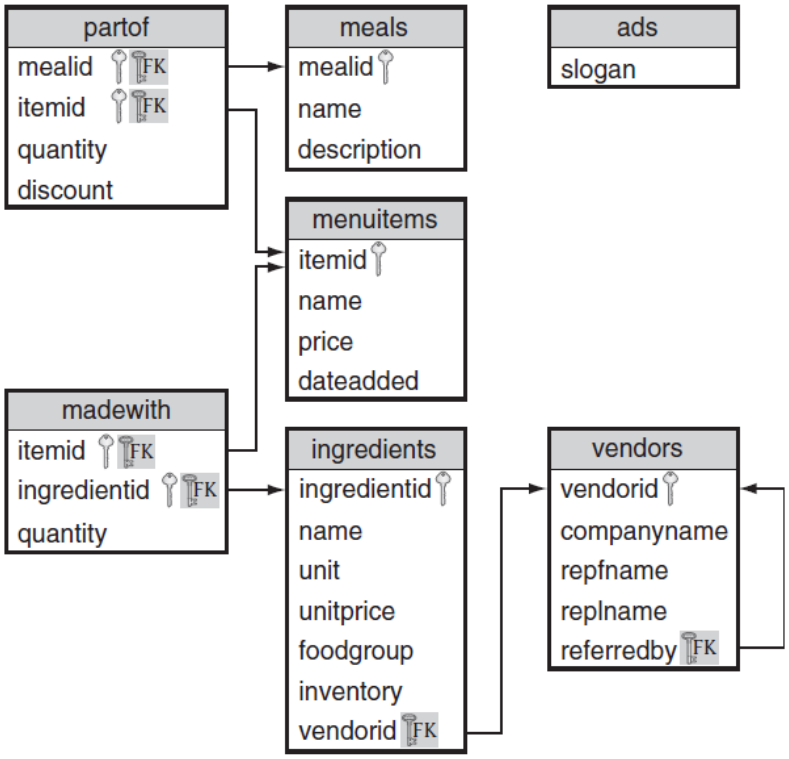
Birla Institute of Technology and Science, Pilani.
Database Systems
Lab No #6

Views:

- We will learn about views, transactions and practice them on a schema of a restaurant chain.
- The table schema is given below.



ABOUT VIEWS

- A view is a virtual table defined by a query. It provides a mechanism to create alternate ways of working with the data in a database. A view acts much like a table. We can query it with a SELECT, and some views even allow INSERT, UPDATE, and DELETE. There are several uses for a view.
- **Usability**—We can use a view as a wrapper around very complex SELECT statements to make our system more usable.
- **Security**—If we need to restrict the data a user can access, we can create a view containing only the permitted data. The user is given access to the view instead of the base table(s).
- **Reduced Dependency**—The database schema evolves over time as our enterprise changes. Such changes can break existing applications that expect a certain set of tables with certain columns. We can fix this by having our applications access views rather than base tables. When the base tables change, existing applications still work as long as the views are correct.

CREATING VIEWS

```
CREATE VIEW <view name> [(<column list>)] AS <SELECT statement>
```

- This creates a view named <view name>. The column names/types and data for the view are determined by the result table derived by executing <SELECT statement>. Optionally, we can specify the column names of the view in <column list>. The number of columns in <column list> must match the number of columns in the <SELECT statement>.

Now let us see an example, try out the following query which creates a view called vrs showing the ingredients (ingredient ID,name, inventory, and inventory value) supplied to us by Veggies_R_Us.

```
CREATE VIEW vrs AS
SELECT ingredientid, name, inventory, inventory * unitprice AS value
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Veggies_R_Us';
```

SELECT FROM VIEWS:

- Selecting from views is just similar to selecting from a table. View is just a virtual table.

```
SELECT * from vrs;
```

- The above query selects all rows from the view vrs. We can also give WHERE conditions as in tables.

