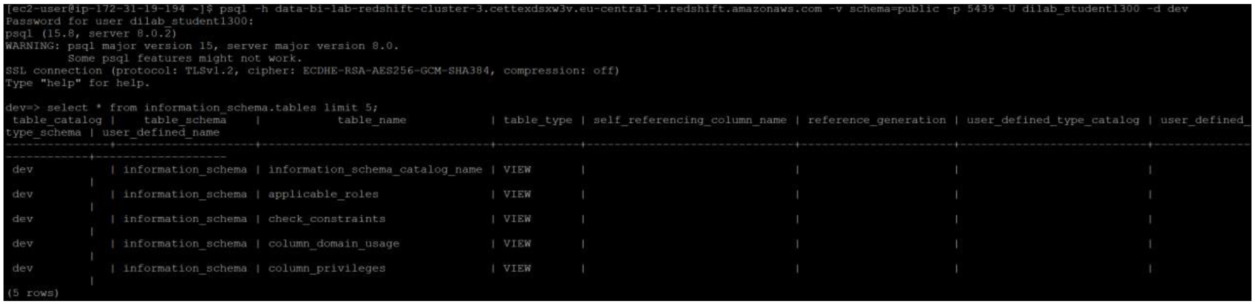
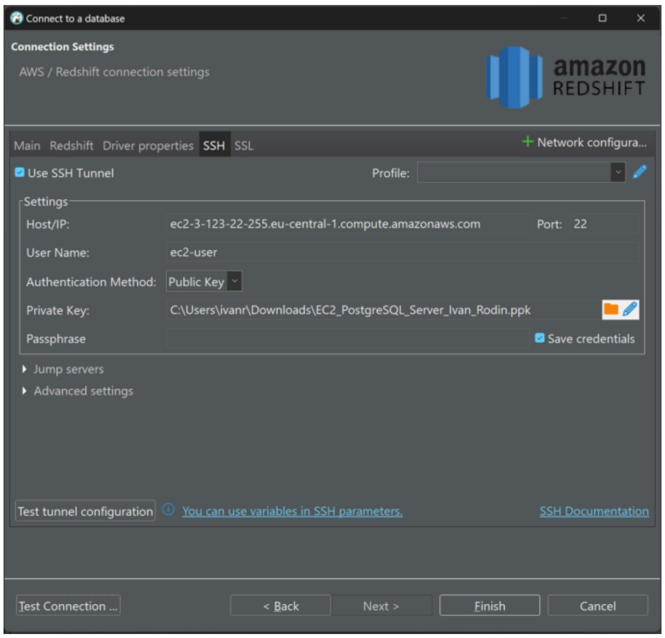
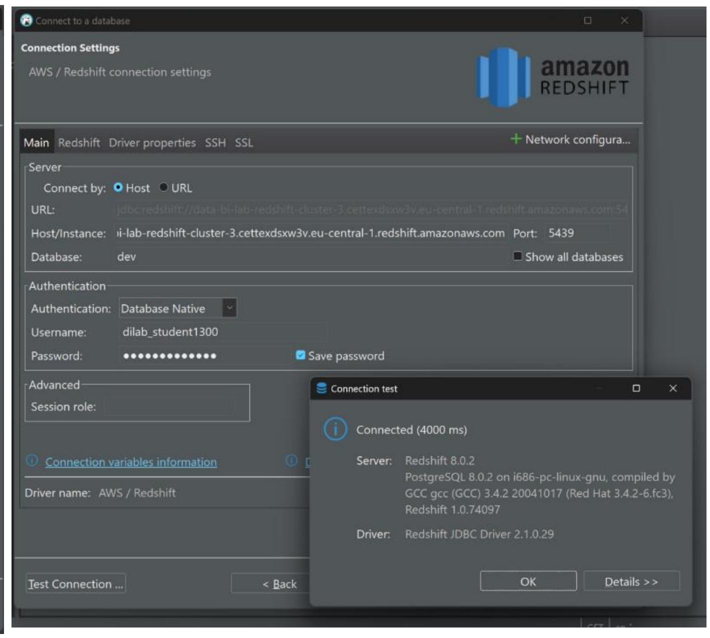
Created new EC2 instance with postgresql server and connected to Redshift in this EC2 instance:

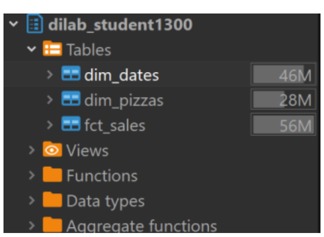


Connected Redshift Server using SSH tunnel with Dbeaver.





