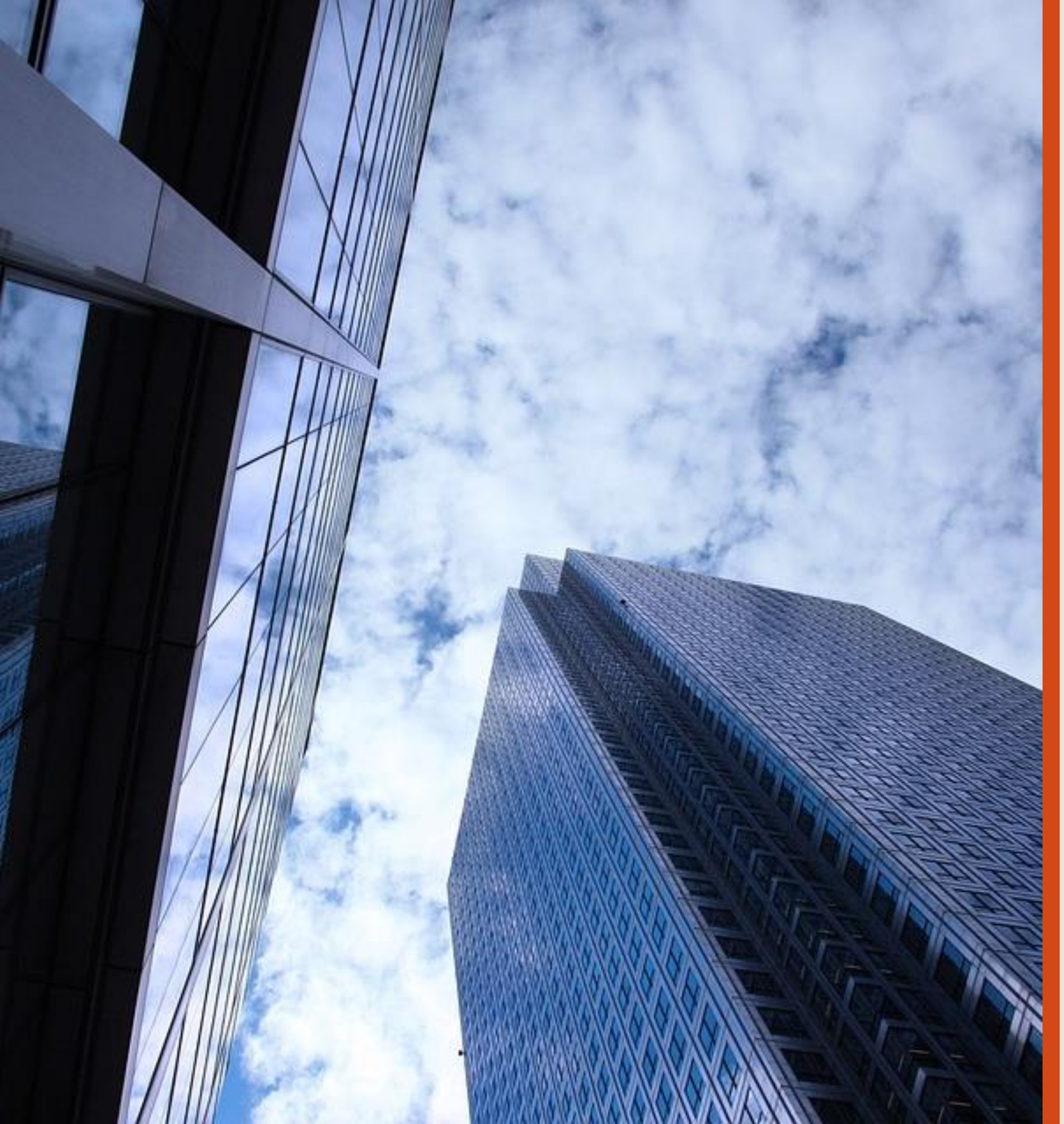


SQL
PART 05

Jayathma Chathurangani

ejc@ucsc.cmb.ac.lk

OUTLINE

DDL Continues

✓ **Indexes**

✓ **Views**

- What is a view?
- Views using SQL
  - Creating view
  - Dropping view
- View Updatability and WITH CHECK OPTION in SQL
- View Materialization

1

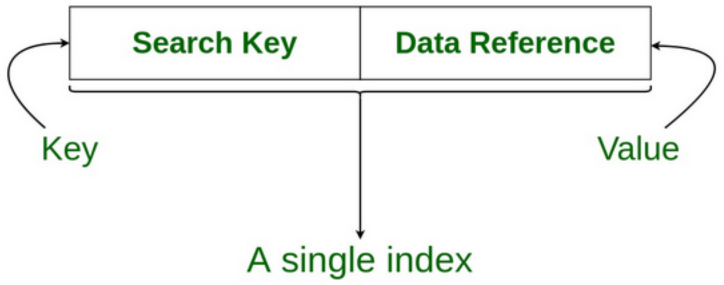
INDEXES

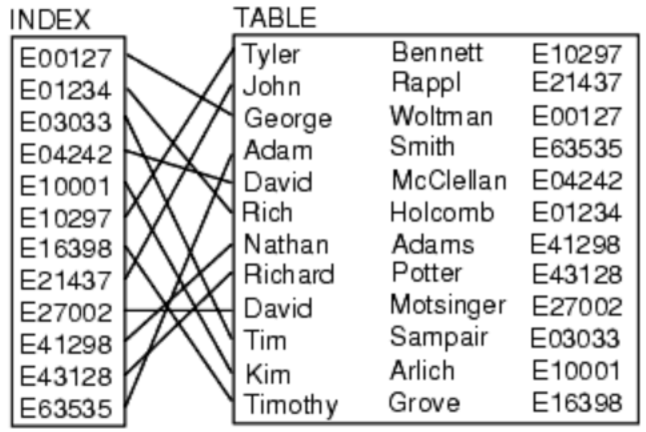
1.1 INTRODUCTION

- An **index** is a structure that provides accelerated access to the rows of a table based on the values of one or more columns.
- The presence of an index can significantly improve the performance of a query.
- **However, as indexes may be updated by the system every time the underlying tables are updated, additional overheads may be incurred.**
- Indexes are usually created to satisfy particular search criteria after the table has been in use for some time and has grown in size.
- **To view a visual explain execution plan containing all these,** execute your query from the SQL editor and then select Execution Plan within the query results tab. The execution plan defaults to Visual Explain, but it also includes a Tabular Explain view that is similar to what you see when executing EXPLAIN in the MySQL client.