Find all ingredients provided by Veggies_R_Us with an inventory of more than 100?

```
SELECT name
FROM vrs
WHERE inventory > 100;
```

- Notice that the above query gives the same result when we use the original tables in the query with out using the view.

```
SELECT name
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Veggies_R_Us'
AND inventory > 100;
```

- Because the view is just a virtual table, any changes to the base tables are instantly reflected in the view data.
- See the following query updates the inventory and these changes are reflected the view vrs.

```
UPDATE ingredients
SET inventory = inventory * 2
WHERE ingredientid = 'TOMTO';
```

- The inventory is updated in the view

```
SELECT * from vrs;
```

- Now, lets see another example which shows how to create view on a complex query .

Let's create a new view called menuitems that lists all of the items we have for sale, including meals and items, and how much they cost.

```
CREATE VIEW menuitems (menuitemid, name, price) AS
(SELECT m.mealid, m.name, CAST(SUM(price * (1-discount)) AS NUMERIC(5,2))
FROM meals m LEFT OUTER JOIN partof p ON m.mealid = p.mealid
LEFT OUTER JOIN items i ON p.itemid = i.itemid
GROUP BY m.mealid, m.name)
UNION
(SELECT itemid, name, price
FROM items);
```

- After creating the view , we can easily list our menu items.

Find all menu items

```
SELECT * from menuitems;
```

Find the most expensive menu item

```
SELECT name
FROM menuitems
WHERE price =
(SELECT MAX(price)
FROM menuitems);
```

Find the priceless menu items

```
SELECT COUNT(*)
FROM menuitems
WHERE price IS NULL;
```

- Notice that these queries would have been more complex without creating views

Restrictions on DML operations for views

- Restrictions on DML operations for views use the following criteria in the order listed:

- If a view is defined by a query that contains `SET` or `DISTINCT` operators, a `GROUP BY` clause, or a group function, then rows cannot be inserted into, updated in, or deleted from the base tables using the view.
- If a view is defined with `WITH CHECK OPTION`, a row cannot be inserted into, or updated in, the base table (using the view), if the view cannot select the row from the base table.
- If a `NOT NULL` column that does not have a `DEFAULT` clause is omitted from the view, then a row cannot be inserted into the base table using the view.
- If the view was created by using an expression, such as `DECODE(deptno, 10, "SALES", ...)`, then rows cannot be inserted into or updated in the base table using the view.

Updating a Join View

- An updatable join view (also referred to as a modifiable join view) is a view that contains more than one table in the top-level `FROM` clause of the `SELECT` statement, and is not restricted by the `WITH READ ONLY` clause.
- The rules for updatable join views are shown in the following table. Views that meet these criteria are said to be inherently updatable.

Rule	Description
General Rule	Any `INSERT`, `UPDATE`, or `DELETE` operation on a join view can modify only one underlying base table at a time.
`UPDATE` Rule	All updatable columns of a join view must map to columns of a key-preserved table*. If the view is defined with the `WITH CHECK OPTION` clause, then all join columns and all columns of repeated tables are not updatable.
`DELETE` Rule	Rows from a join view can be deleted as long as there is exactly one key-preserved table in the join. The key preserved table can be repeated in the `FROM` clause. If the view is defined with the `WITH CHECK OPTION` clause and the key preserved table is repeated, then the rows cannot be deleted from the view.
`INSERT` Rule	An `INSERT` statement must not explicitly or implicitly refer to the columns of a non-key-preserved table. If the join view is defined with the `WITH CHECK OPTION` clause, `INSERT` statements are not permitted.

* The concept of a key-preserved table is fundamental to understanding the restrictions on modifying join views. A table is key-preserved if every key of the table can also be a key of the result of the join. So, a key-preserved table has its keys preserved through a join.

- Finally, even if a view is updatable, not all columns within the view may be updatable. For example, derived columns such as *value* in the *vrs* cannot be updated. Thus, our *menuitems* view cannot be directly updated because it contains attributes from the *items, meals,* and *partof* tables. The *vrs* view can be updated as if it were a base table

```
UPDATE vrs SET inventory = inventory * 2;
SELECT * FROM vrs;
```

- It is important to note that updates through views can have unexpected consequences, depending on the behavior of a DBMS. For example, a DBMS might allow an INSERT on vrs,

such as the following insert. The underlying ingredients table would be updated with the provided values, but the vendorid would be set to the default value (in our case, NULL).

```
INSERT INTO vrs(ingredientid, name, inventory) VALUES
'NEWIN','New ingredient',100);


SELECT * FROM vrs;
```

- Observe the result. Is there anything wrong it? If it is Why? Because the query specification for vrs requires the vendorid to be VGRUS, our new row does NOT appear in the view.

