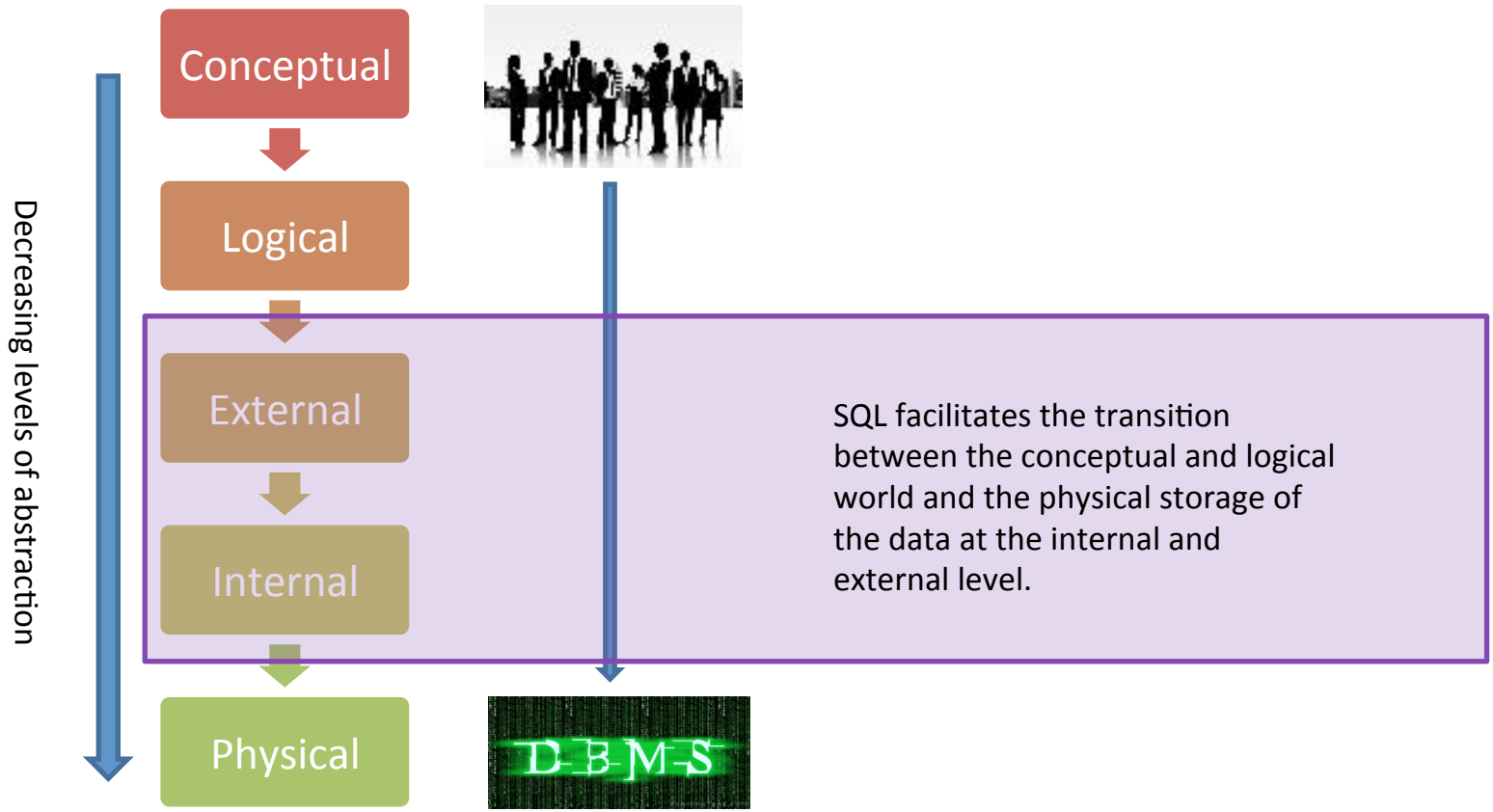
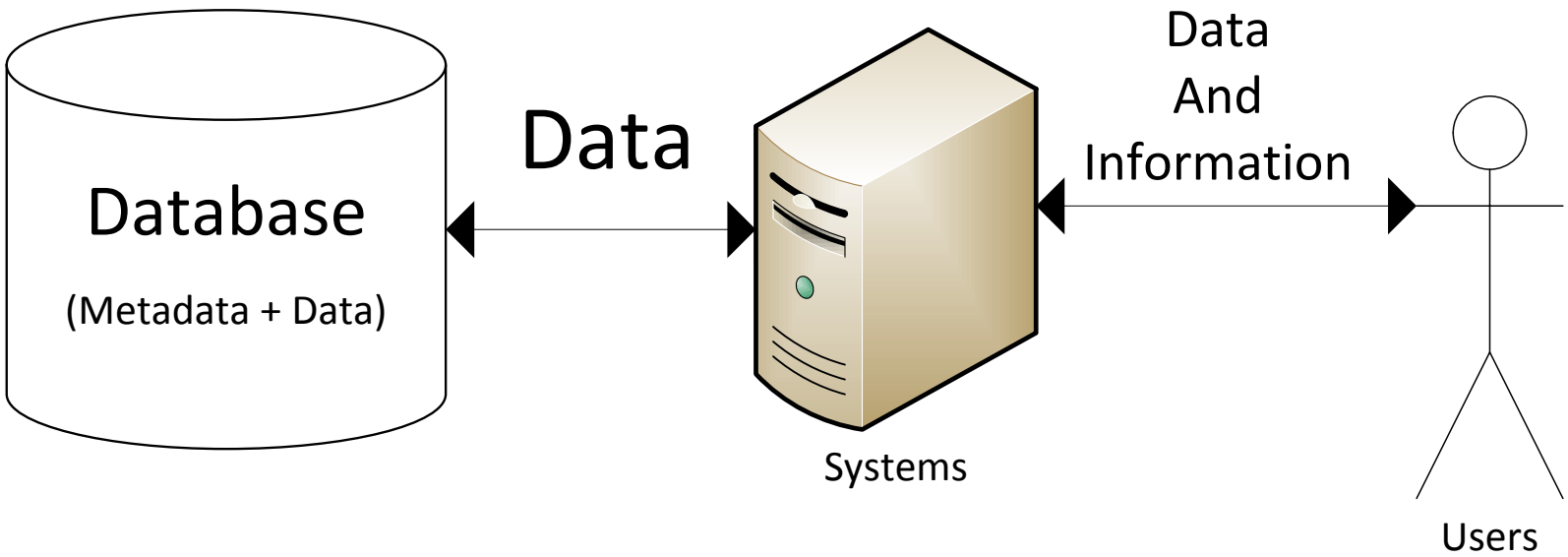
The External Data Model

