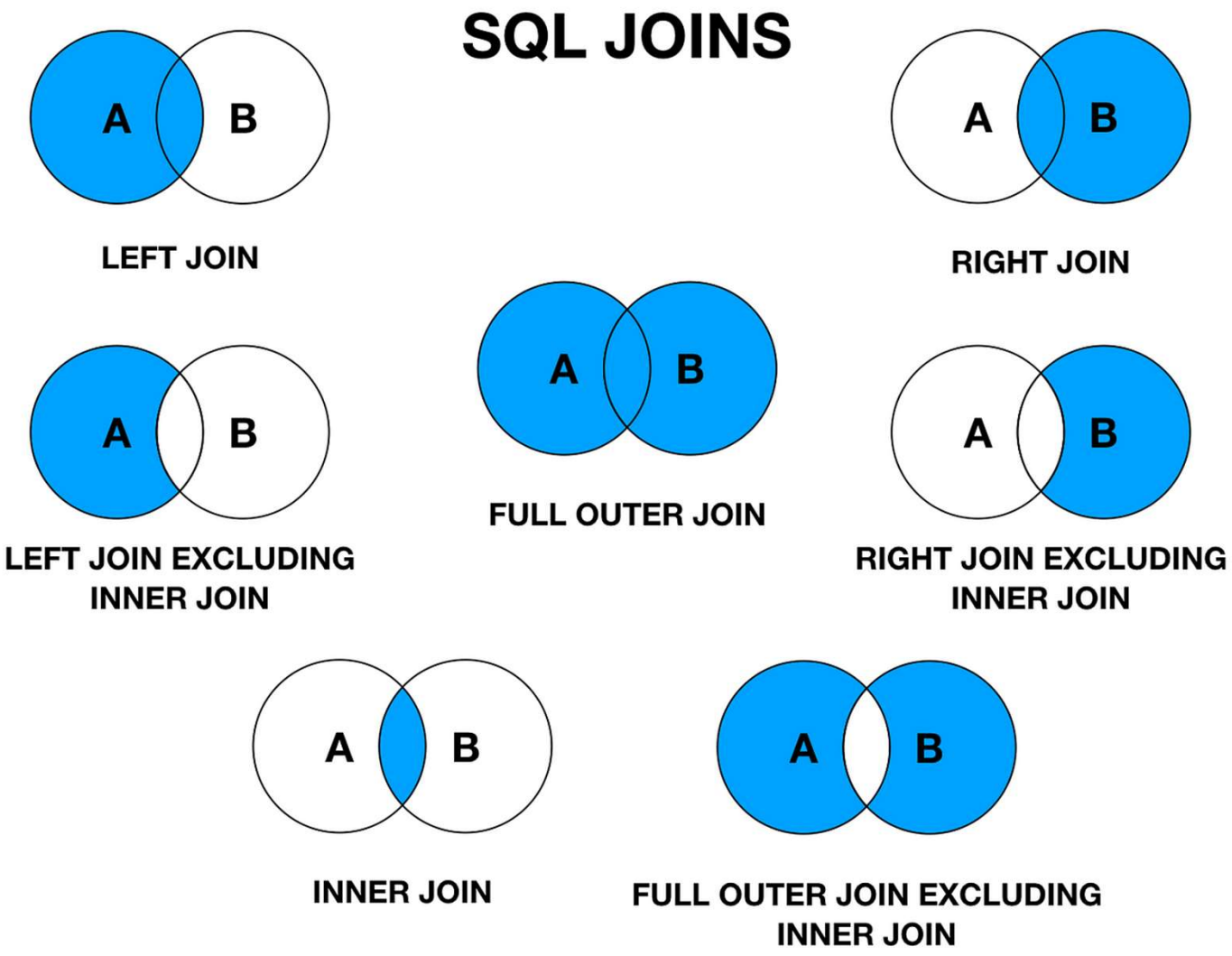
Section6:
Views, Transactions, Basic Authorization

Fanavaran Anisa
Iran Linux House
Linux & Open Source Training Center

Outline

- Views
- Transactions
- Integrity Constraints
- Domain vs User Defined Type
- Authorization

Joins n SQL



Outer Join Examples

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that

  course information is missing CS-347

  prereq information is missing CS-315

Left Outer Join

- *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

- In relational algebra: *course* ⟕ *prereq*

Right Outer Join

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- In relational algebra: *course* ⟖ *prereq*

Full Outer Join

- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

- In relational algebra: *course* ⟗ *prereq*

Full Outer Join

- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

- In relational algebra: *course* ⟗ *prereq*

Joined Types and Conditions

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Multiple Joins

You can join more than two tables together. First, two tables are joined, then the third table is joined to the result of the previous joining.

