USTH
VIETNAM FRANCE UNIVERSITY

FUNDAMENTALS OF DATABASES

Non-table Objects

NGUYEN Hoang Ha

Email: nguyen-hoang.ha@usth.edu.vn

INDEX

Why indexing?

- Indexing are one of the most important and useful tools for achieving high performance in a relational database
- Many database administrators consider indexes to be the single most critical tool for improving database performance
- An index is a data structure that contains a copy of some of the data from one or more existing database tables
- A database index provides an organizational framework that the DBMS can use to quickly locate the information that it needs
- This can vastly improve the speed with which SQL queries can be answered

Without index

- Query for a random name within the table
- What is the average search time if the process is repeated many times?

$$average = \frac{n+1}{2}$$

- What is the maximum search time?

$$Maximum = n$$

Row Position	Last Name
6	Al Rabeeah
16	Beena
13	Doshi
10	Flores
11	Fung
19	Gani
8	Garcia
21	Hu
22	Israr
9	Johnson
18	Ly
2	Mishra
17	Ngo
20	Pham
3	Salehian
1	Schluter
14	Scruton
12	Spievak
4	Vu
5	Wah
15	Winter
7	Wong

With index

- Query for a random name within the table
- What is the average search time if the process is repeated many times?

$$average = \log_2(n) - 1 = 3.5$$

- What is the maximum search time?

$$Maximum = \log_2(n) = 4.5$$

Row Position	Last Name
1	Schluter
2	Mishra
3	Salehian
4	Vu
5	Wah
6	Al Rabeeah
7	Wong
8	Garcia
9	Johnson
10	Flores
11	Fung
12	Spievak
13	Doshi
14	Scruton
15	Winter
16	Beena
17	Ngo
18	Ly
19	Gani
20	Pham
21	Hu
22	Israr

Index concepts

- Indexes are created on one or more columns in a table
  - For example:
    - An index is created on a PK column
    - The index will contain the PK value for each row in the table, along with each row's ordinal position (row number) within the table
    - When a query involving the PK is run, the DBMS will find the PK value within the index. The DBMS will then know the position of the row within the table
    - The DBMS can then quickly locate the row in the table that is associated with the PK value
- Without an index, the DBMS has to perform a table scan in order to locate the desired row

Index concepts

- An index can be created on most, but not all, columns. Whether an index can be created on a column depends on the column's datatype
- Columns with large object data types cannot be indexed without employing additional mechanisms These data types include:
  - Text
  - ntext
  - Image
  - varchar (max)
  - Nvarchar(max)
  - varbinary(max)

Index concepts

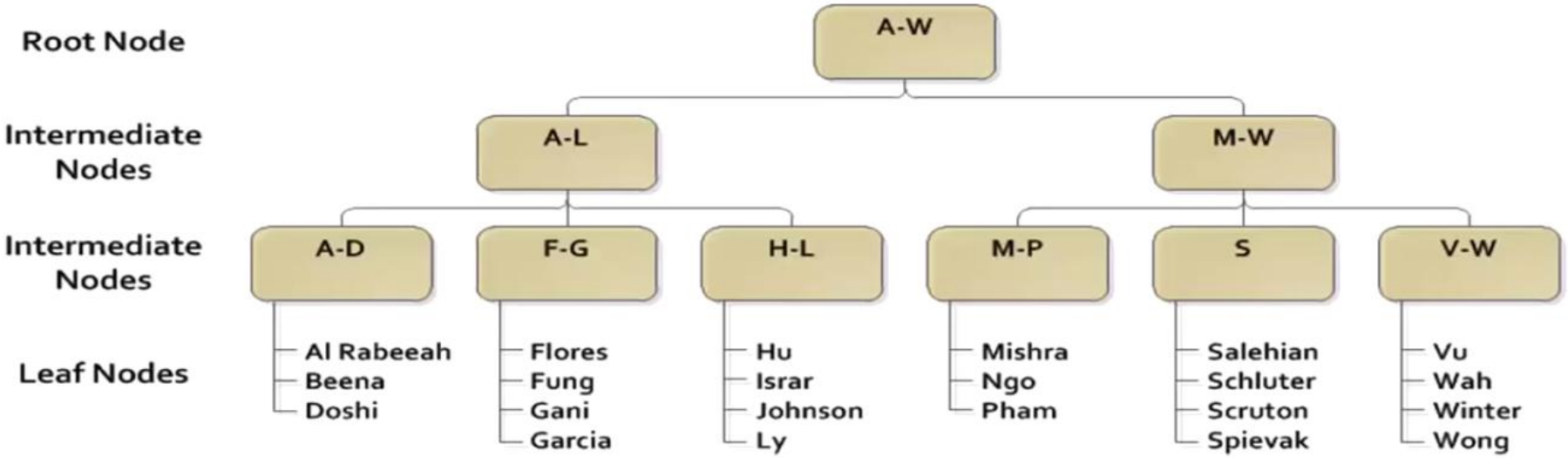
- Creating an index increases the amount of storage space required by the database
  - This occurs because an index contains a copy of some of the data in a table
  - To estimate the storage space requirements of an index, we can use the following formula:

