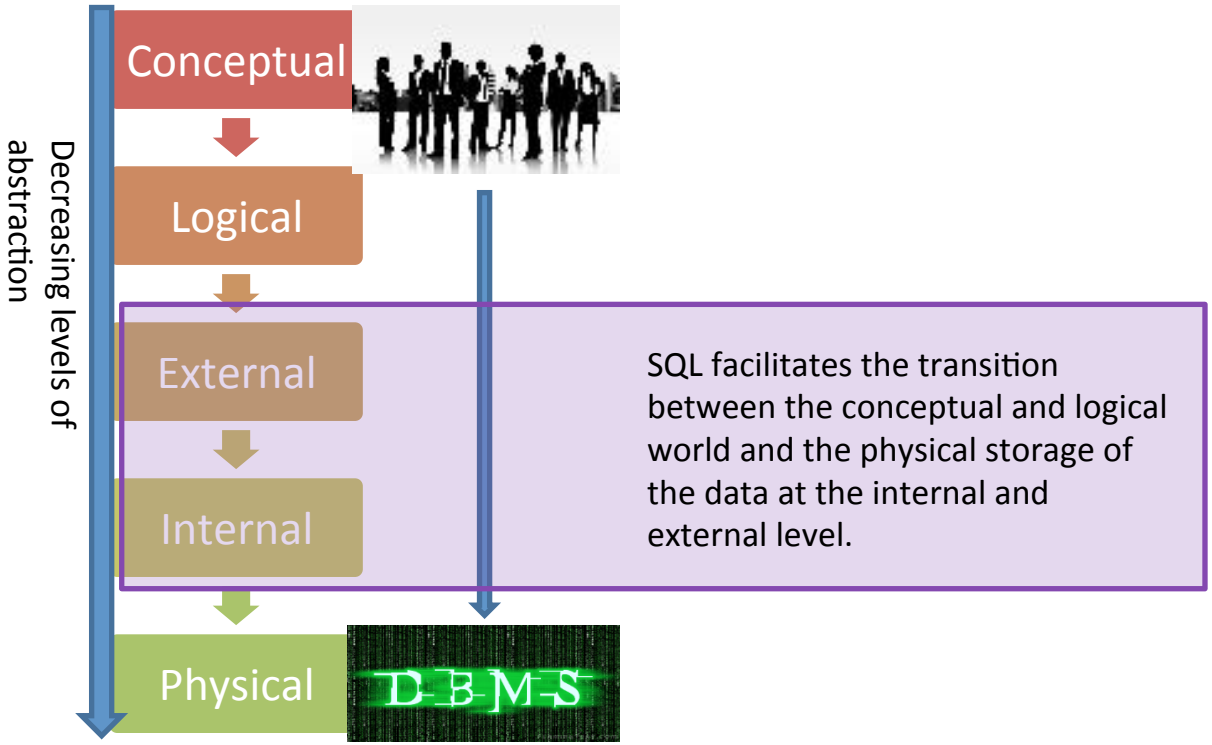
The Internal Model

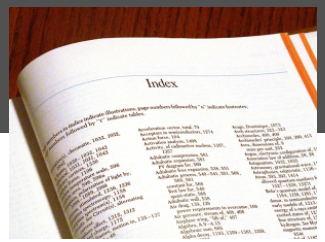
Levels of Data Model Abstraction



Internal Model

- Provides directives to the physical domain on how the data are to be organized.
  - Indexes
  - Clustering data to keep rows of frequently joined tables near one another
  - Maintain statistics about tables and their indexes

Indexes



- Indexes provide a way for the DBMS to quickly access rows of data without having to do full table scans. There are two types, clustered and nonclustered.
- Clustered indexes tell the DBMS the order in which rows should be stored. Each table can only have one clustered index (usually the primary key).
- Nonclustered indexes are separate objects that contain sorted pointers to the main table. There can be up to 249 nonclustered indexes.
- Note: Indexes add performance overhead. Use them wisely.

Primary Keys as Indexes

- A primary key adds a clustered index unless you tell it to be nonclustered.
- Choose primary keys wisely. (Use the identity property.)