1.1.1 INDEXES – A DATA STRUCTURE

- Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.
- Indexes are created using a few database columns.
  - The first column is the **Search key** that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly. Note: The data may or may not be stored in sorted order.
  - The second column is the **Data Reference or Pointer** which contains a set of pointers holding the address of the disk block where that particular key value can be found.




INDEX

E00127
E01234
E03033
E04242
E10001
E10297
E16398
E21437
E27002
E41298
E43128
E63535

TABLE

Tyler	Bennett	E10297
John	Rappl	E21437
George	Woltman	E00127
Adam	Smith	E63535
David	McClellan	E04242
Rich	Holcomb	E01234
Nathan	Adams	E41298
Richard	Potter	E43128
David	Motsinger	E27002
Tim	Sampair	E03033
Kim	Arlich	E10001
Timothy	Grove	E16398

1.1.2 INDEXES – IMPROVING RECORD SELECTION PERFORMANCE

- For indexes to improve the performance of selections, the index expression must match the selection condition exactly.
- For example, if you have created an index whose expression is last_name, the following Select statement uses the index:

SELECT * FROM emp **WHERE** last_name = 'Smith'

- This Select statement, however, does not use the index (does not match the index expression LAST_NAME):

SELECT * FROM emp **WHERE** UPPER(last_name) = 'SMITH'

1.1.3 INDEXES – INDEXING MULTIPLE FIELDS

- If you often use Where clauses that involve more than one field, you may want to build an index containing multiple fields.
- **Concatenated indexes** can also be used for Where clauses that contain only the first of two concatenated fields.
- Consider the following Where clause:

  WHERE last_name = 'Smith' **AND** first_name = 'Thomas'

  For this condition, the optimal index field expression is LAST_NAME, FIRST_NAME. This creates a concatenated index.

- The LAST_NAME, FIRST_NAME index also improves the performance of the following Where clause (even though no first name value is specified):

  last_name = 'Smith'

1.1.3 INDEXES – INDEXING MULTIPLE FIELDS (CONTINUED)

- ❖If your index fields include all the conditions of the Where clause in that order, the driver can use the entire index.
- ❖If, however, your index is on two nonconsecutive fields, say, LAST_NAME and FIRST_NAME, the driver can use only the LAST_NAME field of the index.
- Consider the following Where clause:

WHERE last_name = 'Smith' **AND** middle_name = 'Edward’ **AND** first_name = 'Thomas'

1.1.3 INDEXES – INDEXING MULTIPLE FIELDS (CONTINUED)

❖ **The driver uses only one index when processing Where clauses.**

- If you have complex Where clauses that involve a number of conditions for different fields and have indexes on more than one field, the driver chooses an index to use.
- **The driver attempts to use indexes on conditions that use the equal sign as the relational operator rather than conditions using other operators (such as greater than).**
- Assume you have an index on the EMP_ID field as well as the LAST_NAME field and the following Where clause:

  WHERE emp_id >= 'E10001' **AND** last_name = 'Smith'

- In this case, the driver selects the index on the LAST_NAME field.

1.1.3 INDEXES – INDEXING MULTIPLE FIELDS (CONTINUED)

- **If no conditions have the equal sign, the driver first attempts to use an index on a condition that has a lower and upper bound, and then attempts to use an index on a condition that has a lower or upper bound.**

- The driver always attempts to use the most restrictive index that satisfies the Where clause.
- **In most cases, the driver does not use an index if the Where clause contains an OR comparison operator.**
- For example, the driver does not use an index for the following Where clause:

WHERE emp_id >= 'E10001' **OR** last_name = 'Smith'

1.1.4 INDEXES – DECIDING WHICH INDEX TO CREATE

- Before you create indexes for a database table, consider how you will use the table.
- The two most common operations on a table are to:

1. Insert, update, and delete records
2. Retrieve records

1.1.4 INDEXES – DECIDING WHICH INDEX TO CREATE (CONTINUED)

- **If you most often insert, update, and delete records, then the <u>fewer indexes</u> associated with the table, the better the performance.**
  - This is because the driver must maintain the indexes as well as the database tables, thus slowing down the performance of record inserts, updates, and deletes.
  - It may be more efficient to drop all indexes before modifying a large number of records, and re-create the indexes after the modifications.
- **If you most often retrieve records, you must look further to define the criteria for retrieving records and create indexes to improve the performance of these retrievals.**

1.1.5 INDEXES – IMPROVING JOIN PERFORMANCE

- When joining database tables, index tables can greatly improve performance.
- Unless the proper indexes are available, queries that use joins can take a long time.
- Assume you have the following Select statement:

  SELECT * FROM dept, emp **WHERE** dept.dept_id = emp.dept

  - In this example, the DEPT and EMP database tables are being joined using the department ID field. When the driver executes a query that contains a join, **it processes the tables from <u>left to right</u> and uses an index on the second table's join field (the DEPT field of the EMP table).**
  - To improve join performance, you need an index on the join field of the second table in the From clause.
- If there is a third table in the From clause, the driver also uses an <u>index on the field in the third table</u> that joins it to any previous table. For example:

  SELECT * FROM dept, emp, addr
  WHERE dept.dept_id = emp.dept **AND** emp.loc = addr.loc

  - In this case, you should have an index on the EMP.DEPT field and the ADDR.LOC field.

1.2 CREATING AN INDEX – CREATE INDEX

- The creation of indexes is not standard SQL.
- However, most dialects support at least the following capabilities:

  CREATE [**UNIQUE**] **INDEX** IndexName
  ON TableName (columnName [**ASC** | **DESC**] [, . . .])

- The specified columns constitute the index key and should be listed in major to minor order.
- **Indexes can be created only on base tables not on views.**
- **If the UNIQUE clause is used, uniqueness of the indexed column or combination of columns will be enforced by the DBMS.** This is certainly required for the primary key and possibly for other columns as well (for example, for alternate keys).
  - Although indexes can be created at any time, we may have a problem if we try to create a unique index on a table with records in it, because the values stored for the indexed column(s) may already contain duplicates.
  - Therefore, it is good practice to create unique indexes, at least for primary key columns, when the base table is created and the DBMS does not automatically enforce primary key uniqueness.

