SQL Course

Introduction to Views

- To introduce the concept of a view defined on one or more tables.
- To point out the possible advantages and disadvantages of using views.
- To demonstrate how the RDBMS processes views.

Sometimes in practise the user of a database may want to reference data regularly from different tables and see this data on screen at the same time. This can be achieved using a select statement by selecting the data the user wants from the tables in question. If the same data is being accessed regularly then one could ask the question: " Why not create a table with the data that the user wants as the columns in the table?".

- The problem with this is that each time data is added or updated to the original tables the new table must be checked to see if this data should be added or updated in the new table. This causes a degradation in performance.
- Another potential problem is that if another user wants to see the data in a different format, another table must be created and maintained, adding redundancy and making the problem even worse.

To provide the desired functionality without the drawbacks that were just mentioned, SQL provides a mechanism that enables you to create virtual tables, called views, that can be queried as regular tables but not stored as such. The contents of these virtual tables change when the contents of the regular tables on which they are defined change. This happens logically: updates are made on regular, stored tables and the system computes dynamically the contents of a view, when needed. Since the view mechanism recomputes the result when it is accessed , it may be slower than using a stored table which materialises the view and which is kept up to date at each update.

When you become familiar with the **SELECT** statement in later modules you will see the real power and benefits of using views in the relational database model. Creating a view based on a **SELECT** statement gives you an easy way to examine and handle just the data you need - no more, no less. A view can be used in an SQL statement just as if it were a table. This means that a view can be used in other SQL statements and can even be used to create other views. The rows and columns of data visible through the view are the query results produced by the query that defines the view.

Advantages:

- It saves the user from having to type in long queries each time they wish to see a subset of data across a number of tables.
- Views can be used in other SQL statements as if it were a table in the original database.
- Views allow different users to focus in on particular data and tasks of interest or concern. No extraneous or distracting information gets in the way. Using views, the database can be customised and so tailored to suit a variety of users with dissimilar interests and skill levels.
- Views provide security by hiding sensitive or irrelevant parts of the database. Access privileges can be granted and revoked on views as they would be on normal base tables.
- A view can be seen to be independent from underlying changes in structure to the original

tables. This means that if base tables are changed the view can be re-composed by altering the **SELECT** statement that was used to create the view.
- Views simplify database access by presenting the structure of the stored data in the way that is most natural for each user.
- A view can simplify multi table queries by drawing data from several tables and presenting them in a single table, thus turning the multi table query into a single table query against the view.
- If data is accessed and entered through a view, the DBMS can automatically check the data to ensure that it meets specified integrity constraints.

Disadvantages:

- As the database must translate queries against the view to queries against the source tables it means a degradation of performance and if a view is defined by a complicated query it may take a long time to complete.
- When a user tries to update the rows of a view, the database must translate the request into an update on rows of the underlying source tables. This is possible for simple views, but more complex views cannot be updated.

How it Works

When the database encounters a reference to a view in a SQL statement, it finds the definition of the view stored in the database. The database then translates the request that references the view into an equivalent request against the source tables of the view and carries out the equivalent request. In this way the database maintains the illusion of the view while maintaining the integrity of the source tables. For simple views the database will draw the rows of the view on the fly but for more complex views the database will first need to store the results of the query that defines the view in a temporary table. The database fills your requests for view access from this temporary table and discards the table when it is no longer needed.

SQL Course

Create View Statement

- To show how to create views in SQL syntax.
- To illustrate the syntax of the SQL statement.
- To demonstrate the use of the **CREATE VIEW** statement using examples.

- Views are created using the **CREATE VIEW** command.
- Each view consists of at least one column.
- Each column has a :
  - Name.
  - Type.
  - Size (depending on the type).
  - Optional integrity constraints.

The creation of a view assigns a name to the view, a name to each of its column, and a virtual content. The content is defined by an SQL **SELECT** statement. The types of each column are automatically derived from the **SELECT** statement. Assigning alias names to a views column is optional. If you don't give names in the **CREATE VIEW** clause, the views columns inherit their names from the columns in the underlying table(s).
The create view command has the form:

> **CREATE VIEW** *viewname* (*column-name*, *column-name*, ...) **AS SELECT** *statement* [**WITH** [**LOCAL**|**CASCADED**] CHECK OPTION]**;**

In certain circumstances names must be assigned to the columns of the view.
The circumstances in which this would arise would be:

- When one or more of the views columns are derived from an arithmetic expression, a built in function or a constant.
- When two or more of the columns in the view will end up with the same name.

