily located on the same volumes.
Terminology (2 of 2)

- Co-located — Two objects (tables and indexes) are co-located; if they are co-partitioned and their corresponding partitions are located on the same volumes.
- Decoupling — Decoupling refers to the decoupling of the clustering key and partitioning key. This feature allows the partitioning key and the clustering key to be different sets of columns.
- Partial decoupling — Partial decoupling is a restricted form of decoupling in which the partitioning key columns must be a subset of the clustering key columns. This is the only form of decoupling which will be supported by SQL/MX.
Co-Partitioning

The exact definition of co-partitioning depends on the partitioning scheme:

- For range partitioning, two objects are co-partitioned if:
  - The corresponding partitioning key columns have compatible data types.
  - The numbers of partitions are equal.
  - The first key values are the same.

- For hash partitioning, two objects are co-partitioned if:
  - The corresponding partitioning key columns are identical data types.
  - The numbers of partitions are equal.
Creating a Partitioned Table with a Decoupled Partitioning Key

CREATE TABLE <table-name>
    (<column-definitions>)
    (<constraint definitions>)

    [STORE BY [PRIMARY KEY | (<key-column-list>)]]
    LOCATION [node.$volume$.subvolume.file-name]
    [NAME partition-name]
    [ATTRIBUTE(S) attribute [ , attribute] ...]
    [HASH [RANGE] PARTITION [BY (partitioning-column [, partitioning-column] ... )]]
    [ADD [FIRST KEY (<first-key-values>) LOCATION <location>]
     [, ADD [FIRST KEY (<first-key-values>) LOCATION <location>] ...]

• The basic extension to support decoupled keys is the “BY (<partitioning_columns>)” clause.
• The <partitioning_columns> are chosen from the list of the clustering key columns.
• The default partitioning key is the clustering key (excluding SYSKEY).

This slide illustrates the addition of the BY clause to the HASH or RANGE PARTITION clause in the CREATE TABLE syntax to support decoupled keys. You can specify any column or subset of columns from the clustering key columns in any order as the partitioning key columns. If you do not specify the BY clause, the clustering key excluding the SYSKEY is used as the default partitioning key.
This slide illustrates an example of a range-partitioned decoupled table. The primary key is (A, B, C), the clustering key specified in the STORE BY is also (A, B, C), and the table is range partitioned by column C (the partitioning key), which is one of the clustering key columns.
Creating a Partitioned Index with a Decoupled Partitioning Key

CREATE INDEX [UNIQUE] index-name ON <table-name>
  ( column-name [ASC[ENDING]] | [DESC[ENDING]]
    [, column-name [ASC[ENDING]] | [DESC[ENDING]] ]
  ...
) LOCATION [node.$volume.[subvolume.file-name]
[NAME partition-name]
partn-file-option
  [ [ HASH | RANGE ] PARTITION [ BY (<partitioning-columns>) ]
    ( ADD [FIRST KEY (<first-key-values>) ] LOCATION <location>
      , ADD [FIRST KEY (<first-key-values>) ] LOCATION <location>
    )
  ]

- The basic extension to support decoupled keys is the “BY (<partitioning-columns>)” clause.
- The <partitioning_columns> can be chosen from the list of the clustering key columns.
- The default partitioning key is the clustering key.

This slide illustrates the addition of the BY clause to the HASH or RANGE PARTITION clause in the CREATE INDEX syntax to support decoupled keys. You can specify any column or subset of columns from the clustering key columns in any order as the partitioning key columns. If you do not specify the BY clause, the clustering key excluding the SYSKEY is used as the default partitioning key.
This slide illustrates an example of a decoupled index I1 created on table T1. The primary key of table T1 is also the clustering key, that is, columns (A, B, C). Table T1 has a decoupled partitioning key, this the table is partitioned on column C, which is one of the clustering key columns.

Index I1 is created on column D of table T1. The indexes’ clustering key columns are columns (D, A, B, C), D is the indexed column, and columns A, B, and C are the clustering key columns of table T1, which are also copied to the index. The index is partitioned on column C and therefore co-partitioned with table T1. The table and index have the same partitioning key, same data type, same number of partitions, and the first key values for each partition are the same.
Partitioning Key Columns

In order to know which columns are available for partitioning, it is necessary to know how SQL/MX constructs the clustering key for a table or index.

- Depends on the STORE BY specification.
- Depends on the PRIMARY KEY specification.
- For an index, it depends on the clustering key of the table.

In order to know which columns can be used for a partitioning key, you must know what the columns of the clustering key are for a table or an index. This depends on:

- The store by specification for the table.
- The primary key of a table and whether it is droppable or not droppable.
- For an index it depends on the clustering key of its table.
This slide and the next illustrate the methods and characteristics of specifying a clustering key for a table.

SQL/MX bases table partitioning on clustering key columns, specified by the STORE BY clause or, if there is no STORE BY clause, the primary-key columns. If you do not specify the STORE BY or PRIMARY KEY columns on a table, SQL/MX cannot partition the table.

Use the PARTITION BY clause to decouple the partitioning key from the clustering key. Without the PARTITION BY clause, the partitioning columns of the table are same as the clustering key columns.

No STORE BY clause

- If you do not specify a primary key, SQL/MX uses a SYSKEY column as the clustering key. You cannot partition a table on a SYSKEY column.
- If you specify a primary key which is droppable, SQL/MX uses a SYSKEY column as the clustering key and enforces the primary-key constraint with a unique index. You cannot partition a table on a SYSKEY column.
- If you specify a primary key which is NOT DROPPABLE, SQL/MX uses the primary-key columns as the clustering key.

