

ΑΛΕΞΑΚΗ ΒΑΣΙΛΙΚΗ ΑΜ: 1097464
ΞΗΝΤΑΡΟΠΟΥΛΟΥ ΑΣΗΜΙΝΑ ΑΜ: 1093452

ΠΕΡΙΕΧΟΜΕΝΑ

Στοιχεία.....	3
Συνημμένα αρχεία κώδικα.....	4
Τεχνικά χαρακτηριστικά περιβάλλοντος λειτουργίας.....	4
Ερώτημα 1: Παραγωγή δεδομένων.....	5
• Αρχείο κώδικα hotel_sim.py.....	5
• Αρχείο κώδικα kafka_producer.py.....	7
• Αρχείο κώδικα kafka_consumer.py.....	10
Ερώτημα 2: Κατανάλωση και επεξεργασία με Spark.....	12
• Αρχείο κώδικα spark.py.....	12
Ερώτημα 3: Αποθήκευση σε MongoDB.....	16
• Αρχείο κώδικα spark.py (συνέχεια για MongoDB).....	16
• Αρχείο κώδικα mongoQueries.py.....	22
Σχολιασμός αποτελεσμάτων.....	25
• Γενική ροή.....	25
• Δεδομένα.....	26
Βιβλιογραφία.....	27

Συστήματα Διαχείρισης Δεδομένων Μεγάλου Όγκου

Εργαστηριακή Άσκηση 2024/25

Στοιχεία

Όνομα	Επώνυμο	ΑΜ
ΑΣΗΜΙΝΑ	ΞΗΝΤΑΡΟΠΟΥΛΟΥ	1093452
ΒΑΣΙΛΙΚΗ	ΑΛΕΞΑΚΗ	1097464

Βεβαιώνω ότι είμαι συγγραφέας της παρούσας εργασίας και ότι έχω αναφέρει ή παραπέμψει σε αυτήν, ρητά και συγκεκριμένα, όλες τις πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών, προτάσεων ή λέξεων, είτε αυτές μεταφέρονται επακριβώς (στο πρωτότυπο ή μεταφρασμένες) είτε παραφρασμένες. Επίσης βεβαιώνω ότι αυτή η εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για το συγκεκριμένο μάθημα/σεμινάριο/πρόγραμμα σπουδών.

Έχω ενημερωθεί ότι σύμφωνα με τον εσωτερικό κανονισμό λειτουργίας του Πανεπιστημίου Πατρών άρθρο 50§6, τυχόν προσπάθεια αντιγραφής ή εν γένει φαλκίδευσης της εξεταστικής και εκπαιδευτικής διαδικασίας από οιονδήποτε εξεταζόμενο, πέραν του μηδενισμού, συνιστά βαρύ πειθαρχικό παράπτωμα.

Υπογραφή

ΞΗΝΤΑΡΟΠΟΥΛΟΥ ΑΣΗΜΙΝΑ



Υπογραφή

ΑΛΕΞΑΚΗ ΒΑΣΙΛΙΚΗ



13 / 06 / 2025

Συνημμένα αρχεία κώδικα

Μαζί με την παρούσα αναφορά υποβάλλουμε τα παρακάτω αρχεία κώδικα

Αρχείο	Αφορά το ερώτημα	Περιγραφή/Σχόλιο
hotel_sim.py	1	Εξομοιωτής για παραγωγή δεδομένων (csv).
kafka_producer.py	1	Εκτελεί την αποστολή δεδομένων στον broker.
kafka_consumer.py	1	Έλεγχος ότι η αποστολή και η κατανάλωση δεδομένων γίνεται σωστά στον broker.
spark.py	2-3	Διεργασία Spark η οποία επεξεργάζεται τα εισερχόμενα δεδομένα και παράγει ένα Spark Dataframe. Περιέχει και την αποθήκευση σε MongoDB στα raw και aggregated data.
mongoQueries.py	3	Python script με το οποίο κάνουμε queries στην MongoDB και απαντάμε στα ακόλουθα ερωτήματα

****Υποβάλλουμε ακόμη τα αρχεία *bigdata.city_stats.json* και *bigdata.raw_events.json* που είναι τα *collections* απο την βάση που αποθηκεύουμε τα δεδομένα, καθώς και το *hotel_clickstream.csv* που παράχθηκε απο τον εξομοιωτή για την επεξεργασία.**

Τεχνικά χαρακτηριστικά περιβάλλοντος λειτουργίας

Χαρακτηριστικό	Τιμή
CPU model	AMD Ryzen 5 7520U with Radeon Graphics
CPU clock speed	2.8GHz
Physical CPU cores	4

Logical CPU cores	8
RAM	8
Secondary Storage Type	HDD

Ερώτημα 1: Παραγωγή δεδομένων

- *Αρχείο κώδικα `hotel_sim.py`*

Αρχικά για να εξάγουμε το **csv** τρέχουμε τον **εξομοιωτή** ώστε να μας παράξει τα δεδομένα. Ο κώδικας προσομοιώνει **ρεαλιστικά δεδομένα** περιήγησης χρηστών σε **site κρατήσεων ξενοδοχείων**, περιλαμβάνοντας πολλές συμπεριφορές και καταστάσεις και τα εξάγει σε αρχείο.

Καθορίζονται διάφορες παράμετροι για τη δημιουργία των δεδομένων:

NUM_USERS = 1000: Συνολικοί χρήστες.

SESSIONS_PER_USER = (1, 5): Πόσες συνεδρίες έχει κάθε χρήστης.

ACTIONS_PER_SESSION = (5, 30): Πόσες ενέργειες γίνονται ανά συνεδρία.

LOCATIONS, HOTEL_IDS, ROOM_TYPES: Προκαθορισμένες επιλογές για τοποθεσίες, ξενοδοχεία, τύπους δωματίων.

```
# --- Configuration ---
NUM_USERS = 1000 # Number of unique users to simulate
SESSIONS_PER_USER = (1, 5) # Range of sessions per user
ACTIONS_PER_SESSION = (5, 30) # Range of actions per session
TIME_BETWEEN_ACTIONS_SECONDS = (1, 600) # Range of seconds between actions
MAX_HOTELS_VIEWED_AFTER_SEARCH = 5 # Maximum number of hotels a user might view after search
```

- ❖ Η συνάρτηση `generate_event_details(event_type, context)`

Παράγει **λεπτομέρειες** για κάθε **event**:

Π.χ. για το event `“search_hotels”` δημιουργεί τυχαία τοποθεσία, ημερομηνίες check-in/out, αριθμό ατόμων, ενώ για τα event `“view_hotel_details”` και `“add_to_cart”`, χρησιμοποιεί τα **context** δεδομένα προηγούμενων ενεργειών για **ρεαλισμό**.

```
elif event_type == "search_hotels":
    details["location"] = random.choice(LOCATIONS)
    start_date = datetime.date.today() +
datetime.timedelta(days=random.randint(1, 365))
    end_date = start_date + datetime.timedelta(days=random.randint(1,
14))
    details["check_in_date"] = start_date.strftime("%Y-%m-%d")
```

```
details["check_out_date"] = end_date.strftime("%Y-%m-%d")
details["num_guests"] = random.randint(1, 4)
```

❖ Η συνάρτηση `simulate_user_session(user_id, session_id, start_time)` δημιουργεί μια συνεδρία (session) για έναν συγκεκριμένο χρήστη, κατά την οποία επιλέγεται μια συγκεκριμένη ροή συμπεριφοράς με βάση προκαθορισμένες πιθανότητες.

```
chosen_flow_name = random.choices(list(USER_FLOWS.keys()),
weights=list(FLOW_PROBABILITIES.values()), k=1)[0]
chosen_flow = USER_FLOWS[chosen_flow_name]
```

Στη συνέχεια, παράγει τα επιμέρους **γεγονότα** (events) της συνεδρίας, προσθέτοντας μεταξύ τους ρεαλιστικά χρονικά διαστήματα. Όταν ο χρήστης πραγματοποιεί αναζήτηση ξενοδοχείων (search_hotels), το σύστημα εισάγει **δυναμικά** επιπλέον γεγονότα view_hotel_details, τα οποία **προσομοιώνουν** την προβολή λεπτομερειών διαφορετικών ξενοδοχείων που ενδέχεται να του προταθούν.

```
if event_type == "search_hotels":
    # Generate search details and store them in context
    details = generate_event_details(event_type, context)
    context["search_location"] = details.get("location")
    context["search_check_in_date"] = details.get("check_in_date")
    context["search_check_out_date"] = details.get("check_out_date")
    context["search_num_guests"] = details.get("num_guests")

    event = {
        "user_id": user_id,
        "session_id": session_id,
        "timestamp": current_time.isoformat(),
        "event_type": event_type,
        "event_details": details
    }
    clickstream_events.append(event)
```

Καθ' όλη τη διάρκεια της συνεδρίας, το σύστημα ενημερώνει ένα λεξικό `context = {}` με κρίσιμες πληροφορίες όπως η τελευταία αναζήτηση ή το τελευταίο ξενοδοχείο που προβλήθηκε, έτσι ώστε τα επόμενα γεγονότα να σχετίζονται ρεαλιστικά μεταξύ τους και να δίνουν την αίσθηση **φυσικής αλληλουχίας** στις ενέργειες του χρήστη.