1.2 CREATING AN INDEX – CREATE INDEX (CONTINUED)

- For the Staff and PropertyForRent tables, we may want to create at least the following indexes:

CREATE UNIQUE INDEX StaffNoInd **ON** Staff (staffNo);

CREATE UNIQUE INDEX PropertyNoInd **ON** PropertyForRent (propertyNo);

- For each column, we may specify that the order is ascending (ASC) or descending (DESC), with ASC being the default setting.
- For example, if we create an index on the PropertyForRent table as:

CREATE INDEX RentInd **ON** PropertyForRent (city, rent);

- then an index called RentInd is created for the PropertyForRent table. Entries will be in alphabetical order by city and then by rent within each city.

1.3 DROPPING AN INDEX – DROP INDEX

- If we create an index for a base table and later decide that it is no longer needed, we can use the DROP INDEX statement to remove the index from the database.
- DROP INDEX has the following format:

DROP INDEX IndexName

- The following statement will remove the index created in the previous example:

DROP INDEX RentInd;

2

VIEWS

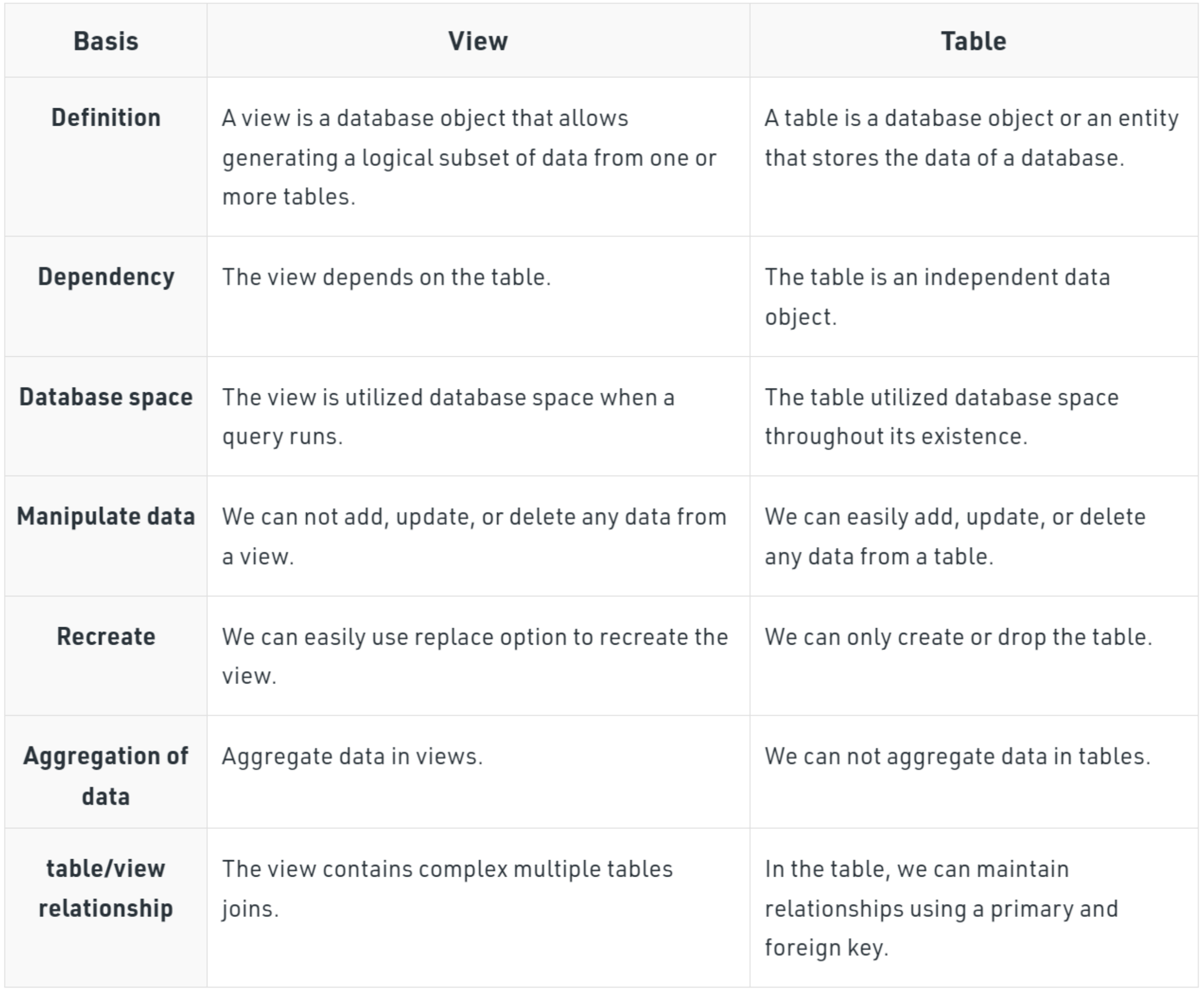
2.1 INTRODUCTION

- The dynamic result of one or more relational operations operating on the base relations to produce another relation.
- **A view is a virtual relation that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request.**
- To the database user<u>, a view appears just like a real table, with a set of named columns and rows of data.</u>
- However, unlike a base table, a view does not necessarily exist in the database as a stored set of data values. Instead, a view is defined as a query on one or more base tables or views.

2.1 INTRODUCTION (CONTINUED)

- The DBMS stores the definition of the view in the database.
- When the DBMS encounters a reference to a view, one approach is to look up this definition and translate the request into an equivalent request against the source tables of the view and then perform the equivalent request. This merging process, called **view resolution.**
- An alternative approach, called **view materialization**, stores the view as a temporary table in the database and maintains the currency of the view as the underlying base tables are updated.

2.2 VIEW VS TABLE



Basis	View	Table
Definition	A view is a database object that allows generating a logical subset of data from one or more tables.	A table is a database object or an entity that stores the data of a database.
Dependency	The view depends on the table.	The table is an independent data object.
Database space	The view is utilized database space when a query runs.	The table utilized database space throughout its existence.
Manipulate data	We can not add, update, or delete any data from a view.	We can easily add, update, or delete any data from a table.
Recreate	We can easily use replace option to recreate the view.	We can only create or drop the table.
Aggregation of data	Aggregate data in views.	We can not aggregate data in tables.
table/view relationship	The view contains complex multiple tables joins.	In the table, we can maintain relationships using a primary and foreign key.