STORE BY PRIMARY KEY columns

- If you do not specify a primary key, then the STORE BY PRIMARY KEY specification is not applicable and SQL/MX returns an error.
- If you specify a primary key that is DROPPABLE, SQL/MX returns an error. This is not supported.
- If you specify a primary key that is NOT DROPPABLE, SQL/MX uses the primary-key columns as the clustering key.
Specifying a Clustering Key (2 of 2)

<table>
<thead>
<tr>
<th>STORE BY &lt;key-column-list&gt;</th>
<th>Primary Key Specified</th>
<th>DROPPABLE Attribute</th>
<th>Clustering Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Not applicable</td>
<td>Key column list + SYSKEY</td>
<td></td>
</tr>
<tr>
<td>Yes DROPPABLE</td>
<td>Key column list + SYSKEY Primary key enforced by unique index</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes NOT DROPPABLE</td>
<td>If the STORE BY column list matches the primary-key column list, SQL/MX uses the STORE BY column list. If the STORE BY column list is a prefix of or the same as the primary-key column list, SQL/MX uses the primary-key column list. Any other combination is not supported and will receive an error.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you specify the STORE BY <key-column-list> and

- You do not specify a primary key, SQL/MX adds a SYSKEY column and uses the key-column-list and the SYSKEY as the clustering key for the table.
- You specify a primary key which is DROPPABLE, SQL/MX adds a SYSKEY column and uses the key-column-list and the SYSKEY as the clustering key for the table. SQL/MX also creates a unique index to enforce the primary-key constraint.
- You specify a primary key which is NOT DROPPABLE and
  - The STORE BY key-column-list matches the primary-key column list, SQL/MX uses the STORE BY column-list as the clustering key.
  - The STORE BY key-column-list is a prefix of or the same as the primary-key column list, SQL/MX uses the primary-key column list as the clustering key.
  - Any other combination is not supported and will receive an error.
Key Columns for Indexes

<table>
<thead>
<tr>
<th></th>
<th>Unique Index</th>
<th>Non-Unique Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering Key</td>
<td>&lt;indexed columns&gt;</td>
<td>&lt;indexed columns&gt; + &lt;Clustering Key of Table&gt;</td>
</tr>
<tr>
<td>Default Partitioning</td>
<td>&lt;indexed columns&gt;</td>
<td>&lt;indexed columns&gt; + &lt;Clustering Key of Table&gt;</td>
</tr>
</tbody>
</table>

- Unique indexes, in general, cannot be co-partitioned with their table, because the table's clustering key columns are not available for partitioning.
- It is possible for indexes to have multiple instances of the same column if an indexed column is also part of the clustering key of the table.

For indexes there are two cases: unique indexes and non-unique indexes.

- For unique indexes the clustering key is simply the index column. If you create a unique index on column A, the clustering key is going to be column A. This means that in general you cannot co-locate a unique index with its base table because the clustering key of the base table is not available.
- For non-unique indexes the clustering key is the index columns plus the clustering key of the table. In this case, you can co-locate index with its base table because the full clustering key of the table is available.
This slide illustrates a co-partitioned and co-located table and index. The table’s primary key and clustering key columns are columns (A, B, C). The partitioning key is column C and the table has three partitions.

The index is created on column D of the table, which makes the clustering key of the index columns (D, A, B, C). The index is also range partitioned exactly like the table and the index partitions are co-located with the table’s partitions.
This slide illustrates two co-partitioned and co-located hash-partitioned tables, T4 and T5. Table T4’s primary key and clustering key are columns (A, B). The partitioning key column is column B, which makes table T4 partially decoupled.

Table T5’s primary key, clustering key, and partitioning key is column D. Table T5 is not decoupled.

Tables T4 and T5 are hash partitioned on columns B and D respectively. Columns B and D have the same data type, DATE, and the tables have the same number of partitions (three); therefore, they are co-partitioned. The partitions are located on the same volume; therefore, they are also co-located.
Creating SQL/MX Triggers

Module 4

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice.
Module Objectives

Describe and implement triggers on HP NonStop SQL/MX tables.

References

HP NonStop SQL/MX Installation and Management Guide.
A trigger is a mechanism that sets up the database system to perform certain actions automatically in response to the occurrence of specified events. Triggers execute in a event-condition-reaction paradigm. When certain events happen, the database contains rules that you specify to react to those events.

An event is a SQL operation (INSERT, UPDATE, or DELETE) that activates the trigger. When the event happens, an optional condition is evaluated, and if the condition is true the action is executed. The condition determines whether the trigger should be executed. The action is a SQL statement.
Trigger Terminology (1 of 2)

- Triggering-SQL-statement — The statement that caused the event (INSERT, UPDATE, or DELETE statement).
- Triggering-event — Specifies the type of triggering-SQL-statement (INSERT, DELETE, or UPDATE) that can cause the trigger to execute.
- Triggered-SQL-statement — Specifies the statement to be performed by the trigger.
- Trigger-action-time — Specifies when the trigger fires, BEFORE or AFTER the triggering-SQL-statement.
- Trigger-action or trigger type
  - Row-level-trigger — A trigger whose triggered-SQL-statement is executed for every row modified by the triggering statement.
  - Statement-trigger — A trigger whose triggered-SQL-statement is executed only once every time the triggering-SQL statement is executed.

This slide and the next define some basic terms used with triggers.

Triggering-SQL-statement is the statement that causes the trigger to execute, which is either an INSERT, DELETE, or UPDATE statement. For an UPDATE, you can specify that the trigger execute on the UPDATE of a column or set of columns.

Triggering-event — When you define a trigger you specify the type of triggering-SQL-statement that will cause the trigger to execute.

Triggered-SQL-statement — This is the statement that the trigger will execute when the triggering event occurs.

Trigger-action-time — When you design a trigger you specify when the triggered-SQL-statement should execute.
- BEFORE if you want the trigger action to occur before the triggering event.
- AFTER if you want the trigger action to occur after the triggering event.

Triggered-action or trigger type — When you design the trigger, you specify how the triggered-SQL-statement is executed, that is,
- Row-level trigger — The triggered-SQL-statement is executed for every row modified by the triggering statement. For example, if four rows are modified in a table, the triggered-SQL-statement is executed for each of those four rows.
- Statement-level trigger — The triggered-SQL-statement is executed once every time the event (triggering-SQL-statement) occurs.
Trigger Terminology (2 of 2)

- Subject table — The table the trigger is defined on.
- Referencing — The ability for triggers to reference old images or new images of the data (at row level or table level) before or after the triggering event.
  - The correlation name is used to reference the old or new row image (transition variable) that a row-trigger acted on.
  - The table alias is used to reference the OLD or NEW table image (transition table) that a statement trigger acted on.
- Note — The transition variable and transition table are supported by creating a SQL/MX temporary table for the subject table. The temporary table is created when you create the first trigger on the subject table. It is deleted when you drop the last trigger on the subject table.