- In general, updates through views work best when the view is defined as a subset of a table and all attributes that determine if a row is in a view are updatable.

- **Note:** the update operations supported on views are generally specific to the SQL vendor.

ALTER A VIEW:

- Altering view can also be done by dropping it and recreating it. but that would drop the associated granted permissions on that view. By using alter clause, the permissions are preserved.

```
ALTER VIEW [ schema_name . ] view_name [ ( column [ ,...n ] ) ]
[ WITH <view_attribute> [ ,...n ] ]
AS select_statement
[ WITH CHECK OPTION ] [ ; ]
<view_attribute> ::=
{
    [ ENCRYPTION ]
    [ SCHEMABINDING ]
    [ VIEW_METADATA ]
}
```

To alter the view vrs to display the ingredients supplied by 'Spring Water Supply'.

```
CREATE OR REPLACE VIEW vrs AS
SELECT ingredientid, name, inventory, inventory * unitprice AS value
FROM ingredients i, vendors v
WHERE i.vendorid = v.vendorid AND companyname = 'Spring Water Supply';
```

DROP A VIEW:

```
DROP VIEW <view name> [CASCADE | RESTRICT];
```

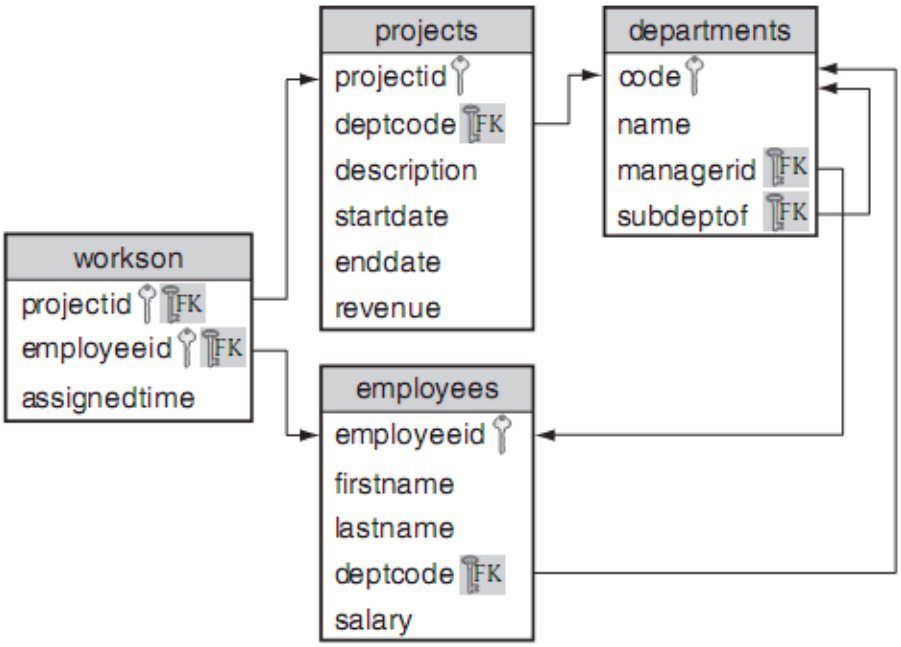
- This removes the specified view; however, it does not change any of the data in the database. SQL does not allow a view to be dropped if view is contained in the SELECT statement of another view. This is default behavior and is equivalent to the effect of using RESTRICT Option. If we want to drop a view which is being used in a select command we must use the CASCADE option.

```
drop VIEW vrs;
```

- This drops the view vrs if it exists.

EXERCISES ON EMPLOYEES DATABASE:

- Write a SQL query for each of the following based on employee database created in the previous labs. The scheme for employee database is shown below.



