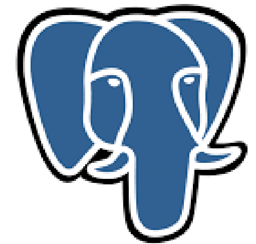
DATA MODEL & Partitioning

IDEO-LAB
13th September 2025

VERSION 1.3
By *Guillaume Oneill*



AGENDA

AGENDA

- PARTITIONING GENERAL CONSIDERATIONS
- Partitioning Implementation
- DJANGO MAIN CONCERNS
- MIGRATION PLAN
- PERFORMANCE CONCERNS
- QUESTIONS & REMARKS

PARTITIONING GENERAL CONSIDERATIONS

Partitioning Key's Advantages

- **Query performance can be improved dramatically** in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning effectively substitutes for the upper tree levels of indexes, making it more likely that the heavily-used parts of the indexes fit in memory.

- When queries or updates access a large percentage of a single partition, performance can be improved by using a **sequential scan of that partition instead of using an index**, which would require random-access reads scattered across the whole table.

- **Bulk loads** and deletes can be accomplished by adding or removing partitions, if the usage pattern is accounted for in the partitioning design. Dropping an individual partition using DROP TABLE, or doing ALTER TABLE DETACH PARTITION, is far faster than a bulk operation. These commands also entirely avoid the VACUUM overhead caused by a bulk DELETE.

- Seldom-used data can be migrated to cheaper and slower storage media.

QUERIES & BIG DATA
Challenges

- 3 SQL Tables are concentrating an important volume of essential Data :
  - accomodationConsumption : **3.4** Millions of Rows
  - accomodationTemperature : **13,2** Millions of Rows
  - accomodationHumidity : **12.7** Millions of Rows
- **85 %** of our queries are accessing theses 3 Tables :
  - complex and costly Where arguments :
    - Multiple Join (*Nested Mode*), Outer and Inner Join
    - Between clause on DateTime Range -> will be removed soon
    - Group By and Having group on Residence, Date, Accomodation, ...
    - Distinct Clauses
    - Order By
- Multiple indexs on lot of columns, most of them are ineffective:
- Indexs scan on Giant Index (***from 3 to 13 Millions of rows***)
- Use a large amout of shared Buffers
- Exposure to excessive Locking and Deadlock (*While Cron are running*)
- Timeout on specifc costly Functions :
  - Import Excel
  - **Export Excel (UR)**
  - Index calculating
- Pre-Computing are mostly ineffective !

How it works

- The partitioned accomodation table - the table we will split into smaller tables by “residence key”-
- It doesn't hold any data. It exists as a parent to the partitions and a blueprint for the table schema.
- One important thing to understand about a partitioned table is that the partitions themselves are also tables
- They are created individually, and we can query them separately

Solutions considered

- Table segregation :
  - By Residence & Year
  - Use **LIST** Partitioning

- Outsource Monthly records
- Optimize Index Scan (*Cf Query Planner settings*),
  - Partitioned Index
  - Prefer multi-column indexes
  - Schedule the creation of indexes on requests and specific queries

- Optimizing our query to benefit from partitioning,
- Optimize our Logical keys, size and format :
  - Remove all Long Choices List defined in VarChar by : **PositiveInteger**

