Database Management
Create Tables

By: Meyer Tanuan (2022), Rick Kozak (2019), Glenn Paulley (2015), John Mckay (2011)

SQL Data Types

- Each column in a table has a data type
- The data type determines the type and amount of data you can store in a column
- Different types have different semantics and support different sets of operations

Character Types

ASCII

- CHAR(n) – Fixed length max 8K
- VARCHAR(n) – Variable length max 8K
- TEXT – Variable length max 2GB

Unicode

- NCHAR(n) – Fixed length max 4000
- NVARCHAR(n) – Variable length max 2GB
- NTEXT – Variable length max 2^30-1

Binary Types

- BINARY(n) – fixed length max 8K bytes
- VARBINARY(n) – variable length max 8K
- IMAGE – variable length max 2GB
- BIT – 0, 1, or NULL

Exact Numeric Types

- INT – 32 bit signed number
- BIGINT – 64 bit signed number
- SMALLINT – 16 bit signed number
- TINYINT – 8 bit unsigned number

Decimal Numeric Types

- DECIMAL and NUMERIC are synonyms
- DECIMAL is an exact numeric type that supports numeric values with fractional component
- Declaration specifies *precision*(between 1 and 38) and an optional *scale*
- DECIMAL(7,2)
  - 7 significant digits (precision), 2 of these digits after the decimal point (scale)
- DECIMAL(10)
  - 10 significant digits, no fractional component
- Range: +/-10^38
- Storage: between 5 and 17 bytes
- Default: if a value is declared DECIMAL, assumed to be DECIMAL(18)

Approximate Numeric Types

- Approximate numeric types store floating-point values; specification uses scientific notation (mantissa and exponent)
- Not all numbers can be represented exactly (mantissa is in base-2)
- Should never be used for primary keys
- REAL
  - Storage: 4 bytes
  - Range: -3.40E + 38 to -1.18E -38, 0 and 1.18E -38 to 3.40E + 38
- FLOAT(n)
  - Value of n is between 1 and 53; number of bits used to store the mantissa of the number
  - Storage: 4 bytes if n <= 24; 8 bytes if n > 24
- DOUBLE PRECISION
  - Synonym for FLOAT(53)

Date Type

- Range: 1 Jan 0001 AD - 31 Dec 9999
  - Cannot be used for dates BC
- Date only, no time portion
- Default string format: 'YYYY-MM-DD'
- Storage size: 3 bytes
- Default value: 1 January 1900

Time Type

- 00:00:00.0000000 - 23:59:59.9999999
- A TIME type can have msec precision
- Default is TIME(7), 100ns precision
- Time only, no date portion
- Default string format: 'hh:mm:ss[.nnnnnnn]'
- Storage size: between 3 and 5 bytes, depending on precision
- Default value: 00:00:00

DateTime Type

- Date range: Jan 1, 1753 - Dec 31, 9999
- Time range: 00:00:00 and 23:59:59:997
  - Approximately 1/300 second precision
- Character length: 19 to 23 characters
- String: 'YYYY-MM-DD hh:mm:ss[.nnnnnnn]'
- Storage size: 8 bytes
- Default value: 1900-01-01 00:00:00.000
- SET DATEFORMAT option controls interpretation of DATETIME values when specified as string literals

SmallDateTime Type

- Date range: January 1, 1900 - June 6, 2079
- Time range: 00:00:00 - 23:59:59
  - Approximately one minute precision
- Character length: 19 characters maximum
- Default format: 'YYYY-MM-DD hh:mm:ss'
- Storage size: 4 bytes
- Default value: 1900-01-01 00:00:00

DateTime2 Type

- Equivalent to the TIMESTAMP type from the ISO SQL Standard
- Now recommended for all applications instead of DATETIME
- Date range: January 1, 0001 - December 31, 9999
  - Cannot be used for dates BC
- Time range: 00:00:00 - 23:59:59:9999999
- DATETIME2 allows time precision, up to 100ns
  - DATETIME2(7) is the default (100ns precision)
- Character length: 19 to 27 characters
- Default string format: 'YYYY-MM-DD hh:mm:ss[.nn]'
- Storage size: 6 - 8 bytes depending on time precision
- Default value: 1900-01-01 00:00:00.0000000