1. Create a view containing all of the employees assigned to the 'Robotic Spouse' project. Include the percent time they are assigned to the project.
2. Query your view created in the previous question to find the employee first and last name with the greatest amount of time assigned to 'Robotic Spouse'.
3. Create a view of employees with their department name.
4. Query your view to find all the first and last names of employees in the Consulting department.
5. Create a view showing all of the projects assigned to Abe Advice, including his percentage time on each project.
6. Query your view to find the total amount of time Abe is assigned to projects.
7. Create an updatable view showing employees and their salaries. Give everyone a 10% raise by updating the view

DERIVED ATTRIBUTES

Data can be somewhat raw. SQL provides the ability to create attributes in our result table that are derived using operations and functions over existing attributes and literals. The default column name of a derived attribute is system dependent; however, a name can be assigned using a column alias.

❖ Numeric

SQL can evaluate simple arithmetic expressions containing numeric columns and literals. Following table shows the SQL arithmetic operators in precedence order from highest to lowest. Unary +/- have the highest precedence. Multiplication and division have the next highest precedence. Addition and subtraction have the lowest precedence. Operators with the same precedence are executed left to right. We can control the order of evaluation using parentheses. SQL evaluates an expression for each row in the table specified in the FROM clause that satisfies the condition in the WHERE clause. Let's look at an example.

Find the value of your pickle inventory if you double your stock of pickles

```
SELECT ingredientid, inventory * 2 * unitprice AS "Inventory Value"
FROM ingredients
WHERE name = 'Pickle';
```

Operator	Returns	Example	Precedence
+*numexp* –*numexp*	*numexp* with sign unchanged *numexp* with negated sign	+inventory –3	Highest
lnumexp * *rnumexp* *lnumexp* / *rnumexp*	Product of *lnumexp* and *rnumexp* Division of *lnumexp* by *rnumexp*	inventory * unitprice * quantity / 3	Middle
lnumexp + *rnumexp* *lnumexp* – *rnumexp*	Sum of *lnumexp* and *rnumexp* Difference of *lnumexp* and *rnumexp*	inventory + 10 discount – 0.1	Lowest

How does this work? For each row in the *ingredients* table that satisfies the WHERE condition, SQL computes the value of the expression(s) in the attribute list. Let's try another example.*Arithmetic literal example*

```
SELECT 5 - 4 + 8 / 4 * 2 AS "Example Equation"
FROM vendors
WHERE referredby IS NOT NULL;
```

Why does the same value show up so many times in Query? There are 5 rows in the *vendors* table that satisfy the predicate `referredby IS NOT NULL`. SQL evaluates the expression for *each* of these

rows and reports the results. Note that the expression itself is evaluated according to operator precedence, not simply from left to right.
SQL also includes many standard mathematic functions. Table below contains some of the more common functions. The exact set of available functions is DBMS dependent. In fact, your DBMS will likely have additional functions.

Function	Returns	Example
ABS(N)	Absolute value of N	ABS(inventory – 100)
CEIL[ING](N)	Ceiling of N	CEILING(inventory/10)
EXP(N)	eN	EXP(5)
FLOOR(N)	Floor of N	FLOOR(inventory/10)
LN(N)	Natural log of N	LN(5)
MOD(N, D)	Remainder of N divided by D	MOD(11, 3)
POWER(B, E)	B to the power of E (BE)	POWER(2, 3)
SQRT(N)	√N	SQRT(4)

What about the infamous NULL? An arithmetic expression evaluated with NULL for any value returns NULL. Arithmetic functions given a NULL parameter value return NULL, as in following query.

Find the inventory value of each ingredient in both dollars and euros

```
SELECT name, inventory * unitprice AS Dollars,
CEIL(inventory * unitprice * 1.2552) AS euros, 1.2552 AS "Exchange Rate"
FROM ingredients;
```

We compute our inventory value in both U.S. dollars and euros. To hedge our bets against fluctuations in the exchange rate, we take the ceiling of the European currency value. As you can see in the example query results, if either inventory or unitprice is NULL, their product is NULL. In the case of a NULL product of inventory and unitprice, the ceiling function also returns NULL. We name our computed columns using column aliases.

❖ Character String

The typical database is full of character strings, such as names, addresses, and ingredients. The string you really want may be a combination of data strings, substrings, string literals, and so on. Perhaps you want to generate address labels or salutations (e.g., “Dear first name last name,”). SQL provides a wide range of mechanisms for combining and manipulating character strings.