Number of rows in table x Average number of bytes required per row for the indexed columns

B-Tree Index

- Balance-Tree: the most common type of database indexing
- B-trees use pointers and several layers of nodes in order to quickly locate desired data
- Root node
- Intermediate nodes
- Leaf nodes
- When the DBMS processes a query which includes an indexed column, it starts at the root node of the B-tree and navigates downward until it finds the desired leaf

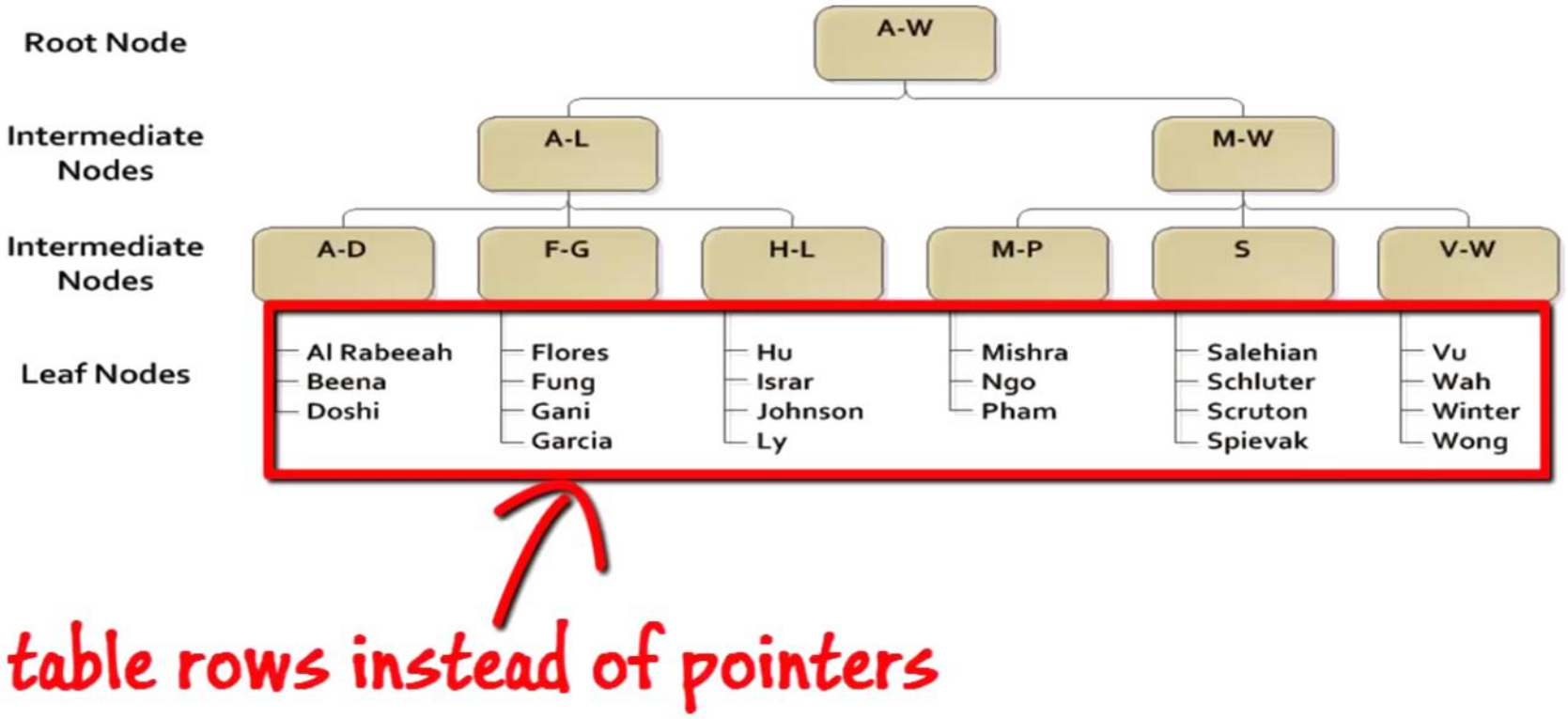
B-tree example



Clustered Indexes

- In a clustered index, the actual data rows that comprise the table are stored at the leaf level of the index
- The indexed values are stored in a sorted order
  - This means that there can be only one clustered index per table
  - PK columns are good candidates for clustered indexes

