مشروع تحويل التفاعل اللحظي للمستخدمين

Prepared by
Khaled Alruwita

Completed on
Jan 10, 2026

1) الفكرة

بناء نظام بثّ بيانات (Streaming) يلتقط الصفوف الجديدة من جدول engagement_events في Postgres، ثم يثري ويحوّل كل حدث عبر ربطه بجدول content، ثم يرسل الحدث المُثرى مباشرة إلى ثلاث وجهات:

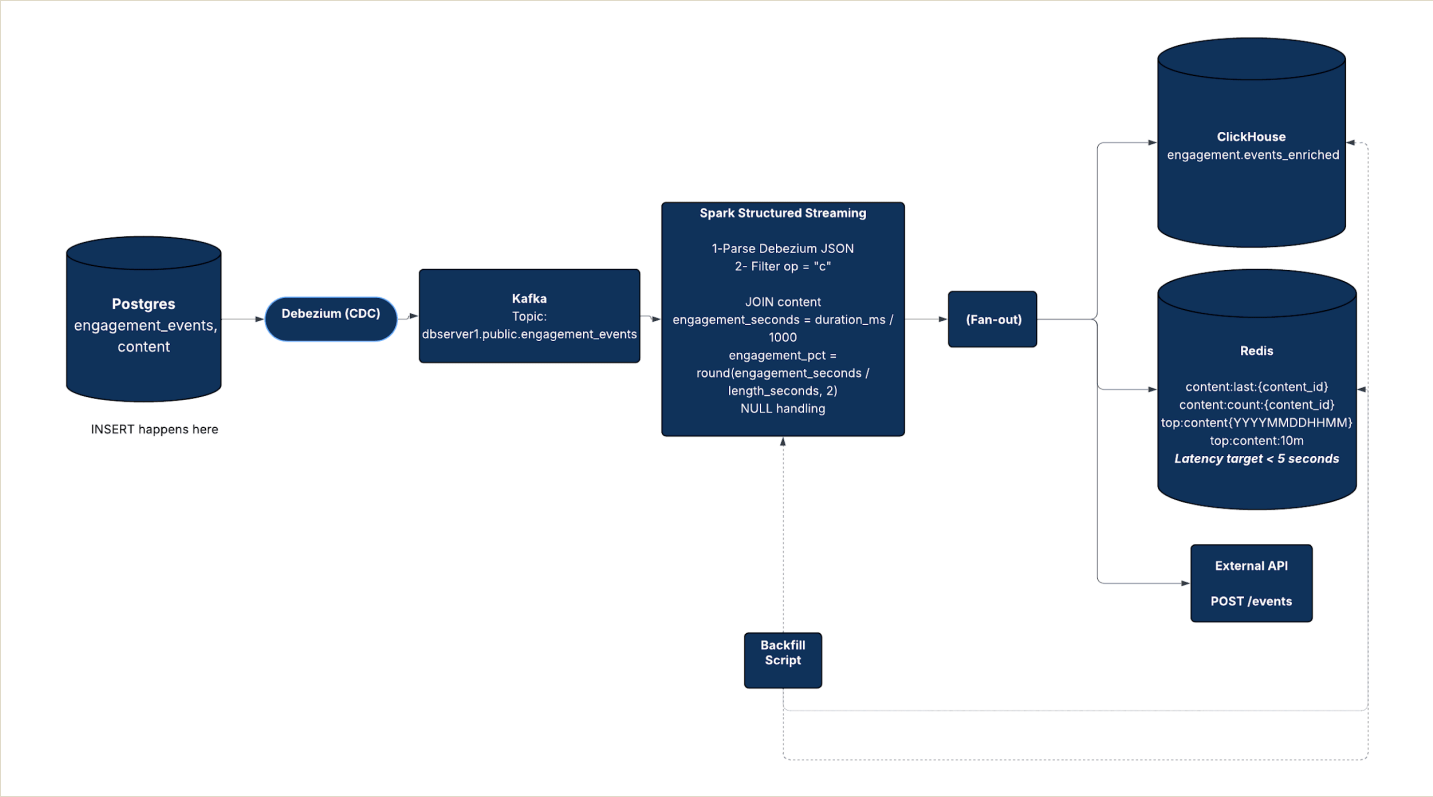
1. قاعدة بيانات عمودية للتحليلات (ClickHouse بديل BigQuery)
2. Redis ك "الأكثر تفاعلًا آخر 10 دقائق"
3. نظام خارجي (External API) لا نتحكم به