Concatenating Strings With ||

We begin with the || (concatenation) operator to construct a new string from a combination of string expressions.

Create a mailing label for each store

```
SELECT manager, address || ' ' || city || ' ' || state || ' ' || zip
|| ' USA' as mail
FROM stores;
```

There are several things worth noting from above Query:

1. You may use string literals in the SELECT list either by themselves or in concatenations.
2. The concatenation operator can take both literals and columns. In fact, it takes any expression that returns (or can be coerced to) a string.
3. Concatenation with NULL is always NULL.
4. String concatenation does not add a space. Any spacing must be explicitly added.
5. Because strings of type CHAR are padded with trailing blanks, concatenation with such strings includes these extra blanks. See the spaces between the zip code and USA in some of the rows.
6. VARCHARs are not padded with trailing spaces; they only include the explicitly specified characters.

SUBSTRING: Getting the String within the String

```
SUBSTRING(<source> FROM <start> [FOR <length>])
```

SUBSTRING returns the substring from <source> string, starting from the character at position <start> (numbering from 1) and containing up to <length> characters. The <source> may be any string expression. <start> and <length> may be any integer expression. If <length> is not specified, SUBSTRING returns all characters from <start> to the end of <source>. If the <source> string is empty or if the <start> position is beyond the end of the <source> string, SUBSTRING returns an empty string (length of 0). If any of the parameters are NULL, SUBSTRING returns NULL.

SUBSTRING example

```
SELECT SUBSTRING(repfname FROM 1 FOR 1) || '. ' || replname AS name
FROM vendors;
```

The SUBSTR Function

This function is a commonly used function that extracts a set of characters from a string. The extraction is started at a specific position. The function has three parameters:

- String Name
- Extraction start position
- Numbers of characters to extract

```
substr(string, start_position, length)
```

SUBSTR example

```
SELECT substr('SAFETY',2,4) "Substring" from dual;
```

TRIM: Removing Unwanted Leading and Trailing Characters

```
TRIM([[LEADING | TRAILING | BOTH] [<trim characters>] FROM] <source>)
```

TRIM returns the <source> after removing the longest substring of leading and/or trailing sequences of <trim characters>. The <source> and <trim characters> can be any string expression. If LEADING, TRAILING, or BOTH is not specified, the default is BOTH. If <trim characters> is not specified, the default is a single space. If any of the parameters are NULL, TRIM returns NULL.

TRIM example

```
SELECT DISTINCT ingredientid, foodgroup || '.' AS "with trailing",
TRIM(TRAILING ' ' FROM foodgroup) || '.' AS "without trailing"
FROM ingredients
WHERE inventory > 500;
```

The LTRIM Function

This function is used to remove characters from the left side of a character string, and it does so until it encounters a target string character that does not exist in the target set of characters.

```
ltrim(char,set)
```

LTRIM example

```
SELECT ltrim('NISHANT','N') "LTRIM" from dual;
```

The RTRIM Function

This function is used to trim the characters from the right side of a value. This function evaluates string characters from left to right. It removes the characters if they match a character in the target set. Removal is stopped when a character that does not exist in the target list of characters is encountered.

```
rtrim(char,set)
```

RTRIM example

```
SELECT rtrim('RAMESHA','A') "RTRIM" from dual;
```

UPPER and LOWER: Controlling Character Case

```
LOWER(<source>)
UPPER(<source>)
```

LOWER returns the <source> string with all alphabetic characters changed to lowercase. UPPER returns the <source> string with all alphabetic characters changed to uppercase. The <source> can be any string expression. LOWER and UPPER do not change nonalphabetic characters. If <source> is NULL, UPPER and LOWER return NULL.

UPPER and LOWER example

```
SELECT UPPER(repfname || ' ' || replname) AS rep, LOWER(companyname) AS
company
FROM vendors WHERE referredby = 'VGRUS';
```

The INITCAP Function

The INITCAP function is used to modify characters values. This function capitalizes the first letter of each set of characters in a character string. The characters following the capitalized characters are made lowercase.

```
Initcap(char)
```

INITCAP example

```
SELECT initcap('GEORGE BUSH') "Title" from dual;
```

POSITION: Finding Where a Substring Begins

```
POSITION(<substring> IN <source>)
```

POSITION returns a number representing the character position (numbering from 1) of the first character of the first occurrence of <substring> in <source>. The <source> and the <substring> can be any string expression. If <substring> does not appear in <source>, POSITION returns 0. If <substring> or <source> are NULL, POSITION returns NULL.