❖ Η συνάρτηση `simulate_clickstream(num_users, ndays)` προσομοιώνει την αλληλεπίδραση πολλών χρηστών με μια πλατφόρμα κρατήσεων κατά τη διάρκεια **ενός καθορισμένου χρονικού διαστήματος**. Για κάθε χρήστη, δημιουργεί έναν ή περισσότερους κύκλους συνεδριών (sessions), αξιοποιώντας τη συνάρτηση `simulate_user_session` για την παραγωγή ρεαλιστικών γεγονότων ανά συνεδρία.

```
for i in range(num_users):
```

```

user_id = f"user_{i+1:05d}"
num_sessions = random.randint(*SESSIONS_PER_USER)

for j in range(num_sessions):
    session_id =
f"session_{user_id}_{j+1:03d}_{uuid.uuid4().hex[:6]}"
    # Start subsequent sessions later
    session_start_time = start_time +
datetime.timedelta(days=random.randint(0, ndays-1),
seconds=random.randint(0, 86400))
    session_events = simulate_user_session(user_id, session_id,
session_start_time)
    all_clickstream_events.extend(session_events)

```

Όλα τα παραγόμενα γεγονότα συγκεντρώνονται σε μία λίστα, η οποία στη συνέχεια **ταξινομείται με βάση** το χρονικό τους στίγμα (**timestamp**), εξασφαλίζοντας ότι το σύνολο των δεδομένων clickstream εμφανίζεται **με χρονολογική σειρά**. Το αποτέλεσμα είναι ένα πλήρες και χρονολογικά σωστό σύνολο δεδομένων που προσομοιώνει την πραγματική συμπεριφορά χρηστών σε βάθος χρόνου.

```

all_clickstream_events.sort(key=lambda x: x["timestamp"])
return all_clickstream_events

```

- **Αρχείο κώδικα *kafka_producer.py***

Εισαγωγή βιβλιοθηκών

Χρησιμοποιήσαμε τις εξής βιβλιοθήκες: Το **pandas** για ανάγνωση και διαχείριση του αρχείου **CSV**, τις **datetime** και **time** για τον χρονισμό της εξομοίωσης, την **json** για μετατροπή των εγγραφών σε **JSON** μορφή και τέλος την **KafkaProducer** για την αποστολή των μηνυμάτων στον **Kafka broker**.

```

import pandas as pd
import json
import time
from datetime import datetime
from kafka import KafkaProducer

```

Ορισμός παραμέτρων και Φόρτωση δεδομένων

Ορίζουμε βασικές ρυθμίσεις όπως το όνομα του αρχείου εισόδου, το όνομα του **Kafka topic** που θα χρησιμοποιηθεί, η διεύθυνση του **Kafka broker** που μας δόθηκε και το χρονικό διάστημα αποστολής μεταξύ παρτίδων (σε δευτερόλεπτα) **N=5**.

(Δημιουργήσαμε νέο **Kafka topic** για τα τελικά μας δεδομένα.)

Το αρχείο **CSV** φορτώνεται σε **pandas DataFrame**, ενώ η στήλη **timestamp** μετατρέπεται σε **datetime** αντικείμενα. Έτσι, τα δεδομένα ταξινομούνται χρονικά ώστε να προσομοιωθεί μια ρεαλιστική ροή γεγονότων.


```

CSV_FILE = "hotel_clickstream.csv"
TOPIC = "SDMD-1097464-final" # νέο, καθαρό topic
BROKER = "150.140.142.67:9094"
INTERVAL = 5 # δευτερόλεπτα

print("Φόρτωση CSV...")
df = pd.read_csv(CSV_FILE, parse_dates=["timestamp"], keep_default_na=False)
df = df.sort_values("timestamp").reset_index(drop=True)

```

Χρονική μετατόπιση

Υπολογίζεται η χρονική μετατόπιση μεταξύ του πρώτου **timestamp** των δεδομένων (**t1**) και της τρέχουσας ώρας (**t2**). Αυτή η διαφορά προστίθεται σε όλα τα **timestamps** για να φαίνεται ότι τα γεγονότα ξεκινούν "τώρα". Με αυτό τον τρόπο, **επιτυγχάνεται ο ρεαλιστικός συγχρονισμός** και ροή δεδομένων, σαν να προέρχονταν ζωντανά από πραγματική πηγή.

```

t1 = df["timestamp"].iloc[0]
t2 = datetime.now()
delta = t2 - t1
df["adjusted_timestamp"] = df["timestamp"] + delta

```

Ρύθμιση του Kafka Producer

Ο Kafka producer αποτελεί την οντότητα που αναλαμβάνει να "παράγει" και να στέλνει δεδομένα στον Kafka broker. Ρυθμίζεται ώστε να επικοινωνεί με τον broker και μέσω της παραμέτρου **value_serializer**, διασφαλίζεται ότι κάθε εγγραφή μετατρέπεται αυτόματα από **Python dictionary** σε **JSON format** πριν σταλεί στο **Kafka topic**.

```

producer = KafkaProducer(
    bootstrap_servers=[BROKER],
    value_serializer=lambda x: json.dumps(x).encode("utf-8")
)

```

Έναρξη της εξομοίωσης αποστολής και επιλογή εγγραφών

Η διαδικασία αποστολής ξεκινά και επαναλαμβάνεται όσο υπάρχουν εγγραφές. Ο υπολογισμός του "τώρα" εξασφαλίζει ότι κάθε παρτίδα στέλνεται με την **σωστή χρονική καθυστέρηση**, όπως αν συνέβαινε ζωντανά.

Επίσης, απομονώνονται οι εγγραφές των οποίων η προσαρμοσμένη χρονική στιγμή είναι μικρότερη ή ίση με την "τρέχουσα" προσομοιωμένη χρονική στιγμή. Οι υπόλοιπες θα αποσταλούν σε επόμενο κύκλο.

Με αυτόν τον τρόπο διατηρείται ο χρονικός συγχρονισμός και η σταδιακή αποστολή παρτίδων δεδομένων.

```

start_time = datetime.now()
while not df.empty:
    now = datetime.now()
    elapsed = now - start_time

```



```
current_virtual_time = t2 + elapsed
```

```
ready = df[df["adjusted_timestamp"] <= current_virtual_time]  
df = df[df["adjusted_timestamp"] > current_virtual_time]
```

Καθαρισμός και αποστολή μηνυμάτων

Κάθε επιλεγμένη εγγραφή μετατρέπεται σε **dictionary**, καθαρίζεται από άδειες ή μη έγκυρες τιμές και στέλνεται ως ξεχωριστό **JSON** μήνυμα στο **Kafka topic**.

Κάθε επιλεγμένη εγγραφή μετατρέπεται σε **dictionary** (event) και μετατρέπεται το **timestamp** της σε **ISO** μορφή. Εκτελείται έλεγχος και καθαρισμός των τιμών ώστε να αντικατασταθούν άκυρες, άδειες ή NaN με **None**, για αποφυγή σφαλμάτων κατά την αποστολή. Το **KafkaProducer** στέλνει κάθε μήνυμα ξεχωριστά στο καθορισμένο topic και η διαδικασία αυτή εξομοιώνει τη συνεχή και αξιόπιστη ροή γεγονότων, διατηρώντας την πληρότητα και ακεραιότητα των δεδομένων.

```
for _, row in ready.iterrows():  
    event = row.drop("timestamp").to_dict()  
    event["timestamp"] = row["adjusted_timestamp"].isoformat()  
    # Καθαρισμός  
    for k, v in event.items():  
        if isinstance(v, (datetime, pd.Timestamp)):  
            event[k] = v.isoformat()  
        elif pd.isna(v) or v == "":  
            event[k] = None  
    producer.send(TOPIC, value=event)  
    print(f"Sent: {event}")
```

Αναμονή πριν τον επόμενο κύκλο

Η εντολή **time.sleep(INTERVAL)** εφαρμόζεται για να επιβραδύνει την αποστολή και να διατηρήσει το προκαθορισμένο χρονικό διάστημα ανάμεσα στις παρτίδες. Ο producer περιμένει για το διάστημα που έχουμε ορίσει 5 sec (**INTERVAL**), προτού επαναλάβει τη διαδικασία με τις επόμενες εγγραφές.

Αυτό μιμείται την αληθινή ροή **streaming** δεδομένων, αποτρέποντας την υπερφόρτωση του **Kafka** με μαζική αποστολή σε μικρό χρόνο.

```
time.sleep(INTERVAL)
```

Τερματισμός και καθαρισμός

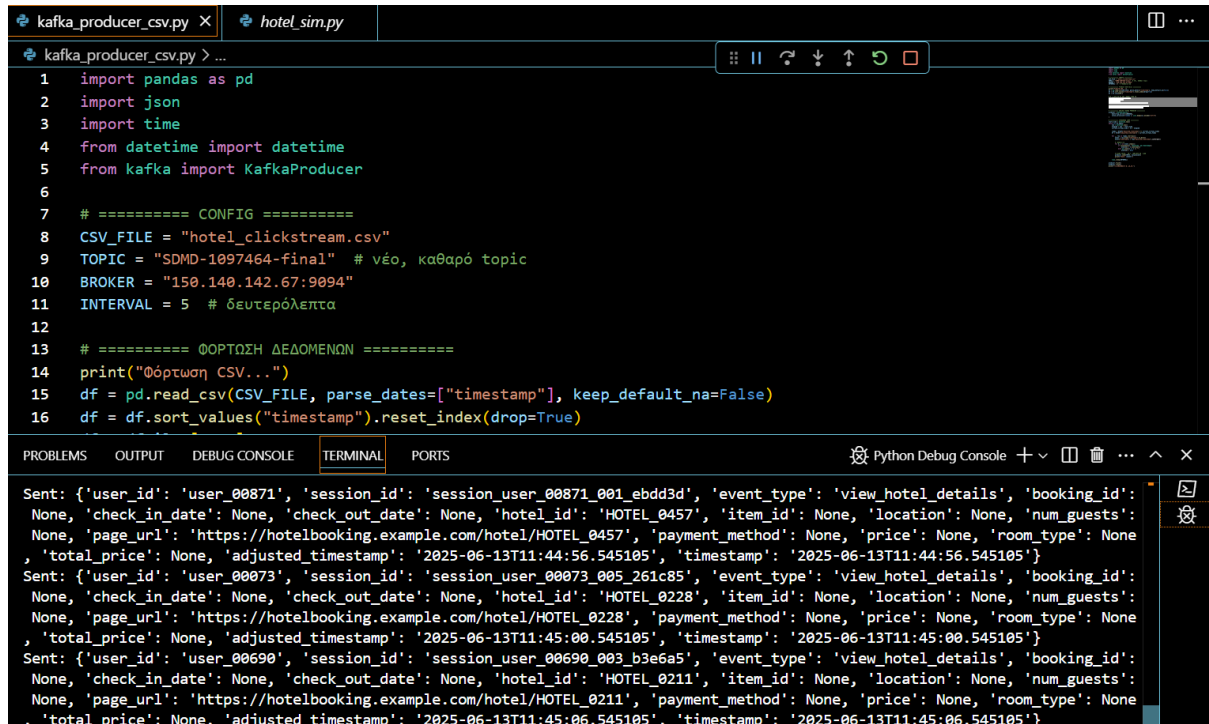
Με την ολοκλήρωση της αποστολής ενεργοποιείται η εντολή **flush()**, η οποία διασφαλίζει ότι όλα τα εν αναμονή μηνύματα έχουν αποσταλεί. Η **close()** αποδεσμεύει τους πόρους και κλείνει σωστά τη σύνδεση με τον **Kafka broker**.