The subject table is the table that the trigger is defined on. Triggers have the ability to reference the old images or new images of a subject table’s data (at the row level or table level) BEFORE or AFTER the triggering event. The trigger can use a correlation name for a row image or a table alias for a table image to reference these images in the trigger’s conditional statement or triggered-SQL-statement. They use transition tables or transition variables to access old and new states of the table or row. Statement triggers use transition tables. Row triggers use transition variables. When you create a trigger on a subject table, SQL/MX creates a temporary trigger table to support the transition variables and table.
CREATE TRIGGER Statement — Syntax
(1 of 2)

```
CREATE TRIGGER <trigger-name>
(BEFORE | AFTER)
{ INSERT | DELETE | UPDATE [OF (columns)] }
ON table-name
  REFERENCING NEW [AS] <new-ref> [, OLD [AS] <old-ref>]
    [FOR EACH { ROW | STATEMENT }]
    [WHEN (<search-condition>)]
<triggered-SQL-statement>;
```

<table>
<thead>
<tr>
<th>columns is:</th>
<th>column-name, columns</th>
<th>column-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>old-new-alias-list is:</td>
<td>old-new-alias, old-new-alias-list</td>
<td>old-new-alias</td>
</tr>
<tr>
<td>old-new-alias is:</td>
<td>OLD [AS] correlation-name</td>
<td>NEW [AS] correlation-name</td>
</tr>
</tbody>
</table>

This slide illustrates the syntax for a CREATE TRIGGER statement.

- **trigger-name** (catalog-name.schema-name.trigger-name) — Specifies the ANSI logical name of the trigger to be added.

- **BEFORE or AFTER** — Specifies the activation time of the trigger.

- `{INSERT | DELETE | UPDATE [OF (columns)]}` — Specifies the type of triggering event (INSERT, UPDATE, or DELETE) which causes the trigger to execute. For an UPDATE, you can explicitly specify the columns you are considering.

  - **column-name** — For a UPDATE trigger event, you can specify the ANSI logical name of the column or columns in the subject table, that when updated, cause the trigger to be activated, of the form: [(catalog-name.schema-name.)column-name].

- **ON table-name** — Specifies the ANSI logical name of the subject table that this trigger is defined on, [(catalog-name.schema-name.)table-name].

- `[REFERENCING NEW [AS] <new-ref> [, OLD [AS] <old-ref>] ]` — Specify correlation names for the new or old row (transition variable) reference or table-alias for the new or old transition table. Note that the REFERENCE clause is optional.

- `[ FOR EACH { ROW | STATEMENT } ]` — Specify whether the trigger is a ROW trigger or a STATEMENT trigger. If you do not specify this clause, the default is ROW for a BEFORE trigger and STATEMENT for an AFTER trigger.

- `[WHEN (<search-condition>) ]` — Specify the search condition for the trigger. If the search condition is TRUE, the trigger action will occur; if the search condition is FALSE, the trigger action does not occur. If you omit the search condition, the trigger action occurs as soon as the trigger is activated.
triggered-SQL-statement — The trigger action, the SQL statement to be performed when this trigger is activated.

- searched-update-statement — Is an update statement to be performed when this trigger is activated.
- searched-delete-statement — Is a delete statement to be performed when this trigger is activated.
- insert-statement — Is an insert statement to be performed when this trigger is activated.
- set-new-statement — Is an assignment statement that can be used as a BEFORE-trigger action to assign values to transition variables representing columns in the subject table modified by the triggering action.
- signal-statement — Allows a trigger execution to raise an exception that causes both the triggered and triggering statements to fail. The SIGNAL statement sends a SQLSTATE and error text.
  - quoted-sqlstate — Is the 5-digit SQLSTATE to be passed to SIGNAL. Use the GET DIAGNOSTICS command to retrieve quoted-string-expr (as message-text) and quoted-sqlstate.
  - quoted-string-expr — Is a string expression.
This slide illustrates an example of a trigger temporary table (SAMDBCAT.PERSNL.EMPLOYEE_TEMP) created for the subject table, SAMDBCAT.PERSNL.EMPLOYEE, when the first trigger was created on the subject table. The trigger temporary table holds temporary values, transition variables for row triggers, or transition table for statement triggers. Transition variables consist of the old and new row values for row triggers. Transition tables consist of the old and new rows affected by the statement trigger. The trigger action specified in the WHEN clause or the triggered SQL statement can reference the transition variables for row triggers or the transition table values for statement triggers.

The trigger temporary table contains all of the subject table columns plus two unifier fields, which consist of two additional columns:

- **@UNIQUE_EXECUTE_ID** — Distinguishes a particular trigger execution, for example, when a trigger is executed multiple times.
- **@UNIQUE_UID_ID** — Distinguishes a particular trigger action, for example, when you have cascading triggers.

The primary key for a trigger temporary table is a composite key, that is, the @UNIQUE_EXECUTE_ID and @UNIQUE_UID_ID values.

There is one trigger temporary table for each subject table, not for each trigger. The temporary trigger table is dropped when the last trigger is dropped from the subject table.

Note — The temporary table may also have a SYSKEY column, if the subject table does not have a primary key defined on it.
Trigger Temporary Table (2 of 3)

- Temporary table that contains all the subject table’s columns and a uniquifier.
  - A user column “SYSKEY” is also added if the subject table has no primary key.
- Regular SQL table with a different namespace.
- One trigger temporary table for each subject table, not each trigger.
  - Temporary tables are only accessible by triggers.
- Used by triggers to hold temporary values during execution.
  - Transition variables for row triggers.
  - Transition table for statement triggers.
- Triggers can reference transition variables (for row triggers) and transition tables (for statement trigger) in the trigger’s condition or actions.