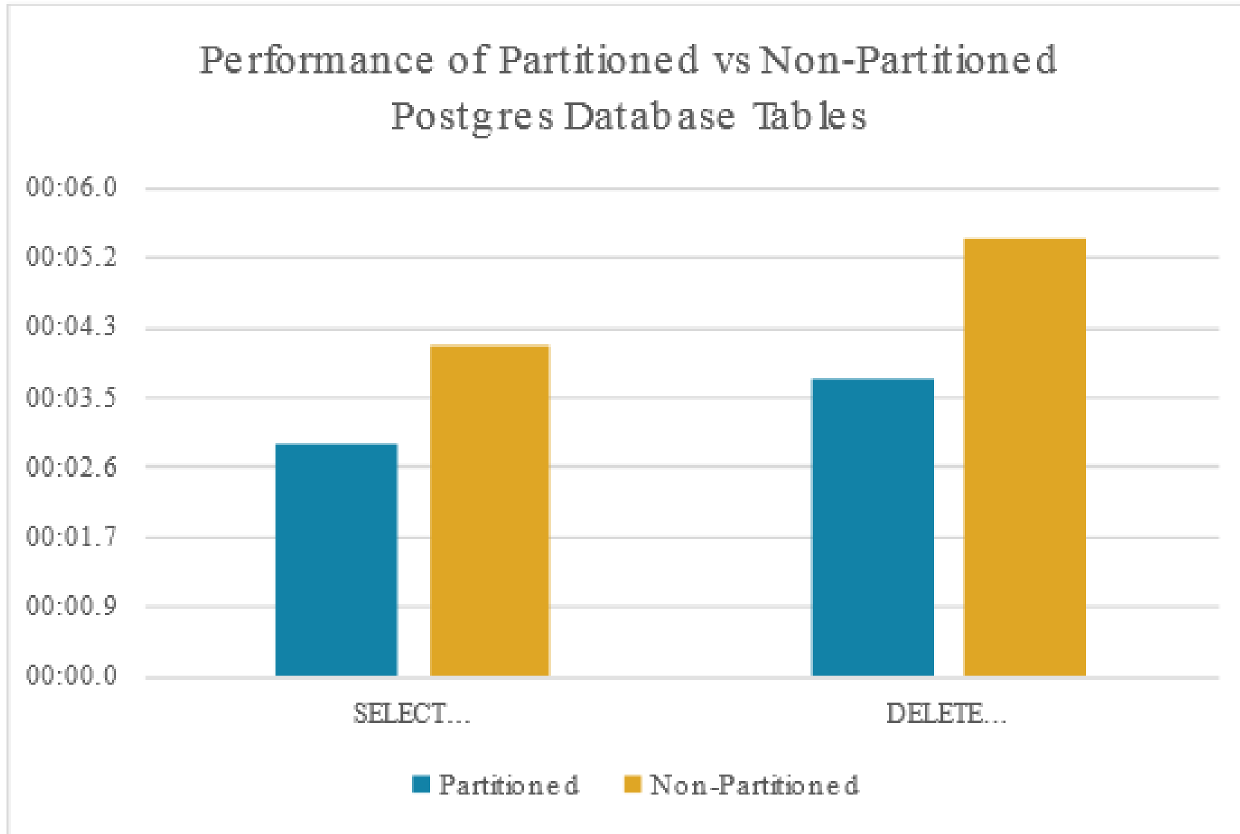
Base Rules

- Django's ORM doesn't have built-in support for partitioned tables,

- We need for using partitions in our Kocliko application, little extra work.

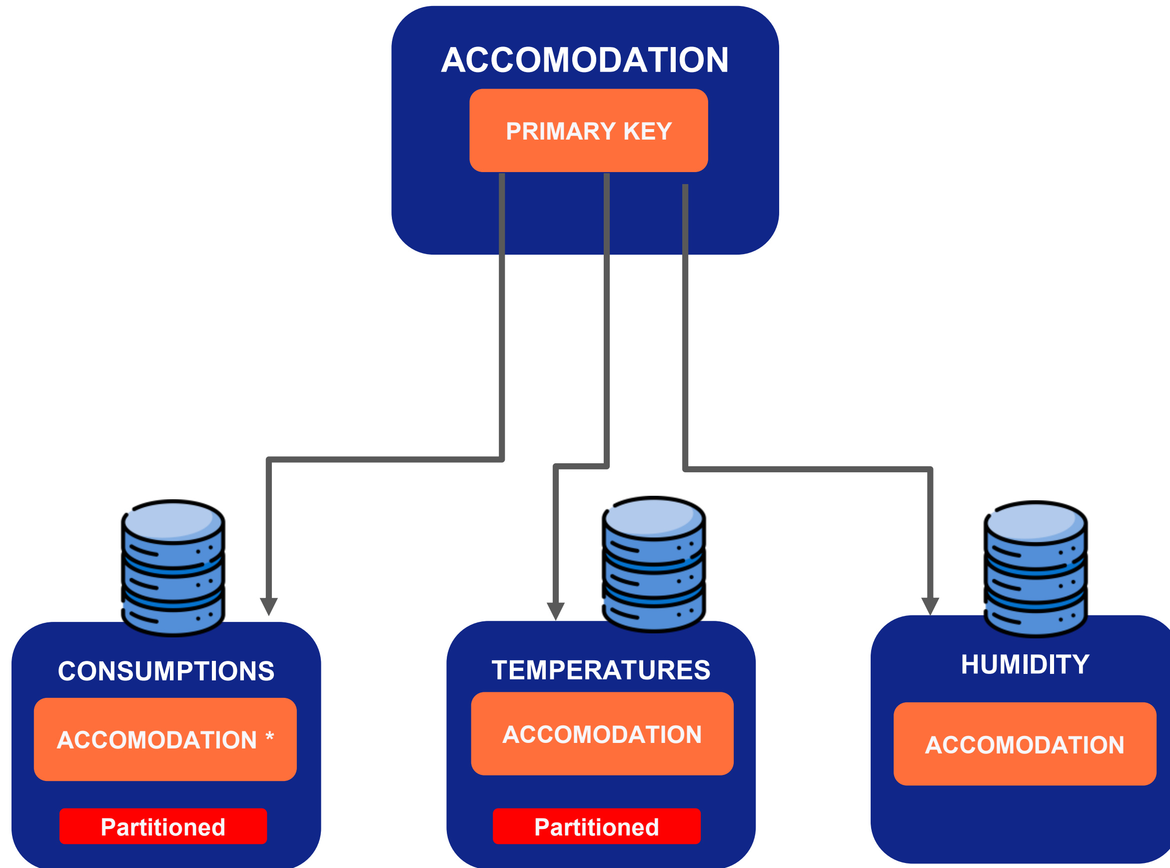
- Another option is to use a package called [django-postgres-extra]

Performances Improvement



Partitioning Implementation

Accomodation Partitioned Data Models



MAJOR DATA MODELS STATISTICS

RESIDENCES ACTIVES	:	121
ACCOMODATIONS	:	13357
SENSORS	:	20312
COUNTERS BUILDING	:	277
AVERAGE READ PROPERTIES	:	39970
CONSUMPTIONS RECORDS *	:	3 219 923
TEMPERATURES RECORDS	:	**13 120 091**
HUMIDITY RECORDS	:	12 451 418
API DATA SENSORS	:	**6 906 075**

Table Partitioning

- **Table Partitioning:** When you deal with large volumes of data, consider using PostgreSQL's partitioning functionality. This involves splitting large tables into smaller parts, which can significantly speed up queries and reduce maintenance overhead.