Αυτή η διαδικασία είναι απαραίτητη σε κάθε **Kafka** εφαρμογή για την αποφυγή **data loss** ή **memory leaks**.

```
producer.flush()
```

```
producer.close()
print("Ολοκληρώθηκε η αποστολή.")
```

Ενδεικτικά αποτελέσματα κατά την εκτέλεση του Kafka Producer



```
kafka_producer_csv.py X hotel_sim.py
kafka_producer_csv.py > ...
1 import pandas as pd
2 import json
3 import time
4 from datetime import datetime
5 from kafka import KafkaProducer
6
7 # ===== CONFIG =====
8 CSV_FILE = "hotel_clickstream.csv"
9 TOPIC = "SDMD-1097464-final" # νέο, καθαρό topic
10 BROKER = "150.140.142.67:9094"
11 INTERVAL = 5 # δευτερόλεπτα
12
13 # ===== ΦΟΡΤΩΣΗ ΔΕΔΟΜΕΝΩΝ =====
14 print("Φόρτωση CSV...")
15 df = pd.read_csv(CSV_FILE, parse_dates=["timestamp"], keep_default_na=False)
16 df = df.sort_values("timestamp").reset_index(drop=True)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python Debug Console

```
Sent: {'user_id': 'user_00871', 'session_id': 'session_user_00871_001_ebdd3d', 'event_type': 'view_hotel_details', 'booking_id':
None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HOTEL_0457', 'item_id': None, 'location': None, 'num_guests':
None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0457', 'payment_method': None, 'price': None, 'room_type': None
, 'total_price': None, 'adjusted_timestamp': '2025-06-13T11:44:56.545105', 'timestamp': '2025-06-13T11:44:56.545105'}
Sent: {'user_id': 'user_00873', 'session_id': 'session_user_00873_005_261c85', 'event_type': 'view_hotel_details', 'booking_id':
None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HOTEL_0228', 'item_id': None, 'location': None, 'num_guests':
None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0228', 'payment_method': None, 'price': None, 'room_type': None
, 'total_price': None, 'adjusted_timestamp': '2025-06-13T11:45:00.545105', 'timestamp': '2025-06-13T11:45:00.545105'}
Sent: {'user_id': 'user_00690', 'session_id': 'session_user_00690_003_b3e6a5', 'event_type': 'view_hotel_details', 'booking_id':
None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HOTEL_0211', 'item_id': None, 'location': None, 'num_guests':
None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0211', 'payment_method': None, 'price': None, 'room_type': None
, 'total_price': None, 'adjusted_timestamp': '2025-06-13T11:45:06.545105', 'timestamp': '2025-06-13T11:45:06.545105'}
```

- **Αρχείο κώδικα kafka_consumer.py**

Ρύθμιση παραμέτρων Kafka Consumer

Η αντίστροφη διαδικασία της παραγωγής είναι η **κατανάλωση** (consumption):

Ορίζουμε το όνομα του **Kafka topic** και τη διεύθυνση του **broker** από τον οποίο θα λάβουμε τα μηνύματα.

Ο **consumer** θα "ακούει" το συγκεκριμένο **stream** δεδομένων και θα συλλέγει τις εγγραφές με τη σειρά που εστάλησαν.

Είναι απαραίτητο ώστε ο **consumer** να γνωρίζει ποιο **stream** δεδομένων να παρακολουθεί και από πού.

```
from kafka import KafkaConsumer
import json
```

```
TOPIC = "SDMD-1097464-final"
BROKER = "150.140.142.67:9094"
```

Δημιουργία Kafka Consumer

Ο **Kafka Consumer** ρυθμίζεται να διαβάζει μηνύματα από τον **broker**. Η επιλογή **auto_offset_reset="earliest"** εξασφαλίζει ότι θα διαβάσει όλα τα υπάρχοντα μηνύματα από την αρχή, καθώς τον τρέχουμε αφού έχουμε τρέξει τον **kafkaProducer**. Με **enable_auto_commit=True**, αποθηκεύει αυτόματα τη θέση ανάγνωσης (offset), ώστε να μην διαβάζει τα ίδια μηνύματα ξανά. Το **group_id** επιτρέπει την παρακολούθηση της

προόδου ανά ομάδα **consumers**. Τέλος, το **value_deserializer** μετατρέπει τα **JSON bytes** που φτάνουν από τον **broker** σε **Python dictionaries** για ευκολότερη επεξεργασία. Αυτή η ρύθμιση εξασφαλίζει ότι τα μηνύματα αναλύονται σωστά και είναι άμεσα προσβάσιμα προς επεξεργασία.

```
consumer = KafkaConsumer(  
    TOPIC,  
    bootstrap_servers=[BROKER],  
    auto_offset_reset="earliest",  
    enable_auto_commit=True,  
    group_id="consumer-read",  
    value_deserializer=lambda x: json.loads(x.decode("utf-8"))  
)
```

Ορίζουμε έναν μετρητή που θα μας βοηθήσει να καταγράψουμε πόσα μηνύματα λάβαμε συνολικά κατά τη διάρκεια της εκτέλεσης. Είναι χρήσιμο για έλεγχο και αξιολόγηση.

```
message_count = 0
```

Ανάγνωση μηνυμάτων

Ο **Kafka consumer** λειτουργεί ως **iterator** δηλαδή “ακούει” διαρκώς το **topic** για νέα μηνύματα. Αυτό είναι βασικό χαρακτηριστικό της **real-time streaming** αρχιτεκτονικής, όπου η επεξεργασία δεν γίνεται τμηματικά, αλλά συνεχώς.

Οπότε, μπαίνουμε σε ατέρμονο βρόχο όπου κάθε νέο μήνυμα από το **topic** διαβάζεται, αποθηκεύεται προσωρινά ως **dictionary (event)**, και εμφανίζεται στην οθόνη. Ο μετρητής αυξάνεται για κάθε μήνυμα που λαμβάνεται επιτυχώς.

Το **Kafka** διατηρεί συνδέσεις μέσω **TCP**. Το **close()** αποδεσμεύει τη σύνδεση από τον **broker** και απελευθερώνει τους πόρους του συστήματος. Το **finally** εγγυάται ότι η αποσύνδεση θα γίνει ανεξαρτήτως σφάλματος ή διακοπής.

Μετά τον τερματισμό, ο **consumer** κλείνει σωστά τη σύνδεση με τον **Kafka broker** και εμφανίζεται το σύνολο των μηνυμάτων που έχουν ληφθεί. Αυτό διασφαλίζει σωστή απελευθέρωση πόρων και μας επιτρέπει να αξιολογήσουμε την κατανάλωση.

```
try:  
    for message in consumer:  
        event = message.value  
        message_count += 1  
        print(f"Received: {event}")  
except KeyboardInterrupt:  
    print("Consumer stopped manually.")  
finally:  
    consumer.close()  
    print(f"Συνολικός αριθμός μηνυμάτων που διαβάστηκαν: {message_count}")
```

Ενδεικτικά αποτελέσματα κατά την εκτέλεση του Kafka Consumer:

```
20 kafka_consumer_csv.py > ...
21
22 message_count = 0
23
24
25 try:
26
27     PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
28
29     Received: {'user_id': 'user_00547', 'session_id': 'session_user_00547_004_3993af', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
30 TEL_0456', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0456', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
31 None, 'adjusted_timestamp': '2025-06-13T09:05:26.545105', 'timestamp': '2025-06-13T09:05:26.545105'}
32
33     Received: {'user_id': 'user_00547', 'session_id': 'session_user_00547_004_3993af', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
34 TEL_0254', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0254', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
35 None, 'adjusted_timestamp': '2025-06-13T09:05:42.545105', 'timestamp': '2025-06-13T09:05:42.545105'}
36
37     Received: {'user_id': 'user_00547', 'session_id': 'session_user_00547_004_3993af', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
38 TEL_0463', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0463', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
39 None, 'adjusted_timestamp': '2025-06-13T09:06:47.545105', 'timestamp': '2025-06-13T09:06:47.545105'}
40
41     Received: {'user_id': 'user_00934', 'session_id': 'session_user_00934_002_0c18af', 'event_type': 'page_view', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': None, 'item
42 id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/', 'payment_method': None, 'price': None, 'room_type': None, 'total_price': None, 'adjusted_timestamp': '2
43 025-06-13T09:10:57.545105', 'timestamp': '2025-06-13T09:10:57.545105'}
44
45     Received: {'user_id': 'user_00547', 'session_id': 'session_user_00547_004_3993af', 'event_type': 'logout', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': None, 'item_id'
46 None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/logout', 'payment_method': None, 'price': None, 'room_type': None, 'total_price': None, 'adjusted_timestamp':
47 '2025-06-13T09:11:12.545105', 'timestamp': '2025-06-13T09:11:12.545105'}
48
49     Received: {'user_id': 'user_00469', 'session_id': 'session_user_00469_003_4facfe', 'event_type': 'page_view', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': None, 'item
50 id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/', 'payment_method': None, 'price': None, 'room_type': None, 'total_price': None, 'adjusted_timestamp': '2
51 025-06-13T09:11:26.545105', 'timestamp': '2025-06-13T09:11:26.545105'}
52
53     Received: {'user_id': 'user_00469', 'session_id': 'session_user_00469_003_4facfe', 'event_type': 'search_hotels', 'booking_id': None, 'check_in_date': '2025-12-13', 'check_out_date': '2025-12-18', 'hot
54 el_id': None, 'item_id': None, 'location': 'London', 'num_guests': '1', 'page_url': None, 'payment_method': None, 'price': None, 'room_type': None, 'total_price': None, 'adjusted_timestamp': '2025-06-1
55 3T09:15:15.545105', 'timestamp': '2025-06-13T09:15:15.545105'}
56
57     Received: {'user_id': 'user_00469', 'session_id': 'session_user_00469_003_4facfe', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
58 TEL_0255', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0255', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
59 None, 'adjusted_timestamp': '2025-06-13T09:16:04.545105', 'timestamp': '2025-06-13T09:16:04.545105'}
60
61     Received: {'user_id': 'user_00469', 'session_id': 'session_user_00469_003_4facfe', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
62 TEL_0020', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0020', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
63 None, 'adjusted_timestamp': '2025-06-13T09:17:50.545105', 'timestamp': '2025-06-13T09:17:50.545105'}
64
65     Received: {'user_id': 'user_00469', 'session_id': 'session_user_00469_003_4facfe', 'event_type': 'view_hotel_details', 'booking_id': None, 'check_in_date': None, 'check_out_date': None, 'hotel_id': 'HO
66 TEL_0185', 'item_id': None, 'location': None, 'num_guests': None, 'page_url': 'https://hotelbooking.example.com/hotel/HOTEL_0185', 'payment_method': None, 'price': None, 'room_type': None, 'total_price
67 None, 'adjusted_timestamp': '2025-06-13T09:19:10.545105', 'timestamp': '2025-06-13T09:19:10.545105'}
```

Ερώτημα 2: Κατανάλωση και επεξεργασία με Spark

Η υλοποίηση πραγματοποιήθηκε στην πλατφόρμα **Databricks**, η οποία παρέχει ενσωματωμένο περιβάλλον **Spark** και πλήρη υποστήριξη για σύνδεση με εξωτερικά **Kafka** Brokers, όπως αυτό που μας δόθηκε (**150.140.142.67:9094**). Δημιουργήσαμε ένα λοιπόν **Spark cluster**, το οποίο αξιοποιήθηκε για real-time επεξεργασία ροών δεδομένων. Πιο συγκεκριμένα:

Runtime Version: *Databricks Runtime 12.2 LTS (includes Apache Spark 3.3.2, Scala 2.12)*

Spark Version: 3.3.2

Language: Python (PySpark)

Driver Type: Community Optimized (15.3GB Memory , 2 cores)

Notebook περιβάλλον: Databricks Web UI

- **Αρχείο κώδικα *spark.py***

Η διαδικασία ξεκινά με την εγκατάσταση της βιβλιοθήκης **kafka-python**, η οποία χρησιμοποιείται προαιρετικά για επικοινωνία με **Kafka** μέσω **Python**.