Levels of Data Model Abstraction



Systems Use DBMS



Users use Systems

External Data Model

- People use software; software uses databases.
- The external model provides an interface between databases and the systems that use them.
- Provides mechanisms to control access to data.

SQL Programming Objects

Programming Objects

- Help provide access to and control of data
  - Enhanced data safety and security
  - Abstracts the data model from its use
- Views
- Stored procedures
- Functions

Views

Views

- A **VIEW** stores a predefined **SELECT** statement.
- Helps to provide easier access to more complicated internal models of data.
- Creating a VIEW:

```
CREATE VIEW viewname AS
       SELECT {colname [, ..n] | * }
       FROM tablename
       [JOIN tablename ON colPK = colFK ..n]
       [WHERE condition]
       [GROUP BY colname [, ..n]]
       [HAVING condition]
```

Views

First, code the SELECT statement.

```
SELECT
    MemberID
    , MemberLastName
    , MemberFirstName
    , MemberEmail
    , ClassName
    , COUNT(AttendanceID) as TimesAttended
FROM Members
INNER JOIN Attendance ON AttendanceMemberID = MemberID
INNER JOIN Classes ON ClassID = AttendanceClassID
GROUP BY
    MemberID
    , MemberLastName
    , MemberFirstName
    , MemberEmail
    , ClassName
```

Views

Then, add the CREATE clause.

```
CREATE VIEW MemberAttendanceCount AS
SELECT
    MemberID
    , MemberLastName
    , MemberFirstName
    , MemberEmail
    , ClassName
    , COUNT(AttendanceID) as TimesAttended
FROM Members
INNER JOIN Attendance ON AttendanceMemberID = MemberID
INNER JOIN Classes ON ClassID = AttendanceClassID
GROUP BY
    MemberID
    , MemberLastName
    , MemberFirstName
    , MemberEmail
    , ClassName
```

Views

SELECT from a view just like a table.