- https://www.postgresql.org/docs/current/ddl-partitioning.html
- https://rodoq.medium.com/partition-an-existing-table-on-postgresql-480b84582e8d

- **Candidates SQL Tables :**
  - **AccomodationConsumption**
  - **AccomodationTemperature**
  - **AccomodationHumidity**
  - **apiDataSensor**
  - **Various logging and Messages Tables**
  - **Sensor + Sensor Status**

Tablespace Partition Key

- Several Tracks on the way :
  - By Year
  - By Year – Month
  - By Residence

- The Partition key should be scalable
- Building a Partition simulator

Kocliko Base Partitioning Key Segregation

- Use Cron to quantify or estimate Records per each Residence and Year.

- Load Balance and group « Residence » per Partition, one or more.

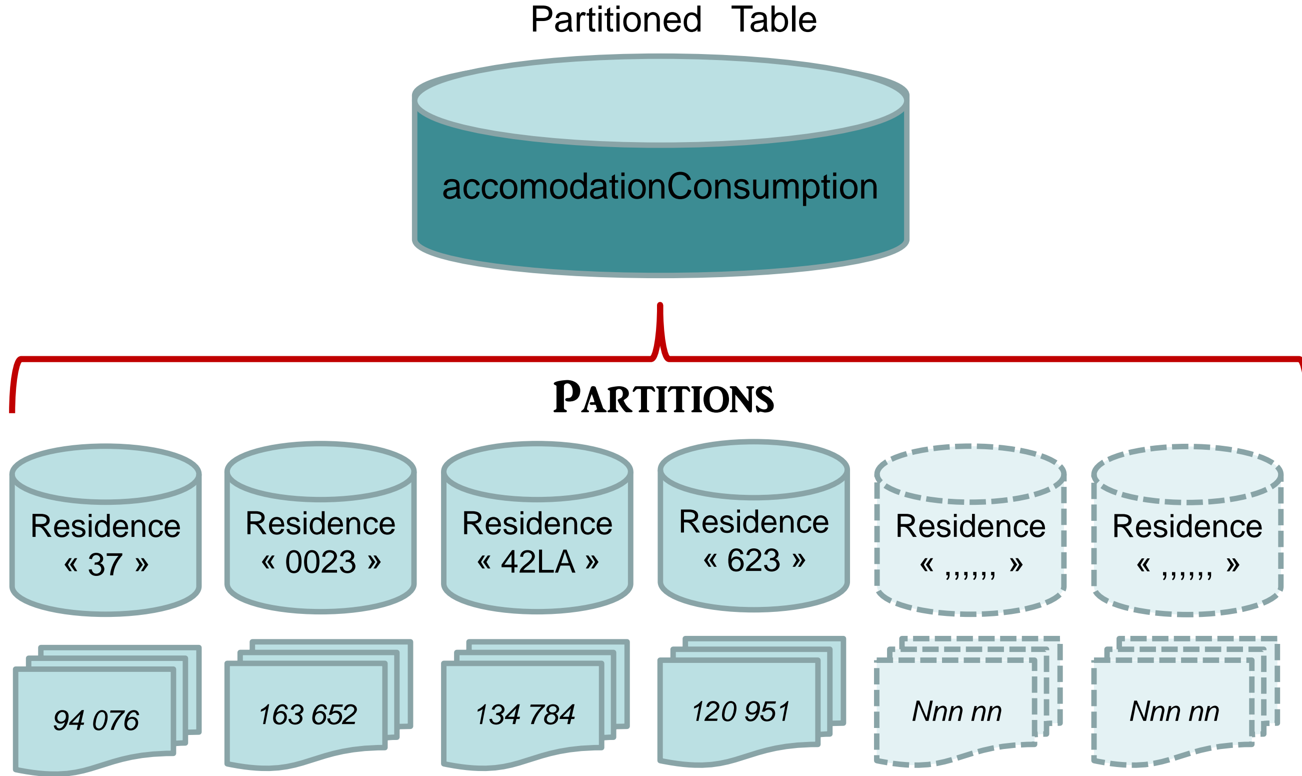
- Automate this Partitioning script within the **Residence Backoffice** based :
  - Number of Accomodations
  - Number of Sensors
  - Number of Properties per Sensors
  - Number of Day’s Collect (*By Default : 365*)

- For each new residence :
  - Edit and Format new Migration script manually
  - Django Migrate execution.