There is one trigger temporary table for each subject table. The trigger temporary table contains all of the columns that are in the base table. In addition, the temporary table has a uniquifier field, which is composed of two columns. The temporary table is created when the first trigger is created and the trigger temporary table is dropped when the last trigger is dropped from the subject table.

The temporary table may also have a SYSKEY column, if the subject table does not have a primary key defined on it.

The temporary table is a regular SQL table but with a different namespace; therefore, temporary tables are only accessible by trigger code.

Triggers need old and new values (images) of the affected set. Triggers use a trigger temporary table to hold the transition variables or transition table during execution. Triggers can reference transition variables (for row triggers) or reference transition tables (for statement triggers) in the trigger’s condition or actions.
The uniquifier is used to identify the rows that originate from a specific statement execution. There is only one trigger temporary table per subject table; therefore, if you have multiple instances of the same statement execution, for example, “you might be inserting, updating, or deleting rows to the temporary table”, you have to know which row belongs to which statement. The uniquifier, the unique_Execute_Id, is used to distinguish a particular trigger execution. It is a combination of cluster node number, the process ID, and the address of the rowtcb.

However, this actually cannot guarantee uniqueness, because you might have multiple triggers defined on the same table, for example, Fan insert trigger and an update trigger. Because of the cascading features of triggers, you might actually have both of them trigger the same statement. Therefore, you need to identify which one of these triggers actually have rows in the trigger temporary table. Another uniquifier called the unique_UID_ID is used for that purpose.

The uniquifier has a combined length of 16 bytes. When you design a subject table and you plan to define triggers on it, keep in mind that SQL/MX has a limitation on the maximum row length, which is 4048 bytes. Therefore, make sure you have space for the additional 16 bytes for the trigger temporary table rowsize. Remember the trigger temporary table contains all columns from the subject table plus the additional 16 bytes for the uniquifier. Also the primary key of the trigger temporary table is composed of the uniquifier and the primary key of the subject table. The maximum length for a primary key is 255 bytes, but if you are going to create triggers on a table, the maximum length of the primary key is limited to 239 bytes because the primary key of the trigger temporary table is composed of the subject table’s primary-key length plus the 16 bytes for the uniquifier. If the primary key of the subject table is greater than 239 bytes, the trigger temporary table cannot be created because the primary-key length for the trigger temporary table will exceed 255 bytes.
TEMPORARY_TABLE_HASH_PARTITIONS
— System Default

- TEMPORARY_TABLE_HASH_PARTITIONS system default describes the partitioning scheme for trigger temporary tables by listing volumes across which the temporary tables can be hash partitioned.
- Specify the volumes names in single quotes. You can specify multiple locations separated by commas or colons.
- Insert global values into the SYSTEM_DEFAULTS table:
  ```sql
  INSERT INTO SYSTEM_DEFAULTS (ATTRIBUTE, ATTR_VALUE)
  VALUES ('TEMPORARY_TABLE_HASH_PARTITIONS', '$CLASS1, $CLASS2');
  ```
- Examples using CONTROL QUERY DEFAULT
  ```sql
  CONTROL QUERY DEFAULT TEMPORARY_TABLE_HASH_PARTITIONS 'SUNNY.$CLASS1';
  CONTROL QUERY DEFAULT TEMPORARY_TABLE_HASH_PARTITIONS '$CLASS1, $CLASS2';
  ```

This slide illustrates the system default TEMPORARY_TABLE_HASH_PARTITIONS, which is used to specify the volumes for the temporary tables hash partitions.

If you specify more than one volume, the temporary table is hash partitioned over all those partitions. Range partitioning is not supported.

If no system default is specified, SQL/MX uses the default location of the creator of the first trigger. If the default is changed, the change affects temporary tables created after the change. Previously created temporary tables will retain the previous setting. The partitioning scheme of the trigger subject table is unrelated to the temporary table.
This slide illustrates that BEFORE triggers can reference old and or new row images (transition variables) based on the triggering event and the activation time. They can use these references in a condition clause (WHEN) or in the triggered-SQL-statement.

- **BEFORE INSERT – NEW ROW** — A row trigger can use a correlation name, for example, NEW newR, to access the new row before the new row is inserted into the table.

- **BEFORE UPDATE – OLD ROW, NEW ROW** — A row trigger can use correlation names, for example, NEW newR, OLD oldR, to access the new row and or old row before the update of the old row in the subject table. For an UPDATE, the alias can also be used with the column name or column names to reference specific columns in the new or old row.

- **BEFORE DELETE – OLD ROW** — A row trigger can use a correlation name, for example, OLD oldR, to access the old row before it is deleted from the subject table.

### Referencing for a BEFORE Triggering Event

<table>
<thead>
<tr>
<th>Triggering Event and Activation Time</th>
<th>Row Trigger Can Use These References</th>
<th>Statement Trigger Can Use These References</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEFORE INSERT</td>
<td>NEW ROW</td>
<td>Invalid</td>
</tr>
<tr>
<td>BEFORE UPDATE</td>
<td>OLD ROW, NEW ROW</td>
<td>Invalid</td>
</tr>
<tr>
<td>BEFORE DELETE</td>
<td>OLD ROW</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
BEFORE Triggers — Syntax

CREATE TRIGGER <trigger-name>
BEFORE { INSERT | DELETE | UPDATE [ OF (columns) ] }
ON table-name
  REFERENCING NEW [AS] <new-ref> [, OLD [AS] <old-ref>]
  FOR EACH ROW
  [WHEN (<search-condition>)]
  <triggered-SQL-statement>;

columns is: column-name, columns | column-name

old-new-alias-list is: old-new-alias, old-new-alias-list | old-new-alias
  old-new-alias is:
    OLD [AS] correlation-name | NEW [AS] correlation-name
    OLD [AS] table-alias
    NEW [AS] table-alias

triggered-SQL-statement is:
  set-new-statement | signal-statement

This slide illustrates the syntax for a BEFORE trigger. BEFORE triggers are always row triggers and can only use the SET or SIGNAL statement as the triggered SQL statement.
**BEFORE UPDATE ROW Trigger with SET Statement**

```sql
CREATE TRIGGER monitor_salary_increase
BEFORE UPDATE OF (salary)
ON employee
REFERENCING new AS newR, old AS oldR
FOR EACH ROW
WHEN ((newR.salary > oldR.salary) AND (newR.salary > (oldR.salary * 1.04)))
SET newR.salary = oldR.salary * 1.04;
```