Clustered B-Tree Example



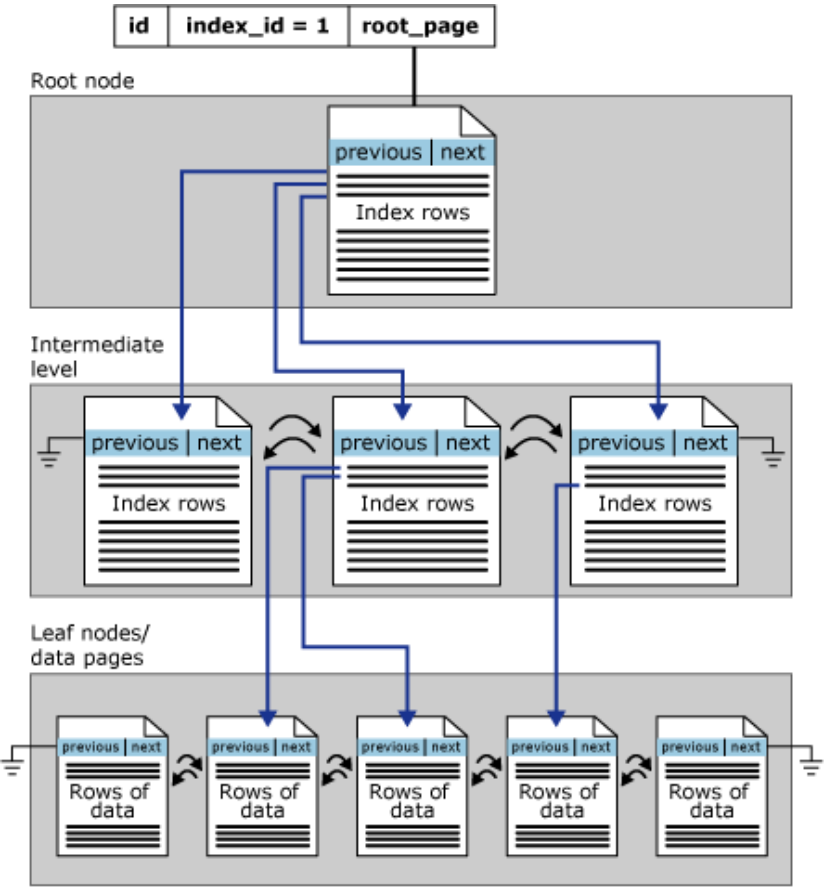
Non-clustered index

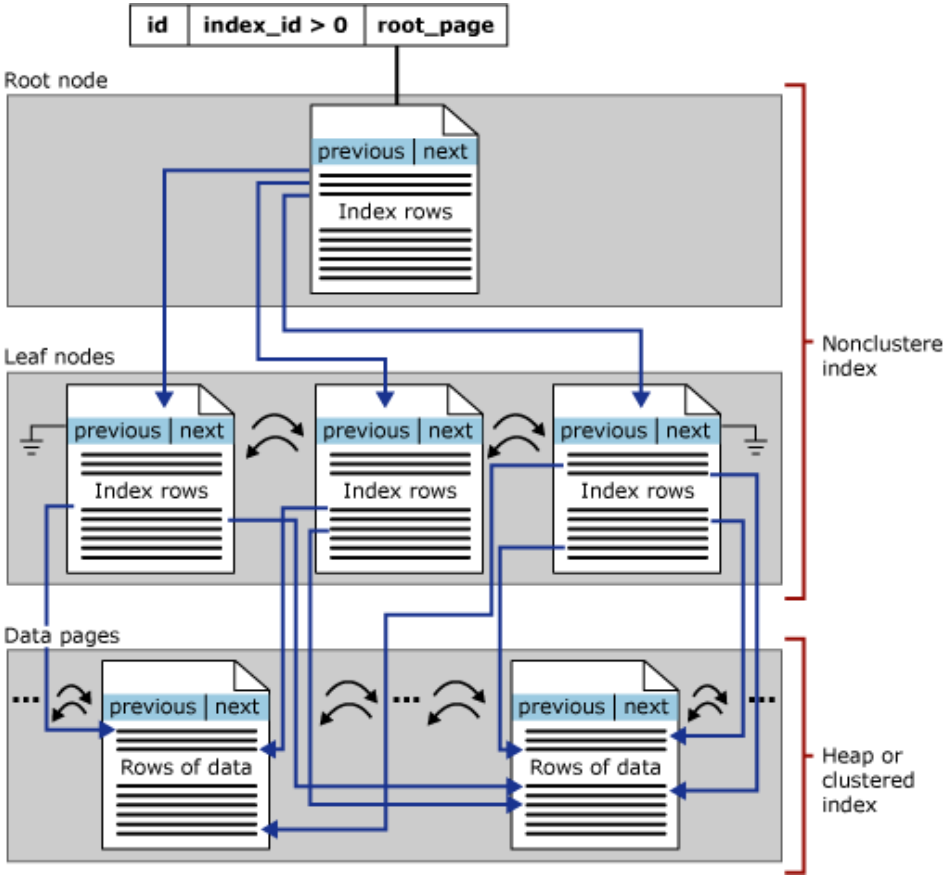
- The Non-Clustered index is an index structure separate from the data stored in a table

→ A table can have more than one non-clustered index

- Non-clustered indexes are slower than clustered indexes because the DMBS must follow a pointer to retrieve the actual data row.
- The leaf nodes of a non-clustered index can optionally contain values from non-indexed columns