TOY AS t

toy_id	toy_name	cat_id
1	ball	3
2	spring	NULL
3	mouse	1
4	mouse	4
5	ball	1

CAT AS c

cat_id	cat_name	mom_id	owner_id
1	Kitty	5	1
2	Hugo	1	2
3	Sam	2	2
4	Misty	1	NULL

OWNER AS o

id	name
1	John Smith
2	Danielle Davis

JOIN & JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
JOIN cat c
  ON t.cat_id = c.cat_id
JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis

JOIN & LEFT JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
JOIN cat c
  ON t.cat_id = c.cat_id
LEFT JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL

LEFT JOIN & LEFT JOIN

```
SELECT
  t.toy_name,
  c.cat_name,
  o.name AS owner_name
FROM toy t
LEFT JOIN cat c
  ON t.cat_id = c.cat_id
LEFT JOIN owner o
  ON c.owner_id = o.id;
```

toy_name	cat_name	owner_name
ball	Kitty	John Smith
mouse	Kitty	John Smith
ball	Sam	Danielle Davis
mouse	Misty	NULL
spring	NULL	NULL

NON-EQUI (Theta) JOIN

You can use a **non-equality** in the ON condition, for example, to show **all different pairs** of rows.

TOY AS a

toy_id	toy_name	cat_id
3	mouse	1
5	ball	1
1	ball	3
4	mouse	4
2	spring	NULL

TOY AS b

cat_id	toy_id	toy_name
1	3	mouse
1	5	ball
3	1	ball
4	4	mouse
NULL	2	spring

```
SELECT
  a.toy_name AS toy_a,
  b.toy_name AS toy_b
FROM toy a
JOIN toy b
  ON a.cat_id < b.cat_id;
```

cat_a_id	toy_a	cat_b_id	toy_b
1	mouse	3	ball
1	ball	3	ball
1	mouse	4	mouse
1	ball	4	mouse
3	ball	4	mouse

LearnSQL.com

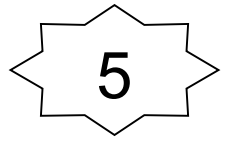
SQL EXECUTION ORDER

Order of a SQL Query

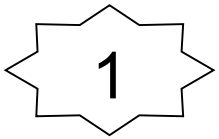
```
SELECT DISTINCT column, AGGREGATE(column)
FROM table1
JOIN table2
ON table1.column = table2.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count;
```

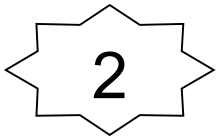
Order of a SQL Query

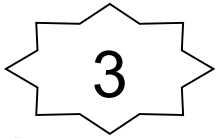
```
SELECT DISTINCT column, AGGREGATE(column)
FROM table1
JOIN table2
ON table1.column = table2.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count;
```

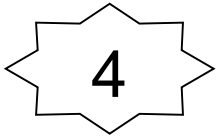


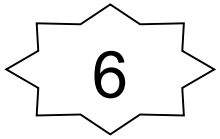
Window Functions Happen Here
Distinct Applies After This