Triggering event (query):
```sql
UPDATE EMPLOYEE
SET SALARY = SALARY * 1.10
WHERE EMPNUM = 1
```

This slide illustrates a **BEFORE UPDATE ROW trigger** for the salary column on the `EMPLOYEE` table. The trigger is executed before the salary column in the `EMPLOYEE` table is updated. The updated row to be updated is copied to the trigger’s temporary table. The trigger generates a new row in the temporary table with the new salary value. The trigger uses correlation names `newR` to reference the new row in the temporary table and `oldR` to reference the old row in the temporary table. The triggers uses the `newR.salary` value and `oldR.salary` value in the trigger condition to compare the salary increase before the `UPDATE` statement is executed. If the new salary increase is greater than 4 percent of the old salary, the trigger executes the SET statement to set the `newR.salary` value to 1.04 times the `oldR.salary` value before the `UPDATE` statement is executed.

**BEFORE-type trigger operations** are exercised as tentative executions. The triggering statement is executed but assigns values to the **NEW ROW transition variables** rather than to the subject table. That table appears not to be affected by the tentative execution. When it is accessed by the trigger action, it shows values in place before the action of the trigger. Because **BEFORE**-triggers can only be row triggers, they use transition variables to access old and new states of the row.

**BEFORE-type triggers do not modify tables.** However, by using the **SET statement**, they can assign new values only to the **NEW ROW transition variables**. As a result, a **BEFORE type trigger** can override the effect of the original trigger statement.

- The triggering statement executes only after the trigger is executed.
- Only row granularity is allowed.
- Only the **NEW ROW transition variable** can be modified.
- **BEFORE-type triggers** cannot be cascading.
This slide illustrates a **BEFORE UPDATE ROW** trigger for the qty_available column in the PARTS table.

```sql
CREATE TRIGGER monitor_part_availability
BEFORE UPDATE OF (QTY_AVAILABLE)
ON PARTS
REFERENCING new AS newR
FOR EACH ROW
WHEN (newR.qty_available = 5)
SIGNAL SQLSTATE 'S9999' ('Check part availability');

EXEC SQL UPDATE PARTS
SET QTY_AVAILABLE = QTY_AVAILABLE - :qty_shipped
WHERE PARTNUM = :partnum;
```
### BEFORE Triggers — Considerations (1 of 2)

- BEFORE triggers are used for one of these purposes:
  - To generate an appropriate signal when an insert, update, or delete operation is applied and a certain condition is satisfied (using the SIGNAL statement as an action).
  - To massage data prior to the insert or update operation that caused the trigger to be activated (using the SET statement as an action).
- BEFORE triggers use only SIGNAL and SET statements.
- BEFORE triggers execute before the transition variables are inserted into the temporary table.
- BEFORE triggers use only the OLD ROW and or NEW ROW transition variables and not transition tables.
- BEFORE triggers cannot modify data already in the database.
- BEFORE triggers do not cascade.

This slide lists the characteristics of BEFORE triggers.

BEFORE triggers execute before you actually put data in the trigger temporary table. The trigger temporary table holds temporary transition values that you can use; but BEFORE triggers actually execute before you put the transition variables in the trigger temporary table.

BEFORE triggers do not cascade because the action of the trigger cannot have inserts, updates, or deletes which are events of other triggers. They only use SET or SIGNAL statements.

BEFORE triggers cannot modify data in the database. Because the SET statement is applied to data that is not in the database yet, it is modifying data that has not made it to the database yet.
BEFORE Triggers — Considerations (2 of 2)

- BEFORE triggers may change any user columns including primary or clustering key columns.
  - Primary-key and clustering key columns are only allowed for insert triggers.

BEFORE triggers is the only case where you might actually modify user columns including the primary-key or clustering key columns. But this is only for inserts. What this means is that you really do not have any value yet for those primary or clustering key specific columns. You can change them before you actually store them. After you store them you cannot change them. For an update or a delete you cannot change the primary-key clustering key. But for an insert, because you have not inserted the column values yet, you can actually change them before hand.
This slide illustrates that AFTER triggers can reference old and or new row images (transition variables) or old and or new table images (transition table) based on the triggering event and the activation time. They can use these references in a condition clause (WHEN) or in the triggered-SQL-statement.

- **AFTER INSERT – NEW ROW**
  - A row trigger can use a correlation name, for example, NEW newR, to access the new row (transition variable) after the row is inserted.
  - A statement trigger can use a table alias, for example, NEW newT, to access the new table image (transition table) after the INSERT statement has executed.

- **AFTER UPDATE**
  - A row trigger can use correlation names, for example, NEW newR, OLD oldR, to access the new row and or old row AFTER the update of the old row in the subject table. For an UPDATE, the alias can also be used with the column name or column names to reference specific columns in the new or old row.
  - A statement trigger can use a table alias, for example, NEW newT, OLD oldT, to access the new row and or old row AFTER the UPDATE statement has executed. For an UPDATE, the alias can also be used with the column name or column names to reference specific columns in the new or old row.

- **AFTER DELETE**
  - A row trigger can use a correlation name, for example, OLD oldR, to access the old row (transition variable) after the row is deleted.
  - A statement trigger can use a table alias, for example, OLD oldT, to access the old table image (transition table) after the DELETE statement has executed.
This slide illustrates the syntax for an AFTER trigger. An AFTER trigger can be row or statement trigger and can use all of the triggered SQL statements, that is, searched update, searched delete, insert, set, and signal.

Note — BEFORE triggers are row triggers and only use set and signal statements as the triggered SQL statement.
This slide illustrates an example of an AFTER INSERT ROW trigger that logs a portion of a new employee record to a log file, EMP_LOG table, plus the user that generated the event and the timestamp of the event.