Clustered vs. Nonclustered indexes





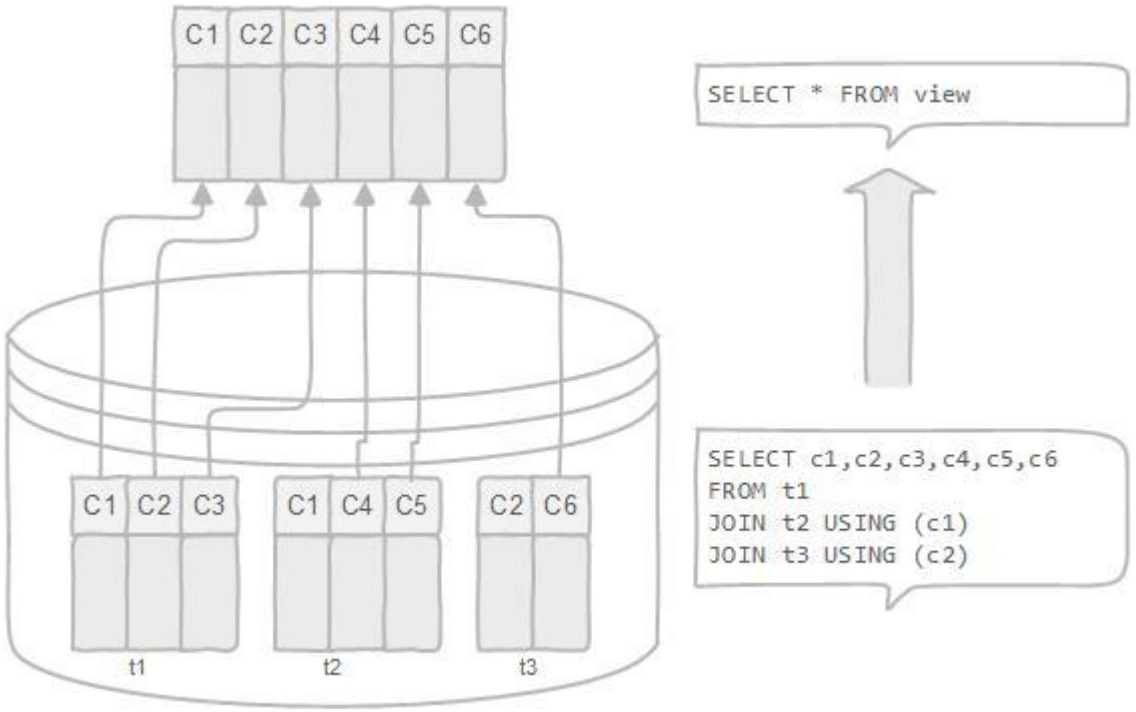
Indexing Guidelines

- If a table is heavily updated, index as few columns as possible
- If a table is updated rarely, use as many indexed columns as necessary to achieve maximum query performance
- Clustered indexes are best used on columns that do not allow null values and whose value are unique
- The performance benefits on an index are related to the uniqueness of the values in the indexed column
  - Index performance is poor when an indexed column contains a large proportion of duplicate values
  - Index performance is best when an indexed column contains unique values

VIEW

View concept

- A view is a "virtual" or logical table that is derived from other tables



View

- Uses:
  - Restrict data access
  - Hide sensitive data
    - Names of tables and columns
  - Simplify data
  - Reuse complex queries

Syntax

```
CREATE [ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
VIEW view_name [(column_list)]
AS
select-statement [WITH CHECK OPTION];
```

- ALGORITHM
  - MERGE: MySQL combines input query with the select-statement. MERGE is not allowed if the SELECT statement contains aggregate functions or `DISTINCT, GROUP BY, HAVING, LIMIT, UNION, UNION ALL`, subquery, SELECT statement refers to no table.
  - TEMPTABLE: MySQL creates a temporary table based on the SELECT statement that defines the view, then performs query against this temporary table.
  - UNDEFINED: MySQL makes choice of MERGE or TEMTABLE.

Pros vs. Cons

- Pros:
  - Simplify complex queries
  - Enable computed columns
  - Provide a security layer: hide sensitive data
  - Enable backward capability
- Cons
  - Performance
  - Table dependency: table changes → need to change views

View examples

- Computed columns

```
CREATE VIEW sale_per_order AS
SELECT order_id, SUM(quantity * unit_price * (1-discount)) total
FROM order_details
GROUP BY order_id
ORDER BY total DESC;
```

- Based on a sub query

```
CREATE VIEW above_avg_products AS
SELECT product_code, product_name, list_price
FROM products
WHERE list_price > (SELECT AVG(list_price)
FROM products)
ORDER BY list_price DESC;
```

- Based on another view

```
CREATE VIEW big_sale_orders AS
SELECT order_id, ROUND(total,2) AS total
FROM sale_per_order
WHERE total > 1000;
```

Updatable views

- SELECT statement defining the view must not contain following elements:
  - Aggregate functions such as MIN, MAX, SUM, AVG, and COUNT.
  - DISTINCT
  - GROUP BY clause.
  - HAVING clause.
  - UNION or UNION ALL clause.
  - Left join or outer join.
  - Subquery in the SELECT clause or in the WHERE clause that refers to the table appeared in the FROM clause.
  - Reference to non-updatable view in the FROM clause.
  - Reference only to literal values.
  - Multiple references to any column of the base table