Created tables dim_dates, dim_pizzas and fct_sales to analyze types/sizes/pizzaname sales throughout periods of times:



Filled it with data from S3 platform using script:

(you can find full sql script in folder SQL_Scripts/Redshift_Script.sql)

```sql
copy dilab_student1300.dim_dates (
     date_id,
     date_year,
     date_month,
     date_monthname,
     date_monthday,
     date_yearday,
     date_weekdayname,
     date_calendarweek,
     date_formatteddate,
     date_quarter,
     date_yearquarter,
     date_yearmonth,
     date_yearcalendarweek,
     date_weekend,
     date_georgianholiday,
     date_period,
     date_cwstart,
     date_cwend,
     date_monthstart,
     date_monthend)
 from 's3://rodinivanbucket/pizzaplace_dwh/schema_bl_dm/dim_dates/dim_dates.csv'
 credentials
 'aws_iam_role=arn:aws:iam::260586643565:role/dilab-redshift-role'
 region 'eu-central-1'
 delimiter ','
 csv
 IGNOREHEADER 1;
```

```sql
copy dilab_student1300.fct_sales(
     customer_surr_id,
     order_surr_id,
     pizza_surr_id,
     event_time,
     address_surr_id,
     event_dt,
     quantity,
     price,
     fct_cost_order,
     insert_dt,
     update_dt)
 from 's3://rodinivanbucket/pizzaplace_dwh/schema_bl_dm/fct_sales/fct_sales.csv'
 credentials
 'aws_iam_role=arn:aws:iam::260586643565:role/dilab-redshift-role'
 region 'eu-central-1'
 delimiter ','
 csv
 IGNOREHEADER 1;
```

```sql
copy dilab_student1300.dim_pizzas(
     pizza_surr_id,
     pizza_src_id,
     source_system,
     source_entity,
     pizza_name,
     pizza_type_src_id,
     pizza_type_name,
     pizza_size_src_id,
     pizza_size_name,
     insert_dt,
     update_dt)
 from 's3://rodinivanbucket/pizzaplace_dwh/schema_bl_dm/dim_pizzas/dim_pizzas.csv'
 credentials
 'aws_iam_role=arn:aws:iam::260586643565:role/dilab-redshift-role'
 region 'eu-central-1'
 delimiter ','
 csv
 IGNOREHEADER 1;
```

Prepared analyzation:

Dim_dates:

	column	type	encoding	sortkey
1	date_id	date	az64	0
2	date_year	integer	az64	0
3	date_month	integer	az64	0
4	date_monthname	character varying(255)	lzo	0
5	date_monthday	integer	az64	0
6	date_yearday	integer	az64	0
7	date_weekdaynan	character varying(255)	lzo	0
8	date_calendarwee	integer	az64	0
9	date_formattedda	character varying(255)	lzo	0
10	date_quarter	character varying(255)	lzo	0
11	date_yearquarter	character varying(255)	lzo	0
12	date_yearmonth	character varying(255)	lzo	0
13	date_yearcalenda	character varying(255)	lzo	0
14	date_weekend	character varying(255)	lzo	0
15	date_georgianhol	character varying(255)	lzo	0
16	date_period	character varying(255)	lzo	0
17	date_cwstart	date	az64	0
18	date_cwend	date	az64	0
19	date_monthstart	date	az64	0
20	date_monthend	timestamp without time zone	az64	0

	table	diststyle
1	dim_dates	AUTO(ALL)

Dim_pizzas:

	column	type	encoding	sortkey
1	pizza_surr_id	bigint	az64	0
2	pizza_src_id	character varying(255)	lzo	0
3	source_system	character varying(255)	lzo	0
4	source_entity	character varying(255)	lzo	0
5	pizza_name	character varying(255)	lzo	0
6	pizza_type_src_id	integer	az64	0
7	pizza_type_name	character varying(255)	lzo	0
8	pizza_size_src_id	integer	az64	0
9	pizza_size_name	character varying(255)	lzo	0
10	insert_dt	timestamp without time zone	az64	0
11	update_dt	timestamp without time zone	az64	0

	table	diststyle
1	dim_pizzas	AUTO(ALL)