You issue the CREATE TRIGGER statement with the trigger name, "emp_insert" and then the trigger action time which is AFTER. Next, specify the operation or the event that will activate the trigger, which is an INSERT event. The subject table the INSERT event is acting on is the EMPLOYEE table. In this trigger you want to log a portion of the new row inserted into the EMPLOYEE table into a the EMP_LOG table; therefore, you need to specify a REFERENCE clause using the correlation name, such as, "new AS newR", to access this row. Next you specify that this trigger should execute for each row inserted, making this a row trigger. There are no conditions for this trigger; the WHEN clause has not been specified.

The trigger-SQL-statement is the INSERT statement used to insert information into the EMP_LOG table. The INSERT statement obtains the empnum (newR.empnum) and deptnum (newR.deptnum) using correlation names to access the column values from the transition variable in the trigger temporary table. The INSERT statement also uses the CURRENT_USER and CURRENT_TIMESTAMP functions to obtain the user that generated the event and caused the trigger to execute and the timestamp of the trigger event.
This slide illustrates an example of an AFTER INSERT trigger. When an order is inserted into the ORDERS table and the associated ordered items are inserted into the ODETAIL table, the quantity ordered in the ODETAIL table is used to decrement the quantity available in the PARTS table.
### AFTER UPDATE Trigger with an Update Action

```sql
CREATE TRIGGER INVENTORY_UPDATE2
AFTER UPDATE ON SAMDBCAT.INVENT.PARTSUPP
REFERENCING new as newT
FOR EACH ROW
  UPDATE SAMDBCAT.SALES.PARTS
  SET QTY_AVAILABLE = QTY_AVAILABLE + newT.QTY_RECEIVED
  WHERE PARTNUM = newT.PARTNUM
;

BEGIN WORK;

UPDATE SAMDBCAT.INVENT.PARTSUPP
SET QTY_RECEIVED = 5
WHERE PARTNUM = 2001
  AND SUPPNUM = 2;

COMMIT WORK;
```

This slide illustrates an example of an AFTER UPDATE row triggers. When parts are received and the quantity received in the PARTSUPP table is updated, the quantity available in the PARTS table is increment by the quantity received for each part number received.
This slide illustrates an example of an AFTER DELETE trigger. When an employee is deleted from the EMPLOYEE table, a copy of the deleted employee record plus the user ID that deleted the employee and the timestamp when the deletion occurred is logged to the EMP_DEL_LOG table.
AFTER Triggers — Considerations

- AFTER triggers encode actual application logic.
  - AFTER triggers execute after the triggering-SQL statement.
- After triggers execute after constraints.

This slide lists the characteristics of AFTER triggers.
This slide illustrates cascading triggers, which means that one trigger action (trig1) will trigger another trigger (trig2). For example, when you insert a new employee into the EMPLOYEE table, trig1 is activated, which updates the budget for a project in the PROJECT table. The update to the PROJECT table activates trig2, which updates the budget of the department in the DEPT table. This is an example of cascading triggers.
Desired Characteristics of Triggers — Termination and Confluence (1 of 2)

- When triggers are designed two desired characteristics need to be satisfied to ensure:
  - Termination.
  - Confluence.
- Termination — Iterative or recursive triggering of triggers must eventually terminate.
  - Termination is guaranteed by a maximum recursion limit of 16.
  - SQL/MX raises a compilation warning if the compiler detects a potential cycle that exceeds the maximum cycles limit.
  - SQL/MX raises an error during execution if the maximum limit is reached.

When designing or developing triggers, there are two theoretical characteristics that are desired and should be present in your trigger design, termination and confluence.

- **Termination** — Make sure that your trigger execution eventually gets to a quiescent state, which means that the triggers terminate and that they do not get into a cycle that never finishes.

- **Confluence** — Make sure that your database is always consistent. If you have multiple triggers executing at the same time, make sure that when they finish execution you eventually get to a consistent state again.

Termination means that the iterative or recursive triggering of triggers eventually terminates. You have to make sure that you get to a quiescent state. SQL/MX guarantees this characteristic by limiting up to 16 levels of recursion. When you compile a statement, you get a warning if you reach the limit, 16 levels of recursion. However, you get an error, when you reach the level during execution. The reason for the difference is that you might have some condition on your trigger that stops the recursion before you reach the limit.
### Desired Characteristics of Triggers — Termination and Confluence (2 of 2)

- Confluence — Certifies that when several nonprioritized triggers are triggered at the same time, the final state at the termination of the sequence of triggers does not depend on the order in which these triggers are executed.
  - Confluence is guaranteed by a total order on triggers in the system. Trigger creation time (guaranteed to be unique) is used to prioritize conflicting triggers.
  - SQL/MX supports executing triggers in parallel only if their actions do not conflict.

---

Confluence — Certify that when several nonprioritized triggers are at the same time, the final state at the termination of the sequence of triggers does not depend on the order in which these triggers are executed. When several nonprioritized triggers are triggered at the same time, the final state of the execution will be deterministic.

For example, if you are at a specific state and then you have an event that fires several triggers, then once they are executed you get to final state. If you get to the same state again and the same triggers are executed, you want to make sure that the final state is similar to the other state also. Therefore, you always get a deterministic and predictable state.

This is implemented in SQL/MX by guaranteeing some total order of the triggers in the system, which means every time you have a certain number of triggers executing at the same time, you have a predetermined way of executing them, that is, a specific total order of which one should be executed first, when all of them are triggered at the same time. The way to guarantee this is by the trigger creation time, which is guaranteed to be unique and is used to prioritize conflicting triggers.

Note that SQL/MX supports executing triggers in parallel, but only if they are guaranteed not to be conflicting.
Conflicting Triggers and Access Sets

(1 of 2)

- Triggers can only be allowed to execute in parallel when concurrency does not compromise the semantic correctness of the execution.
- To check for conflicts, "access sets" are constructed for each trigger.
  - The set of table columns (access set) read by the trigger or its descendants (a descendant is a trigger cascading from this trigger directly or indirectly).
  - The set of table columns (access set) written by a trigger or its descendants.
- Two triggers are conflicting when their "access sets" intersect on columns and at least one column is for writing.
- Within a conflict-set, the order of execution is determined by the trigger's timestamp creation — older triggers are executed first.