WITH CHECK OPTION Clause

- Role: to prevent updating or inserting rows that are not visible through the view
- Example

```
CREATE OR REPLACE VIEW northwind_products
AS
SELECT id, product_code, product_name
FROM products
WHERE product_name LIKE 'Northwind%'
WITH CHECK OPTION;

INSERT INTO northwind_products (product_code, product_name)
VALUES('HNB', 'Hanoi Beer');-- This is invalid

INSERT INTO northwind_products (product_code, product_name)
VALUES('NWnew', 'Northwind Beer');

UPDATE northwind_products
SET product_name = 'Nwd beer'
WHERE product_code = 'NWnew'; --- WITH CHECK OPTION will prevent this statement from running
```

View management

- Show view definition
  - `SHOW CREATE VIEW [database_name].[view_ name];`
- Delete view:
  - `DROP VIEW [IF EXISTS] view_name`
- Change view

```
ALTER[ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
VIEW view_name [(column_list)]
AS
select-statement [WITH CHECK OPTION];
```

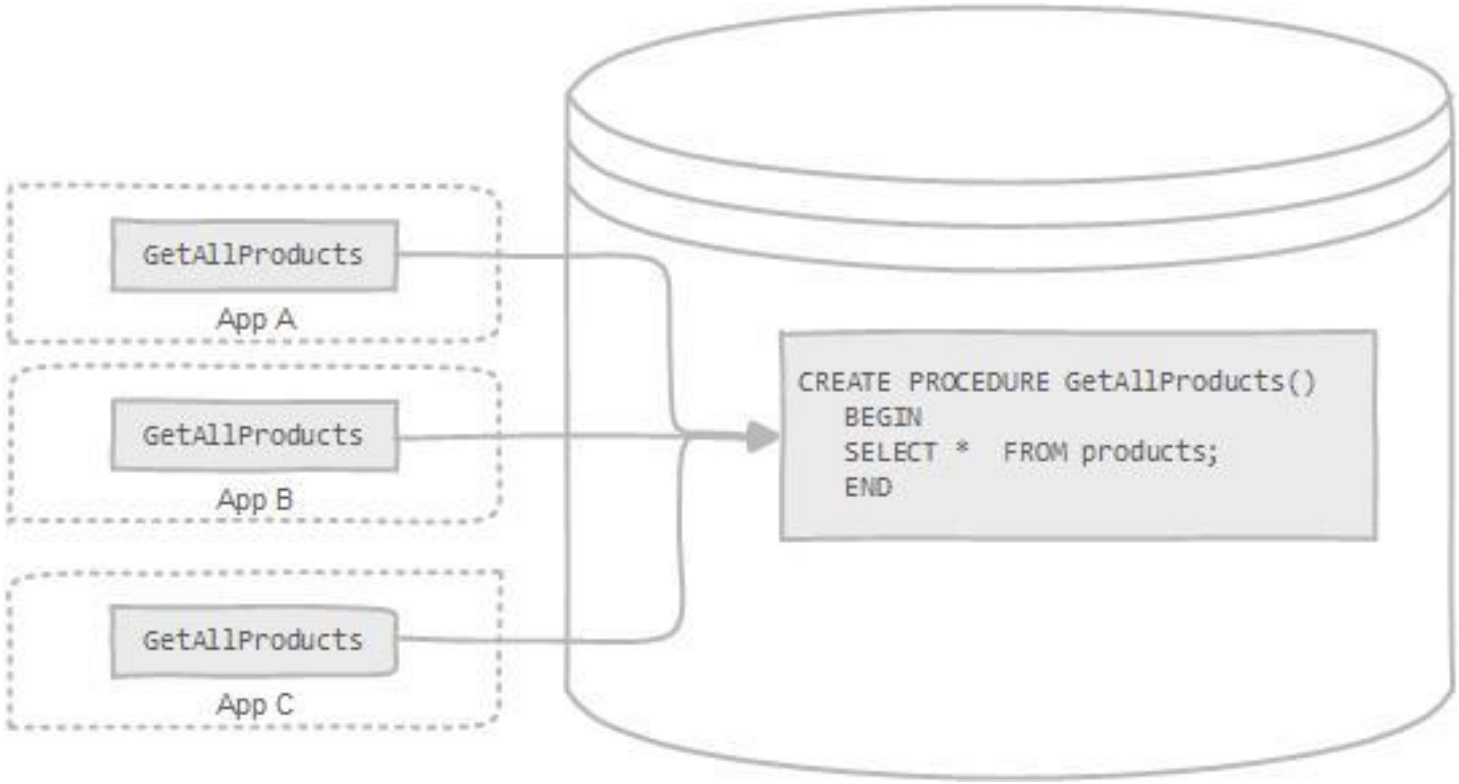
  - OR: CREATE OR REPLACE VIEW
- List all views with updateable information (SQL Server)

```
SELECT table_name, is_updatable
FROM information_schema.views
```

STORED PROCEDURE

Concept

- A stored procedure is a segment of SQL statements stored inside the database catalog