2.3 CREATING A VIEW – CREATE VIEW

```
CREATE VIEW ViewName [(newColumnName [, . . . ])]
AS subselect [WITH [CASCADED | LOCAL] CHECK OPTION]
```

- A view is defined by specifying an SQL SELECT statement.
- A name may optionally be assigned to each column in the view.
- If a list of column names is specified, it must have the same number of items as the number of columns produced by the subselect.
- If the list of column names is omitted, each column in the view takes the name of the corresponding column in the subselect statement.
- The list of column names must be specified if there is any ambiguity in the name for a column.

```
CREATE VIEW Manager3Staff
AS SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo = ‘B003’;
```

2.3 CREATING A VIEW – CREATE VIEW (CONTINUED)

- The subselect is known as the defining query.
- **WITH CHECK OPTION** is an optional clause on the CREATE VIEW statement. (Let's discuss more details in Section 2.10)
  - It specifies the level of checking when data is inserted or updated through a view.
  - The option cannot be specified if the view is read-only.
  - The definition of the view must not include a subquery.
  - SQL ensures that if a row fails to satisfy the WHERE clause of the defining query of the view, it is not added to the underlying base table of the view.
- It should be noted that to create a view successfully, you must have SELECT privilege on all the tables referenced in the subselect and USAGE privilege on any domains used in referenced columns.

2.4 CREATE A HORIZONTAL VIEW

- **A horizontal view restricts a user's access to selected rows of one or more tables.**

Query: ***Create a view so that the manager at branch B003 can see the details only for staff who work in his or her branch office.***

```
CREATE VIEW Manager3Staff
AS SELECT *
FROM Staff
WHERE branchNo = 'B003';
```

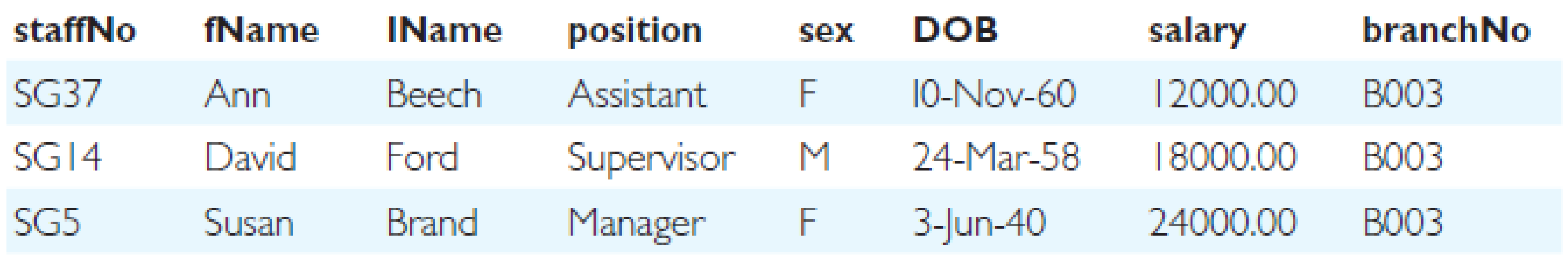
- This creates a view called Manager3Staff with the same column names as the Staff table but containing only those rows where the branch number is B003.

2.4 CREATE A HORIZONTAL VIEW (CONTINUED)

- Consider the statement

```
SELECT * FROM Manager3Staff;
```

- To ensure that the branch manager can see only these rows, the manager should not be given access to the base table Staff.
- Instead, the manager should be given access permission to the view Manager3Staff.
- This, in effect, gives the branch manager a customized view of the Staff table, showing only the staff at his or her own branch.



staffNo	fName	lName	position	sex	DOB	salary	branchNo
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000.00	B003
SG5	Susan	Brand	Manager	F	3-Jun-40	24000.00	B003

2.5 CREATE A VERTICAL VIEW

- **A vertical view restricts a user's access to selected columns of one or more tables.**

Query: ***Create a view of the staff details at branch B003 that excludes salary information, so that only managers can access the salary details for staff who work at their branch.***

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
FROM Staff
WHERE branchNo = 'B003';
```

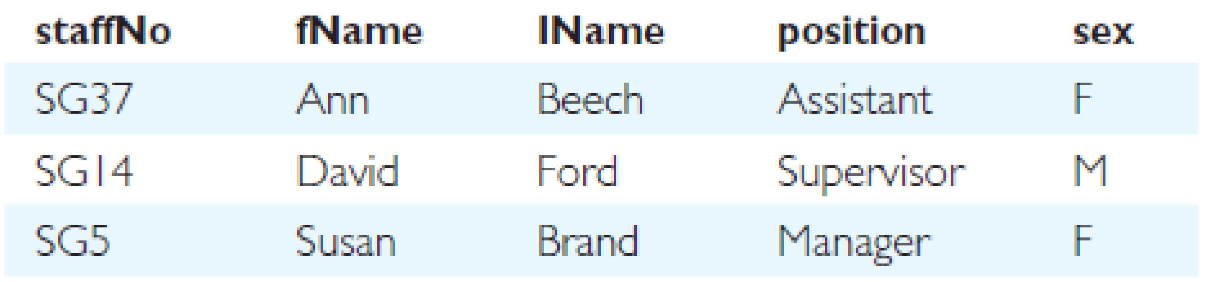
- Note that we could rewrite this statement to use the Manager3Staff view instead of the Staff table, thus:

```
CREATE VIEW Staff3
AS SELECT staffNo, fName, lName, position, sex
FROM Manager3Staff;
```

- Either way, this creates a view called Staff3 with the same columns as the Staff table, but excluding the salary, DOB, and branchNo columns.

2.5 CREATE A VERTICAL VIEW (CONTINUED)

- To ensure that only the branch manager can see the salary details, staff at branch B003 should not be given access to the base table Staff or the view Manager3Staff.
- Instead, they should be given access permission to the view Staff3, thereby denying them access to sensitive salary data.
- **Vertical views are commonly used where the data stored in a table is used by various users or groups of users.**
- They provide a private table for these users composed only of the columns they need.



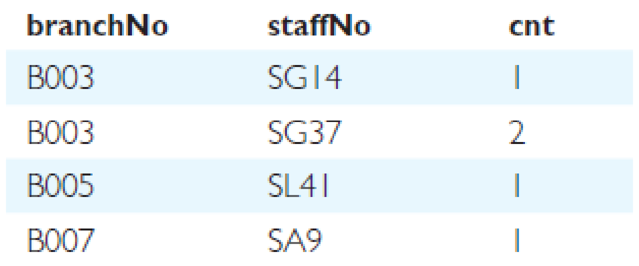
staffNo	fName	lName	position	sex
SG37	Ann	Beech	Assistant	F
SG14	David	Ford	Supervisor	M
SG5	Susan	Brand	Manager	F

2.5 GROUPED AND JOINED VIEWS

- It illustrates the use of a subselect containing a GROUP BY clause (**giving a view called a grouped view**), and containing multiple tables (**giving a view called a joined view**).
- **One of the most frequent reasons for using views is to simplify multi-table queries.**
- **Once a joined view has been defined, we can often use a simple single-table query against the view for queries that would otherwise require a multi-table join.**
- Note that we have to name the columns in the definition of the view because of the use of the unqualified aggregate function COUNT in the subselect.

Query: ***Create a view of staff who manage properties for rent, which includes the branch number they work at, their staff number, and the number of properties they manage***

```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*) AS cnt
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;
```



branchNo	staffNo	cnt
B003	SG14	1
B003	SG37	2
B005	SL41	1
B007	SA9	1

2.6 REMOVING A VIEW (DROP VIEW)

DROP VIEW ViewName **[RESTRICT | CASCADE]**

- DROP VIEW causes the definition of the view to be deleted from the database.
- For example, we could remove the Manager3Staff view using the following statement:

DROP VIEW Manager3Staff;

- If **CASCADE** is specified, DROP VIEW deletes all related dependent objects; in other words, all objects that reference the view.
- **This means that DROP VIEW also deletes any views that are defined on the view being dropped.**
- If **RESTRICT** is specified and there are any other objects that depend for their existence on the continued existence of the view being dropped, the command is rejected.
- The default setting is RESTRICT.

2.7 VIEW RESOLUTION

Definition:
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, **COUNT**(*) AS cnt
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;

Query we are going to execute(View is Utilized):
SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = ‘B003’
ORDER BY staffNo;

- **Step 01:** The view column names in the SELECT list are translated into their corresponding column names in the defining query.

  SELECT s.staffNo **AS** staffNo, **COUNT**(*) **AS** cnt

- **Step 02:** View names in the FROM clause are replaced with the corresponding FROM lists of the defining query:

  FROM Staff s, PropertyForRent p

- **Step 03:** The WHERE clause from the user query is combined with the WHERE clause of the defining query using the logical operator AND.

  WHERE s.staffNo = p.staffNo **AND** branchNo = ‘B003’

- **Step 04:** The GROUP BY and HAVING clauses are copied from the defining query.

  GROUP BY s.branchNo, s.staffNo

2.7 VIEW RESOLUTION (CONTINUED)

Definition:
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, **COUNT**(*)
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;

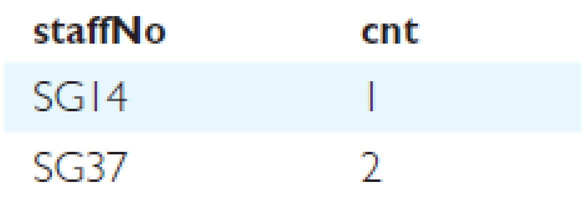
Query we are going to execute:
SELECT staffNo, cnt
FROM StaffPropCnt
WHERE branchNo = ‘B003’
ORDER BY staffNo;

- **Step 05:** Finally, the ORDER BY clause is copied from the user query with the view column name translated into the defining query column name

ORDER BY s.staffNo

- **Step 06:** The final merged query becomes:

SELECT s.staffNo **AS** staffNo, **COUNT**(*) **AS** cnt
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo **AND** branchNo = ‘B003’
GROUP BY s.branchNo, s.staffNo
ORDER BY s.staffNo;



staffNo	cnt
SG14	1
SG37	2

2.8 RESTRICTION ON VIEWS

```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;
```

The ISO standard imposes several important restrictions on the creation and use of views, although there is considerable variation among dialects.

❖ If **a column in the view is based on an aggregate function**, then the column may appear only in SELECT and ORDER BY clauses of ***queries that access the view***.

- In particular, such a column may not be used in a WHERE clause and may not be an argument to an aggregate function in *any query based on the view*.
  - Consider the below example which has a column cnt based on the aggregate function COUNT. (consider the view StaffPropCnt we talked about earlier)

```
#1 SELECT COUNT(cnt)
     FROM StaffPropCnt;
```

```
#2 SELECT *
     FROM StaffPropCnt
     WHERE cnt > 2;
```

  - #1 would fail because we are using an aggregate function on the column cnt, which is itself based on an aggregate function.
  - #2 would fail because we are using the view column, cnt, derived from an aggregate function, on the left-hand side of a WHERE clause.

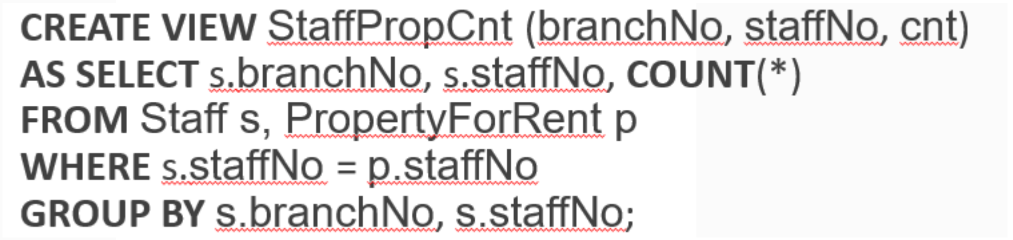
2.8 RESTRICTION ON VIEWS (CONTINUED)

The ISO standard imposes several important restrictions on the creation and use of views, although there is considerable variation among dialects.

- A **grouped view may never be joined with a base table or a view.**

- For example, the StaffPropCnt view is a grouped view, so any attempt to join this view with another table or view fails.

2.9 VIEW UPDATABILITY