```
%pip install kafka-python
```

Για την επεξεργασία των εισερχόμενων **streaming** δεδομένων χρησιμοποιήθηκαν δύο βασικά **modules** της βιβλιοθήκης **PySpark**: **pyspark.sql.types** και **pyspark.sql.functions**. Από το πρώτο, αξιοποιήθηκαν οι τύποι **StructType**, **StructField**, **StringType** και **DoubleType**, προκειμένου να οριστεί το **schema**, δηλαδή η δομή των **JSON** μηνυμάτων

που αναμένονται από το **Kafka topic**. Το **schema** αυτό περιλαμβάνει τα πεδία **όπως ζητήθηκαν στην εκφώνηση, user_id, event_type, timestamp, location, total_price** και άλλα, τα οποία αντιστοιχούν σε στοιχεία αναζήτησης ή κράτησης σε μια πλατφόρμα κρατήσεων ξενοδοχείων. Τα περισσότερα πεδία ορίζονται ως **StringType**, ενώ το **total_price** ορίζεται ως **DoubleType** ώστε να υποστηρίζει αριθμητικούς υπολογισμούς, όπως ο υπολογισμός του συνολικού τζίρου (sales volume). Από το **pyspark.sql.functions** χρησιμοποιήθηκαν οι συναρτήσεις **from_json** (για μετατροπή των **JSON** σε **DataFrame**), **col** (για αναφορά σε στήλες), **to_timestamp** (για μετατροπή χρονικών στιγμών σε τύπο **timestamp**), **when** (για την κανονικοποίηση τιμών του πεδίου **event_type**), καθώς και οι **count, sum, coalesce** και **window**, που χρησιμοποιούνται για την παραγωγή συγκεντρωτικών στατιστικών ανά χρονικά διαστήματα 10 λεπτών και ανά τοποθεσία. Ο συνδυασμός αυτών των επιτρέπει την αποτελεσματική και σε βάθος ανάλυση των δεδομένων σε πραγματικό χρόνο.

```
# === 2. Imports για Spark ===
from pyspark.sql.types import *
from pyspark.sql.functions import *

# === 3. Schema για τα Kafka JSON ===
schema = StructType([
    StructField("user_id", StringType()),
    StructField("session_id", StringType()),
    StructField("timestamp", StringType()),
    StructField("event_type", StringType()),
    StructField("booking_id", StringType()),
    StructField("check_in_date", StringType()),
    StructField("check_out_date", StringType()),
    StructField("hotel_id", StringType()),
    StructField("item_id", StringType()),
    StructField("location", StringType()),
    StructField("num_guests", StringType()),
    StructField("page_url", StringType()),
    StructField("payment_method", StringType()),
    StructField("price", StringType()),
    StructField("room_type", StringType()),
    StructField("total_price", DoubleType())
])
```

Σύνδεση Spark με Kafka

Το **Spark** συνδέεται στον **Kafka broker** στη διεύθυνση **150.140.142.67:9094** και "ακούει" το **Kafka topic** με όνομα **SDMD-1093452-final**, ώστε να **επεξεργάζεται** τα δεδομένα που στέλνονται **live-stream** εκεί.

```
raw_kafka_df = (
    spark.readStream
    .format("kafka")
    .option("kafka.bootstrap.servers", "150.140.142.67:9094")
    .option("subscribe", "SDMD-1097464-final")
    .load())
```

)

Ανάλυση και μετατροπή Kafka μηνυμάτων

Καθώς τα δεδομένα στο **Kafka** αποστέλλονται σε μορφή **JSON**, κάθε μήνυμα μετατρέπεται από **bytes** σε **string**, και στη συνέχεια αναλύεται (parsed) με βάση ένα ορισμένο **schema**. Το **schema** περιλαμβάνει όλα τα πιθανά πεδία που μπορεί να περιέχει μια εγγραφή χρήστη, όπως **user_id**, **timestamp**, **event_type**, **location**, **total_price** κ.ά.

```
parsed_df = (  
    raw_kafka_df  
    .selectExpr("CAST(value AS STRING)")  
    .select(from_json(col("value"), schema).alias("data"))  
    .select("data.*")  
    .withColumn("timestamp", to_timestamp("timestamp"))  
)
```

Κανονικοποίηση των δεδομένων

Για την **ευκολότερη ομαδοποίηση** και υπολογισμό στατιστικών, η τιμή του πεδίου **event_type** κανονικοποιείται ώστε τα γεγονότα τύπου **search_hotels** μετονομάζονται σε **search** και τα γεγονότα **book_hotel** σε **booking**.

```
parsed_df = parsed_df.withColumn(  
    "event_type",  
    when(col("event_type") == "search_hotels", "search")  
    .when(col("event_type") == "complete_booking", "booking")  
    .otherwise(col("event_type"))  
)
```

Ορισμός window και watermark

Η ροή αποκτά **watermark** διάρκειας **30 λεπτών**, το οποίο καθορίζει πόσο πίσω στον χρόνο μπορεί να δεχθεί δεδομένα που "καθυστερούν".

Αυτό είναι σημαντικό γιατί τα **streams** δεν φτάνουν πάντοτε με τη σωστή σειρά ή στον σωστό χρόνο και το **watermark** θέτει τα όρια μέχρι τότε το **Spark** θα τα δέχεται.

```
parsed_df = parsed_df.withWatermark("timestamp", "30 minutes")
```

Aggregation

Τα δεδομένα φιλτράρονται ώστε να διατηρηθούν μόνο τα **search** και **booking events** με μη κενή τοποθεσία (**location**). Στη συνέχεια ομαδοποιούνται ανά παράθυρο **50** λεπτών (**window**) και ανά πόλη (**location**)

Για κάθε τέτοιο **group** υπολογίζονται: **Ο αριθμός των αναζητήσεων** (**search_volume**), **ο αριθμός των κρατήσεων** (**bookings_volume**) και **το άθροισμα των τιμών κρατήσεων** (**sales_volume**).

****Αυτό μας επιτρέπει να βλέπουμε για παράδειγμα κάθε 50 λεπτά πόσες κρατήσεις και αναζητήσεις έγιναν ανά πόλη και πόσα χρήματα συγκεντρώθηκαν από κρατήσεις. Βάλαμε 50 λεπτά γιατί τα δεδομένα φτάνουν αραιά και τα φίλτρα είναι αυστηρά με αποτέλεσμα σε παράθυρο 10 λεπτών να βλέπαμε 1 εγγραφή ανα πόλη συνεχώς.**

```

from pyspark.sql.functions import coalesce, lit
agg_df = (
    parsed_df
    .filter(col("event_type").isin("search", "booking"))
    .filter(col("location").isNotNull())
    .groupBy(window("timestamp", "50 minutes"), col("location"))
    .agg(
        count(when(col("event_type") == "search", True)).alias("search_volume"),
        count(when(col("event_type") == "booking",
True)).alias("bookings_volume"),
        sum(when(col("event_type") == "booking",
col("total_price"))).alias("sales_volume")
    )
    .select(
        col("window.start").alias("time"),
        col("location").alias("destination_name"),
        col("search_volume"),
        col("bookings_volume"),
        col("sales_volume")
    )
    .select(
        "time",
        "destination_name",
        coalesce("search_volume", lit(0)).alias("search_volume"),
        coalesce("bookings_volume", lit(0)).alias("bookings_volume"),
        coalesce("sales_volume", lit(0.0)).alias("sales_volume")
    )
)

```

Εκκινούμε ένα **streaming query** στο Spark το οποίο διαβάζει το `agg_df` DataFrame και μας εκτυπώνει πληροφορίες. Με αυτό τον τρόπο βλέπουμε αν υπάρχουν δεδομένα aggregated.

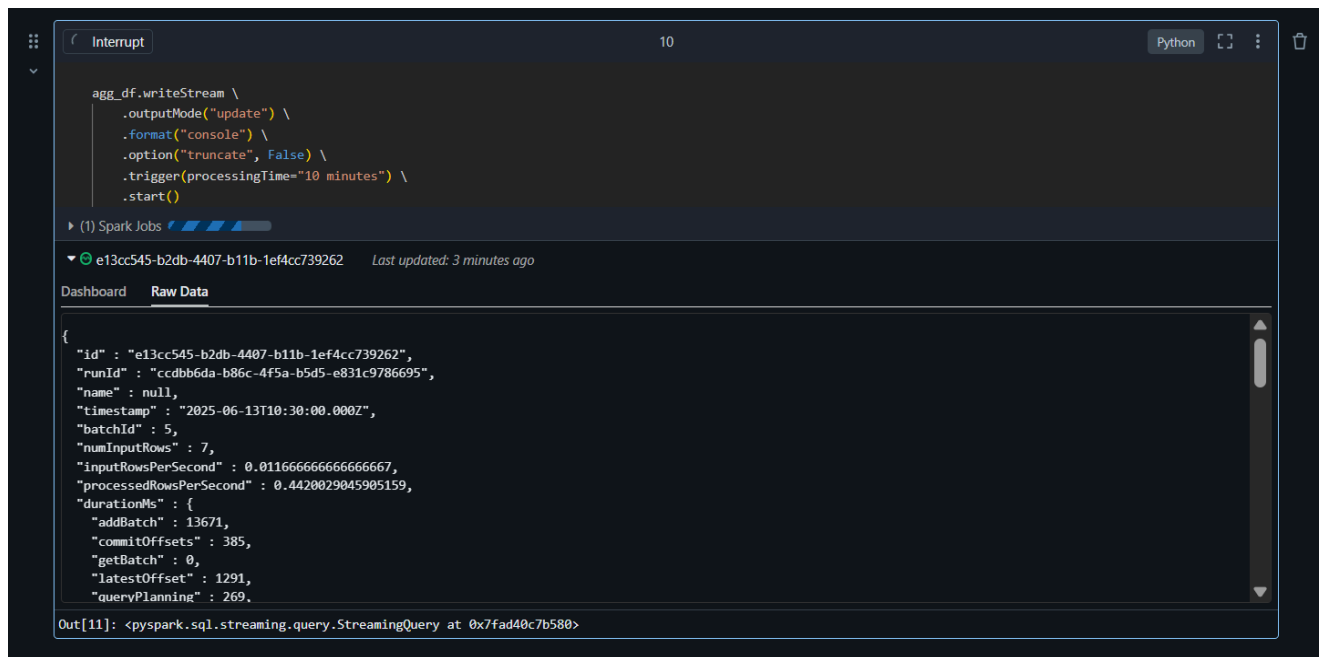
```