DateTimeOffset Type

- Equivalent to the TIMESTAMP WITH TIME ZONE type from the ISO SQL Standard
- Adds a time zone component to a DATETIME2 type
- Date range: January 1, 0001 - December 31, 9999
  - Cannot be used for dates BC
- Time range: 00:00:00 - 23:59:59:9999999
  - allows time precision, up to 100ns
- DATETIMEOFFSET(7) is the default (100ns precision)
  - Time zones: -14h through +14h
  - Character length: 26 - 34 characters
  - Format: YYYY-MM-DD hh:mm:ss.nnnnnnn{+|-}hh:mm
  - Storage size: 10 bytes
- Default value: 1900-01-01 00:00:00.0000000 +00.00

Other Types

- MONEY

  Money amounts to the trillions with 4 decimal places accuracy

- ROWVERSION
  - A database-wide unique row number

CREATE TABLE

- Basic CREATE TABLE syntax and format:

```
CREATE TABLE TableName
(column1 TYPE [qualifiers],
 column2 TYPE,
 column3 TYPE [qualifiers],
. . .
 [, table-level constraints]
)
```

- It is customary, but not essential, to put primary key columns at the beginning of a table

CREATE TABLE: Example

```
CREATE TABLE Product
(
   productId VARCHAR(11),
   description VARCHAR(75),
   vendorId VARCHAR(4),
   vendorPartNumber VARCHAR(20),
   price DECIMAL(7,2),
   reorderThreshold INT,
   productCategoryCode CHAR(1)
)
```

NULL , NOT NULL

Each column definition can include:

- NULL to indicate that NULL values are permitted
  - This is the default and is therefore seldom seen
- NOT NULL to indicate that NULL values are prohibited

```
vendorName VARCHAR(50) NOT NULL,
```

CREATE TABLE

All database systems have various optional clauses on CREATE TABLE that make these statements non-portable across various database management systems

- CREATE TABLE can contain vendor-specific syntax for:
  - Column and/or table-level encryption
  - Column compression
  - FOREIGN KEY constraints and various options
  - Table partitioning specification
  - Indexing options
  - Use of system-generated values for primary keys
  - SQL Server supports both IDENTITY values and SEQUENCE types

SELECT INTO

```
SELECT <select list>
INTO <table>
FROM …
```

- The columns in <table> have the same names and data types as the attributes in the query's SELECT list
- Rows are populated from the result of the SELECT statement

SELECT INTO Limitations

- Cannot be used to create a partitioned table
- Indexes, constraints, and triggers defined in the source table are not transferred to the new table, nor can they be specified in the SELECT...INTO statement.
  - If these objects are required, you can create them after executing the SELECT...INTO statement.
- Specifying an ORDER BY clause does not guarantee the rows are inserted in the specified order.
- When a computed column is included in the select list, the corresponding column in the new table is not a computed column. The values in the new column are the values that were computed at the time SELECT...INTO was executed.

DROP TABLE

DROP TABLE <table>

- Completely removes table from database

IF EXISTS

Use code like this in a script to drop a table if it already exists:

```
IF EXISTS (
  SELECT name
  FROM sysobjects
  WHERE name = 'Mytable')
DROP TABLE Mytable;
```

CONSTRAINTS

A. Primary Key (PK)
B. Unique (UX)
C. Foreign Key (FK)
D. Check (CK)
E. Default (DF)

CONSTRAINTS Types

DEFAULT

- The value stored in a column on INSERT if no value is specified

PRIMARY KEY

- Identifies one or more columns that uniquely identify each row in the table
- All values in a PRIMARY KEY cannot be NULL