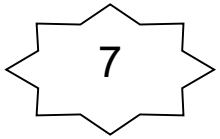






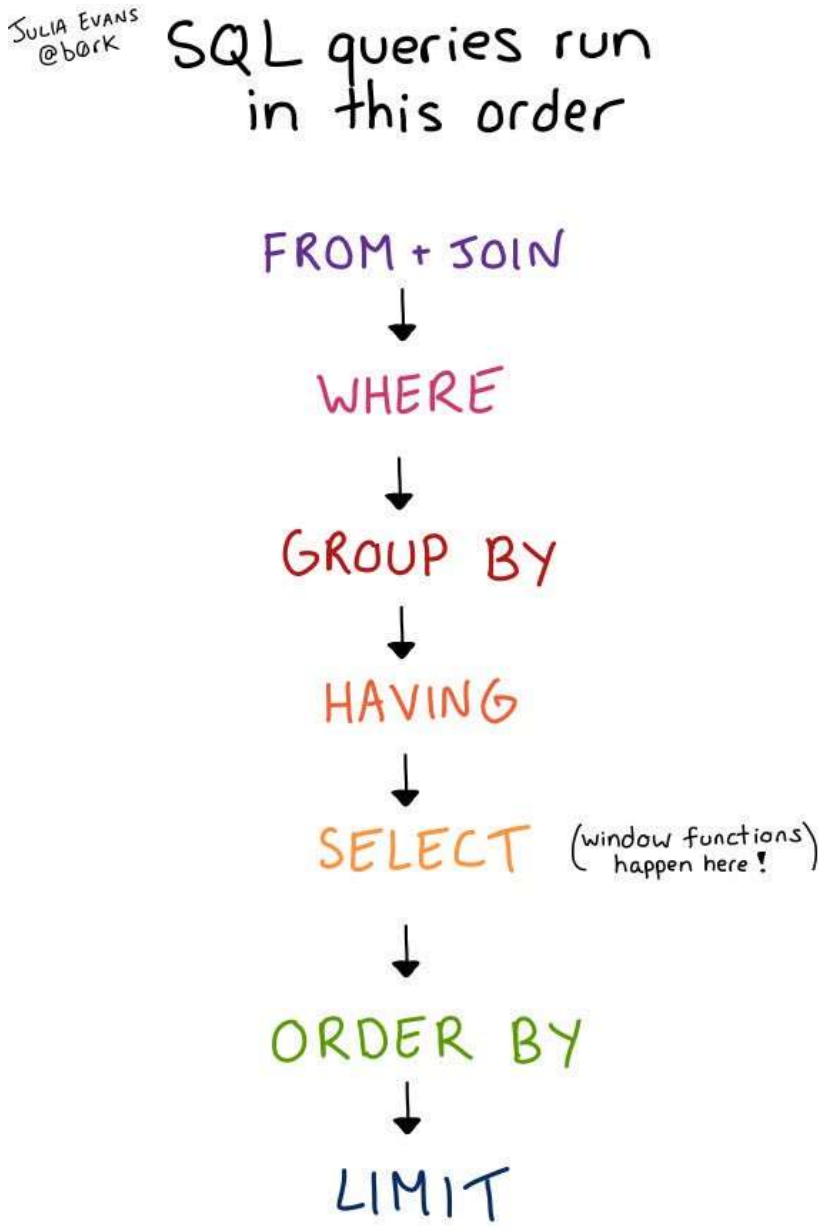




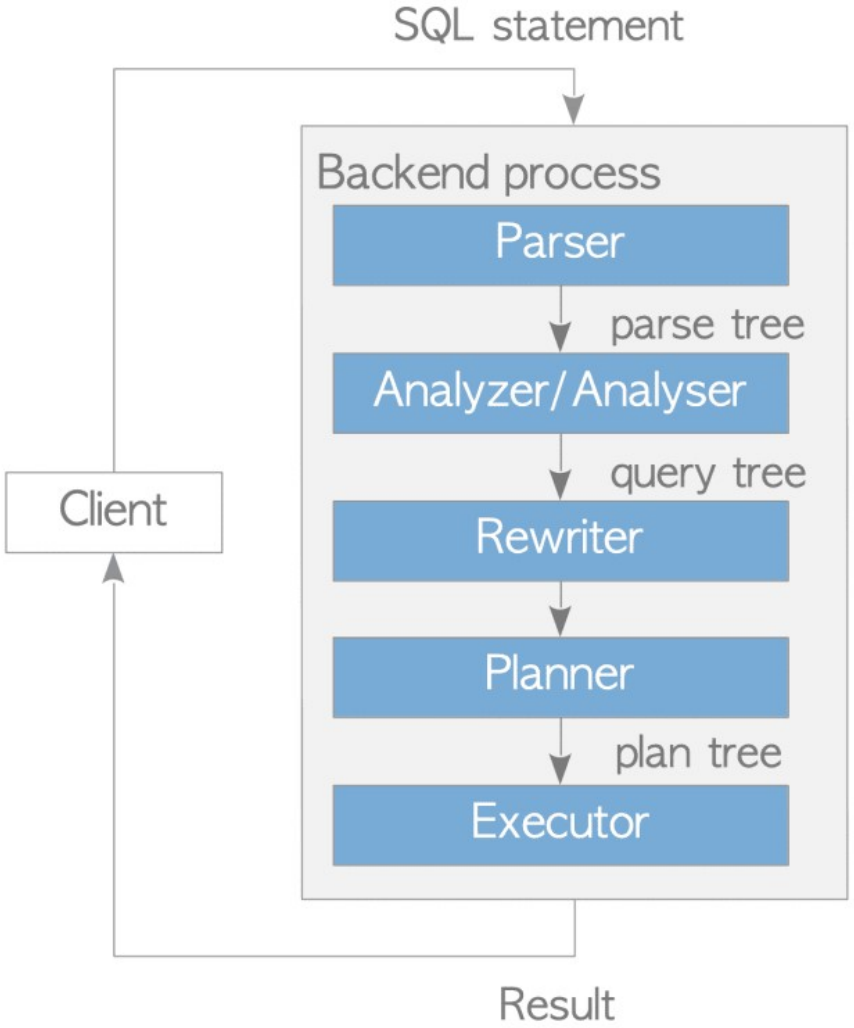


SQL Execution Order

1. FROM
2. ON
3. JOIN
4. WHERE
5. GROUP BY
6. WITH CUBE or WITH ROLLUP
7. HAVING
8. SELECT
9. Window Functions
10. DISTINCT
11. ORDER BY
12. TOP/LIMIT



Order of a SQL Query – Backend Tasks



Order of a SQL Query - Parser

Action:

- Tokenizes the SQL statement into individual elements (keywords, identifiers, operators).
- Generates a parse tree with nodes representing the structure of the SQL statement.

Result:

- **Parse tree with nodes for SELECT, column names, FROM, table name, WHERE clause, and ORDER BY.**

```
SELECT * FROM mytable
```

```
SelectStmt
├─ targetList
│  └─ ResTarget
│     └─ ColumnRef
│        └─ A_Star
└─ fromClause
   └─ RangeVar
      └─ relname = mytable
```

```
DELETE FROM abc WHERE id = 1
```

```
DeleteStmt
├─ relation
│  └─ RangeVar
│     └─ relname = abc
└─ whereClause
   └─ A_Expr
      ├─ kind = AEXPR_OP
      └─ ...
```

Order of a SQL Query - Analyzer

Action:

- Examines the parse tree.
- Performs semantic analysis.
- Resolves references to tables, columns, and other database objects.

Result:

- Ensures that "employee_id," "first_name," "last_name," "employees," and "department_id" exist in the database schema.
- **Generates a query tree with resolved references.**

Order of a SQL Query - Rewriter

- **Action:**
  - Analyzes the query tree.
  - Applies rules and transformations if necessary.
- **Result:**
  - May **rewrite the query tree to optimize performance** based on rules stored in the pg_rules system catalog.
  - Example: If there's a rule to use an index for the "department_id" filter, the rewriter might modify the query tree accordingly.

Order of a SQL Query - Rewriter

```
CREATE TABLE shoelace_log (
    sl_name     text,           -- shoelace changed
    sl_avail    integer,        -- new available value
    log_who     text,           -- who did it