SQL INSERT Statement

INSERT INTO Syntax

```
INSERT INTO table_name
    (column1,column2,column3,...)
VALUES
    (value1,value2,value3,...)
```

- Example

```
INSERT INTO Customer
    (CustName, Address, City, State, PC)
VALUES
    ('XYZ Company', '123 Main St', 'Anytown', 'PA', '23456')
```

Add Data to students_years

```
insert into student_years (student_year, student_year_sort)
   values ('Unknown', '0');

insert into student_years (student_year, student_year_sort)
   values ('Freshman', '1');

insert into student_years (student_year, student_year_sort)
   values ('Sophomore', '2');

insert into student_years (student_year, student_year_sort)
   values ('Junior', '3');

insert into student_years (student_year, student_year_sort)
   values ('Senior', '4');
```

Add Data to students

```
insert into students
(student_name, student_email, student_gpa, student_year,
   student_is_ischool, student_dob)
   values
('Dinah
   Sores','dsores@syr.edu','3.4','Freshman','1','12/5/1982
   ');

insert into students
(student_name, student_email, student_gpa, student_year,
   student_is_ischool, student_dob)
   values
('Ella
   Mentry','ementry@syr.edu','3.7','Junior','1','1/15/1981
   ');
```

Time Permitting . . . The SELECT

Show all rows and columns from the students table.

```
SELECT * FROM students
```

The SQL SELECT Statement

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

- The basic **SELECT** statement is a simple structure but can be as complicated as the request and the database you're querying.
- To the left is the syntax template for a **SELECT** statement.
- Let's go through it.
- Our **data question** is:
  - "Which **customers** are in which **cities** in **New York State**?"
  - Assume we have a table called Customer and we need the CustomerName and CustomerCity columns. We'll need CustomerState as well.

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT
```

- We start with the word SELECT.

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT CustomerName, CustomerCity
```

- Next we specify which column names we wish to retrieve from the database. This syntax template reads like this:
  - "Either list one or more **column names**, each separated by a comma, or use a *** for all column names**."
- We need to see the customer's name and city, so we specify those columns after the SELECT keyword.

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT CustomerName, CustomerCity
FROM Customer
```

- The keyword `FROM` indicates we are about to start our list. In what table is our data stored?
- We specify that table's name in the place where it says "`tablename`".
- Our customer data is stored in a table called Customer, so we specify that here.

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT CustomerName, CustomerCity
FROM Customer
WHERE
```

- The `WHERE` clause, represented by `[WHERE condition]` in the template, is optional. But we need to filter our data set by state (we only want NY customers), so let's build it.
- First, we start with the keyword `WHERE`.
- Now we need a condition.

The SQL SELECT Statement

- Conditionals
  - Compare two or more terms for equivalence. Used in the `WHERE` clause to filter rows in the result set.
  - Compare two or more terms using logical operators.

The SQL SELECT Statement

SQL Operators (WHERE Clause Fun)		
Logical Operator	**Example Syntax**	**Description (Say it like…)**
=	`A = B`	Equal (“A is equal to B”)
<>	`A <> B`	Not equal (“A is not equal to B”)
>	`A > B`	Greater than (“A is greater than B”)
<	`A < B`	Less than (“A is less than B”)
>=	`A >= B`	Greater than or equal to (“A is greater than or equal to B”)
<=	`A <= B`	Less than or equal to (“A is less than or equal to B”)
BETWEEN	`A BETWEEN B AND C`	Between two values (“A is between B and C”)
LIKE	`A LIKE B`	Matches a pattern (“A matches the pattern B”)
IN	`A IN (1, 2, 3, 4, 9)`	The predicate is in the list provided (“A is in the list 1, 2, 3, 4, or 9”)
AND	`A = B AND C = D`	Both conditions must be true (Both A equals B and C equals D)
OR	`A = B OR C = D`	One of the two conditions must be true (Either A equals B or C equals D)
( )	`((A = 5) OR A = B) AND B = C`	Use parentheses to group terms! (Either A equals 5 or A equals B, only if B equals C)

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT CustomerName, CustomerCity
FROM Customer
WHERE CustomerState = 'NY'
```

- We need to specify the condition that state is “NY”.
- Referencing our table of operators, we know that = means “equal to”.
- Our field is CustomerState and it must be equal to “NY”.
- Let’s run it!

The SQL SELECT Statement

```
SELECT CustomerName, CustomerCity
FROM Customer
WHERE CustomerState = 'NY'
```

100 %

Results | Messages

	CustomerName	CustomerCity
1	Acme, inc.	Watertown
2	Widget Corp	Troy
3	123 Warehousing	Schenectady
4	Demo Company	Olean
5	Smith and Co.	Amsterdam
6	Foo Bars	Port Jervis
7	ABC Telecom	Batavia
8	Fake Brothers	Amsterdam
9	QWERTY Logistics	Johnstown
10	Demo, inc.	Beacon
11	Sample Company	Port Jervis
12	Sample, inc	Utica
13	Acme Corp	Oneonta
14	Allied Biscuit	Buffalo

- It works! But it's not very orderly.
- The cities and companies are all over the place. Wouldn't it be nice to see them in some easily viewable order?
- Let's fix that by ORDERing them by city then name.

The SQL SELECT Statement