Pros vs. Cons

- Pros:
  - Better performance
  - Reduce traffic
  - Be reusable and transparent
  - Provide a secure way to access data
- Cons
  - CPU usage can increase if logical operators are overused
  - Hard to debug, maintain

Example

```
DROP PROCEDURE IF EXISTS count_products;
delimiter //
CREATE PROCEDURE count_products (OUT param1 INT)
BEGIN
    SELECT COUNT(*) INTO param1
    FROM products;
END//
delimiter ;

CALL count_products(@a); SELECT @a;
```

Input parameter

```
DELIMITER //
CREATE PROCEDURE get_customers_by_city(IN search_city nvarchar(255))
AS
BEGIN
        SELECT * FROM customers
        WHERE city = search_city;
END //
DELIMITER ;


CALL get_customers_by_city('Seattle');
```

Stored Procedure (SP)

- SP is a collection of T-SQL statements that SQL Server compiles into a single execution plan.
- SP is stored in cache area of memory when it is first executed so that it can be used repeatedly, not need recompiled
- Parameters:
  - Input
  - Output

SP Syntax

[ENCRYPTION]
[RECOMPILE]
[EXECUTE AS username]

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [schema_name.] procedure_name
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS
{ [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
```

```
DROP PROC [schema_name.] procedure_name
```

Stored Procedure vs. SQL Statement

SQL Statement

First Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Second Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Stored Procedure

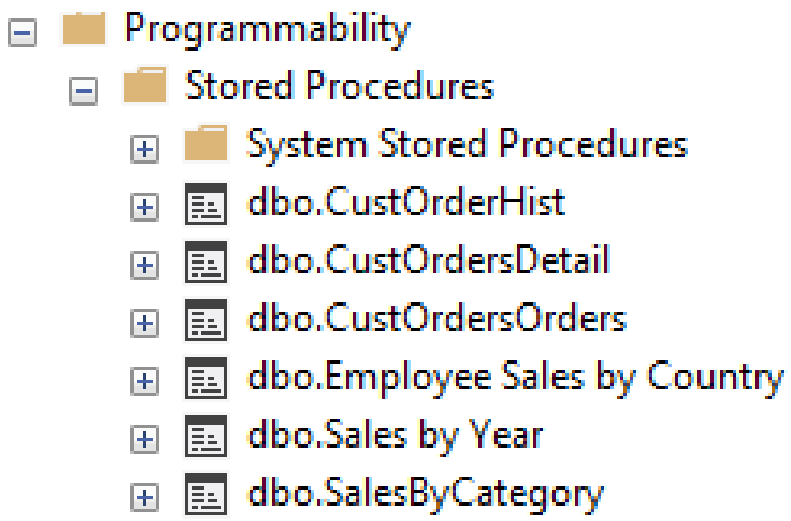
Creating
- *Check syntax*
- *Compile*

First Time
- *Be loaded*
- *Execute*
- *Return data*

Second Time
- *Execute*
- *Return data*

Types of SP

- System stored procedure:
  - Name begins with sp_
  - Created in master database
  - For application in any database
  - Often used by sysadmins
- Local stored procedure:
  - Defined in the local database



Executing a SP

- EXEC pr_GetTopProducts
- With parameters
  - By Name:

```
EXEC pr_GetTopProducts
   @StartID = 1, @EndID = 10
```

  - By Position:

```
EXEC pr_GetTopProducts 1, 10
```

  - Leveraging Default values

```
EXEC pr_GetTopProducts @EndID=10
```

  - Place parameters with default values at the end of the list for flexibility of use

Output parameters

- Used to send non-recordset information back to client
- Example: returning identity field

```
CREATE PROC InsertSuppliers
@CompanyName nvarchar(40), @returnID int OUTPUT
AS
INSERT INTO Suppliers(CompanyName) VALUES (@CompanyName)
SET @returnID = @@IDENTITY

GO

DECLARE @ID int
EXEC InsertSuppliers @CompanyName = 'NewTech', @returnID = @ID OUTPUT
SELECT @ID
```

Encrypting stored procedures

- When the stored procedures created, the text for them is saved in the *SysComments* table.
- If the stored procedures are created with the "WITH ENCRYPTION" then the text in *SysComments* is not directly readable
- "WITH ENCRYPTION" is a common practice for software vendors