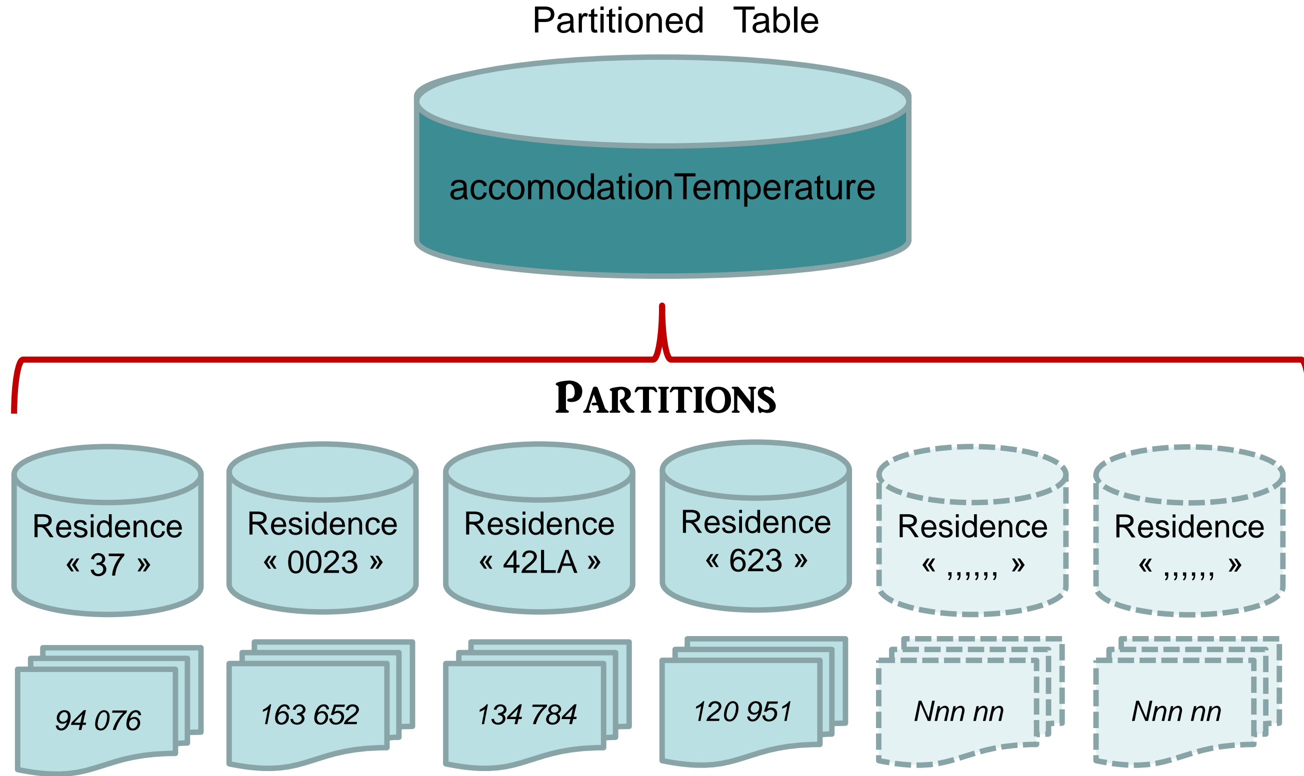
Sample of Partitioned Data Model

```
class AccomodationConsumption_Partitioned(AccomodationConsumptionBase, PostgresPartitionedModel, checkSumData):
    #  ====================================================
    #  ==== ATTACHED FOREIGN KEYS                      ===
    #  ====================================================
    accomodation            = models.ForeignKey(Accomodation, on_delete=models.CASCADE, related_name='%(class)s_related'
    residence               = models.ForeignKey('api.residence', null=True, blank=True, on_delete=models.CASCADE,   rel

    class PartitioningMeta:
        method = PostgresPartitioningMethod.LIST
        key    = ["residence_id"]

    class Meta:
        app_label           = 'api'
        verbose_name        = _('CONSUMPTION  PARTITIONED')
        verbose_name_plural = _('CONSUMPTIONS PARTITIONED')
```

Conso - Partitioned Table



Temp - Partitioned Table



DJANGO MAIN CONCERNS

Creating a Partitioned Table - Use PG Extra

- **Create new Partitioned Main Data Model**
  - **Use Partitioning Key Method : "LIST"**
  - **Add Partition Key : if Foreign Key use SQL Real DDL Definition**
  - **ADD Partition Mixin : PostgresPartitionedModel (*Django PostgreSQL Extra*)**

- **Django PgMakeMigrations**
  - **Create Main Partition Data Model**
  - **Create Default Partition**

- **Django migrate**
- **Adding Manually extra LIST partition :**
  - **One or more Residence Code/ID per partition**

Django and Partitioning

django-postgres-extra
master

Search docs

Overview

Installation
Managers & Models
HStore
Indexes
Conflict handling
Deletion
Table partitioning
Creating partitioned tables
Automatically managing partitions
Manually managing partitions
Expressions
Annotations
Locking
Schema
Settings
API Reference
Major releases

Read the Docs v: master

Warning

Table partitioning is a relatively new and advanded PostgreSQL feature. It has plenty of ways to shoot yourself in the foot with.

We HIGHLY RECOMMEND you only use this feature if you're already deeply familiar with table partitioning and aware of its advantages and disadvantages.

Do study the PostgreSQL documentation carefully.

Table partitioning

`PostgresPartitionedModel` adds support for PostgreSQL Declarative Table Partitioning.