UNIQUE

- Prohibits duplicate values

FOREIGN KEY

- Used to enforce referential integrity

CHECK

- Used to enforce domain integrity

CONSTRAINTS: Naming

- Constraints should be named
- If you don’t name a constraint, the database engine synthesizes a long and ugly name
- Often, by convention, PRIMARY KEY, UNIQUE KEY, FOREIGN KEY, DEFAULT and CHECK constraint names **start** or **end** in PK, UX, FK, DF and CK respectively, for example:

```
CONSTRAINT CK_Guests_NameLength
CHECK …
```

CONSTRAINTS Categories

Constraints are one of two categories:

- Column constraint
  - A constraint that applies to one and only one column. Often NOT NULL is specified as a column constraint for any given column
- Table constraint
  - A constraint specified at the table level
  - Must be used if the constraint involves multiple columns, for example a multi-column PRIMARY KEY or FOREIGN KEY constraint

TABLE CONSTRAINT: Example

All constraints *can* be specified at the table level

- Multi-column constraints *must* be done at the table level
- Table constraint example:

```
CREATE TABLE OrderItem
(
  invoiceNumber INT NOT NULL,
  productId VARCHAR(11) NOT NULL,
  quantity INT,
  discount NUMERIC(3,3)
  CONSTRAINT PK_OrderItem
  PRIMARY KEY (invoiceNumber,
    productId)
)
```

CONSTRAINTS: Examples

```
CREATE TABLE Product
(
  productId VARCHAR(11) NOT NULL
  CONSTRAINT PK_Product PRIMARY KEY,

  price NUMERIC(7,2)
  CONSTRAINT DF_Product_Price DEFAULT 0.00,

  reorderThreshold INT
  CONSTRAINT CK_Product_Threshold_GtZero
  CHECK (reorderThreshold > 0),

  categoryCode CHAR(1)
  CONSTRAINT CK_Product_CategoryCode_Valid
  CHECK (categoryCode IN ('S','H'))
)
```

CONSTRAINTS:
A. PRIMARY KEY

- PRIMARY KEY values cannot be NULL
  - Typically one ensures this by adding NOT NULL constraints to each primary key column
- A primary key uniquely identifies each row in a table

CONSTRAINTS: B. UNIQUE

A UNIQUE constraint differs slightly from a PRIMARY KEY:

- A table can have only one PRIMARY KEY, but many UNIQUE constraints
- UNIQUE constraints can specify nullable columns
  - Only one (1) NULL value is permitted in the index
  - This restriction is not enforced by other DBMS products, for example SAP SQL Anywhere

As with PRIMARY KEYs:

- A UNIQUE constraint specification results in the creation of a unique index on those columns
- UNIQUE constraints can be referenced by FOREIGN KEY constraints in other tables

CONSTRAINTS: C. FOREIGN KEY

- A FOREIGN KEY constraint provides referential integrity for the data in the column(s).
- FOREIGN KEY constraints require that each value in the column exist in the specified column(s) in the referenced table.
- The referenced columns in the referenced table are typically those specified in that table's PRIMARY KEY constraint

Referential Integrity

- Referential integrity (or RI) means that a row cannot be inserted into the referencing table unless a row with matching values exists in the referenced table
  - Often this is referred to as a “parent” and “child” relationship amongst tables in a schema

Example:

```
INSERT INTO Parent VALUES(1, 'Fred'); -- OK
INSERT INTO Child VALUES( 1 ); -- OK
INSERT INTO Child VALUES( 2 ); -- Error
```

Foreign Key Constraint Options

SQL Server supports the following options for ON DELETE and ON UPDATE rules:

- NO ACTION: the operation is not permitted. In other systems this is called RESTRICT, as in “ON DELETE RESTRICT”. This is the default.
- CASCADE: the (delete or update) operation is “cascaded” to the referencing table. Rarely used in practice.
- SET NULL: Rather than leave orphaned rows in the child table, the column value(s) in the referencing row are set to NULL.
- SET DEFAULT: Similar to SET NULL, but instead the default values for the columns in the child table are used.