```
SELECT * FROM sys.syscomments
```

100 %

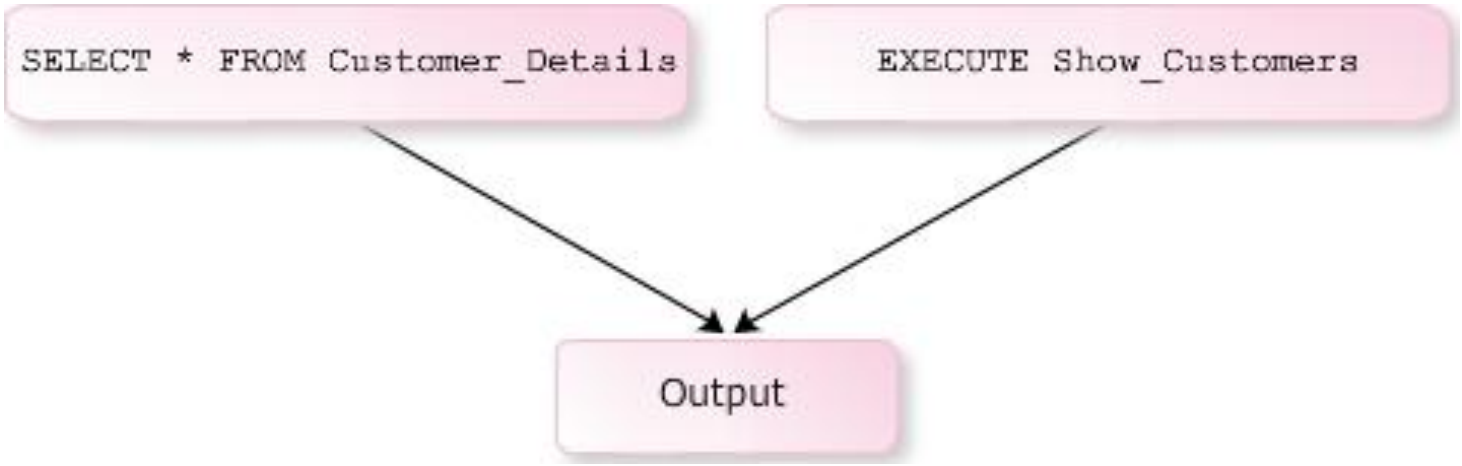
Results | Messages

		texttype	language	encrypted	compressed	text
1	5006100740065002000700072006F006300650...	2	0	0	0	create procedure sys.sp_MSalreadyhavegeneration (@genguid..
2	5006100740065002000700072006F006300650...	2	0	0	0	create procedure sys.sp_MSwritemergeperfcounter (@agent_..
3	5004100540045002000560049004500570020 0...	2	0	0	0	CREATE VIEW INFORMATION_SCHEMA.TABLE_PRIVILEGES ..
4	0006C007300650074007300790006E006300730...	2	0	0	0	replsetsyncstatus extended procedure
5	3007200650061007400650020007000720006F0...	2	0	0	0	create procedure sys.sp_replshowcmds (@maxtrans int = 1 ..
6	F002A00200046006F00720020006200610063...	2	0	0	0	/* For backward compatible */ create procedure sys.sp_publishd..
7	5006100740065002000700072006F006300650...	2	0	0	0	create procedure sys.sp_addqueued_artinfo (@artid ..
8	7006E0075006C006C002700200063006F006C...	2	0	0	0	N'null' collate database_default) select @owner = schema_...

Advantages of SP

- Security
- Code reuse, modular programming
- Performance
- Reduce traffic

Example: Reduced traffic



- Each time Client wants to execute the statement "SELECT * FROM customer_details", it must send this statement to the Server.
- Of course, we see that, the length of that statement is longer than the length of "Show_Customers"

Control of flow – SQL Programming

- Still somewhat limited compared to other languages
  - WHILE
  - IF ELSE
  - BEGIN END block
  - CASE
  - WAITFOR
  - CONTINUE/BREAK

Variables

- Declare a variable:

```
DECLARE @limit money

DECLARE @min_range int, @hi_range int
```

- Assign a value into a variable:

```
SET @min_range = 0, @hi_range = 100

SET @limit = $10
```

- Assign a value into a variable in SQL statement:

```
SELECT @price = price FROM titles
WHERE  title_id = 'PC2091'
```

Control of Flow

BEGIN…END

IF…ELSE

CASE … WHEN

RETURN [n]

WHILE

PRINT

PRINT

- Display message in SQL Query Analyze (Console)

```
USE AdventureWorks2008R2;
GO
IF (SELECT SUM(i.Quantity)
    FROM Production.ProductInventory i
    JOIN Production.Product p
    ON i.ProductID = p.ProductID
    WHERE Name = 'Hex Nut 17'
    ) < 1100
    PRINT N'There are less than 1100 units of Hex Nut 17 in stock.'
GO
```

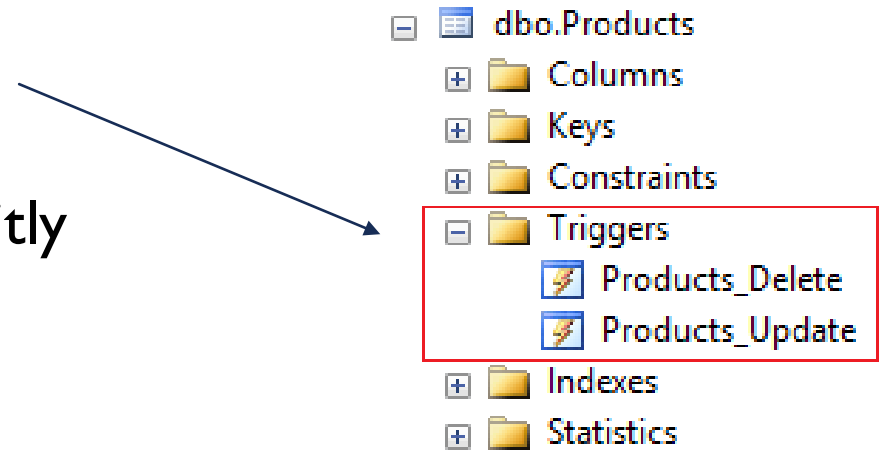
TRY CATCH structure