```
CREATE VIEW StaffPropCnt (branchNo, staffNo, cnt)
AS SELECT s.branchNo, s.staffNo, COUNT(*)
FROM Staff s, PropertyForRent p
WHERE s.staffNo = p.staffNo
GROUP BY s.branchNo, s.staffNo;
```

- All updates to a base table are immediately reflected in all views that encompass that base table.
- Similarly, we may expect that if a view is updated, the base table(s) will reflect that change.
- However, consider again the view StaffPropCnt we discussed earlliar.
  - Consider what would happen if we tried to insert a record that showed that at branch B003, staff member SG5 manages two properties, using the following insert statement:

    INSERT INTO StaffPropCnt
    VALUES (‘B003’, ‘SG5’, 2);
  - We do not know the underlying table structure Eg- Primary Keys, Null Allowed or not etc.
  - The ISO standard specifies the views that must be updatable in a system that conforms to the standard.

For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

2.9 VIEW UPDATABILITY (CONTINUED)

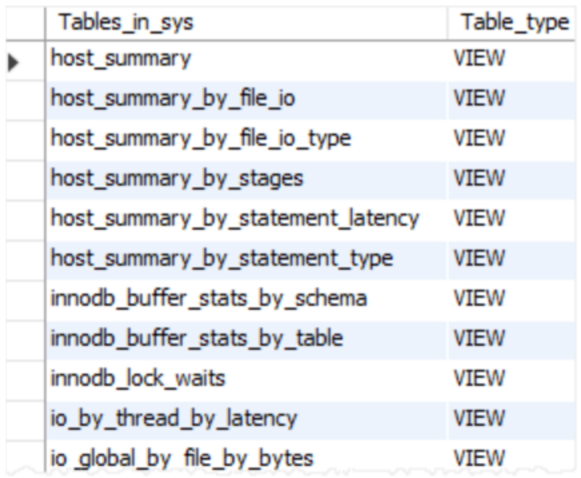
- The definition given in the ISO standard is that a view is updatable if and only if:

1. **DISTINCT is not specified**; that is, duplicate rows must not be eliminated from the query results.
2. **Every element in the SELECT list of the defining query is a column name** (rather than a constant, expression, or aggregate function) and no column name appears more than once.
3. **The FROM clause specifies only one table**; that is, the view must have a single source table for which the user has the required privileges. If the source table is itself a view, then that view must satisfy these conditions. This, therefore, excludes any views based on a join, union (UNION), intersection (INTERSECT), or difference (EXCEPT).
4. **The WHERE clause does not include any nested SELECTs that reference the table in the FROM clause.**
5. **There is no GROUP BY or HAVING clause** in the defining query.

- In addition, every row that is added through the view **must not violate the integrity constraints** of the base table.

For your knowledge…

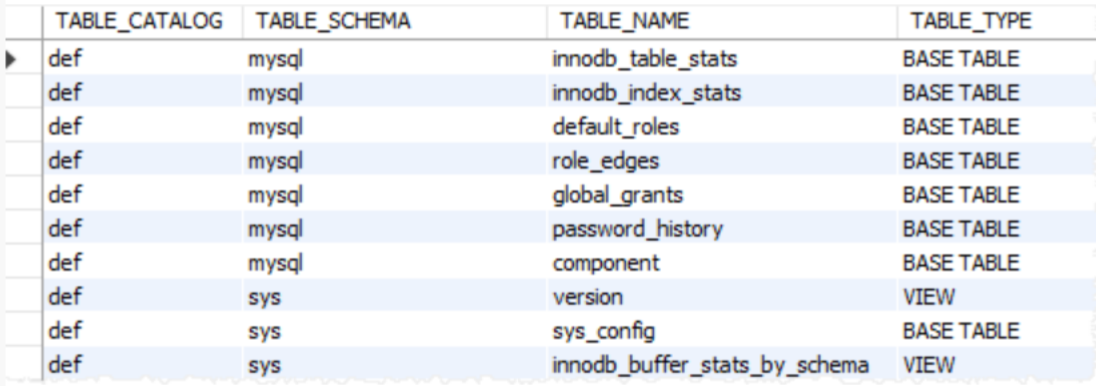
```
SHOW FULL TABLES IN sys
WHERE table_type='VIEW';
```



Tables_in_sys	Table_type
host_summary	VIEW
host_summary_by_file_io	VIEW
host_summary_by_file_io_type	VIEW
host_summary_by_stages	VIEW
host_summary_by_statement_latency	VIEW
host_summary_by_statement_type	VIEW
innodb_buffer_stats_by_schema	VIEW
innodb_buffer_stats_by_table	VIEW
innodb_lock_waits	VIEW
io_by_thread_by_latency	VIEW
io_global_by_file_by_bytes	VIEW

```
SELECT *
FROM information_schema.tables;
```

Here's the partial output:



TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE
def	mysql	innodb_table_stats	BASE TABLE
def	mysql	innodb_index_stats	BASE TABLE
def	mysql	default_roles	BASE TABLE
def	mysql	role_edges	BASE TABLE
def	mysql	global_grants	BASE TABLE
def	mysql	password_history	BASE TABLE
def	mysql	component	BASE TABLE
def	sys	version	VIEW
def	sys	sys_config	BASE TABLE
def	sys	innodb_buffer_stats_by_schema	VIEW

2.10 WITH CHECK OPTION

- Rows exist in a view, because they satisfy the WHERE condition of the defining query.
- If a row is altered such that it no longer satisfies this condition, then it will disappear from the view.
- Similarly, new rows will appear within the view when an insert or update on the view causes them to satisfy the WHERE condition.
- **The rows that enter or leave a view are called migrating rows.**
- Generally, the WITH CHECK OPTION clause of the CREATE VIEW statement prohibits a row from migrating out of the view.
- **Simply, this WITH CHECK OPTION prevents you from updating or inserting rows that are not visible through the view.**
- When an INSERT or UPDATE statement on the view violates the WHERE condition of the defining query, the operation is rejected. This behavior enforces constraints on the database and helps preserve database integrity.
- **The WITH CHECK OPTION can be specified only for an updatable view**.

2.10 WITH CHECK OPTION (CONTINUED)

The optional qualifiers LOCAL/ CASCADED are applicable to *view hierarchies*, that is, a view that is derived from another view.

- In this case, if **WITH LOCAL CHECK OPTION** is specified, then any row insert or update on this view, and on any view directly or indirectly defined on this view, must not cause the row to disappear from the view, unless the row also disappears from the underlying derived view/table.