CONSTRAINTS: D. CHECK

- A CHECK constraint is a constraint that the database verifies with each INSERT, UPDATE, or MERGE statement that affects the columns referenced by the constraint
- CHECK constraints are not verified by DELETE operations (why?)
- CHECK constraints are violated only when they evaluate to FALSE
- A CHECK constraint that evaluates to UNKNOWN is considered valid

CONSTRAINTS: CHECK Example

Example:

```
CHECK( x < 15)
```

- In this case, the server will verify that the X-value in each row is less than 15 after each update modification
- Otherwise the INSERT, UPDATE, or MERGE statement will get an error
- Expressions in CHECK constraints (X in this example) refer to column values in the particular row being modified
- However: CHECK constraints support sub-queries – hence the search condition in the CHECK constraint may be as complex as desired

CONSTRAINTS: E. DEFAULT

In practice, the DEFAULT constraint is often not named.

- You may see either of these alternatives:

```
price NUMERIC(7,2)
CONSTRAINT DF_Product_Price
DEFAULT 0.00

price NUMERIC(7,2)
DEFAULT 0.00
```

- DEFAULT constraints are useful for situations where not all values are known at the time a row is inserted, but is desirable that the column not contain NULL

Requirements for Keys (1)

Uniqueness

- An obvious requirement for a key
- Names, phone numbers are usually inappropriate, though still serve as useful, if not essential, search terms
  - Bell Canada: maintains both a client ID, and an account number, along with phone numbers

Control

- Is the key value under the application's complete control? Or is the value under the control of an independent organization, for example the Province of Ontario or the Government of Canada?

Requirements for Keys (2)

Immutability

- Updating primary keys is rarely a good idea
  - System maintenance can be complex – need to update all related tuples in other tables
  - Potential loss of historical information

Data type

- Alphabetic keys have implications for internationalization
- Never use floating point values for keys

Requirements for Keys (3)

Entropy and range

- Running out of “room” is problematic because it necessitates both schema and application changes
- How many bits does it take to store the key?
- How many bits are required to effect a change in the key value?
- How many bits are different between the letters ‘a’ and ‘b’ in 7-bit ASCII?

Generation technique

- System-generated or application-generated?
- There are significant performance implications given the choices because an application-generated key will likely require additional SQL requests to insert a new row

Requirements for Keys (4)

Format

- Different key formats can be useful to differentiate between business entities
  - GHK009 – client identifier (note: alphanumeric; Latin characters; no vowels)
  - 897765 – group insurance policy number
- Fast, easy differentiation is useful

Self-checking

- Self-checking keys can aid in administering client data
- Examples: American Express numbers, Canadian SIN numbers
- SINs in Canada: 8 digits plus a “check” digit
  - Computed using Luhn’s algorithm: see http://en.wikipedia.org/wiki/Luhn_Algorithm

Requirements for Keys (5)

Embedded information

- Often embedded information is held within an identifier; SIN is one example
  - First digit of a Canadian SIN number indicates the region in which the number was issued

Input difficulty

- Can help to reduce data entry errors; example: Canadian postal codes
  - Very difficult to touch-type a postal code even for superb typists
  - Significant reduction in misdirected mail

Requirements for Keys (6)

Composite or simple

- Simple keys of a single column are significantly easier to deal with in an application
- Simplifies the logical and physical schema
  - uses less storage

IDENTITY

- IDENTITY columns are typically used as primary keys
- The auto-generated value becomes a surrogate key for the row
- SQL Server generates a unique, sequential, integer identity value for each row inserted in a table
- IDENTITY columns are usually INTEGER but they can be stored using other numeric data types

IDENTITY: Example

```
CREATE TABLE Wedding
(
  id INT IDENTITY NOT NULL
  CONSTRAINT PK_Wedding
  PRIMARY KEY,
  partner1 VARCHAR(50) NOT NULL,
  partner2 VARCHAR(50) NOT NULL,
  date SMALLDATETIME NOT NULL,
  location VARCHAR(30) NOT NULL
)
```

INSERT with IDENTITY

- Usually, you must omit the identity column from the INSERT statement
- This is because SQL Server is responsible for generating key values, not you
- However, you can explicitly enable the ability to override this behaviour and INSERT explicit values to IDENTITY columns
- This is done using the SET IDENTITY_INSERT statement:

```
SET IDENTITY_INSERT Guest ON;
```