agg_df.writeStream \
    .outputMode("update") \
    .format("console") \
    .option("truncate", False) \
    .trigger(processingTime="10 minutes") \
    .start()

```

Παρακάτω η ροή εκτελείται με trigger κάθε 10 λεπτά και εκτυπώνει στην κονσόλα τα ενημερωμένα αποτελέσματα (aggregation updates). Η συγκεκριμένη παρτίδα (**batchId: 5**) δείχνει ότι διαβάστηκαν **7 νέα μηνύματα από το Kafka** (offsets 991–998), με ταχύτητα επεξεργασίας περίπου 0.44 γραμμές/δευτερόλεπτο. Τα μηνύματα είναι τόσα καθώς είναι αραιά και εφαρμόζονται και πάνω τους φίλτρα για το **event_type** και **location**.

******Εγινε στο τέλος δοκιμαστικά για την απόδειξη **καθαρά** της λειτουργίας του spark, καθώς εμείς την επιβεβαιώναμε απο την αποθήκευση των δεδομένων στην MongoDB



The screenshot shows a Databricks notebook interface. At the top, there's a toolbar with an 'Interrupt' button, a progress bar at 10%, and a 'Python' language selector. Below the toolbar, a code cell contains the following Python code:

```
agg_df.writeStream \
    .outputMode("update") \
    .format("console") \
    .option("truncate", False) \
    .trigger(processingTime="10 minutes") \
    .start()
```

Below the code cell, a 'Spark Jobs' section shows a job with ID 'e13cc545-b2db-4407-b11b-1ef4cc739262' that was last updated 3 minutes ago. Underneath, there are tabs for 'Dashboard' and 'Raw Data'. The 'Raw Data' tab is selected, displaying a JSON object representing the job's progress:

```
{
  "id" : "e13cc545-b2db-4407-b11b-1ef4cc739262",
  "runId" : "ccdbb6da-b86c-4f5a-b5d5-e831c9786695",
  "name" : null,
  "timestamp" : "2025-06-13T10:30:00.000Z",
  "batchId" : 5,
  "numInputRows" : 7,
  "inputRowsPerSecond" : 0.011666666666666667,
  "processedRowsPerSecond" : 0.4420029045905159,
  "durationMs" : {
    "addBatch" : 13671,
    "commitOffsets" : 385,
    "getBatch" : 0,
    "latestOffset" : 1291,
    "queryPlanning" : 269,
  }
}
```

At the bottom, the output of the last cell is shown: 'Out[11]: <pyspark.sql.streaming.query.StreamingQuery at 0x7fad40c7b580>'.

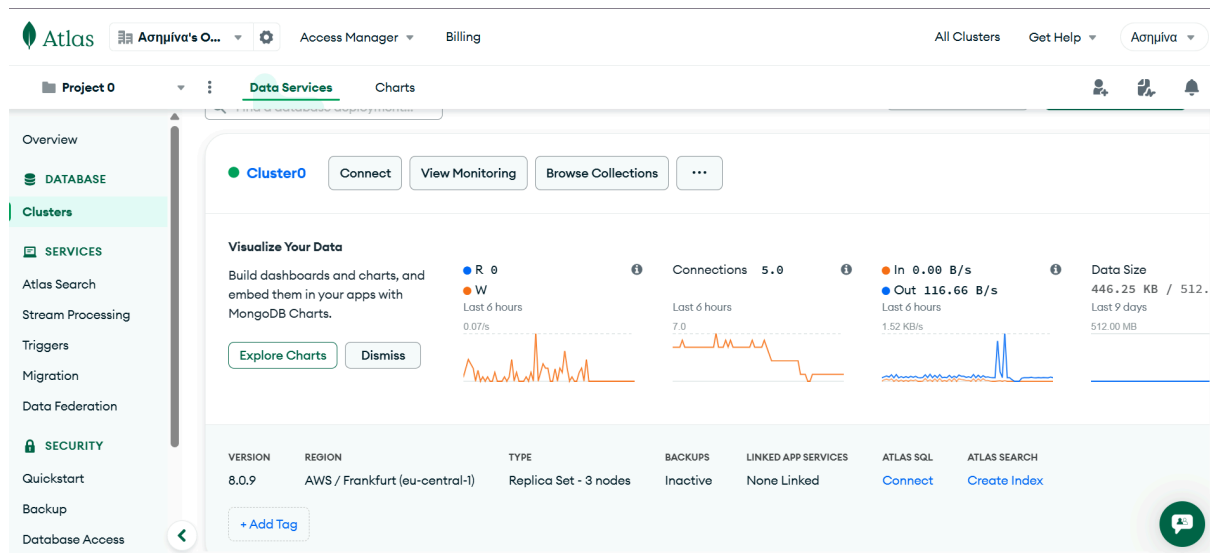
Ερώτημα 3: Αποθήκευση σε MongoDB

- *Αρχείο κώδικα spark.py (συνέχεια για MongoDB)*

Ρύθμιση σύνδεσης προς MongoDB Atlas

Χρησιμοποιήθηκε το **URI** σύνδεσης που παρέχει το **MongoDB Atlas** όταν φτιάξαμε το cluster και επιτρέψαμε την πρόσβαση από όλες τις IP (0.0.0.0/0) για διευκόλυνση της σύνδεσης από **Spark**.

```
spark.conf.set("spark.mongodb.write.connection.uri",
"mongodb+srv://xintaropoulouasimina:Asimina2003@cluster0.n6928z1.mongodb.net/?retryWrites=true&w=majority")
```



–Cluster στο MongoDB Atlas

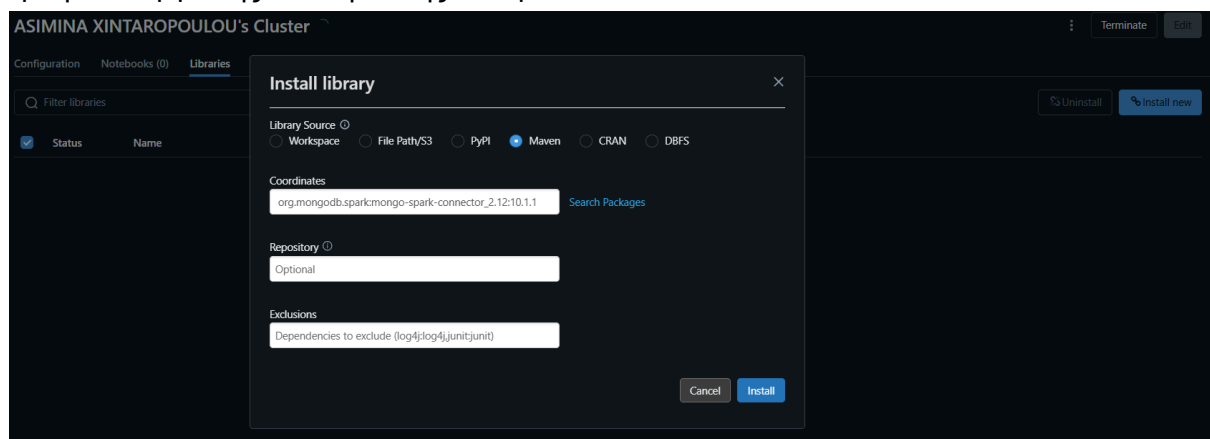
Επίσης χρειάστηκε να προσθέσουμε το **URI** και στις ρυθμίσεις του cluster στο Databricks.

```
Spark JDBC/ODBC

Spark config ⓘ

spark.databricks.rocksDB.fileManager.useCommitService false
spark.mongodb.write.connection.uri
mongodb+srv://xintaropoulosasimina:Asimina2003@cluster0.n69
28z1.mongodb.net/?retryWrites=true&w=majority
```

Επίσης χρειάστηκε να κατεβάσουμε και **spark-mongodb connector** στο cluster για την ορθή λειτουργία της αποθήκευσης δεδομένων.



Στο MongoDB Atlas φτιάξαμε την NoSQL βάση “bigdata” και τα επεξεργασμένα δεδομένα αποθηκεύτηκαν σε δύο collections που δημιουργήσαμε:

- ❖ **raw_events**: όλα τα αρχικά JSON γεγονότα.
- ❖ **city_stats**: συνοπτικά στατιστικά ανά πόλη και χρονικό παράθυρο.

Εγγραφή αποτελεσμάτων στη MongoDB

Χρησιμοποιούνται οι μέθοδοι **foreachBatch** για την αποθήκευση των δεδομένων όπως ζητήθηκε και τα ακατέργαστα δεδομένα (**raw events**) αποθηκεύονται στη συλλογή **raw_events** ενώ τα συγκεντρωτικά στατιστικά (**aggregated data**) αποθηκεύονται στη συλλογή **city_stats** της βάσης δεδομένων **bigdata**.

Η εγγραφή πραγματοποιείται κάθε 10 λεπτά, όπως καθορίζεται από το **trigger(triggerTime="10 minutes")**.

****Όταν κάνουμε `.write.format("mongodb"). > .mode("append") > .save()`:**

αν δεν υπάρχει η συλλογή **raw_events** ή **city_stats**, η MongoDB την δημιουργεί αυτόματα. Η συλλογή δημιουργείται **δυναμικά** βάσει των εγγραφών που αποθηκεύονται, **χωρίς να χρειάζεται να ορίσουμε σχήμα εκ των προτέρων**.