```
SELECT {colname [, ..n] | * }
FROM tablename
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

So far we have:

```
SELECT CustomerName, CustomerCity
FROM Customer
WHERE CustomerState = 'NY'
ORDER BY CustomerCity, CustomerName
```

- ORDER BY is an optional term, but it's helpful for making the data more usable.
- We start with the ORDER BY keywords and then list the column names in the order we want them sorted (separated by a comma).
- DESC is an optional keyword that will sort the results in reverse (descending) order (Z to A instead of A to Z). We don't need it here, so we omit DESC.
- Let's run it!

The SQL SELECT Statement

```sql
SELECT CustomerName, CustomerCity
FROM Customer
WHERE CustomerState = 'NY'
ORDER BY CustomerCity, CustomerName
```

100 %

Results | Messages

	CustomerName	CustomerCity
1	Spade and Archer	Albany
2	Big T Burgers and Fries	Amsterdam
3	Fake Brothers	Amsterdam
4	General Forge and Foundry	Amsterdam
5	Initrode	Amsterdam
6	Smith and Co.	Amsterdam
7	The Legitimate Businessmens Club	Amsterdam
8	Chasers	Auburn
9	Sombra Corporation	Auburn
10	ABC Telecom	Batavia
11	General Products	Batavia
12	Sirius Cybernetics Corporation	Batavia
13	Burleigh and Stronginthearm	Beacon

- **BOOM**! Now our rows are sorted by city name first, then by company name within those cities.

FROM Clause (with JOINs)

FROM Clause

- In a properly normalized database, related data will exist in more than one table.
- In order to effectively run queries against these tables, we need to JOIN them in the FROM clause.

JOIN

```
SELECT
    product_vendor_id
    , product_name
    , product_retail_price
    , product_wholesale_price
FROM fudgemart_products
ORDER BY product_vendor_id
```

Results | Messages

	product_vendor_id	product_name	product_retail_price	product_w
1	1	DVD Player	45.00	30.00
2	1	HD-DVD Player	150.00	100.00
3	1	Blu-Ray DVD Player	150.00	100.00
4	1	[illegible]	[illegible]	[illegible]
5	1	50" LCD HD TV	1300.00	1100.00
6	1	65" LCD HD TV	1900.00	1700.00
7	2	Dri-Fit Tee	20.00	8.00
8	2	Running Pants	35.00	12.00
9	2	Wool Socks	8.00	2.00
10	2	Squeaky Speaks	65.00	20.00

```
SELECT
    *
FROM fudgemart_vendors
```

	vendor_id	vendor_name	vendor_phone	vendor_web
1	1	Soney	555-2939	http://www
2	2	Mikey	555-2870	http://mike
3	3	Stanlee	555-9920	NULL
4	5	Mikerosoft	555-2220	http://www
5	6	Fudgeman	555-1239	http://www
6	7	Leaveeyes	555-2931	NULL
7	8	Weebock	555-0002	http://www
8	9	Fudgeoco	555-0232	http://www
9	10	Blackened-Deckhand	555-9922	NULL

JOIN

- Syntax template

```
SELECT {colname [, ..n] | * }
FROM tablename
[[LEFT | RIGHT | INNER | FULL OUTER] JOIN table_name ON colname =
colname] ..n]
[WHERE condition]
[ORDER BY col [DESC] [, ..n]]
```

JOIN

```
SELECT vendor_name, product_name,
product_retail_price,
product_wholesale_price,
product_retail_price -
product_wholesale_price as
product_markup
FROM fudgemart_vendors
JOIN fudgemart_products ON
   vendor_id = product_vendor_id
ORDER BY vendor_name
```

Part of `fudgemart_vendors`

Part of `fudgemart_products`

JOIN

```sql
SELECT
    vendor_name
    , product_name
    , product_retail_price
    , product_wholesale_price
    , product_retail_price - product_wholesale_price as product_markup
FROM fudgemart_vendors
JOIN fudgemart_products ON vendor_id = product_vendor_id
ORDER BY vendor_name
```