```
CREATE VIEW LowSalary
AS SELECT *
FROM Staff
WHERE salary > 9000;
```

```
CREATE VIEW HighSalary
AS SELECT *
FROM LowSalary
WHERE salary > 10000
WITH LOCAL CHECK OPTION;
```

```
CREATE VIEW Manager3Staff
AS SELECT *
FROM HighSalary
WHERE branchNo = ‘B003’;
```

- Consider below update

```
UPDATE Manager3Staff
SET salary = 9500
WHERE staffNo = ‘SG37’;
```

- This update would fail: row disappear from the view **HighSalary**, the row would not disappear from the table LowSalary that HighSalary is derived from.

2.10 WITH CHECK OPTION (CONTINUED)

The optional qualifiers LOCAL/ CASCADED are applicable to *view hierarchies*, that is, a view that is derived from another view.

- If the **WITH CASCADED CHECK OPTION (Default of CHECK OPTION)** is specified , then any row insert or update on this view and on any view directly or indirectly defined on this view must not cause the row to disappear from the view.
  - Consider the previous example update, if the view HighSalary had specified WITH CASCADED CHECK OPTION, then setting the salary to either 9500 or 8000 would be rejected, because the row would disappear from HighSalary.
  - Therefore, to ensure that anomalies like this do not arise, each view should normally be created using the WITH CASCADED CHECK OPTION.
- **WITH LOCAL CHECK OPTION** specifies that the search conditions of only those dependent views that have the WITH LOCAL CHECK OPTION or WITH CASCADED CHECK OPTION are checked when a row is inserted or updated.
- **WITH CASCADED CHECK OPTION** specifies that the search conditions of all dependent views are checked when a row is inserted or updated.

2.10 WITH CHECK OPTION (CONTINUED) - CASCADED

- Example:

```
CREATE TABLE t1 (
    c INT
);
```

```
CREATE OR REPLACE VIEW v1
AS
    SELECT
        c
    FROM
        t1
    WHERE
        c > 10;
```

No CHECK OPTION SPECIFIED

```
INSERT INTO v1(
VALUES (5);
```

OK

```
CREATE OR REPLACE VIEW v2
AS
    SELECT c FROM v1
WITH CASCADED CHECK OPTION;
```

```
INSERT INTO v2(c)
VALUES (5);
```

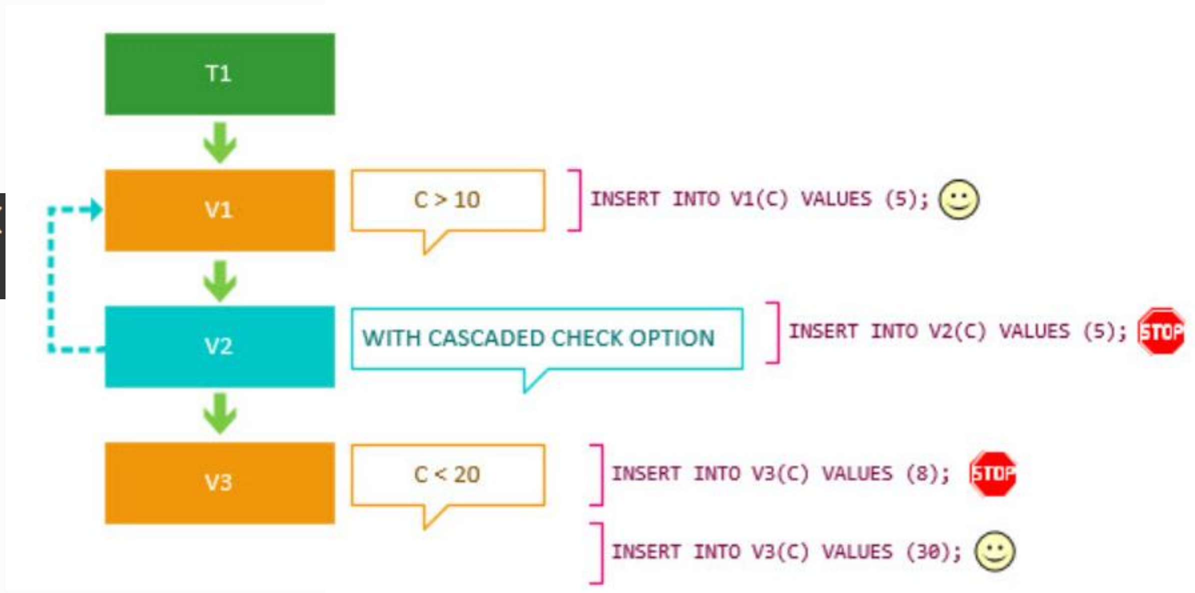
ERROR

```
CREATE OR REPLACE VIEW v3
AS
    SELECT c
    FROM v2
    WHERE c < 20;
```

No CHECK OPTION SPECIFIE D

```
INSERT INTO v3(c)
VALUES (8);
```

ERROR



```
INSERT INTO V1(C) VALUES (5);
INSERT INTO V2(C) VALUES (5);
INSERT INTO V3(C) VALUES (8);
INSERT INTO V3(C) VALUES (30);
```

```
INSERT INTO v3(c) VALUES (30);
```

OK

2.10 WITH CHECK OPTION (CONTINUED) - LOCAL

- Example:

```
CREATE TABLE t1 (
    c INT
);
```

```
CREATE OR REPLACE VIEW v1
AS
    SELECT
        c
    FROM
        t1
    WHERE
        c > 10;
```

No CHECK OPTION SPECIFIED