The following partitioning methods are available:

`PARTITION BY RANGE`

`PARTITION BY LIST`

`PARTITION BY HASH`

Note

Although table partitioning is available in PostgreSQL 10.x, it is highly recommended you use PostgresSQL 11.x. Table partitioning got a major upgrade in PostgreSQL 11.x.

PostgreSQL 10.x does not support creating foreign keys to/from partitioned tables and does not automatically create an index across all partitions.

Django Partitioning - Get’s started

```
from psqlextra.types import PostgresPartitioningMethod

from psqlextra.models import PostgresPartitionedModel

class accomodationConsumption_partitioned(PostgresPartitionedModel):
    class PartitioningMeta:
        method = PostgresPartitioningMethod.RANGE
        key       = [‘residence_id’]

    uid                      = models.TextField()
    residence_id     = models.integerField()
    nnnnnn.....          = models.--------------
    ..........
```

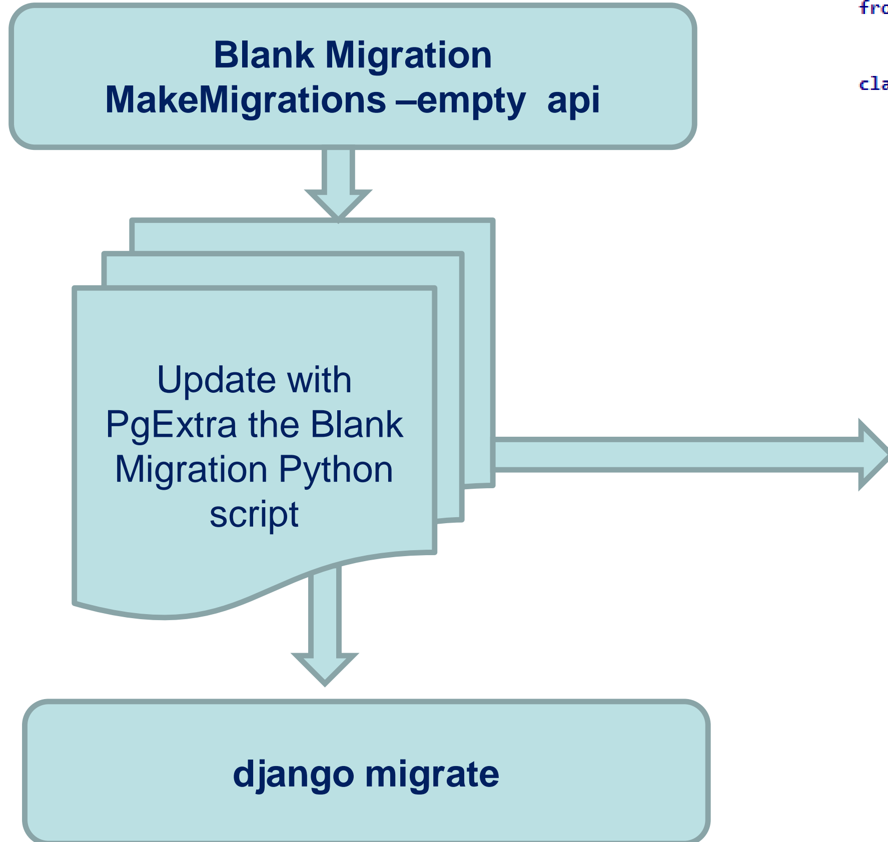
MIGRATION PLAN

Migration Plan

- Install New Django addon : **django-postgres-extra**
- Create a Base Model for the 3 concerned Tables
  - Use new Mixin for partitioned Table : **PostgresPartitionedModel**
  - Use new Postgresql Engine : **psqlextra.backend**

- Generating PG migration of the new partitioned Tables :
  - **Consumption**
  - **Temperature**
  - Humidity
  - Create a separated Migration Plan for each Partition List Key
- Creation of a **Cron** to calculate statistics and Partition Key affectation.
- Create new specific partitioned Indexs,
- Creation of a **Cron** to ensure a *flexible, smooth, fast and reliable migration*

Create a Migration Plan for each New Partition




```python
from django.db import migrations,  models
from psqlextra.backend.migrations.operations import PostgresAddListPartition


class Migration(migrations.Migration):

    dependencies = [
        ('api', '0005_alter_accomodationconsumption_partitioned_accomodation_and_more'),
    ]

    operations = [
        PostgresAddListPartition(
           model_name="AccomodationConsumption_Partitioned",
           name="residence_0023",
           values=[145, ],
        ),
        PostgresAddListPartition(
           model_name="AccomodationConsumption_Partitioned",
           name="residence_42LA",
           values=[59, ],
        ),
        PostgresAddListPartition(
           model_name="AccomodationConsumption_Partitioned",
           name="residence_623",
           values=[40, ],
        ),
        PostgresAddListPartition(
           model_name="AccomodationConsumption_Partitioned",
           name="residence_37",
           values=[9, ],
        ),
        PostgresAddListPartition(
           model_name="AccomodationConsumption_Partitioned",
           name="residence_360-400",
           values=[46, ],
        ),
```