100 %

Results | Messages

	vendor_name	product_name	product_retail_price	product_wholesale_price	product_markup
1	Blackened-Deckhand	19.2v Drill Driver Set	90.00	45.00	45.00
2	Blackened-Deckhand	10" Miter Saw	200.00	140.00	60.00
3	Blackened-Deckhand	Lazer Level	45.00	25.00	20.00
4	Blackened-Deckhand	Table Saw	290.00	180.00	110.00
5	Blackened-Deckhand	Power Washer	290.00	180.00	110.00
6	Blackened-Deckhand	Belt Sander	250.00	180.00	70.00
7	Blackened-Deckhand	Crock Pot	25.00	10.00	15.00
8	Blackened-Deckhand	Monsignor Coffee	20.00	10.00	10.00
9	Blackened-Deckhand	Electric Griddle	20.00	10.00	10.00
10	Blackened-Deckhand	Steam Iron	15.00	5.00	10.00
11	Blackened-Deckhand	Blender	45.00	20.00	25.00
12	Leaveeyes	Cool Jeans	45.00	10.00	35.00
13	Leaveeyes	Denim Jacket	60.00	15.00	45.00

The SQL UPDATE Statement

To Change Data, We Use UPDATE

SQLQuery7.sql - CH...US\cahar_000 (55))* | SQLQuery6.sql - CH...US\cahar_000 (54))*

```sql
select
    product_id
    , product_department
    , product_name
    , product_vendor_id
from fudgemart_products
where product_vendor_id = 3
```

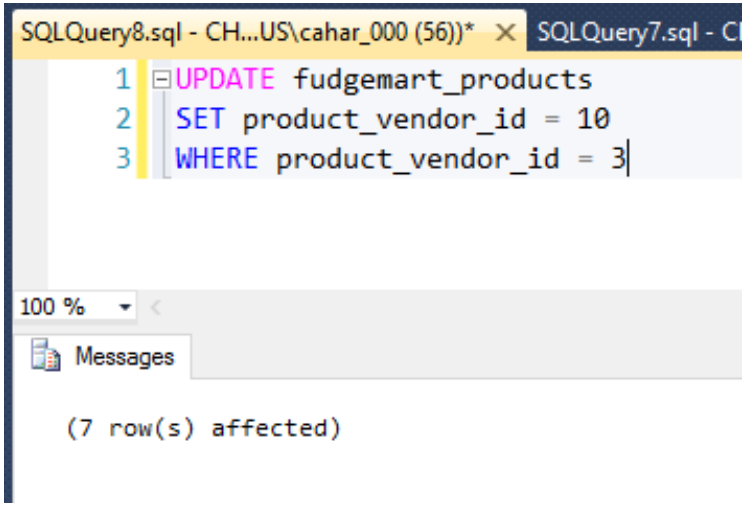
100 %

Results | Messages

	product_id	product_department	product_name	product_vendor_id
1	3	Hardware	Rip Claw Hammer	3
2	12	Clothing	Work Pants	3
3	13	Clothing	Work Gloves	3
4	29	Hardware	18v Drill Driver Set	3
5	35	Hardware	Cold Chisel Set	3
6	36	Hardware	Screwdriver Set	3
7	37	Hardware	Drill Bit Set	3

UPDATE

```
UPDATE table_name
SET field_name1 = value1 [, field_name2 = value2 ..n]
[WHERE condition]
```



Change Is Permanent and Immediate

SQLQuery8.sql - CH...US\cahar_000 (56))* SQLQuery7.sql - CH...US\cahar_000

```sql
select
    product_id
    , product_department
    , product_name
    , product_vendor_id
from fudgemart_products
where product_vendor_id = 3
```