View definitions may not be self-recursive, i.e. the view name of the view being defined may not be used directly or indirectly in the definition.

The **WITH** CHECK OPTION may only be specified if the view is update-able and a view is update-able only if the query expression is update-able. In general, if a **WITH** CHECK OPTION is specified then no **INSERT** or **UPDATE** statement may result in a row being created that will not appear in the view. If the **LOCAL** option is used then the constraint specifically provides that, if the row appears in the directly underlying table or view after the change, it must also appear in the view itself. If the **CASCADED** option is specified then the same check is carried out for each underlying view regardless of whether it has a **WITH** CHECK OPTION and of which sort.

The structure of the view is defined at the time at which the view is created. If the query expression had been of the form "**SELECT** * **FROM** *t* ..." then the * means all the columns of t that are defined at the moment the **CREATE VIEW** is executed. If columns are subsequently added to t then these will not appear in the view, unless the view is subsequently dropped and recreated.
This statement creates a view called `inserviceplanes` which takes its values from the

`aircraft` and `aircraft_flight` tables, using the field `call_sign` in both tables. This produces a view of all Planes inservice.

CREATE VIEW *inserviceplanes* (*inser_name*, *inser_model*, *inser_flight_no*) **AS SELECT** *aircraft_name*, *model*, *flight_no* **FROM** *aircraft*, *aircraft_flight* **WHERE** *aircraft.call_sign* = *aircraft_flight.call_sign*;

SQL Course

An Introduction to Joins

- Introduce the concept of a multi-table query in the form of a join.
- Explain why joins are necessary and how the DBMS handles them.
- Note some important points about joins.

- So far we have looked at single table queries (i.e. queries involving only one table). This is not sufficient as we shall on many occasions need to retrieve information from several tables at the same time.

- We call the combining of information from different tables a **JOIN**.

- Values can be displayed from more than one table by "joining" the rows using columns of the same type and size.

- Tables can be joined by adding a condition to the **WHERE** clause.

- Joins complete the triad of operations that a relational query language must provide: selection, projection, and join. The join operation lets you retrieve and manipulate data from more than one table in a single **SELECT** statement. Joining two tables is a little like making a seam to join two pieces of material. Once you have made the seam, you can work with the whole cloth. You specify joins in the **WHERE** clause of a **SELECT** statement. Like projections, which are specified in the select list of the **SELECT** statement, joins are expressed implicitly rather than explicitly.

- You specify each join on two tables at a time, using a column from each table as a connecting column or join column. A connecting column should have values that match or compare easily, representing the same or similar data in each of the tables participating in the join.

- Connecting columns almost always have the same data type. The values in connecting columns are join compatible: their values come from the same general class of data.

- A skeleton version of join syntax is:
  SELECT *select_list* **FROM** *table_1, table_2* [,*table_3* ...] **WHERE** *[table_1.]column* **JOIN_OPERATOR** *[table_2.]column*;

- The **FROM** clause table list must contain at least two tables, and the columns specified in the **WHERE** clause must be join compatible.

- When the join columns have identical names you must qualify the columns with their table names in the select list and in the **WHERE** clause.

- You can use any of the relational operators to express the relationships between the join columns as the join operator but equality is the most common. However it may be one of the following :
  < > < >= <> or **!=**

- You don't have to name the join column twice in the select list, or even include it in the results at all, in order for the join to succeed. However you may need to qualify the name of a join column with its table name in the select list or in join specification of the **WHERE** clause.
- As many tables as desired may be joined together.
- The matching criteria for the tables is called the **join predicate** or **join criteria**.
- More than one pair of columns may be used to specify the join condition between any two tables.
- Columns from any tables may be named in the **SELECT** clause.
- Columns which have the same name in the multiple tables named in the **FROM** clause must be uniquely identified by specifying **tablename.columnname**.
- If the columns are uniquely named across the tables, it is not necessary to specify **tablename.columnname**. However it improves readability of the **SELECT** statement.
- When n tables are joined, it is necessary to have at least n-1 table join conditions to avoid the Cartesian product. (Four table joins requires specification of a join criteria for three pairs of tables).
- When an = is used as the join operator the join is called an **Equi-Join** or **Equal-Join** and is used to retrieve rows which have a matching values in each column named in the join clause.
- The joining of tables can be based on any criteria you choose (not only equality comparison).
- The high level join construct hides all asymmetry of the implementation. A join is commutative. This means A join B = B join A. Therefore, the order in which the names appear in the **FROM** clause is irrelevant.
- The notion of a join generalises to more than two tables. To perform a join between A, B and C, two tables are first joined and then the result is joined with the third table. The join operation is also associative : (A join B) join C = A join (B join C).