This slide defines conflicting triggers and access sets.

To check whether triggers are conflicting or not, SQL/MX USES “access sets” to check if triggers are conflicting.

First, the set of columns (access set) that every trigger is reading or one of its descendants is reading is collected. Also, the set of table columns (access set) written by a trigger or one of its descendants is also collected.

Then, two triggers are conflicting if they intersect in the set of columns (access set) and at least one of them is writing. But if they all are reading the intersecting column, they are not conflicting.

If two triggers intersect and one of them is writing and the other one is reading or both are writing, then they are conflicting and have to be executed in sequence. They cannot be executed in parallel. If you have a conflicting set of triggers, SQL/MX uses the trigger’s creation timestamp to determine the order of execution.
Triggers with the same activation time (BEFORE or AFTER) are in a conflict set. However, if you have BEFORE and AFTER triggers, they do not conflict because they execute at two different times.

Note: The trigger that you create first executes first. But that is not totally true because triggers can execute concurrently as long as they do not conflict.
Conflicting Triggers — Example

- Suppose that an update to table T1 is driving the following triggers:
  - Trig1 is an AFTER trigger that updates T2(a).
  - Trig2 is an AFTER trigger that updates T2(b).
  - Trig3 is an AFTER trigger that updates T3.
  - Trig4 is an AFTER trigger that updates T2(a).
  - Trig5 is a BEFORE trigger that updates T2(b).
  - Trig6 is a BEFORE trigger that updates T2(a,b).
- We have two “potential” conflict-sets:
  - Trig1, Trig2, Trig3, and Trig4.
  - Trig5 and Trig6.
- True conflicts:
  - Trig1 and Trig4.
  - Trig5 and Trig6.
- No “true” conflicting triggers can execute in parallel.

This slide gives an example of conflicting trigger sets.

Suppose that we have an update on table T1 and it is driving six triggers. Four of the triggers are AFTER triggers and two of them are BEFORE triggers; therefore, there are two potential conflict sets (BEFORE and AFTER).

Two triggers that have different activation times cannot be conflicting. Therefore, the AFTER triggers Trig1, Trig2, Trig3, and Trig4 might be potentially conflicting, but they cannot conflict with the BEFORE triggers Trig5 and Trig6 because of the trigger activation time. Trig5 and Trig6 are potentially conflicting because they are both BEFORE triggers.

True conflicts — Trig1 and Trig4 because their actions update the same column. Trig1 updates T2(a) and Trig4 updates T2(a).

Trig1 and Trig2 are not conflicting because Trig1 updates column T2(a) and Trig2 updates column T2(b) even though they are on the same table. They are not updating the same column.

True conflicts — Trig5 and Trig6 are also conflicting because Trig5 updates column T2(b) and Trig6 updates columns T2(a,b). The common or intersection set is column "b". They are both writing to column "b" and they are conflicting.

If they are true conflicting triggers, they cannot execute in parallel; they have to executed in sequence.
This slide illustrates the threshold to keep in mind when designing triggers.

- You can have up to 256 triggers per statement.
- There is no maximum number of triggers that you can create on a specific table; however, you can only have 256 triggers per statement. When an event (INSERT, UPDATE, or DELETE) happens you cannot have more than 256 triggers.
- With the recursive level, you can have up to 16 recursive levels per single recursive trigger. However, with 16 recursive levels, it is easy to reach the 256 triggers per statement.
- Disabling triggers does not mean they are going to be removed from the compilation plan. If you disable triggers, they still show up in the query plan.
- Use SHOWDDL to see which triggers are defined on a table. Because triggers on a table might be disabled, you might not be aware of their existence. Use SHOWDDL or check the metadata information to see what triggers are defined on a table.
Error Handling with Triggers

- Signal statement:
  - Trigger action to allow trigger execution to raise exceptions.
  
  ```sql
  SIGNAL SQLSTATE <quoted-sqlstate> (<quoted-string-expr>);
  ```
  - `<quoted-sqlstate>` values are implementation-defined only.
  - The execution of enclosing statement is stopped and the encompassing transaction is aborted.
  - Rollback of entire trigger statement on failure.
    - If any triggered action fails, the entire execution chain starting from the top-level triggering event is rolled back.

This slide illustrates error handling with triggers.

The signal statement allows the trigger action execution to raise exceptions and if the condition is satisfied to nullify or rollback the triggered statement and the triggering statement. This can cascade to any level on top of that. The syntax for a SIGNAL statement is: SIGNAL SQLSTATE and then the quoted SQLSTATE followed by the quoted message or string expression. The quoted SQLSTATE is implementation defined; you cannot use ANSI SQLSTATE. The execution of the enclosing statement is stopped and the encompassing transaction is aborted. On failure, the action of any trigger or cascading trigger or tree can cause the entire tree to rollback. Therefore, if any trigger action fails, the entire execution chain starting from the top-level trigger event is rolled back.
Dropping Triggers and Enabling or Disabling Triggers

- Dropping triggers — Syntax.
  ```sql
  DROP TRIGGER <trigger-name>;
  ```
- You can drop the object (table or view) of a trigger if you use the CASCADE option.
- Enabling and disabling triggers is an HP extension. You must be the owner of the schema or SUPER.SUPER.
  ```sql
  ALTER TRIGGER { ENABLE <trigger-name>
  | DISABLE <trigger-name>
  | ENABLE ALL OF <table-name>
  | DISABLE ALL OF <table-name>
  };
  ```
- ALTER TRIGGER ENABLE <trigger-name>;
- ALTER TRIGGER DISABLE <trigger-name>;
- ALTER TRIGGER ENABLE ALL OF <table-name>;
- ALTER TRIGGER DISABLE ALL OF <table-name>;

This slide illustrates the syntax for dropping and enabling or disabling triggers.

To drop a trigger, use the DROP TRIGGER statement and specify the trigger-name. You can also drop triggers by dropping the dependent object, a table or view, with the CASCADE option, DROP TABLE <table-name> CASCADE, DROP VIEW <view-name> CASCADE.