SQLQuery2.sql - CH...US\cahar_000 (52))* | SQLQuery1.sql - CH...US\cahar_000 (55))* | Object Explorer Details

```
1  SELECT * FROM MemberAttendanceCount
```

100 %

Results | Messages

	MemberID	MemberLastNa...	MemberFirstNa...	MemberEmail	ClassName	TimesAttend...
1	1	Sorum	Matt	Matt.Sorum@lookout.net	Pedal To The Metal	3
2	2	Alexander	Tim	Tim.Alexander@geemail.com	Pedal To The Metal	3
3	3	Benante	Charlie	Charlie.Benante@yaywho.com	Pedal To The Metal	3
4	4	Burr	Clive	Clive.Burr@lookout.net	Pedal To The Metal	3
5	8	Lander	Morgan	Morgan.Lander@geemail.com	Pedal To The Metal	2
6	9	Lander	Mercedes	Mercedes.Lander@lookout.net	Pedal To The Metal	2
7	10	McLeod	Tara	Tara.McLeod@yaywho.com	Pedal To The Metal	2
8	11	Doan	Trish	Trish.Doan@yaywho.com	Pedal To The Metal	2
9	1	Sorum	Matt	Matt.Sorum@lookout.net	Total Carnage	2
10	5	McBrain	Nikko	Nikko.McBrain@yaywho.com	Total Carnage	1
11	6	Davies	Adrienne	Adrienne.Davies@geemail.com	Total Carnage	1
12	7	Fish	Ginger	Ginger.Fish@yaywho.com	Total Carnage	2
13	8	Lander	Morgan	Morgan.Lander@geemail.com	Total Carnage	2
14	9	Lander	Mercedes	Mercedes.Lander@lookout.net	Total Carnage	2

Functions

Functions

- Can be called from within `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statements
- Scalar functions
  - Take 0 or more parameters
  - Return a single distinct value of a specified data type
- Table functions
  - Less common
  - Return tables like our full result sets

Built-In Functions

- Built-in functions can do many things.
  - `GetDate()`
    - Gets the current date
  - `Month('12/1/2015')`
    - Gets the month for the date 12/1/2015
  - `Year('12/1/2015')`
    - Gets the month for the date 12/1/2015
  - `DateDiff(dd, '10/1/2015', '10/31/2015')`
    - Calculates the difference in days between 10/1/2015 and 10/31/2015

User-Defined Functions

- Good news!
- We can write our own functions!
- Syntax template

```
CREATE FUNCTION functionname(
  @parameter1 AS datatype
  , .. n
) RETURNS datatype AS
BEGIN
  sqlstatement
  ..n
  RETURN expression
END
```

User-Defined Functions

- Good news!
- We can write our own functions!
- An example

```
/* Define the Function */
CREATE FUNCTION GetMarkup(@cost AS MONEY, @price AS MONEY)
RETURNS MONEY AS
BEGIN
  RETURN @price - @cost
END;

/* Call the Function */
SELECT dbo.GetMarkup(50, 75);
```

Stored Procedures

Stored Procedures

- Stored procedures allow for multiple SQL statements to be executed in one call.
- We can send parameters to the procedures to change the behavior.

```
CREATE PROCEDURE procedurename(
  @parameter1 AS datatype
  , .. n
) AS
BEGIN
  sqlstatement
  ..n
  [RETURN expression]
END
```

Stored Procedures

- Stored procedures allow for multiple SQL statements to be executed in one call.
- We can send parameters to the procedures to change the behavior.

```
CREATE PROCEDURE procedurename(
  @parameter1 AS datatype
  , .. n
) AS
BEGIN
  sqlstatement
  ..n
  [RETURN expression]
END
```

Variable and parameter names begin with @
To define their data type, we say "AS datatype"

Stored Procedures

- Stored procedures allow for multiple SQL statements to be executed in one call.
- We can send parameters to the procedures to change the behavior.

```
CREATE PROCEDURE procedurename (
  @parameter1 AS datatype
  , .. n
) AS
BEGIN
  sqlstatement
  ..n
  [RETURN expression]
END
```

Opening and closing parentheses surround the parameters.

Stored Procedures

- Stored procedures allow for multiple SQL statements to be executed in one call.
- We can send parameters to the procedures to change the behavior.

```
CREATE PROCEDURE procedurename(
  @parameter1 AS datatype
  , .. n
) AS
BEGIN
  sqlstatement
  ..n
  [RETURN expression]
END
```

Blocks of code must be contained within BEGIN and END statements.