```
def write_raw_to_mongo(batch_df, batch_id):
    print(f"RAW batch {batch_id}")
    batch_df.write.format("mongodb") \
        .option("database", "bigdata") \
        .option("collection", "raw_events") \
        .mode("append") \
        .save()

def write_agg_to_mongo(batch_df, batch_id):
    print(f"AGGREGATED batch {batch_id}")
    batch_df.write.format("mongodb") \
        .option("database", "bigdata") \
        .option("collection", "city_stats") \
        .mode("append") \
        .save()
```

Checkpointing για ανθεκτικότητα

Κατά την εγγραφή ορίζονται φάκελοι **checkpoints** για την αποθήκευση της προόδου του **stream**. Αυτό διασφαλίζει ότι σε περίπτωση επανεκκίνησης της διαδικασίας, το **Spark** θα συνεχίσει από το σωστό σημείο, αποφεύγοντας διπλές εγγραφές ή απώλεια δεδομένων.

```
# RAW stream
parsed_df.writeStream \
    .foreachBatch(write_raw_to_mongo) \
    .outputMode("append") \
    .trigger(triggerTime="10 minutes") \
    .option("checkpointLocation", "/dbfs/checkpoints/raw") \
    .start()

# AGGREGATED stream
agg_df.writeStream \
    .foreachBatch(write_agg_to_mongo) \
    .outputMode("update") \
    .trigger(triggerTime="10 minutes") \
```

```
.option("checkpointLocation", "/dbfs/checkpoints/agg") \  
.start()
```

Έτσι, το παραπάνω **script** συνδέεται επιτυχώς σε **Kafka topic** και καταναλώνει δεδομένα σε πραγματικό χρόνο. Επεξεργάζεται και μετατρέπει **JSON** σε δομημένο **DataFrame** και υπολογίζει στατιστικά ανά 10λεπτο και ανά πόλη. Τέλος, αποθηκεύει τόσο τα **raw** όσο και τα **aggregated** δεδομένα σε **MongoDB**.

Στις παρακάτω 3 εικόνες παρουσιάζονται αποτελέσματα της αποθήκευσης ωμών και επεξεργασμένων δεδομένων ανά 10 λεπτά.

The screenshot shows the Databricks IDE interface. At the top, a code editor displays a Python script for a Spark Streaming job. Below the editor, the 'Spark Jobs' section shows a job with ID '571221c2-610c-48dd-9a21-b1a07ef3ac4c' that is 'Last updated: 6 minutes ago'. The 'Raw Data' tab is selected, displaying a JSON object representing a batch of data. The output at the bottom shows the Spark Streaming query object.

```
.outputMode("update") \  
.trigger(processingTime="10 minutes") \  
.option("checkpointLocation", "/dbfs/checkpoints/agg") \  
.start()
```

(2) Spark Jobs

571221c2-610c-48dd-9a21-b1a07ef3ac4c Last updated: 6 minutes ago

Dashboard Raw Data

```
{  
  "id" : "571221c2-610c-48dd-9a21-b1a07ef3ac4c",  
  "runId" : "27e139dc-048c-4d0f-bfda-5ffff4f5f3a5",  
  "name" : null,  
  "timestamp" : "2025-06-13T01:00:00.001Z",  
  "batchId" : 10,  
  "numInputRows" : 11,  
  "inputRowsPerSecond" : 0.018333302777828703,  
  "processedRowsPerSecond" : 0.4257295456304667,  
  "durationMs" : {  
    "addBatch" : 23468,  
    "commitOffsets" : 120,  
    "getBatch" : 3,  
    "latestOffset" : 1313,  
    "queryPlanning" : 638,  
  },  
}
```

3c934538-4125-4359-8dc7-3fa4cc042f14 Last updated: 6 minutes ago

Out[11]: <pyspark.sql.streaming.query.StreamingQuery at 0x7f7090310250>

The screenshot shows the Databricks IDE interface. At the top, a code editor displays a Python script for a Spark Streaming job. Below the editor, the 'Spark Jobs' section shows two jobs: '571221c2-610c-48dd-9a21-b1a07ef3ac4c' and '3c934538-4125-4359-8dc7-3fa4cc042f14', both 'Last updated: 6 minutes ago'. The 'Raw Data' tab is selected, displaying a JSON object representing a batch of data. The output at the bottom shows the Spark Streaming query object.

```
.outputMode("update") \  
.trigger(processingTime="10 minutes") \  
.option("checkpointLocation", "/dbfs/checkpoints/agg") \  
.start()
```

(2) Spark Jobs

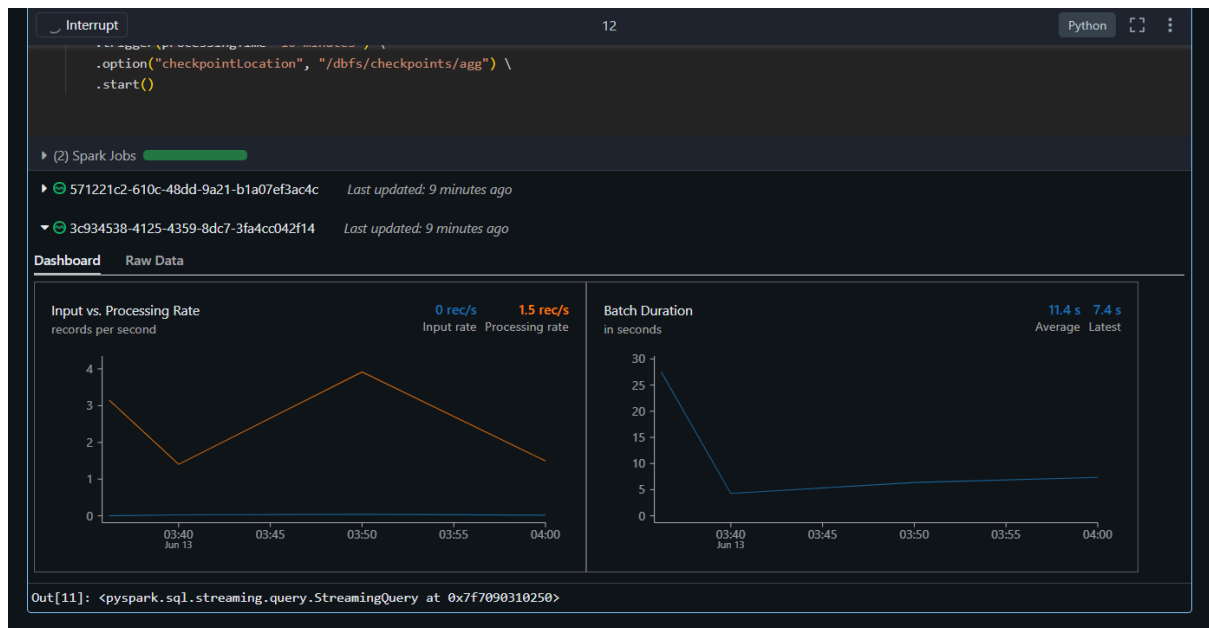
571221c2-610c-48dd-9a21-b1a07ef3ac4c Last updated: 6 minutes ago

3c934538-4125-4359-8dc7-3fa4cc042f14 Last updated: 6 minutes ago

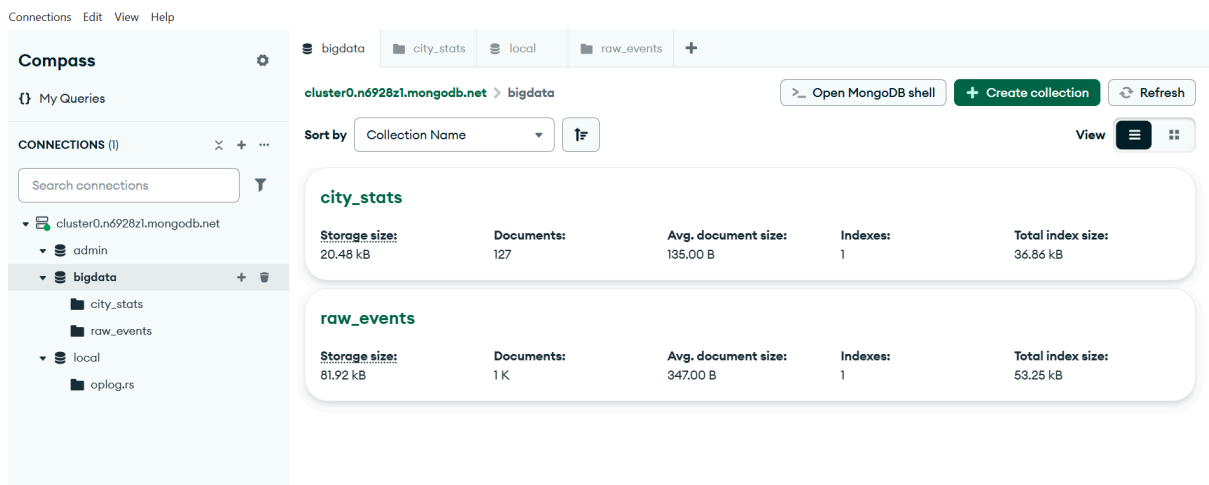
Dashboard Raw Data

```
{  
  "id" : "3c934538-4125-4359-8dc7-3fa4cc042f14",  
  "runId" : "6b4e127b-1561-44dd-9d57-0a1bd6b52ae7",  
  "name" : null,  
  "timestamp" : "2025-06-13T01:00:00.001Z",  
  "batchId" : 9,  
  "numInputRows" : 11,  
  "inputRowsPerSecond" : 0.018333302777828703,  
  "processedRowsPerSecond" : 1.4963950482927493,  
  "durationMs" : {  
    "addBatch" : 5384,  
    "commitOffsets" : 222,  
    "getBatch" : 0,  
    "latestOffset" : 1310,  
    "queryPlanning" : 156,  
  },  
}
```

Out[11]: <pyspark.sql.streaming.query.StreamingQuery at 0x7f7090310250>



Για καλύτερη οπτικοποίηση εγκαταστήσαμε το **MongoDB Compass** όπου μας προσφέρει καλύτερη εικόνα της βάσης και των δεδομένων μας ακόμα και με στατιστικές αναλύσεις.



—Βάση Δεδομένων bigdata με collections city_stats, raw_events

Compass

My Queries

CONNECTIONS (1)

Search connections

cluster0.n6928z1.mongodb.net

- admin
- bigdata
- city_stats**
- raw_events
- local
- oplog.rs

bigdata > city_stats

Documents 123 Aggregations Schema Indexes 1 Validation

Generate query + Explain Reset Find Options

ADD DATA EXPORT DATA UPDATE DELETE 25 76 - 100 of 101