Joins are necessary because:

- In a database that has been designed according to the normalisation rules, one table may not give you all the information you need about a particular entity. For comprehensive data analysis, you must assemble data from several of the tables you created when you normalised your database.
- The relational model - having led you to partition your data into several single subject tables following the rules of normalisation and good clean database design - relies on the join operation to enable you to perform ad hoc queries and produce comprehensible reports.
- Joins are possible in a relational database management system because the relational model's data independence permits you to bring data from separate tables into new and

unanticipated relationships. Relationships among data become explicit when you manipulate the data, not when you create the database.

- Joins are necessary because during the process of analysis that distributes data over the relational database landscape, you cleanly separate information on separate entities. You can get a comprehensive view of your data only if you can reconnect the tables. A corollary of the join is that it gives you unlimited flexibility in adding new kinds of data to your database.

- In relational theory, a join is the projection and restriction of the product. The product or cartesian product, is the set of all possible combinations of the rows from the two tables.

- The system first examines all possible combinations of the rows from two tables, then eliminates all the rows that do not meet the conditions of the projection (columns) and restriction (rows). The actual procedure that the system follows is more sophisticated.

- Conceptually speaking, the first step in processing a join is to form the cartesian product of the tables - all the possible combinations of the rows from each of the tables. Once the system obtains the cartesian product, it uses the columns in the select list for the projection, and the conditions in the **WHERE** clause for the selection, to eliminate the rows that do not satisfy the join.

- The cartesian product is the matrix of all possible combinations that could satisfy the join condition. If there is only one row in each table, then there is only one possible combination. The number of rows in the cartesian product equals the number of rows in the first table multiplied by the number of rows in the second table.

The diagram below shos how 2 tables are joined -

aircraft

aircraft name	model	club seats	econ seats	call sign
Eagle Flyer	ATR42	22	40	N410C
(NULL)	Boeing 707-320C	50	102	9J-AEB
(NULL)	Boeing 727-200	34	100	N7255U
Finians Dream	Boeing 737-200	22	96	DQ-FDM
(NULL)	Boeing 737-200	8	120	N301SW

aircraft_flight

call sign	flight_no	departure date
DQ-FDM	EI 082	01/08/96
N410C	BD 303	11/06/96
N301SW	EI 124	08/07/96
9J-AEB	SK 537	21/05/96
N7255U	EI 989	12/06/96

Join

aircraft name	model	club seats	econ seats	call sign	flight_no	departure date
Eagle Flyer	ATR42	22	40	N410C	BD 303	11/06/96
(NULL)	Boeing 707-320C	50	102	9J-AEB	SK 537	21/05/96
(NULL)	Boeing 727-200	34	100	N7255U	EI 989	12/06/96
Finians Dream	Boeing 737-200	22	96	DQ-FDM	EI 082	01/08/96
(NULL)	Boeing 737-200	8	120	N301SW	EI 124	08/07/96

Table returned

The bottom table is what results after the 2 tables are joined.

SQL Course

An Introduction to Simple Joins

- Introduce the concept of an Equi-Join.
- Note some important considerations when joining tables.
- Illustrate the use of multi-table queries with a few examples.

A join based on an exact match between two columns is more precisely called an **Equi-Join**. Joins can also be based on other kinds of column comparisons. As all of the data in a relational database is stored in its columns as explicit data values, all possible relationships between tables can be formed by matching the contents of related columns. The join thus provides a powerful facility for exercising the data relationships in a database.

The syntax for an equi-join follows :

> **SELECT** *select_list* **FROM** *table_1, table_2* [,*table_3* ...] **WHERE** *[table_1.]column* **=** *[table_2.]column*;

When the join operator is given as an **=**, the join is called an equi-join. The search condition compares columns from two different tables. We call these two columns the matching columns for the two tables. Like all search conditions, this one restricts the rows that appear in the query results.
The most common multi-table queries involve two tables that have a natural parent/child relationship. This is usually achieved using a primary key/foreign key relationship. SQL does not require that the matching columns be included in the results of a multi table query.

The search condition that specifies the matching columns in a multi-table query can be combined with other search conditions to further restrict the contents of the query results. To join tables based on a parent/child relationship where the relationship is specified by more than one column you must join the tables over these multiple columns. Multi column joins are less common than single column joins, and are usually found in queries involving compound keys.