    log_when    timestamp       -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Order of a SQL Query - Planner

Action:

- Generates an execution plan to minimize the overall cost.
- Considers factors like available indexes, join methods, and other optimization strategies.

Result:

- **Creates an execution plan specifying how to access and process the data efficiently.**
- May involve decisions such as using an index scan for the WHERE clause and sorting the result set based on the "last_name" column.

Order of a SQL Query - Executor

- **Action:**
  - Executes the SQL query based on the generated execution plan.
  - Coordinates with other components like the storage manager, buffer manager, and transaction manager.
- **Result:**
  - Retrieves rows from the "employees" table where "department_id" equals 10.
  - Orders the result set by the "last_name" column.
  - **Produces the final result set** with columns "employee_id," "first_name," and "last_name."

ROLLUP & CUBE

Grouping – The Old Way

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    brand,
    segment;
```

brand	segment	quantity
ABC	Premium	100
ABC	Basic	200
XYZ	Premium	100
XYZ	Basic	300

	brand character varying	segment character varying	sum bigint
1	[null]	[null]	700
2	XYZ	Basic	300
3	ABC	Premium	100
4	ABC	Basic	200
5	XYZ	Premium	100
6	ABC	[null]	300
7	XYZ	[null]	400
8	[null]	Basic	500
9	[null]	Premium	200

?

Grouping Set

A **grouping set** is a set of columns by which you group by using the GROUP BY clause.

```
SELECT
    brand,
    segment,
    SUM (quantity)
FROM
    sales
GROUP BY
    GROUPING SETS (
        (brand, segment),
        (brand),
        (segment),
        ()
    );
```

	brand	segment	quantity
▶	ABC	Premium	100
	ABC	Basic	200
	XYZ	Premium	100
	XYZ	Basic	300

	brand character varying	segment character varying	sum bigint
1	[null]	[null]	700
2	XYZ	Basic	300
3	ABC	Premium	100
4	ABC	Basic	200
5	XYZ	Premium	100
6	ABC	[null]	300
7	XYZ	[null]	400
8	[null]	Basic	500
9	[null]	Premium	200