Stored Procedures

- Write a procedure to add a product to our database.

```
CREATE PROCEDURE spAddProduct (
  @department varchar(20)
  , @name varchar(50)
  , @price money
  , @cost money
  , @vendor_id int
) AS
BEGIN
  INSERT INTO products (
    product_department, product_name, product_retail_price,
    product_wholesale_price, product_is_active, product_add_date,
    product_vendor_id
  ) VALUES (
    @department, @name, @price, @cost, 1, GETDATE(), @vendor_id
  )
  RETURN @@identity
END;
```

Stored Procedures

- Using our spAddProduct stored procedure

```
EXEC spAddProduct 'Hardware', 'Level', 10, 6, 2
```

Stored Procedures

- Using our spAddProduct stored procedure

```
EXEC spAddProduct 'Hardware', 'Level', 10, 6, 2
```

These values map to the parameter list in our CREATE PROCEDURE statement.

```
CREATE PROCEDURE spAddProduct (
  @department varchar(20)
  , @name varchar(50)
  , @price money
  , @cost money
  , @vendor_id int
) AS
BEGIN
  …
END;
```

Flow Control

- In programming, occasionally we need to make decisions that will affect how our code runs.
- To make these decisions, we use the **IF** statement.

```
IF conditional
BEGIN
   sql_statement_when_true;
   next sql statement...n
END
ELSE
BEGIN
   sql_statement_when_false;
   next sql statement ...n
END
```

Flow Control

- In programming, occasionally we need to make decisions that will affect how our code runs.
- To make these decisions, we use the **IF** statement.

```
DECLARE @dayofmonth AS INT;
SET @dayofmonth = day(getdate());
IF (@dayofmonth > 15)
BEGIN
  PRINT 'Later in the month';
END
BEGIN
  PRINT 'Earlier in the month';
END
```

Flow Control

- In programming, occasionally we need to make decisions that will affect how our code runs.
- To make these decisions, we use the **IF** statement.

```
DECLARE @dayofmonth AS INT;
SET @dayofmonth = day(getdate());
IF (@dayofmonth > 15)
BEGIN
  PRINT 'Later in the month';
END
BEGIN
  PRINT 'Earlier in the month';
END
```

Which of these lines of code will run today?

SQL Transactions

Transactions

- Transactions are units of work
- Useful when several statements need to execute successfully all at once (or none at all)
  - Insert into products table.
  - Insert into vendor products table.
  - If both succeed, commit the transaction.
  - If either fails, roll back both.

Transactions

- Well-structured transactions follow the ACID principle, which states that transactions should be:
  - Atomic
    - The transaction is the smallest unit of work it can be. It cannot be subdivided.
  - Consistent
    - When a transaction has finished, successfully or otherwise, all data must be left in a consistent state.
  - Isolated
    - Nothing the transaction does should be visible to any other processes until the transaction completes.
  - Durable
    - All changes to the database must be permanent.

```sql
IF EXISTS (SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME = 'Color')
BEGIN
    DROP TABLE Color
END
GO
CREATE TABLE Color (
    ColorID     int identity primary key
    , ColorDesc char(10) unique
)
GO

-- This transaction should succeed
BEGIN TRANSACTION
    DECLARE @rows as int
    SELECT @rows = COUNT(*) FROM Color
    INSERT INTO Color (ColorDesc) VALUES ('Red')
    INSERT INTO Color (ColorDesc) VALUES ('Blue')
    SELECT * FROM Color
    IF @@ROWCOUNT - @rows =2
        COMMIT TRANSACTION
    ELSE
        ROLLBACK TRANSACTION

-- This transaction should fail
BEGIN TRANSACTION
    SELECT @rows = COUNT(*) FROM Color
    INSERT INTO Color (ColorDesc) VALUES ('Red')
    INSERT INTO Color (ColorDesc) VALUES ('Purple')
    SELECT * FROM Color
    IF @@ROWCOUNT - @rows =2
        COMMIT TRANSACTION
    ELSE
    ROLLBACK TRANSACTION

SELECT * FROM Color
```

Results | Messages

	ColorID	ColorDe...
1	2	Blue
2	1	Red

	ColorID	ColorDe...
1	2	Blue
2	4	Purple
3	1	Red

	ColorID	ColorDe...
1	2	Blue
2	1	Red