Create Partition Key Migration Script

145	0023	164304
59	42LA	134784
40	623	121102
9	37	94192

```
PostgresAddListPartition(
    model_name="AccomodationConsumption_Partitioned",
    name="residence_0023",
    values=[145, ],
),
```

```
PostgresAddListPartition(
    model_name="AccomodationConsumption_Partitioned",
    name="residence_42LA",
    values=[59, ],
),
```

```
PostgresAddListPartition(
    model_name="AccomodationConsumption_Partitioned",
    name="residence_623",
    values=[40, ],
),
```

```
PostgresAddListPartition(
    model_name="AccomodationConsumption_Partitioned",
    name="residence_37",
    values=[9, ],
),
```

Automation Migration Plan & Process

- **Create New Cron :**
  - Scan Accomodation Consumption Max records for each residence.
  - Generate Django Migration of each residence and Partition

- **Update Residence Data Model :**
  - add Is partitioned Flag.
  - add partition Key.
  - add consumption records cardinality.

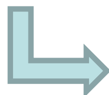
- Installation of **pathlib2** (*pip install*)
- Insert new Migration Plan in SQL Table : dataModelsLink (*DATA MODELS*)
- Run **Clone_postgresql**.py

Data Migration

- **RESET IS PARTITION MIGRATED (*False*):**
  - ***update api_accomodationconsumption set is_partition_migrated = False;***
  - **accomodationConsumption**
  - **accomodationTemperature**
  - **accomodationHumidity**

- **RUN Clone_postgresql.py :**
  - **year = 2024**
  - **group = partition**
  - **partition = y**

- **DataModelsLink :**



Action: --------- Go 0 of 3 selected

☐	MODEL	ORIGIN	PARTITIONING	POSTGRES	TARGET	ACTIVE	MIGRATED	YEAR	UPDATE	GROUP	APPS	ROWS	SEQ	BULK	PROGRESS
☐	api | CONSUMPTION **	-	☑	✔	✖	☑	✖	2024	Modified	Partition	Django	0	0	500	100
☐	api | PRE-CALCULATED	-	☑	✖	✖	☐	✖	2024	Modified	Partition	Django	3413227	0	1000	200
☐	api | CONSUMPTION PARTITIONED	api | CONSUMPTION **	☑	✔	✔	☑	✖	2024	Modified	Partition	Api	0	0	0	10

PARTITIONING PRODUCTION FLOW

- **GIT PULL –V ON NEW PARTITIONING BRANCH**
- **POSTGRESQL DUMP**
- **CLEANUP CURRENT DJANGO MIGRATION / SQL MIGRATION**
  - **Delete all Old Migratin script**
  - **Purge django migration SQL Table**
  - **Apply a Migrate –fake**
- **Pip install pathlib2**
- **pgmakemigrations –empty api**
- **python manage.py generate_partition --force=y --sep='/' --eof=\\n**
- **Execute some DDL on Production Database**
- ***update api_accomodationconsumption set is_partition_migrated = false;***

CONSO - CLONING TOOL PREPARATION

---- DJANGO SYSTEM PARAMETERS & SETTINGS ----

Kocliko API

- DATA MODELS
- DATA MODELS ERRORS
- DATA MODELS M2M
- KOCLIKO APPLICATIONS
- KOCLIKO GROUP PARAMETERS **
- KOCLIKO KEY PARAMETERS **
- KOCLIKO PARAMETERS
- SYSTEM KEY PARAMETERS
- SYSTEM PARAMETERS VALUES
- SYSTEM ROOT PARAMETERS *

Sélectionnez l'objet DATA MODEL à changer

Rechercher 2 résultats (179 résultats)

Action : --------- Envoyer 0 sur 2 sélectionné

	MODEL 2	ORIGIN	PARTITIONING	POSTGRES	TARGET	ACTIVE	MIGRATED	YEAR	UPDATE	GROUP 1	APPS	ROWS	SEQ	BULK	PROGRESS	MIGRATED	REMAINING	ERROR	DUPLICATES	CONSTRAINTS	M2M	F
☐	api | CONSUMPTION **	-	☑	✓	✗	☑	✗	2024	Modified	Partition	Django	0	0	1000	200	0	0	0	0	0		
☐	api | CONSUMPTION PARTITIONED	api | CONSUMPTION **	☑	✓	✓	☑	✗	2024	Modified	Partition	Django	0	0	1000	200	383236	0	0	0	0		

TEMP - CLONING TOOL PREPARATION

---- DJANGO SYSTEM PARAMETERS & SETTINGS ----

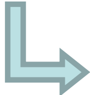
Kocliko API

- DATA MODELS
- DATA MODELS ERRORS
- DATA MODELS M2M
- KOCLIKO APPLICATIONS
- KOCLIKO GROUP PARAMETERS **
- KOCLIKO KEY PARAMETERS **
- KOCLIKO PARAMETERS
- SYSTEM KEY PARAMETERS
- SYSTEM PARAMETERS VALUES
- SYSTEM ROOT PARAMETERS *