POSITION example

```
SELECT name, POSITION('Salad' IN name)
```

```
FROM items;
```

The INSTR Function

The INSTR function returns a numeric value that represents the position of a specified characters residing within a string. The function contains two parameters: The first is the string, and the second is the target set of characters.

INSTR example

```
SELECT instr('george bush','u') "instr" from dual;
```

CHAR[ACTER]_LENGTH: Counting Characters in a String

```
CHARACTER_LENGTH(<source>)
```

CHARACTER_LENGTH returns the number of characters in <source>. <source> can be any string expression. If <source> is NULL, CHARACTER_LENGTH returns NULL. CHARACTER_LENGTH may be abbreviated as CHAR_LENGTH.

CHARACTER_LENGTH example

```
SELECT name, CHAR_LENGTH(name) AS namelen, CHAR_LENGTH(foodgroup) AS fglen
FROM ingredients;
```

The Length function

This function returns the number of positions that are within a character string and that are occupied by an actual value. This function is of value only if the target string is VARCHAR. CHAR data types have all positions occupied by default and have the values returned by the function that will always be the same length as the length specified in the column definition within the database.

```
length(word)
```

Length example

```
SELECT length('SHARMA') from dual;
```

Combining String Functions

The parameters to the various string functions can be any expression returning the correct data type. Such expressions can even include other string functions. Query below is a complex example to show how to combine string functions. This query returns the first word in uppercase of a multiword company name with any trailing's (apostrophe s) removed. The results are in all uppercase.

Combining string functions

```
SELECT vendorid, companyname,
TRIM(TRAILING '''S' FROM
TRIM(SUBSTRING(UPPER(companyname) FROM 1 FOR
POSITION(' ' IN companyname)))) AS "CoName"
FROM vendors;
```

Let's take this step-by-step:

- POSITION (' ' IN companyname) finds the character position of the first occurrence of a space. Let's call this value P. For Ed's Dressing, P is 5.
- UPPER (companyname) returns the company name with all alphabetic characters converted to uppercase. Let's call this string C. For Ed's Dressing, C is ED'S DRESSING.
- SUBSTRING then finds the substring in C starting at character position 1 and ending at position P, inclusive. Let's call this substring S. For Ed's Dressing, S is "ED'S ". Note the trailing space.
- The inner TRIM then eliminates any leading and trailing spaces. If the company name has a space, then S has a space because S includes the character in position C, which must be a space. If the company name does not have a space, then P is 0 and S is the empty string. Let's call this substring T. For Ed's Dressing, T is ED'S. Note that the trailing space has been removed.
- Finally, the outer TRIM eliminates any trailing occurrences of the string 'S. Note that we had to quote the single quote ('). For Ed's Dressing, this final value is ED.

What happened to Veggies_R_Us? Recall that POSITION returns 0 if it cannot find the substring. Given a <length> of 0, SUBSTRING returns an empty string.

Your DBMS will likely have other string manipulation functions. The SQL standard describes a few other string manipulation functions, which are not widely implemented. Such functions include OVERLAY (substituting substrings), CONVERT (changing character encoding), and TRANSLATE (mapping between character sets).

The LPAD Function

The LPAD function is used to pad the left side of a character string. The function makes each of the target strings the same overall length by placing the specified characters to the left of the original value.

```
lpad(char1, n, char2)
```

LPAD example

```
SELECT lpad('page 3',12,'*') "lpad" from dual;
```

The RPAD Function

The RPAD function is used to pad the right side of a character string. The function makes each of the target strings the same overall length by placing the specified characters to the right of the original value.

```
rpad(char1, n, char2)
```

RPAD example

```
SELECT rpad('page 3',12,'*') "rpad" from dual;
```

The TRANSLATE Function

The TRANSLATE function replace a sequence of character string in a string with another set of characters. However it replaces a single character at a time. The function has three parameters:

1. The target character string
2. The search character string
3. The replacement character string

```
translate(string1.string_to_replace,replacement string)
```

TRANSLATE example

```
SELECT TRANSLATE('1ddct568','158','249') "Change" from dual;
```

❖ **Temporal**

Time is no simple concept. To deal with this complexity, SQL provides a wide range of techniques for operating on temporal data types. Unfortunately, temporal types and arithmetic are not supported by all DBMSs. Consult your documentation for the exact limitations and syntax of your system.