Fct_sales:

	column	type	encoding	sortkey
1	customer_surr_id	bigint	az64	1
2	order_surr_id	bigint	az64	0
3	pizza_surr_id	bigint	az64	0
4	event_time	character varying(255)	lzo	0
5	address_surr_id	bigint	az64	0
6	event_dt	date	az64	0
7	quantity	integer	az64	0
8	price	numeric(6,2)	az64	0
9	fct_cost_order	numeric(10,2)	az64	0
10	insert_dt	timestamp without time zone	az64	0
11	update_dt	timestamp without time zone	az64	0

	table	diststyle
1	fct_sales	AUTO(KEY(customer_surr_id))

Analyzed compression of fct_sales:

	Table	Column	Encoding	Est_reduction_pct
1	fct_sales	customer_surr_id	az64	0.00
2	fct_sales	order_surr_id	az64	68.73
3	fct_sales	pizza_surr_id	az64	86.88
4	fct_sales	event_time	zstd	79.69
5	fct_sales	address_surr_id	az64	86.79
6	fct_sales	event_dt	az64	70.54
7	fct_sales	quantity	az64	94.85
8	fct_sales	price	az64	81.54
9	fct_sales	fct_cost_order	az64	79.75
10	fct_sales	insert_dt	az64	61.02
11	fct_sales	update_dt	az64	61.02

Okay, basically whats happening here. We have our RAW encoding in withoutcomp. It means that no encoding or compression is implemented (BTW fun fact, that Redshift by default encodes if we do not specify RAW). So here we see the biggest size of all. In defaultcomp we see default encodings and it works okay, but if we let Redshift "think about it" using ANALYZE COMPRESIION, it can give us better ways to encode. In my case, only one 'text' table was encoded differently (zstd instead of lzo) than default so the size differences are not crucial but still can be seen.

	table	total_size_mb
1	fct_sales_withoutcomp	136
2	fct_sales_defaultcomp	65
3	fct_sales_analyzedcomp	62

Several runs of main_procedure: (in the forst one – evaluating plan, so it's much longer, then its okay, almost even)

```
Query execution time: 4.7317150000000003 seconds
dim_pizzas Distribution style: AUTO(ALL)
dim_dates Distribution style: AUTO(ALL)
fct_sales Distribution style: AUTO(KEY(customer_surr_id))
dim_pizzas Sortkeys: <NULL>
dim_dates Sortkeys: <NULL>
fct_sales Sortkeys: customer_surr_id
```

```
Query execution time: 0.034761 seconds
dim_pizzas Distribution style: AUTO(ALL)
dim_dates Distribution style: AUTO(ALL)
fct_sales Distribution style: AUTO(KEY(customer_surr_id))
dim_pizzas Sortkeys: <NULL>
dim_dates Sortkeys: <NULL>
fct_sales Sortkeys: customer_surr_id
```

```
Query execution time: 0.033945000000000003 seconds
dim_pizzas Distribution style: AUTO(ALL)
dim_dates Distribution style: AUTO(ALL)
fct_sales Distribution style: AUTO(KEY(customer_surr_id))
dim_pizzas Sortkeys: <NULL>
dim_dates Sortkeys: <NULL>
fct_sales Sortkeys: customer_surr_id
```

```
   QUERY PLAN
1  XN Merge  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
2    Merge Key: sum(fs2.quantity)
3    ->  XN Network  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
4          Send to leader
5          ->  XN Sort  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
6                Sort Key: sum(fs2.quantity)
7                ->  XN HashAggregate  (cost=26747.61..26747.61 rows=1 width=21)
8                      ->  XN Hash Join DS_DIST_ALL_NONE  (cost=32.98..26747.23 rows=50 width=21)
9                            Hash Cond: ("outer".pizza_surr_id = "inner".pizza_surr_id)
10                           ->  XN Hash Join DS_DIST_ALL_NONE  (cost=31.99..26705.23 rows=3241 width=12)
11                                 Hash Cond: ("outer".event_dt = "inner".date_id)
12                                 ->  XN Seq Scan on fct_sales fs2  (cost=0.00..11840.37 rows=1184037 width=16)
13                                 ->  XN Hash  (cost=31.97..31.97 rows=5 width=4)
14                                       ->  XN Seq Scan on dim_dates dd  (cost=0.00..31.97 rows=5 width=4)
15                                             Filter: ((upper((date_weekdayname)::text) = 'FRIDAY'::text) AND (date_monthday > 15))
16                           ->  XN Hash  (cost=0.99..0.99 rows=1 width=25)
17                                 ->  XN Seq Scan on dim_pizzas dp  (cost=0.00..0.99 rows=1 width=25)
18                                       Filter: (upper((pizza_type_name)::text) = 'CLASSIC'::text)
```