MODEL 2	ORIGIN	PARTITIONING	POSTGRES	TARGET	ACTIVE	MIGRATED	YEAR	UPDATE	GROUP 1	APPS	ROWS	SEQ	BULK	PROGRESS	MIGRATED
api | TEMPERATURE ****	-	✓	✓	✗	✓	✗	2024	Modified	Partition	Django	4566724	0	2000	10	0
api | TEMPERATURE PARTITIONED *	api | TEMPERATURE ****	✓	✓	✓	✓	✗	2024	Modified	Partition	Api	0	0	0	10	0

Common approach on Multiple Partitions

- **Bases rules :**
  - Now since « Postgresql 12 » 2019, we can manage up to **2000 or 3000 partitions**, without any significant overhead.
  - Make it simple and scalable.
  - One uniq and single Container (*Partition*) for each new Residence.
  - No chance that any Company would wish to manage globaly a group of Residences.
  - Each Residence on data point of view, is a **consistent Group of relationnal Data**.
    - accomodation
    - sensor
    - Data (Conso, Temp, ...)
  - It's possible to alter a Partition with : **Detach / Attach :**
    - Change Residence ID / Code
    - Add to the « LIST KEY », extra residence ID
    - Example : existing Partition to Host Residence Data for « 59 », and we want to group « 41 » to the same partition.



```
PostgresAddListPartition(
    model_name="AccomodationConsumption_Partitioned",
    name="residence_42LA",
    values=[59, ],
),
```

```
PostgresAddListPartition(
     model_name="AccomodationConsumption_Partitioned",
    name="residence_0023_42LA",
    values=[59, 41],
    ),
```

PERFORMANCE CONCERNS

PARTITIONING BEST PRATICES

- **Choose the right partition size**. This is one of the first design questions that will come to your mind when implementing partitioning: what's your ideal partition size? The answer is it depends—you should aim for a balance between too large and too small. While PostgreSQL can handle a high number of partitions, having too many will increase planning time and could negatively affect query time. At the same time, if your partitions are too large, then you won't be able to use ranges to exclude data, and the effectiveness of partition pruning will be minimized.
- **Keep your partition size consistent.** Aim to keep partitions relatively uniform in size, ensuring that maintenance tasks and query performances are consistent across partitions.
- **Choose the right partitioning key**. Opt for a key that aligns with your query patterns. For instance, if most of your queries filter by date, a timestamp or date column would be an ideal partitioning key.
- **Create partitions in advance**. Ensure that partitions for upcoming periods (e.g., future days or months) are created ahead of time so there's no interruption in data ingestion. While you can use a default partition to catch orphaned records, in practice, this introduces a maintenance burden and does not perform well.
- **Take advantage of data retention policies to maintain old partitions**. For example, if you're partitioning by time and data has a limited useful life, schedule regular tasks to drop or archive old partitions.
- **Optimize your queries.** If you're especially interested in optimizing query performance, make sure to analyze and understand the query execution plan to validate that only necessary partitions are being scanned.
- **Properly place partitions across different storage mediums.** If you're using tablespaces to place partitions on different storage engines (e.g., EBS or S3), ensure that frequently accessed partitions are on faster storage and older or less accessed partitions can be on slower, cheaper storage.

ERRORS TO AVOID WHEN PARTITIONING

- **Over-partitioning**. It's tempting to create a high number of small partitions, but as we mentioned earlier, this won't work well—you'll get into query planning and management challenges.
- **Inefficient indexing**. Avoid creating unnecessary indexes on your partitions. Only index the columns that are frequently filtered or joined on.
- **Unoptimized query pattern.** Queries spanning multiple partitions or not using the partition key in the `WHERE` clause might suffer in performance. Ensure that the majority of your queries are optimized for the partitioning scheme.
- **Running out of partitions**. If you insert data with no partition to live in, it will either be rejected or stored in a `DEFAULT` partition. Ensure you either pre-create partitions at a quiet time (as this will lock your table) or use an extension that creates new partitions on the fly.
- **Monitor disk usage (partitions need extra space).** If you're creating many partitions, especially on different tablespaces or disks, monitor disk usage to avoid out-of-space issues.

SQL Cardinalities – *Conso –Temp - Hum*

Kocliko

Residence Code #Conso

residence_id	code	consumption
145	0023	163652
59	42LA	134784
40	623	120951
9	37	94076
46	360-400	77664
75	531-	71125
102	6991	68068
63	FAISANDERIE	67972
112	8401-8402-8403	63364
79	Marne	61009
39	134	60528
66	GR009	59290
53	Symphonie	58911
111	EP	56832
35	12071	56640
119	4138	52173
68	8300	50560
55	5073	50256
29	ICONE	48720
18	0233	48554
10	4	46380
118	4261	45962
120	4238	42720
12	10	40650
25	109-2	40400
121	4198	39672
24	109-1	39641
122	4640	39312
13	1068	38976
28	APT	38955
36	13VB	38880
105	3088	38475
114	BLD-ROI	37740
86	4058	35904
26	109-3	35728
74	3053	32120
144	6234	32032
76	783_6B1	31696
31	027	31668
42	oxygen	31046
69	9728	30429
61	1002	30096
23	8120	29711
17	0006	28608
54	0005	27940
81	3059	27118