You can also enable and disable triggers, which is an HP extension, not part of the ANSI standard.

- Use the ALTER TRIGGER {ENABLE|DISABLE} statement to enable or disable individual triggers.
- Use the ALTER TRIGGER {ENABLE ALL OF <table-name> | DISABLE ALL OF <table-name>}) statement to enable or disable all triggers for the specified table.
This slide illustrates the three SMD tables: TRIGGERS, TRIGGER_USED, and TRIGGER_CAT_USAGE tables for triggers.

TRIGGERS table — Describes information about triggers.
- TRIGGER_UID — UID of trigger object
- SUBJECT_CATALOG_UID — UID of the catalog of the subject table
- SUBJECT_SCHEMA_UID — UID of the schema of the subject table
- SUBJECT_UID — UID of the table on which the trigger is defined
- ACTIVATION_TIME — Activation time: B Before or A After.
- OPERATION — Operation that fires the trigger: I INSERT, D DELETE, or U UPDATE
- GRANULARITY — Granularity: R Row or S Statement
- COLUMNS_IMPLICIT — Relevant only for UPDATE trigger: Y Yes or N No
- ENABLED — Current status of trigger: Y if enabled or N if not
- TRIGGER_CREATED — Timestamp of creation of the trigger

TRIGGER_CAT_USAGE table — Describes a trigger’s use of objects in other catalogs (primarily needed for DROP TRIGGER statement). Triggers can access objects in different (foreign) catalogs than the catalog of the trigger itself.
- TRIGGER_UID — UID of trigger object
- OTHER_CATALOG_UID — UID of the foreign catalog containing schemas containing objects used by this trigger
- OTHER_SCHEMA_UID — UID of the foreign schema containing objects used by this trigger

TRIGGER_USED table — Describes how triggers are used.
- TRIGGER_UID — UID of trigger object
- USED_OBJECT_UID — UID of the object used by the trigger
- USED_COL_NUM — Number of columns used by the trigger
- OPERATION — Operation associated with the trigger
- IS_SUBJECT_TABLE — Indicates whether the trigger is defined on the subject table

* Indicates primary key.
TRIGGER_USED — Contains information on how triggers use objects. It serves three purposes:

- Given a table, return all the triggers of a certain operation that are defined on that table. Those triggers might be in other catalogs.
- For every local object (table or view), check if a trigger anywhere is using it. This query also describes how the trigger is using the object (for example, SELECT), and might be needed to determine if and how a table has been used or modified.
- For an UPDATE trigger on explicit columns in the subject table, only the TRIGGER_USED table keeps the list of those columns (a row for each column).

TRIGGER_USED column definitions:

- TRIGGER_CATALOG_UID — UID of trigger’s catalog
- TRIGGER_SCHEMA_UID — UID of trigger’s schema
- TRIGGER_UID — UID of trigger object
- USED_OBJECT_UID — UID of the local object used by the trigger
- USED_COL_NUM — The column number in USED_OBJECT_UID. When there is no specific column, the value is 1.
- OPERATION — Operation that fires the trigger: U UPDATE, I INSERT, or D DELETE. For the used-object only, the operation performed on the used object: S SELECT or R ROUTINE.
- IS_SUBJECT_TABLE — Y if the USED_OBJECT_UID is the subject table of this trigger or N if the USED_OBJECT_UID is used only by this trigger
Triggers Versus Constraints

• If the same task can be accomplished with a trigger or a constraint, which one to use?
  • Triggers are more flexible.
    – You can use a wide variety of SQL statements in a trigger.
    – You can only use integrity checks and foreign key options with constraints.
  • Constraints are more rigid but:
    – DBMS knows exactly what to do and how to optimize for them.
    – Less chance of errors because of deterministic behavior.
    – Use constraints when applicable triggers are tricky.

Triggers and Constraints are similar in many ways. If the same task can be accomplished with a trigger or constraint, which one should you use? Keep the following in mind:

▪ Triggers are more flexible.
  – You can have a variety of SQL statements as part of the action of the trigger.
  – You can only use integrity checks and foreign key options with constraints.

▪ Constraints are more rigid
  – The database management system knows exactly how they are implemented, so the optimizer can optimize better for them.
  – You have fewer chances of errors because you have a deterministic behavior, because they are implemented in the engine.
  – Use constraints whenever it is possible because triggers can be tricky and can cause problems if they are not designed properly.
This slide illustrates the differences between the implementation of the Publish and Subscribe features and triggers.

Triggers are synchronous — the event happens and immediately the trigger executes. Publish/Subscribe is asynchronous.

Before triggers can condition input or impose constraints. For example, using the SET statement you can actually condition the input before it is actually inserted. You cannot do that for Publish/Subscribe.

Triggers have two granularities, row and statement triggers for at least AFTER. You only have row granularity for BEFORE. Publish and Subscribe can only have row granularity.

Triggers can provide copying and or auditing of data. Publish and Subscribe can only queue or broadcast data.

Triggers action can be done under the security of the original event. The action has to execute under the original security of when the event was raised. For Publish and Subscribe, the subscription might have different security.

Triggers can only have a SQL/MX subject table and can only reference in the condition or action, a SQL/MX table. Publish and Subscribe can access either HP NonStop SQL/MP or SQL/MX tables.
Restrictions on Triggers

- Triggers do not allow the use of:
  - Publish and Subscribe's embedded update and embedded delete statements as triggering events.
  - INSERTs, UPDATEs, and DELETEs found in compound statements delimited by BEGIN … END as triggering events.
  - Compound statements delimited by BEGIN … END as part of a triggered action.
  - CALL statement for the triggered action. However, triggering events can be in the body of a stored procedure in Java.
  - Positioned deletes and updates as triggered statements.
  - Subqueries in search-condition for AFTER triggers.
- Cannot define triggers on SQL/MP objects and SQL/MP objects cannot be referenced in a trigger.
- Do not use triggers on SQL/MX user metadata tables, system metadata tables, and the MXCS metadata tables.