So in this task we are to optimize performance of our query in Redshift. First of all lets think about size of our files and distribution style. Pizzas (66 rows) is the smallest one and easily gets to ALL style. Dates is bigger, but not much (~2000 rows). In future, if we are to collect data for several decades it’s probably better to use EVEN (spread across all compute nodes evenly) but now ALL is good too. However if we pick EVEN a new type of JOIN is introduced DS_BCAST_INNER.

DS_BCAST_INNER is generally more optimized for scenarios where you have a small table being joined with a larger table, as it reduces the need for shuffling large datasets across nodes.

DS_DIST_ALL_NONE is best for operations where data can be processed efficiently without redistribution, particularly for small tables or operations that are already well-optimized for local processing.

So here I've decided to stick to one-node-operations DS_DIST_ALL_NONE, but in future with increasing of data we can look up to DS_BCAST_INNER.

For Sales tables we pick KEY distribution style because it's more preferable, but I changed key to pizza_surr_id because based on predicted queries (only pizza can be analyzed now, based on the tables I've loaded, so ……)

```
ALTER TABLE dilab_student1300.dim_dates ALTER DISTSTYLE ALL;
ALTER TABLE dilab_student1300.dim_pizzas ALTER DISTSTYLE ALL;
ALTER TABLE dilab_student1300.fct_sales ALTER DISTSTYLE KEY DISTKEY pizza_surr_id;

ALTER TABLE dilab_student1300.dim_dates ALTER SORTKEY (date_id);
ALTER TABLE dilab_student1300.dim_pizzas ALTER SORTKEY (pizza_surr_id);
ALTER TABLE dilab_student1300.fct_sales ALTER SORTKEY (event_dt, pizza_surr_id);
```

About Sortkeys its rather simple approach: we have filters (where conditions) only on yet small tables (pizzas and dates), so no actual need to create sortkeys on these filters (planner will still go with seq_scan without any crucial impact on performance), however we still can have sortkeys on join columns (FK's) and it's never a bad idea to have them in case of joins.

```
Query execution time: 0.031433000000000003 seconds
```

```
QUERY PLAN
XN Merge  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
  Merge Key: sum(fs2.quantity)
  -> XN Network  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
      Send to leader
      -> XN Sort  (cost=1000000026747.62..1000000026747.62 rows=1 width=21)
          Sort Key: sum(fs2.quantity)
          -> XN HashAggregate  (cost=26747.61..26747.61 rows=1 width=21)
              -> XN Hash Join DS_DIST_ALL_NONE  (cost=32.98..26747.23 rows=50 width=21)
                  Hash Cond: ("outer".pizza_surr_id = "inner".pizza_surr_id)
                  -> XN Hash Join DS_DIST_ALL_NONE  (cost=31.99..26705.23 rows=3241 width=12)
                      Hash Cond: ("outer".event_dt = "inner".date_id)
                      -> XN Seq Scan on fct_sales fs2  (cost=0.00..11840.37 rows=1184037 width=16)
                      -> XN Hash  (cost=31.97..31.97 rows=5 width=4)
                          -> XN Seq Scan on dim_dates dd  (cost=0.00..31.97 rows=5 width=4)
                              Filter: ((upper((date_weekdayname)::text) = 'FRIDAY'::text) AND (date_monthday > 15))
                  -> XN Hash  (cost=0.99..0.99 rows=1 width=25)
                      -> XN Seq Scan on dim_pizzas dp  (cost=0.00..0.99 rows=1 width=25)
                          Filter: (upper((pizza_type_name)::text) = 'CLASSIC'::text)
```