```
CREATE PROCEDURE dbo.uspTryCatchTest
AS
BEGIN TRY
    SELECT 1/0
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber
     ,ERROR_SEVERITY() AS ErrorSeverity
     ,ERROR_STATE() AS ErrorState
     ,ERROR_PROCEDURE() AS ErrorProcedure
     ,ERROR_LINE() AS ErrorLine
     ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

TRIGGERS

Trigger overview

- Definition: A trigger is a special SP executed automatically as part of a data modification (INSERT, UPDATE, or DELETE)
- Associated with a table
- Invoked automatically
- Cannot be called explicitly



Syntax

```
CREATE TRIGGER trigger_name
ON  <tablename>
<{FOR | AFTER}>
{[DELETE] [,] [INSERT] [,] [UPDATE]}
AS
SQL_Statement [...n]
```

Simplied Syntax

```
CREATE TRIGGER trg_one
ON tablename
FOR INSERT, UPDATE, DELETE
AS
BEGIN
    SELECT * FROM Inserted
    SELECT * FROM Deleted
END
```

Temporary table holding new records

Temporary table holding old, deleted, updated records

Uses of Triggers

- Maintenance of duplicate and derived data
- Ensure integrity
  - Complex column constraints
  - Cascading referential integrity
  - Inter-database referential integrity
- Complex defaults
- Logging/Auditing
- Maintaining de-normalized data

Trigger example

```
Use Northwind
GO
CREATE TRIGGER Cust_Delete_Only1 ON Customers
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 1
BEGIN
   RAISERROR('You are not allowed to delete more than one customer at a
time.', 16, 1)
   ROLLBACK TRANSACTION
END
```

```
DELETE FROM Customers
WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders)
```

Define a trigger preventing users from updating more than 2 records at a time?

INSERT-Trigger example

```
USE Northwind GO
CREATE TRIGGER Order_Insert
ON [Order Details]
FOR INSERT
AS
UPDATE P SET UnitsInStock = (P.UnitsInStock - I.Quantity)
FROM Products AS P INNER JOIN Inserted AS I ON P.ProductID = I.ProductID
```

Order Details

OrderID	ProductID	UnitPrice	Quantity	Discount
10522	10	31.00	7	0.2
10523	41	9.65	9	0.15
10524	7	30.00	24	0.0
10523	2	19.00	5	0.2

ProductID	UnitsInStock	…	…
1	15		
2	5		
3	65		
4	20		

```
INSERT [Order Details] VALUES
(10525, 2, 19.00, 5, 0.2)
```

inserted

10523	2	19.00	5	0.2

UPDATE-Trigger example

```
CREATE TABLE PriceTracking
(ProductID int, Time DateTime, OldPrice money, NewPrice money)

GO

CREATE TRIGGER Products_Update
ON Products FOR UPDATE
AS
INSERT INTO PriceTracking (ProductID, Time, OldPrice, NewPrice)
SELECT I.ProductID, GETDATE(), D.UnitPrice, I.UnitPrice
FROM inserted AS I INNER JOIN Deleted AS D ON I.ProductID = D.ProductID AND
I.UnitPrice <> D.UnitPrice
```

```
UPDATE Products
SET UnitPrice = UnitPrice + 2
```

ProductID	Time	OldPrice	NewPrice
1	2017-10-27 10:46:01.190	18.00	19.00
77	2017-10-27 10:46:24.107	13.00	15.00
76	2017-10-27 10:46:24.107	18.00	20.00
75	2017-10-27 10:46:24.107	7.75	9.75
74	2017-10-27 10:46:24.107	10.00	12.00
73	2017-10-27 10:46:24.107	15.00	17.00
72	2017-10-27 10:46:24.107	34.80	36.80
71	2017-10-27 10:46:24.107	21.50	23.50
70	2017-10-27 10:46:24.107	15.00	17.00
69	2017-10-27 10:46:24.107	36.00	38.00
68	2017-10-27 10:46:24.107	12.50	14.50

Enforcing integrity with Trigger

```
CREATE TRIGGER Products_Delete
ON Products FOR DELETE AS
IF (SELECT COUNT(*)
    FROM [Order Details] OD
    WHERE OD.ProductID = (SELECT ProductID FROM deleted)
    ) > 0
BEGIN
    PRINT 'Violate Foreign key reference. Rollback!!!'
    ROLLBACK TRAN
END
```

```
DELETE Products
WHERE ProductID = 11
```

Performance Considerations

- Triggers work quickly because the Inserted and Deleted tables are in cache
- Execution time is determined by:
  - Number of tables that are referenced
  - Number of rows that are affected
- Actions contained in triggers implicitly are part of a transaction