```
INSERT INTO v1(c)
VALUES (5);
```

OK

```
ALTER VIEW v2 AS
    SELECT
        c
    FROM
        v1
WITH LOCAL CHECK OPTION;
```

```
INSERT INTO v2(c)
VALUES (5);
```

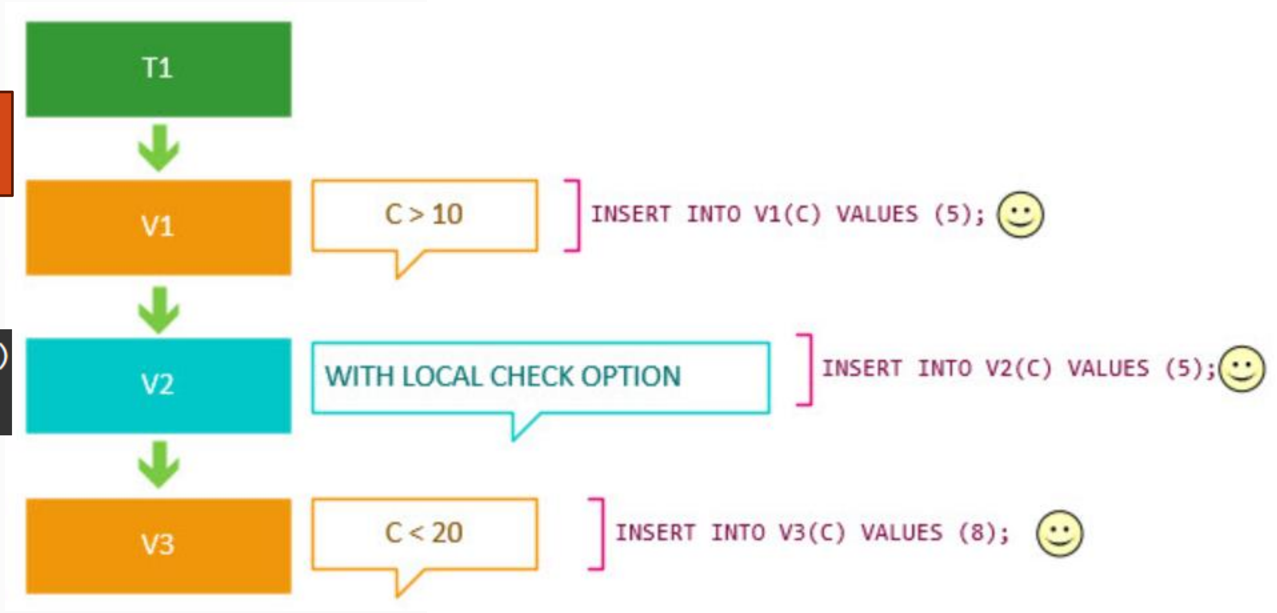
OK

```
CREATE OR REPLACE VIEW v3
AS
    SELECT c
    FROM v2
    WHERE c < 20;
```

```
INSERT INTO v3(c) VALUES (8);
```

OK

No CHECK OPTION SPECIFIED



```
INSERT INTO V1(C) VALUES (5);
INSERT INTO V2(C) VALUES (5);
INSERT INTO V3(C) VALUES (8);
```

2.10 VIEW UPDATABILITY WITH CHECK OPTION (CONTINUED)

- Example

```
CREATE VIEW Manager3Staff
AS SELECT *
FROM Staff
WHERE branchNo = 'B003'
WITH CHECK OPTION;
```

- Consider the queries:

```
UPDATE Manager3Staff
SET branchNo = 'B005'
WHERE staffNo = 'SG37';
```

- WITH CHECK OPTION clause in the definition of the view prevents this from happening, as it would cause the row to migrate from this horizontal view. (as branch B005 is not part of the view)

```
INSERT INTO Manager3Staff
VALUES('SL15', 'Mary', 'Black', 'Assistant', 'F', DATE'1967-06-21', 8000, 'B002');
```

- The specification of WITH CHECK OPTION would prevent the row from being inserted into the underlying Staff table and immediately disappearing from this view (as branch B002 is not part of the view)

2.11 ADVANTAGES OF VIEWS

- **Data independence:** A view can present a consistent, unchanging picture of the structure of the database, even if the underlying source tables are changed. If it affects on the structure view can be redefined.
- **Currency** Changes to any of the base tables in the defining query are immediately reflected in the view.
- **Improved security** Each user can be given the privilege to access the database only through a small set of views that contain the data appropriate for that user, thus restricting and controlling each user's access to the database.
- **Reduced complexity** A view can simplify queries, by drawing data from several tables into a single table, thereby transforming multi-table queries into single-table queries.
- **Convenience** Views can provide greater convenience to users as users are presented with only that part of the database that they need to see. This also reduces the complexity from the user's point of view.
- **Customization** Views provide a method to customize the appearance of the database, so that the same underlying base tables can be seen by different users in different ways.
- **Data integrity** If the WITH CHECK OPTION clause of the CREATE VIEW statement is used, then SQL ensures that no row that fails to satisfy the WHERE clause of the defining query is ever added to any of the underlying base table(s) through the view, thereby ensuring the integrity of the view.

2.12 DISADVANTAGES OF VIEWS

- **Update restriction,** in some cases, a view cannot be updated.
- **Structure restriction** The structure of a view is determined at the time of its creation. If the defining query was of the form SELECT * FROM . . . , then the * refers to the columns of the base table present when the view is created. If columns are subsequently added to the base table, then these columns will not appear in the view, unless the view is dropped and recreated.
- **Performance** There is a performance penalty to be paid when using a view. In some cases, this will be negligible; in other cases, it may be more problematic. For example, a view defined by a complex, multi-table query may take a long time to process, as the view resolution must join the tables together

2.14 VIEW MATERIALIZATION

- The problem with earlier approach is the time taken to perform the view resolution, particularly if the view is accessed frequently.
- An alternative approach, called **view materialization**, is to store the view as a temporary table in the database when the view is first queried. Thereafter, queries based on the materialized view can be much faster than recomputing the view each time.
- The difficulty with this approach is maintaining the currency of the view while the base table(s) are being updated.
- The process of updating a materialized view in response to changes to the underlying data is called **view maintenance**.
  - The basic aim of view maintenance is to apply only those changes necessary to the view to keep it current.
  - Suppose we insert/delete/update a row to the base table that does not meet the View definition it is fine. But if we insert/delete/update a row that satisfy the view definition then view too has to be refreshed. (There are ways to handle this)
- However, unlike PostgreSQL or Oracle Database, **MySQL does not natively support materialized views**, which can be a huge disadvantage. That, however, is no reason to stop using MySQL because it is a great database for major applications.

WRAP UP
THANK YOU!