COPY:

Time of copy of 1 gz file 3.5 gb (9 mins):

Start time	Wed Sep 04 22:46:48 GET 2024
Finish time	Wed Sep 04 22:57:01 GET 2024

Time of copy 4 paraquet files 600 mb each (4 mins):

Start time	Wed Sep 04 22:58:08 GET 2024
Finish time	Wed Sep 04 23:02:22 GET 2024

The Idea behind this task is to analyze why 1 gz file is longer to copy into table in redshift than 4 files of paraquet. The answer is pretty simple: .gz is a compressed text file that we need to decompress in order to retrieve data from it, while .paraquet is a columnar format more preferable in I/O operations and analysis.

EXTERNAL TABLES

Created schema for database in GLUE. (now we see all our tables as external tables from s3 bucket, that crawler collected for us in 1st task of course)

```
CREATE EXTERNAL SCHEMA if not exists user_dilab_student1300_ext
FROM DATA catalog
DATABASE 'pizzaplace_rodin_db'
IAM_ROLE 'arn:aws:iam::260586643565:role/dilab-redshift-role';
```

Here we see procedure of exporting data by months to s3 buckets as a csv files (later be used as partitions):

```
CREATE OR REPLACE PROCEDURE dilab_student1300.export_data_by_month()
LANGUAGE plpgsql
AS $$
DECLARE
    record_month RECORD;
    base_s3_path TEXT := 's3://rodinivanbucket/task_3_redshift/';
    unload_sql VARCHAR(500);
BEGIN
    -- Get distinct months from the table
    FOR record_month IN
        SELECT DISTINCT to_char(event_dt, 'YYYY-MM') AS month
        FROM dilab_student1300.fct_sales
    LOOP
        unload_sql :=
            'UNLOAD (''SELECT * FROM dilab_student1300.fct_sales WHERE to_char(event_dt, ''''YYYY-MM'''') = ''''' ||
            record_month.month || ''''' '') ' ||
            'TO ''s3://rodinivanbucket/task_3_redshift/' || record_month.month || '/' || record_month.month ||'.csv'' '||
            'IAM_ROLE ''arn:aws:iam::260586643565:role/dilab-redshift-role'' ' ||
            'ALLOWOVERWRITE ' ||
            'PARALLEL OFF ' ||
            'DELIMITER AS '','' ' ||
            'FORMAT AS CSV;';
        EXECUTE unload_sql;
    END LOOP;
END $$;
```