100 %

Results | Messages

product_id	product_department	product_name	product_vendor_id

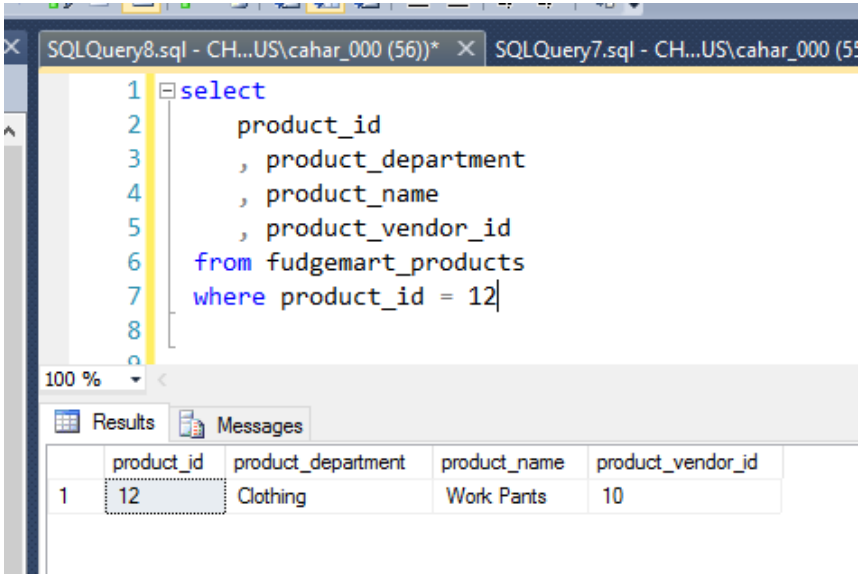
Caution!

- Write the WHERE clause first.
- There is NO undo option.
- There is NO audit trail.
- Consider putting the UPDATE queries in stored procedures.

The SQL DELETE Statement

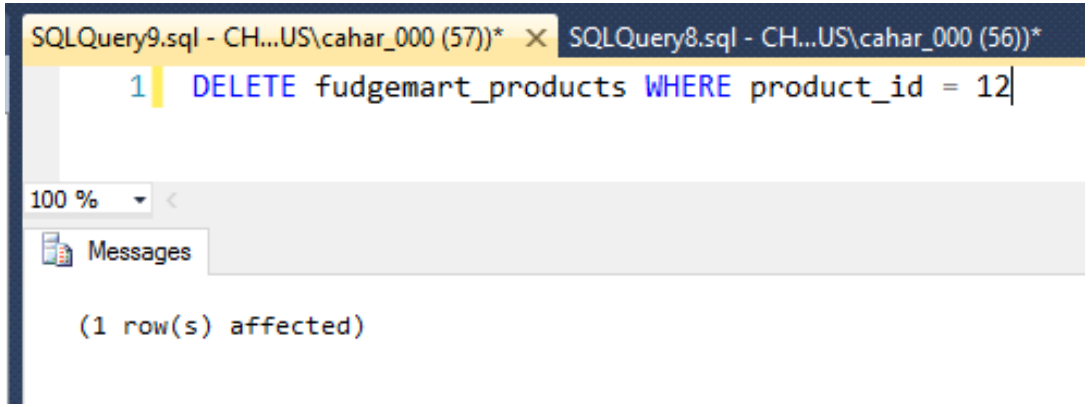
DELETE

Removing data from the database requires a DELETE statement.



DELETE

```
DELETE table_name [WHERE condition]
```



Change Is Permanent and Immediate

SQLQuery9.sql - CH...US\cahar_000 (57))* SQLQuery8.sql - CH...US\cahar_000 (

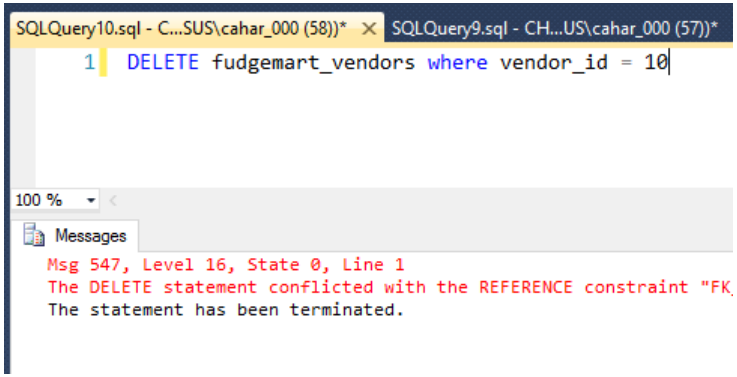
```sql
select
    product_id
    , product_department
    , product_name
    , product_vendor_id
from fudgemart_products
where product_id = 12
```

100 %

Results | Messages

product_id	product_department	product_name	product_vendor_id

Can’t DELETE If Dependencies Exist




```
Msg 547, Level 16, State 0, Line 1
The DELETE statement conflicted with the REFERENCE constraint
"FK_fudgemart_products_fudgemart_vendors1". The conflict occurred
in database “fudgmart", table "dbo.fudgemart_products", column
'product_vendor_id'.
The statement has been terminated.
```

Caution!

- Consider writing your WHERE clause first.
- There is NO undo option.
- There is NO audit trail.
- Consider encapsulating your DELETEs in stored procedures.