INSERT with IDENTITY: Examples

```
INSERT INTO Wedding
(partner1, partner2, date, location)
VALUES
('Jane', 'Bob', '2022-06-01', 'Elmira')

INSERT INTO Wedding
(partner1, partner2, date, location)
VALUES
('Eli', 'Betsy', '2021-12-15', 'St.
Jacobs')

INSERT INTO Wedding
(partner1, partner2, date, location)
VALUES
('Pat', 'Bruce', '2021-06-30', 'Toronto')
```

INSERT with IDENTITY: Result

```
SELECT *
FROM Wedding;
```

```
id partner1 partner2 date       location
----------------------------------------
 1 Jane     Bob      2022-06-01 Elmira
 2 Eli      Betsy    2021-12-15 St. Jacobs
 3 Pat      Bruce    2021-06-30 Toronto
```

Last IDENTITY

The most common way to find the last identity value inserted is with @@IDENTITY:

```
SELECT @@IDENTITY
```

You may want to use an alias:

```
SELECT @@IDENTITY
AS 'Last Identity Value'
```

```
Last Identity Value
------------------------
3
```

Last IDENTITY: Three Ways

There are actually three ways to find the last identity value:

@@IDENTITY

- Returns the identity value generated by the previous INSERT statement

SCOPE_IDENTITY( )

- Same as @@IDENTITY, but restricted to the current scope (useful for stored procedures)

IDENT_CURRENT(table)

- Returns the last identity value created by any INSERT in a particular table

Last IDENTITY in SELECT

You can use @@IDENTITY in a WHERE clause to work with the last row inserted

```
SELECT *
FROM Wedding
WHERE id = @@IDENTITY
```

IDENTITY: IDENT_CURRENT Example

```
CREATE TABLE Guest (
id INT IDENTITY PRIMARY KEY,
name NVARCHAR(10) NOT NULL)

CREATE TABLE Reservation (
id INT IDENTITY PRIMARY KEY,
guestId INT CONSTRAINT
FK_Reservation_Guest FOREIGN KEY
REFERENCES Guest(id),
checkInDate DATE NOT NULL)

DECLARE @GuestId INT;
INSERT INTO Guest VALUES('Meyer');
SET @GuestId =
IDENT_CURRENT('Guest');
INSERT INTO Reservation VALUES
(@GuestId, '2022-01-01');
```

IDENTITY Variation

- Identity column may start with a different value than 0 and increment with a value other than 1, but this is seldom done in practice

```
<column> INTEGER
NOT NULL IDENTITY(15,2)
```

- Initial value of 15, increment by 2

UNIQUE IDENTIFIER

SQL Server UNIQUEIDENTIFIER columns are used to store Globally Unique Identifier (GUID) values

- GUIDs are made up of 32 Hexadecimal (base 16) numbers, for example:

  cd2beaf3-2203-483d-b5a6-8eac46cf8c16

- GUIDs occupy 128 bits / 16 bytes

UNIQUE IDENTIFIER (2)

- The probability of 2 UNIQUEIDENTIFERs having the same value is miniscule
- Many applications treat UNIQUEIDENTIFIER values as unique, without fear of collisions
- UNIQUEIDENTIFIER can be used for the primary key to avoid collisions when data from different sources are combined
- Issues
  - GUID fields are large (16 bytes) compared to INTEGER values
  - Not self-checking
  - Difficult to type, and easy to transpose adjacent characters

UNIQUE IDENTIFIER (3)

- SQL Server has two functions for creating UNIQUEIDENTIFIER values:

NEWID()

- The original function

NEWSEQUENTIALID()

- A function introduced in SQL Server 2014 that makes sure each new value is larger than any previous value on a given computer until the next restart of the Windows operating system

UNIQUE IDENTIFIER: NEWSEQUENTIALID Example

```
CREATE TABLE Software
(
   uniqueId UNIQUEIDENTIFIER NOT NULL

   CONSTRAINT DF_Software_UniqueId
   DEFAULT NEWSEQUENTIALID()

   CONSTRAINT PK_Software
   PRIMARY KEY (uniqueId),

   product NVARCHAR(50) NOT NULL,
…
);
```