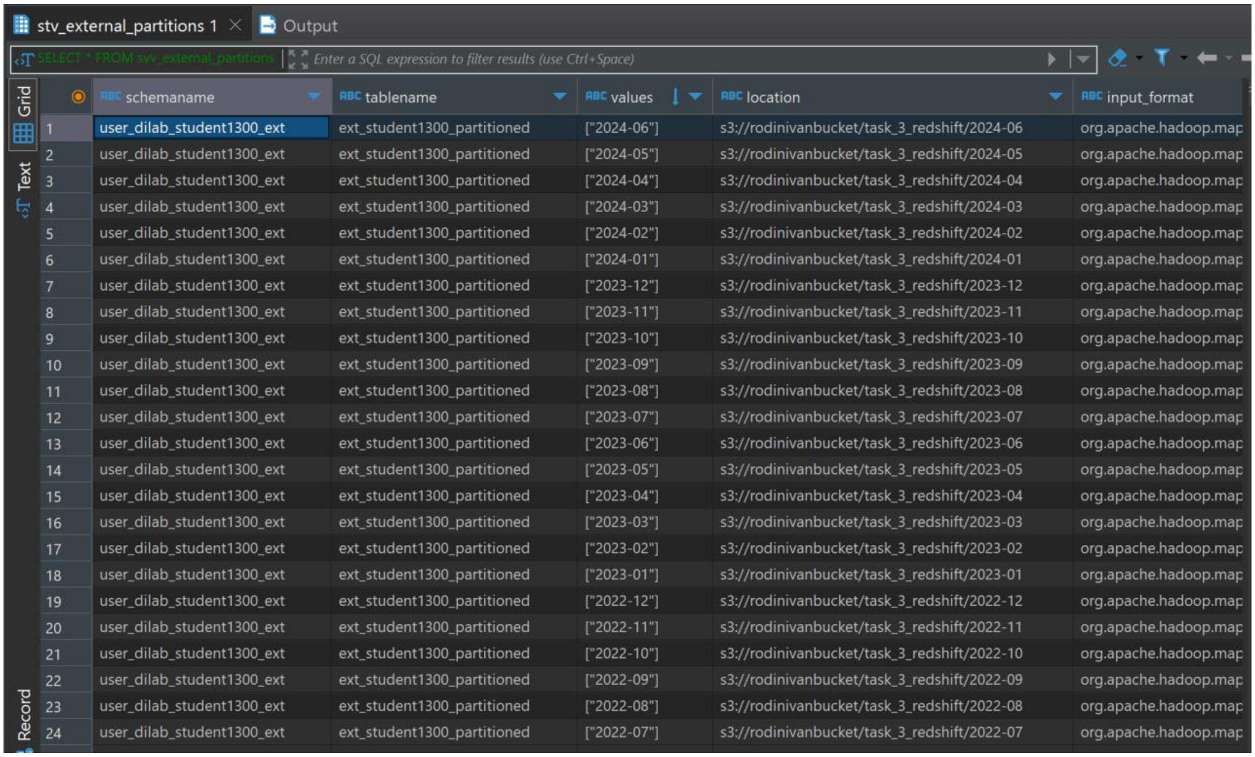
Created external table with partitions by months (in this case just char(10) for now):

```sql
CREATE EXTERNAL TABLE user_dilab_student1300_ext.ext_student1300_partitioned (
        customer_surr_id bigint,
        order_surr_id bigint,
        pizza_surr_id bigint,
        event_time varchar(255),
        address_surr_id bigint,
        event_dt date,
        quantity int,
        price decimal(6,2),
        fct_cost_order decimal(10,2),
        insert_dt timestamp,
        update_dt timestamp
)
PARTITIONED BY (saledate char(10))
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION 's3://rodinivanbucket/pizzaplace_dwh/schema_bl_dm/fct_sales/';
```

A part of partitions created (actually created all, but you can find whole script in SQL_Script/Redshift_Script.sql):

```sql
ALTER TABLE user_dilab_student1300_ext.ext_student1300_partitioned ADD
PARTITION (saledate='2022-07')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-07/'
PARTITION (saledate='2022-08')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-08/'
PARTITION (saledate='2022-09')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-09/'
PARTITION (saledate='2022-10')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-10/'
PARTITION (saledate='2022-11')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-11/'
PARTITION (saledate='2022-12')
LOCATION 's3://rodinivanbucket/task_3_redshift/2022-12/'
PARTITION (saledate='2023-01')
LOCATION 's3://rodinivanbucket/task_3_redshift/2023-01/'
PARTITION (saledate='2023-02')
LOCATION 's3://rodinivanbucket/task_3_redshift/2023-02/'
PARTITION (saledate='2023-03')
```