SQL can combine data from three or more tables using the same basic technique used for two table queries. Look at the following example to see how this is achieved:

> **SELECT** *select_list* **FROM** *table_1, table_2, table_3* **WHERE** *[table_1.]column* **=** *[table_2.]column* **AND** *[table_2.]column* **=** *[table_3.]column*;

The vast majority of multi table queries are based on parent/child relationships, but SQL does not require that the matching columns be related as a foreign key and primary key. Any pair of columns from two tables can serve as matching columns, provided they have comparable data types. Matching columns like this, generate a many to many relationship between the two tables.

- Joins that match primary keys to foreign keys always create one to many parent/child relationships.
- Other joins may also produce one to many relationships if one of the matching columns has unique values for all rows in the table.
- In general, joins on arbitrary matching columns generate many to many relationships.

Non equi-joins are when the tables are joined by another comparison operator other than the equality sign.

What happens when the connecting columns contain a **NULL** value ?

- Most SQL implementations are clear about what happens when nulls are present in columns being joined. If there are nulls in the connecting columns of tables being joined, the nulls will never join because nulls represent unknown or inapplicable values.
- Few SQL implementations support the join on null values. DB2 is one of the few. There is ongoing debate about how a system should deal with joins on null values. The consequence of not supporting a join on nulls is that some needed or useful information would not appear in query results.

The multi table queries do not usually require any special SQL syntax or language features beyond those described for single table queries. However some multi table queries cannot be expressed without the additional SQL language features described as follows :
Qualified Column Names
These are needed to eliminate ambiguous column references. This happens when two tables share the same column name and that column is specified in the select list. This leads SQL to produce an error. To eliminate this ambiguity, you must use a qualified column name to identify the column. This is achieved by specifying the name of the table followed by the column name separated by a full stop. The table specified in the qualified column name must, of course, match one of the tables specified in the **FROM** clause.
All column selection (*)
In a multi table query, the ***** selects all columns from all tables in the **FROM** clause. Many SQL dialects treat the asterix as a special kind of wild-card column name that is expanded into a list of columns. In these dialects, the asterix, can be qualified with a table name. Although this is permitted in most SQL dialects, it is not permitted by the ANSI/ISO standard.
Self Joins
Some multi table queries involve a relationship that a table has with itself. SQL uses an imaginary duplicate table approach to join a table to itself. Instead of actually duplicating the contents of the table, SQL simply lets you refer to it by a different name, called a table alias. The **FROM** clause assigns an alias to each copy of the table by specifying the alias name immediately after the actual table name. When a **FROM** clause contains a table alias, the alias must be used to identify the table in qualified column references.

Table Aliases or Correlation Names
A Correlation name (or alias) allows a convenient shorthand for a table name.

- The correlation name is paired with the table name in the **FROM** clause.
- Correlation names also allow more advanced queries in conjunction with sub queries.
- **SELECT** *af.call_sign, af.departure_time*
  FROM *aircraft_flight af*;
  Displays the call sign and departure time of all flights.

Table aliases are required in queries involving self joins but they can also be used in any query. For example if your query refers to another users table, or if the name of a table is very long, the table name can become tedious to type as a column qualifier. If a table alias is specified, it becomes the table tag, otherwise the tables name, exactly as it appears in the **FROM** clause, becomes the tag.
The SQL2 standard optionally allows the keyword **AS** to appear between a table name and

table alias. While this makes the **FROM** clause easier to read, it may not be yet supported by all implementations of SQL.

The following example retrieves the aircraft name, flight no, departure details and arrival location for all aircraft flights :

> **SELECT** *aircraft.aircraft_name, flight.flight_no, flight.departure_date, flight.departure_time, flight.departure_location, flight.arrival_location* **FROM** *aircraft, aircraft_flight, flight* **WHERE** *aircraft_flight.call_sign* **=** *aircraft.call_sign* **AND** *aircraft_flight.flight_no* **=** *flight.flight_no* **AND** *aircraft_flight.departure_date* **=** *flight.departure_date***;**

We can join tables using more than one criteria, by including another clause in the **SELECT** statement.

The following example will return only airplanes of model 'Boeing 737-200' that are registered for flights :

> **SELECT** *aircraft.aircraft_name, aircraft.model, aircraft.no_club_seats, aircraft.no_economy_seats, aircraft_flight.call_sign, aircraft_flight.flight_no, aircraft_flight.departure_date* **FROM** *aircraft, aircraft_flight* **WHERE** *aircraft_flight.call_sign* **=** *aircraft.call_sign* **AND** *aircraft.model* **=** 'Boeing 737-200'**;**