```
{
  "_id": ObjectId("684bc321282c81426b63cd73"),
  "time": 2025-06-13T08:40:00.000+00:00,
  "destination_name": "London",
  "search_volume": 1,
  "bookings_volume": 2,
  "sales_volume": 0
}
```

```
{
  "_id": ObjectId("684bc7d2282c81426b63cd7f"),
  "time": 2025-06-13T09:30:00.000+00:00,
  "destination_name": "Rio de Janeiro",
  "search_volume": 2,
  "bookings_volume": 0,
  "sales_volume": 0
}
```

```
{
  "_id": ObjectId("684bca2a282c81426b63cd93"),
  "time": 2025-06-13T09:30:00.000+00:00,
  "destination_name": "Rio de Janeiro",
  "search_volume": 3
}
```

–Συλλογή city_stats

Compass

My Queries

CONNECTIONS (1)

Search connections

cluster0.n6928z1.mongodb.net

- admin
- bigdata
- city_stats
- raw_events**
- local
- oplog.rs

bigdata > raw_events

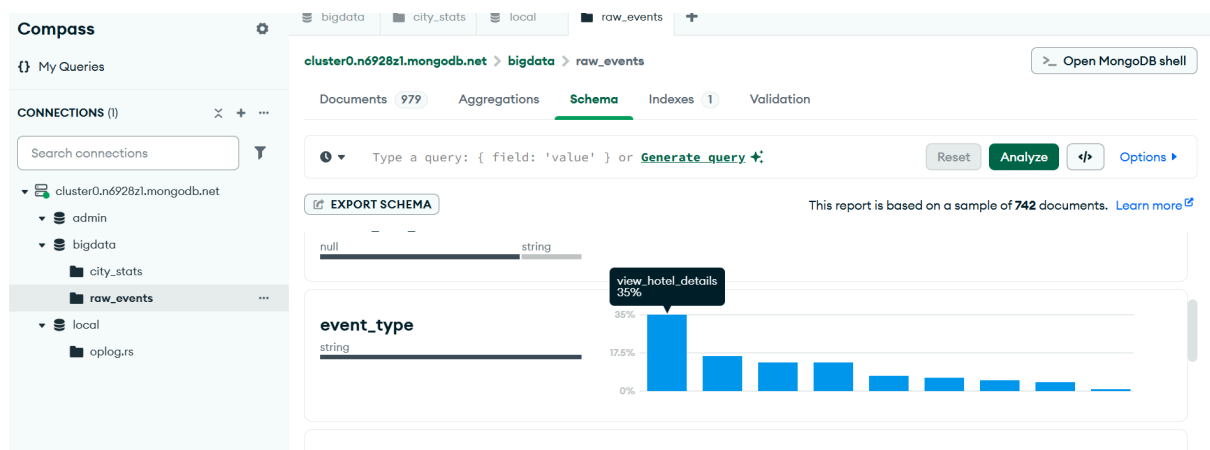
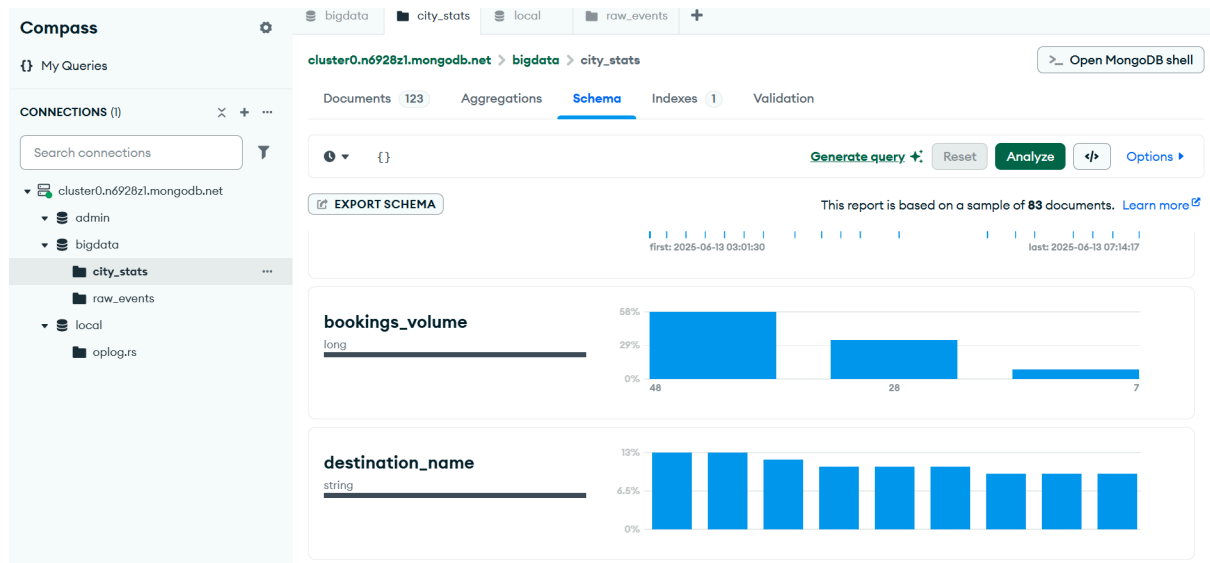
Documents 979 Aggregations Schema Indexes 1 Validation

Type a query: { field: 'value' } or Generate query + Explain Reset Find Options

ADD DATA EXPORT DATA UPDATE DELETE 25 26 - 50 of 576

```
{
  "_id": ObjectId("684b94826233357e7c0b06c1"),
  "user_id": "user_00562",
  "session_id": "session_user_00562_001_d57b6b",
  "timestamp": 2025-06-13T00:33:14.636+00:00,
  "event_type": "view_hotel_details",
  "booking_id": null,
  "check_in_date": null,
  "check_out_date": null,
  "hotel_id": "HOTEL_0291",
  "item_id": null,
  "location": null,
  "num_guests": null,
  "page_url": "https://hotelbooking.example.com/hotel/HOTEL_0291",
  "payment_method": null,
  "price": null,
  "room_type": null,
  "total_price": null
}
```

–Συλλογή raw events



–Στατιστικά για τα αποθηκευμένα δεδομένα

- Αρχείο κώδικα *mongoQueries.py*

Σύνδεση στη MongoDB και επιλογή συλλογής

Γίνεται σύνδεση στον **MongoDB Atlas** μέσω του **MongoClient**, επιλέγεται η βάση **bigdata** και η συλλογή **raw_events**, στην οποία έχουν αποθηκευτεί όλα τα **raw** γεγονότα που προήλθαν από το **Spark Streaming pipeline**.

```
from pymongo import MongoClient
from datetime import datetime
import pprint
# Σύνδεση στη MongoDB Atlas
client =
MongoClient("mongodb+srv://xintaropoulouasimina:Asimina2003@cluster0.n6928z1
.mongodb.net/")
db = client["bigdata"]
raw = db["raw_events"]
```


Ορισμός χρονικού διαστήματος και format εκτύπωσης

Καθορίζεται το χρονικό διάστημα στο οποίο θα εκτελεστούν όλα τα **queries**. Στο παράδειγμα, επιλέγεται η 13η Ιουνίου 2025. Χρησιμοποιείται επίσης η βιβλιοθήκη **pprint** για πιο ευανάγνωστη εμφάνιση των αποτελεσμάτων στο **terminal**.

```
# Ορισμός χρονικού διαστήματος (π.χ. 13 Ιουνίου 2025)
start = datetime(2025, 6, 13, 0, 0, 0)
end = datetime(2025, 6, 13, 23, 59, 59)
pp = pprint.PrettyPrinter(indent=2)
```

Query 1: Πόλη με τις περισσότερες κρατήσεις

Σε αυτό το ερώτημα χρησιμοποιείται το **aggregation pipeline** της **MongoDB** για να βρεθεί η πόλη με τις περισσότερες κρατήσεις εντός της συγκεκριμένης ημερομηνίας. Αρχικά φιλτράρονται τα γεγονότα τύπου **booking**, στη συνέχεια ομαδοποιούνται κατά **location** και μετρώνται οι κρατήσεις ανά πόλη. Τέλος, επιστρέφεται η πόλη με το μεγαλύτερο πλήθος (**limit: 1**).

```
result1 = raw.aggregate([
    {
        "$match": {
            "timestamp": {"$gte": start, "$lte": end},
            "event_type": "booking",
            "location": {"$exists": True, "$ne": None}
        }
    },
    {"$group": {"_id": "$location", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}},
    {"$limit": 1}
])
print("Περισσότερες κρατήσεις:")
pp.pprint(list(result1))
```

Query 2: Πόλη με τις περισσότερες αναζητήσεις

Αυτό το κομμάτι είναι παρόμοιο με το προηγούμενο, αλλά αφορά γεγονότα τύπου **search**. Φιλτράρονται όλα τα **search events** της ημέρας, και υπολογίζεται ο αριθμός τους ανά πόλη. Το αποτέλεσμα επιστρέφει την πόλη με τον μεγαλύτερο αριθμό αναζητήσεων.

```
result2 = raw.aggregate([
    {
        "$match": {
            "timestamp": {"$gte": start, "$lte": end},
            "event_type": "search",
            "location": {"$exists": True, "$ne": None}
        }
    },
    {"$group": {"_id": "$location", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}},
    {"$limit": 1}
])
```

```

    {"$group": {"_id": "$location", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}},
    {"$limit": 1}
])
print("\nΠερισσότερες αναζητήσεις:")
pp.pprint(list(result2))

```

Query 3: Μέση διάρκεια παραμονής ανά πόλη

Σε αυτό το ερώτημα δεν χρησιμοποιείται **aggregation** της **MongoDB**, αλλά επεξεργασία μέσω **Python**, καθώς απαιτείται υπολογισμός ημερών από **check_in_date** και **check_out_date**. Ανακτώνται όλα τα **bookings** που περιλαμβάνουν αυτές τις δύο ημερομηνίες, γίνεται υπολογισμός της διαφοράς και δημιουργείται ένα **dictionary stay_data** με συνολικό αριθμό ημερών και κρατήσεων ανά πόλη. Τέλος, υπολογίζεται ο μέσος όρος ημερών παραμονής **ανά τοποθεσία**.