SEQUENCE

- A SEQUENCE is a database object that generates a sequence of numeric values and are, like IDENTITY columns, used to create surrogate key values
- Advantages:
  - More flexible than IDENTITY
  - Cycling of values can be controlled
  - The sequence value is known **prior** to the execution of the INSERT statement
  - Sequence values can be generated in descending order
  - Can apply to more than one table; a SEQUENCE is an object in the database separate from any table

CREATE SEQUENCE Syntax

```
CREATE SEQUENCE [schema_name.]sequence_name
[AS[integer_type]]
[START WITH <constant>]
[INCREMENT BY <constant>]
[{MINVALUE [<constant>]}|{NOMINVALUE}]
[{MAXVALUE [<constant>]}|{NOMAXVALUE}]
[CYCLE|{NOCYCLE}]
[{CACHE [<constant>]}|{NO CACHE}]
```

SEQUENCE: Integer Types

- A sequence can be defined as any integer type:
  - **Tinyint** - Range 0 to 255
  - **Smallint** - Range -32,768 to 32,767
  - **Integer** - Range -2,147,483,648 to 2,147,483,647
  - **Bigint** - Range -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- Decimal and numeric with a scale of 0.
- Any user-defined data type (alias type) that is based on one of the allowed types.
- If no data type is provided, the **bigint** data type is used as the default.

SEQUENCE: Example

```
CREATE SEQUENCE CountBy1
START WITH 1
INCREMENT BY 1;

CREATE SEQUENCE CountByNeg1
START WITH 0
INCREMENT BY -1;
```

SEQUENCE: NEXT VALUE FOR

- To retrieve the next value from a SEQUENCE object, you code a SELECT statement using the NEXT VALUE FOR function

  ```
  SELECT NEXT VALUE FOR
  Test.CountBy1
  ```

- NEXT VALUE FOR is a function that can be used in any SELECT statement
- Note that NEXT VALUE FOR has many restrictions and cannot be used in a query's WHERE clause
- If there are multiple NEXT VALUE FOR functions over the same sequence in the same SELECT statement, all of the NEXT VALUE FOR functions will return the same value

SEQUENCE: NEXT VALUE FOR Example

A typical use case for a SEQUENCE is to generate surrogate keys for newly-inserted rows:

```
CREATE TABLE TestTable
(
  counterColumn INT PRIMARY KEY,
  name VARCHAR(40) NOT NULL
);

INSERT INTO TestTable
( counterColumn, name )
VALUES
( NEXT VALUE FOR CountBy1,'x' );
```

Database Management
Alter Tables

By: Meyer Tanuan (2022)

Source: Murach’s SQL Server 2019 for Developers

Basic Syntax of the **ALTER TABLE** statement

```
ALTER TABLE table_name [WITH CHECK|WITH NOCHECK]
{ADD new_column_name data_type
[column_attributes] |
 DROP COLUMN column_name |
 ALTER COLUMN column_name new_data_type
[NULL|NOT NULL] |
 ADD [CONSTRAINT] new_constraint_definition |
 DROP [CONSTRAINT] constraint_name}
```

ALTER TABLE Examples (1)

```
-- Add a column
 ALTER TABLE Vendors
 ADD LastTranDate DATE NULL;
-- Drop a column
ALTER TABLE Vendors
DROP COLUMN LastTranDate;
-- Add a new check constraint
ALTER TABLE Invoices WITH NOCHECK
ADD CHECK (InvoiceTotal >= 1);
```

ALTER TABLE Examples (2)

```
-- Add a foreign key constraint
ALTER TABLE InvoiceLineItems WITH CHECK
ADD FOREIGN KEY (AccountNo)
        REFERENCES GLAccounts(AccountNo);

-- Change the data type of a column
ALTER TABLE InvoiceLineItems
ALTER COLUMN InvoiceLineItemDescription
VARCHAR(200);
```