Residence Code #Temp

residence_id	code	temperature
59	42LA	550634
145	0023	524718
102	6991	420750
40	623	385410
119	4138	357948
118	4261	348825
9	37	293774
105	3088	281271
121	4198	270135
86	4058	266832
122	4640	254124
46	360-400	241645
75	531-	219539
39	134	211185
63	FAISANDERIE	206292
35	12071	198052
79	Marne	197556
112	8401-8402-8403	196911
66	GR009	186747
81	3059	186732
53	Symphonie	182453
29	ICONE	179154
111	EP	177276
124	4383	165768
123	4379	162224
125	4388	156780
68	8300	156489
10	4	155131
18	0233	154836
55	5073	154662
120	4238	134715
28	APT	133458
54	0005	133263
12	10	129879
25	109-2	126286
13	1068	124951
24	109-1	124909
36	13VB	122568
126	6283	121291
114	BLD-ROI	117678
26	109-3	110855
144	6234	107356
31	027	106236
127	4255	104256
61	1002	103638
74	3053	98915

residence_id	code	humidity
59	42LA	514130
145	0023	496824
102	6991	420750
119	4138	357948
118	4261	348828
40	623	336788
105	3088	281280
121	4198	270135
86	4058	266832
9	37	262654
122	4640	254124
75	531-	217125
46	360-400	216168
63	FAISANDERIE	206288
79	Marne	195132
112	8401-8402-8403	193812
81	3059	186732
66	GR009	180780
35	12071	179660
39	134	176334
111	EP	174816
124	4383	165769
53	Symphonie	164207
123	4379	162225
125	4388	156779
29	ICONE	154313
68	8300	153840
55	5073	153792
18	0233	135294
10	4	134896
120	4238	131328
54	0005	122370
126	6283	121292
28	APT	119854
114	BLD-ROI	116550
12	10	113183
25	109-2	112910
24	109-1	110658
36	13VB	108098
13	1068	107944
127	4255	104256
26	109-3	99625
144	6234	98382
74	3053	98174
76	783_6B1	96768

Python Code & Query Optimization

- **Accomodation / consumption_for() :**
  - **residence parameter -> Redis Django session (*getResidenceParametres*)**
  - ***query on heating periods (gt/lt)***
  - ***duplicated access on consumption_for and consumption_euro (export_data)***
  - **getFuelPrice**() should be stored into a Redis session
  - ***self.get_accommodation_first_last_consumption_objects(ru_field, custom_period = custom_period) ➔ should pre-computed !***
    - getRepartitionUnitsSumAccordingCoef()
    - get_accommodation_first_last_consumption_objects()
    - get_all_heating_periods(self, self.residence) =➔ Precalcul
    - getCommonHeatingPeriod(first_date_object, residence = residence)

PostGreSQL Settings – **Query Planners**

- Enable : **enable_partition_pruning (*on*)**

- *Enables or disables the query planner's ability to eliminate a partitioned table's partitions from query plans. This also controls the planner's ability to generate query plans which allow the query executor to remove (ignore) partitions during query execution. The default is on.*

Partitionning & Indexing

- get_accommodation_first_last_consumption_objects():
  - query 1 :
    - residence, accomodation, period, repartition_unit_corr_elec_fair, date_2024

- avg_temperature :
  - query 1 :
    - residence, accomodation, period, frequency

Partitioning & Query Planner Impact

- Index Scan or Tablespace Scan ?
- https://www.postgresql.org/docs/current/planner-optimizer.html
- https://www.postgresql.org/docs/current/runtime-config-query.html
- https://pganalyze.com/docs/explain/basics-of-postgres-query-planning
- https://www.postgresql.org/docs/current/using-explain.html

Partitioning & Auto Scaling

Sub-partitioning can be useful to further divide partitions that are expected to become larger than other partitions.

Another option is to use range partitioning with multiple columns in the partition key. Either of these can easily lead to excessive numbers of partitions, so restraint is advisable.

Tablespace Partitioning References

Kocliko

- https://pganalyze.com/blog/postgresql-partitioning-django
- https://django-postgres-extra.readthedocs.io/en/master/table_partitioning.html
- https://www.postgresql.org/docs/current/ddl-partitioning.html
- https://dev.to/carlai/best-practices-for-postgresql-table-partition-managing-n30
- https://stackoverflow.com/questions/69656491/how-to-alter-partition-in-postgresql

Kocliko

Optimization Agenda & Plan

- Partitioning Migration Plan :
  - accomodationConsumption
  - accomodationTemperature
- Check Partitioned Index
- Check classic Index
- Inventory of concerned queries
  - accomodation.py
  - utils.py
  - utils_non_legal.py
- add switch on partitioned Data Model :
  - getPartitionModel()
  - currentConsoModel = getPartitionModel(None, partition = settings.**USE_PARTITIONED_TABLE**)
- Migration with **clone_postgresql** :
  - accomodationConsumption -> accomodationConsumption_Partitioned
  - accomodationTemperature -> accomodationTemperature_Partitioned

QUESTIONS & REMARKS

Issues with Django Partitioning support

- Pseudo and limited Full Partitioning with foreign Key
- Foreign Key column, just identified by their original DDL name.
- « residence » -> « residence_id »
- Need to create manually every new Extra Partition,
- Evaluate manually the LIST argument per each new partition,
- Load Balance manually the attached Residence(s)
- Attach a partitioning script to the Residence Backoffice