```

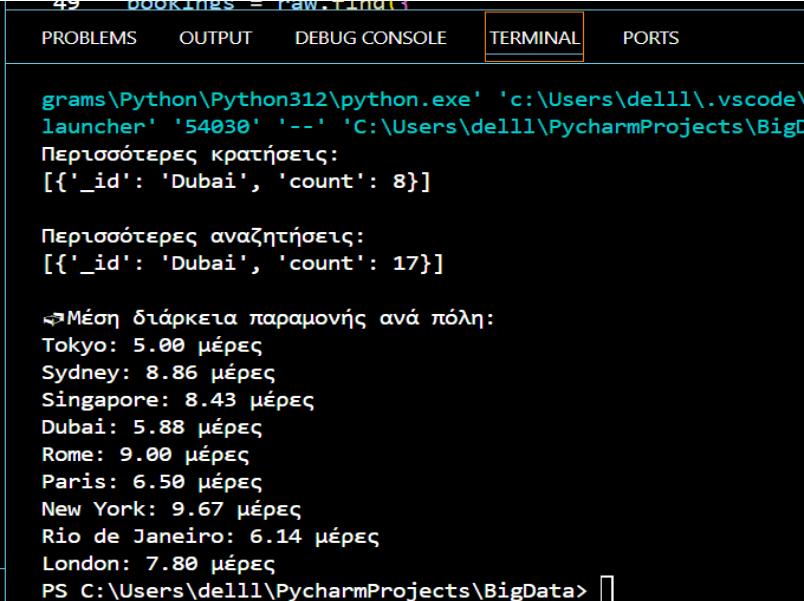
bookings = raw.find({
    "timestamp": {"$gte": start, "$lte": end},
    "event_type": "booking",
    "check_in_date": {"$exists": True, "$ne": None},
    "check_out_date": {"$exists": True, "$ne": None},
    "location": {"$exists": True, "$ne": None}
})
stay_data = {}
for b in bookings:
    try:
        loc = b["location"]
        check_in = datetime.fromisoformat(b["check_in_date"])
        check_out = datetime.fromisoformat(b["check_out_date"])
        stay_length = (check_out - check_in).days
        if stay_length < 0:
            continue
        if loc in stay_data:
            stay_data[loc]["total_days"] += stay_length
            stay_data[loc]["count"] += 1
        else:
            stay_data[loc] = {"total_days": stay_length, "count": 1}
    except Exception as e:
        continue

print("\n📊 Μέση διάρκεια παραμονής ανά πόλη:")
for loc, stats in stay_data.items():
    avg = stats["total_days"] / stats["count"]
    print(f"{loc}: {avg:.2f} μέρες")

```

Αποτελέσματα

Εμφανίζονται αποτελέσματα από την ανάλυση δεδομένων της συλλογής **"raw_events"** της βάσης δεδομένων **MongoDB "bigdata"**, που προέκυψαν από εκτελέσεις σε Python. Τα δεδομένα υποδεικνύουν ότι η πόλη με τις περισσότερες κρατήσεις και αναζητήσεις είναι το **Ντουμπάι**, με 8 κρατήσεις και 17 αναζητήσεις αντίστοιχα. Επιπλέον, παρέχονται στοιχεία για τη **μέση διάρκεια παραμονής ανά πόλη**, με τη Νέα Υόρκη να παρουσιάζει τη μεγαλύτερη μέση διάρκεια (9.67 ημέρες) και το **Τόκιο** τη μικρότερη (5.00 ημέρες), προσφέροντας έτσι μια ολοκληρωμένη εικόνα των ταξιδιωτικών προτιμήσεων και συμπεριφορών των χρηστών.



```
49 books = raw.find()
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
grams\Python\Python312\python.exe' 'c:\Users\del11\.vscode\
launcher' '54030' '--' 'C:\Users\del11\PycharmProjects\BigD
Περισσότερες κρατήσεις:
[{'_id': 'Dubai', 'count': 8}]

Περισσότερες αναζητήσεις:
[{'_id': 'Dubai', 'count': 17}]

➡Μέση διάρκεια παραμονής ανά πόλη:
Tokyo: 5.00 μέρες
Sydney: 8.86 μέρες
Singapore: 8.43 μέρες
Dubai: 5.88 μέρες
Rome: 9.00 μέρες
Paris: 6.50 μέρες
New York: 9.67 μέρες
Rio de Janeiro: 6.14 μέρες
London: 7.80 μέρες
PS C:\Users\del11\PycharmProjects\BigData>
```

Σχολιασμός αποτελεσμάτων

- **Γενική ροή**

Ζωντανή ροή δεδομένων (Kafka Producer)

Ο Kafka Producer στέλνει δεδομένα **σε πραγματικό χρόνο**, προσομοιώνοντας events που παράγονται ζωντανά από ένα σύστημα κρατήσεων ξενοδοχείων και αποστέλλονται σε **live ροή**.

Streaming επεξεργασία με Apache Spark

Ο Spark καταναλώνει τα **Kafka** μηνύματα με τη βοήθεια του **Spark Structured Streaming**. Επεξεργάζεται δυναμικά τη ροή, πραγματοποιεί κανονικοποίηση των τύπων συμβάντων και εκτελεί **παράθυρο-χρονική ομαδοποίηση (windowed aggregations)**.

Ο Spark παράγει ένα aggregating **DataFrame** που περιλαμβάνει:

- time: χρονικό παράθυρο επεξεργασίας
- destination_name: τοποθεσία (πόλη)
- search_volume: αριθμός αναζητήσεων ξενοδοχείων

bookings_volume: αριθμός κρατήσεων
sales_volume: συνολικό ποσό των κρατήσεων (τζίρος)

Αποθήκευση σε MongoDB

Τα αποτελέσματα της επεξεργασίας (aggregations) καθώς και τα **ωμά δεδομένα** (raw Kafka messages) αποθηκεύονται σε δύο διαφορετικές συλλογές σε MongoDB χρησιμοποιώντας **foreachBatch sinks** του Spark.

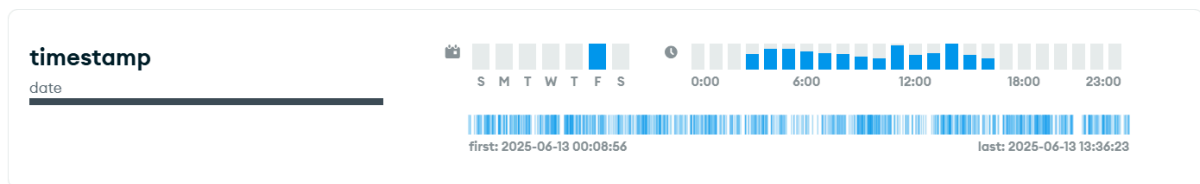
Η υλοποίηση πραγματοποιήθηκε με **παράλληλη εκτέλεση** του Kafka Producer και του Spark Structured Streaming. Ο Producer έστειλε συνεχώς νέα events, ενώ ο Spark τα κατανάλωνε, επεξεργαζόταν και αποθήκευε άμεσα τα αποτελέσματα στη MongoDB. Έτσι, ολόκληρη η ροή – από την παραγωγή έως την αποθήκευση και ανάλυση – πραγματοποιούνταν σε **πραγματικό χρόνο**, επιτρέποντας τη ζωντανή παρακολούθηση των αποτελεσμάτων.

- **Δεδομένα**

Τα **αποτελέσματά μας** και το **pipeline** που πραγματοποιήσαμε θεωρούμε ότι είναι **επιτυχημένα**, καθώς διεκπεραιώνουν **όλες** τις απαιτήσεις της άσκησης και τα Mongo ερωτήματα βγαίνουν ορθά παρά το περιορισμένο αριθμό που στείλαμε στο Kafka Broker και άρα έχουμε διαθέσιμο για ανάλυση.

Όπως φαίνεται στην ανάλυση του **timestamp** στο **MongoDB Compass** τα δείγματα είναι πολύ περιορισμένα σε ημερομηνία.

Στην αρχή, είχαμε κάποιες δυσκολίες με διπλές εγγραφές δεδομένων στον **Kafka Broker** κατά τη διάρκεια των δοκιμών, κάτι που προκάλεσε λάθος αποτελέσματα. Εντοπίσαμε το πρόβλημα, το διορθώσαμε και τρέξαμε ξανά τον κώδικα για να συλλέξουμε δεδομένα (περίπου 13 ώρες- 1K δεδομένα) ώστε να έχουμε όσο το δυνατόν ένα αξιόπιστο δείγμα για την ανάλυση.



Βιβλιογραφία

[1] Streaming Data With Apache Spark and MongoDB

<https://www.mongodb.com/developer/languages/python/streaming-data-apache-spark-mongodb/>

Αυτή η πηγή ήταν ιδιαίτερα σημαντική, καθώς μας βοήθησε να κατανοήσουμε και να εφαρμόσουμε τη διαδικασία **streaming** δεδομένων χρησιμοποιώντας δύο από τις κεντρικές τεχνολογίες της εργασίας: το **Apache Spark** και το **MongoDB**.
Μας έδωσε τη βάση για το πώς να ενσωματώσουμε τα **raw** γεγονότα που προέρχονταν από το **Spark Streaming pipeline** στη συλλογή **raw_events** του **MongoDB**.

[1] Apache Kafka and Spark Streaming Integration Guide

<https://spark.apache.org/docs/latest/streaming/structured-streaming-kafka-integration.html>

Αυτή η πηγή εξηγεί τον τρόπο με τον οποίο γίνεται η σύνδεση μεταξύ Apache Kafka και Spark Structured Streaming. Μας βοήθησε να κατανοήσουμε τις βασικές παραμέτρους της ροής, όπως τα offsets, τα watermarks και την παραθυροποίηση.

[3] MongoDB Aggregation Framework

<https://www.mongodb.com/docs/manual/aggregation/>

Οδηγός που μας εξηγεί τα στάδια aggregation σε MongoDB. Είναι χρήσιμος για την κατασκευή των MongoDB queries στην mongoQueries.py.

[4] Connect via Compass – Atlas

<https://www.mongodb.com/docs/atlas/compass-connection/>

Μας εξηγεί πως αντιγράφεται το connection string από το Atlas, για το MongoDB Compass και πως αποθηκεύεται η σύνδεση για εύκολη πρόσβαση στο Atlas cluster.

[5] ChatGPT

Χρησιμοποιήσαμε το **ChatGPT** για να διορθώσουμε σφάλματα και να βελτιστοποιήσουμε τον κώδικα μας.