Finding the Current Date or Time

Let's start with the basic question: "What time is it?" SQL provides several functions to help you find out.

Temporal Function	Description	Return Type
CURRENT_TIME[(precision)]	Current time with time zone displacement	TIME WITH TIMEZONE
CURRENT_DATE	Current date	DATE
CURRENT_TIMESTAMP[(precision)]	Current date and time with timezone displacement	TIMESTAMP WITH TIMEZONE
LOCALTIME[(precision)]	Current time	TIME WITHOUT TIMEZONE
LOCALTIMESTAMP[(precision)]	Current date	TIMESTAMP WITHOUT TIMEZONE

The optional precision argument specifies the fractional seconds precision. LOCALTIME and LOCALTIMESTAMP are not widely implemented.

Using Arithmetic Operators with Temporal Types

SQL allows the use of basic arithmetic operators with temporal data types. For example, you may want to know the date 3 days from now. To get that, you can add the current date to an interval of 3 days. Of course, not all arithmetic operations make sense with temporal data. For example, dividing a date by an interval makes no sense so it is not allowed by SQL. Here are the allowable operations and the resulting data types:

Expression			Result Type	Precedence
Interval	*	/	Interval	Highest
Numeric			Interval	
Numeric		*		
Interval				
Datetime	+	–	Datetime	Lowest
Interval				
Interval		+	Datetime	
Datetime				
Datetime		–	Interval	
Datetime			Interval	
Interval	+	–		
Interval				

The table doesn't cover all restrictions. The operation must make sense. Subtracting a DATE value from a TIME value has no meaning so it is not allowed by SQL. Let's look at an example.

Find how long each item has been on the menu as of midnight January 2, 2005

```
SELECT name, dateadded, DATE '2005-01-02' - dateadded AS "Days on Menu"
FROM items;
```

According to the 2003 SQL specification, subtracting two dates should result in an interval. Naturally, this must be a DAY–TIME interval, or precision would be lost. Because there is no time data, our DBMS chose to represent the results as a number of days. Your DBMS may be different.

EXTRACT: Getting Fields from Temporal Data

Datetime and interval data are made up of fields such as year, month, and so on. EXTRACT gets a specified field from temporal data.

```
EXTRACT(<field label> FROM <source>)
```

EXTRACT returns a numeric value representing the field specified by <field label> from <source>. The <source> is any expression returning a datetime or interval, and <field label> must be YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, or TIMEZONE_MINUTE. If <source> or <field label> is NULL, EXTRACT returns NULL.

EXTRACT example

```
SELECT name, EXTRACT(YEAR FROM dateadded) AS year,
EXTRACT(MONTH FROM dateadded + INTERVAL '30' DAY) AS month  FROM items;
```

Your DBMS will likely specify other temporal functions. Other SQL standard temporal functions include OVERLAPS (test if two time periods intersect) and ABS (returns the absolute value of an interval); however, these functions are not widely implemented.

Binary

The binary data types are called binary strings because SQL treats them like a strings of 0s and 1s. Most (but not all) things that you can do with a character string, you can do with a binary string using the same operators and functions, including concatenation, trimming (using the hexadecimal value X'00' as the default trim character), using substrings, and using LIKE. Query below is a simple binary string example.

BINARY STRING example

```
SELECT SUBSTRING(X'F0' FROM 3) || B'11' AS bits
FROM stores
WHERE storeid LIKE '#%';
```

One of the differences between character strings and binary strings is that binary strings can only be compared using = and <>; comparisons with > and < are not allowed. Binary strings also cannot follow DISTINCT or appear in the ORDER BY clause or in many other parts of SQL that we discuss later. If you need to use binary strings, consult your DBMS documentation.

---------&---------