By checking svv_external_partitions table we can see our partitions created



	schemaname	tablename	values	location	input_format
1	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-06"]	s3://rodinivanbucket/task_3_redshift/2024-06	org.apache.hadoop.map
2	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-05"]	s3://rodinivanbucket/task_3_redshift/2024-05	org.apache.hadoop.map
3	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-04"]	s3://rodinivanbucket/task_3_redshift/2024-04	org.apache.hadoop.map
4	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-03"]	s3://rodinivanbucket/task_3_redshift/2024-03	org.apache.hadoop.map
5	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-02"]	s3://rodinivanbucket/task_3_redshift/2024-02	org.apache.hadoop.map
6	user_dilab_student1300_ext	ext_student1300_partitioned	["2024-01"]	s3://rodinivanbucket/task_3_redshift/2024-01	org.apache.hadoop.map
7	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-12"]	s3://rodinivanbucket/task_3_redshift/2023-12	org.apache.hadoop.map
8	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-11"]	s3://rodinivanbucket/task_3_redshift/2023-11	org.apache.hadoop.map
9	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-10"]	s3://rodinivanbucket/task_3_redshift/2023-10	org.apache.hadoop.map
10	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-09"]	s3://rodinivanbucket/task_3_redshift/2023-09	org.apache.hadoop.map
11	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-08"]	s3://rodinivanbucket/task_3_redshift/2023-08	org.apache.hadoop.map
12	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-07"]	s3://rodinivanbucket/task_3_redshift/2023-07	org.apache.hadoop.map
13	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-06"]	s3://rodinivanbucket/task_3_redshift/2023-06	org.apache.hadoop.map
14	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-05"]	s3://rodinivanbucket/task_3_redshift/2023-05	org.apache.hadoop.map
15	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-04"]	s3://rodinivanbucket/task_3_redshift/2023-04	org.apache.hadoop.map
16	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-03"]	s3://rodinivanbucket/task_3_redshift/2023-03	org.apache.hadoop.map
17	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-02"]	s3://rodinivanbucket/task_3_redshift/2023-02	org.apache.hadoop.map
18	user_dilab_student1300_ext	ext_student1300_partitioned	["2023-01"]	s3://rodinivanbucket/task_3_redshift/2023-01	org.apache.hadoop.map
19	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-12"]	s3://rodinivanbucket/task_3_redshift/2022-12	org.apache.hadoop.map
20	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-11"]	s3://rodinivanbucket/task_3_redshift/2022-11	org.apache.hadoop.map
21	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-10"]	s3://rodinivanbucket/task_3_redshift/2022-10	org.apache.hadoop.map
22	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-09"]	s3://rodinivanbucket/task_3_redshift/2022-09	org.apache.hadoop.map
23	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-08"]	s3://rodinivanbucket/task_3_redshift/2022-08	org.apache.hadoop.map
24	user_dilab_student1300_ext	ext_student1300_partitioned	["2022-07"]	s3://rodinivanbucket/task_3_redshift/2022-07	org.apache.hadoop.map

After that a procedure to check if number of rows match for fct_sales and ext_student1300_partitioned tables for each month matches created:

```sql
CREATE OR REPLACE PROCEDURE dilab_student1300.partitions_check()
LANGUAGE plpgsql
AS $$
DECLARE
    count_fct_sales int;
    count_ext_table int;
    curr_date date := '2022-07-01'::date;
    end_date date := '2024-06-30'::date;
BEGIN
    LOOP
        SELECT count(*)
        INTO count_fct_sales
        FROM user_dilab_student1300_ext.fct_sales fs2
        WHERE extract(YEAR FROM event_dt::date) = extract(YEAR FROM curr_date)
        AND extract(MONTH FROM event_dt::date) = extract(MONTH FROM curr_date);

        SELECT count(*)
        INTO count_ext_table
        FROM user_dilab_student1300_ext.ext_student1300_partitioned esp
        WHERE saledate = to_char(curr_date, 'YYYY-MM');

        IF count_fct_sales - count_ext_table != 0 THEN
            RAISE NOTICE 'Check Failed for date %', curr_date;
            EXIT;
        ELSE
            RAISE NOTICE 'Check Passed for date %', curr_date;
        END IF;

        curr_date := curr_date + interval '1 month';

        EXIT WHEN curr_date > end_date;
    END LOOP;
END $$;
```

And in this procedure output we see that everything is working as expected:

```
Check Passed for date 2022-07-01
Check Passed for date 2022-08-01
Check Passed for date 2022-09-01
Check Passed for date 2022-10-01
Check Passed for date 2022-11-01
Check Passed for date 2022-12-01
Check Passed for date 2023-01-01
Check Passed for date 2023-02-01
Check Passed for date 2023-03-01
Check Passed for date 2023-04-01
Check Passed for date 2023-05-01
Check Passed for date 2023-06-01
Check Passed for date 2023-07-01
Check Passed for date 2023-08-01
Check Passed for date 2023-09-01
Check Passed for date 2023-10-01
Check Passed for date 2023-11-01
Check Passed for date 2023-12-01
Check Passed for date 2024-01-01
Check Passed for date 2024-02-01
Check Passed for date 2024-03-01
Check Passed for date 2024-04-01
Check Passed for date 2024-05-01
Check Passed for date 2024-06-01
```