CUBE

PostgreSQL CUBE is a subclause of the GROUP BY clause. The CUBE allows you to generate multiple grouping sets (**All Combinations**).

```
CUBE(c1,c2,c3)
```

```
GROUPING SETS (
(c1,c2,c3),
(c1,c2),
(c1,c3),
(c2,c3),
(c1), (c2), (c3), () )
```

brand	segment	quantity
ABC	Premium	100
ABC	Basic	200
XYZ	Premium	100
XYZ	Basic	300

Partial Cube :

```
SELECT c1, c2, c3, aggregate (c4)
FROM table_name
GROUP BY c1, CUBE (c1, c2);
```

brand	segment	sum
ABC	Basic	200
ABC	Premium	100
ABC	(Null)	300
XYZ	Basic	300
XYZ	Premium	100
XYZ	(Null)	400
(Null)	Basic	500
(Null)	Premium	200
(Null)	(Null)	700

Rollup

`Rollup is a subset of` CUBE that takes into the account the hierarchy of colums;

`Rollup(c1,c2,c3)`
generates only four grouping sets, assuming the hierarchy **c1 > c2 > c3** as follows:

```
GROUPING SETS (
(c1, c2, c3)
(c1, c2)
(c1)
()
)
```

brand	segment	quantity
ABC	Premium	100
ABC	Basic	200
XYZ	Premium	100
XYZ	Basic	300

In this case, the hierarchy is the segment > brand.

segment	brand	sum
Basic	ABC	200
Basic	XYZ	300
Basic	(Null)	500
Premium	ABC	100
Premium	XYZ	100
Premium	(Null)	200
(Null)	(Null)	700

Partial Rollup :

```
SELECT c1, c2, c3, aggregate (c4)
FROM table_name
GROUP BY c1, Rollup (c1, c2);
```

VIEWS

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Definition and Use

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to ***depend directly*** on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to ***depend on*** view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be ***recursive*** if it depends on itself.

Views Defined Using Other Views

- **create view *physics_fall_2017* as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id = section.course_id*
  and *course.dept_name* = 'Physics'
  and *section.semester* = 'Fall'
  and *section.year* = '2017';

- **create view *physics_fall_2017_watson* as**
  select *course_id*, *room_number*
  from *physics_fall_2017*
  where *building*= 'Watson';

View Expansion

- Expand the view :

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- To:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

View Expansion (Cont.)

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty*

  values ('30765', 'Green', 'Music');

- This insertion must be represented by the insertion into the *instructor* relation
  - Must have a value for salary.
- Two approaches
  - Reject the insert
  - Insert the tuple

    ('30765', 'Green', 'Music', null)

    into the *instructor* relation

Some Updates Cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name = department.dept_name*;
- **insert into** *instructor_info*
  values ('69987', 'White', 'Taylor');
- Issues
  - Which department, if multiple departments in Taylor?
  - What if no department is in Taylor?

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= 'History';
- What happens if we insert

  ('25566', 'Brown', 'Biology', 100000)

  into *history_instructors?*

View Updates in SQL

- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.

Materialized Views

- Certain database systems allow view relations to be physically stored.
  - Physical copy created when the view is defined.
  - Such views are called **Materialized view**:
- If relations used in the query are updated, the materialized view result becomes out of date
  - Need to **maintain** the view, by updating the view whenever the underlying relations are updated.

```
CREATE MATERIALIZED VIEW view_name
AS
query
WITH [NO] DATA;
```

Refreshing data for materialized views

```
REFRESH MATERIALIZED VIEW view_name;
```

Materialized Views

When you refresh data for a materialized view, PostgreSQL locks the entire table therefore you cannot query data against it. To avoid this, you can use the CONCURRENTLY option.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;
```

With CONCURRENTLY option, PostgreSQL creates a temporary updated version of the materialized view, compares two versions, and performs INSERT and UPDATE only the differences.

to refresh it with CONCURRENTLY option, you need to create a UNIQUE index for the view first.

Materialized Views

```
CREATE MATERIALIZED VIEW rental_by_category
AS
 SELECT c.name AS category,
    sum(p.amount) AS total_sales
   FROM (((((payment p
     JOIN rental r ON ((p.rental_id = r.rental_id)))
     JOIN inventory i ON ((r.inventory_id = i.inventory_id)))
     JOIN film f ON ((i.film_id = f.film_id)))
     JOIN film_category fc ON ((f.film_id = fc.film_id)))
     JOIN category c ON ((fc.category_id = c.category_id)))
  GROUP BY c.name
  ORDER BY sum(p.amount) DESC
WITH DATA;
```