2) اختيار ال Stack

- Debezium: CDC من Postgres بدون polling
- Kafka: ناقل رسائل يفصل المصدر عن المعالجة
- PySpark (Spark Structured Streaming): إطار streaming مع checkpointing لتحقيق دقة عالية + fan-out
- ClickHouse: قاعدة عمودية محلية للتحليلات

- Redis: serving/real-time
- External API: mock HTTP service

3) مسار البيانات



Stage 0 — Source Tables (PostgreSQL)

Tool: PostgreSQL

Tables:

- `engagement_events` (fact table)
- `content` (dimension table)

(INSERT):

```
INSERT INTO engagement_events
(content_id, user_id, event_type, duration_ms, device)
VALUES (1, 777, 'play', 7000, 'web');
```

This represents a user interaction event.

Stage 2 — Change Data Capture (CDC)

Tool: Debezium PostgreSQL Connector

- Debezium reads PostgreSQL

- Every INSERT into `engagement_events` is captured as a change event.

Output: JSON message sent to Kafka.

Stage 2— Message Transport (Kafka)

Tool: Apache Kafka
Topic:

`dbserver1.public.engagement_events`

- Buffers events
- Guarantees ordering within partitions
- Allows replay (offset-based recovery)

Example Debezium message


```
{
  "before": null,
  "after": {
    "id": 16,
    "content_id": 1,
    "user_id": 999,
    "event_type": "play",
    "event_ts": "2026-01-08T14:21:52Z",
    "duration_ms": 7000,
    "device": "web"
  },
```

```
  "op": "c"
}
```

Spark only consumes `after` records where `op = "c"` (create).

Stage 3 — Stream Ingestion

Tool: Spark Structured Streaming (PySpark)

- Spark reads Kafka as a streaming source.
- Parses Debezium JSON.
- Filters only `INSERT` operations.
- Converts the stream into a structured DataFrame.

Stage 4 — Enrichment & Transformation

Tool: Spark (joins + derived columns)

Join:

```
engagement_events.content_id = content.content_id
```

Derived Fields:

- `content_type` ← from `content`
- `length_seconds` ← from `content`
- `engagement_seconds = duration_ms / 1000`
- `engagement_pct = round(engagement_seconds / length_seconds, 2)`

Null Handling Rule:

- If `duration_ms` or `length_seconds` is NULL → `engagement_seconds` and `engagement_pct` are set to NULL.

Example enriched record:

```
{
  "event_id": 16,
  "content_id": 1,
  "user_id": 999,
  "event_type": "play",
  "event_ts": "2026-01-08 14:21:52",
  "duration_ms": 7000,
  "device": "web",
  "content_type": "video",
  "length_seconds": 300,
  "engagement_seconds": 7.0,
  "engagement_pct": 0.02
}
```

4. Multi-Sink Fan-out

Spark uses `foreachBatch` to fan out the same enriched data to three destinations in parallel.

Sink A — ClickHouse (Analytics Store [Column Based])

Table:

`engagement.events_enriched`

Write mode: Append

Example query:

```
SELECT content_id, avg(engagement_pct)
FROM engagement.events_enriched
GROUP BY content_id;
```

Sink B — Redis (Real-Time Layer)

Role: Low-latency serving & aggregation

1. Latest event per content

```
Key: content:last:{content_id}
Type: String (JSON)
TTL: ~1 hour
```

2. Total event counter per content

```
Key: content:count:{content_id}
Type: Integer
```

3. Per-minute aggregation buckets

```
Key: top:content:bucket:{YYYYMMDDHHMM}
Type: ZSET
Member: content_id
Score: number of events in that minute
TTL: 15 minutes
```

4. Last 10 minutes aggregation

```
Key: top:content:10m
Type: ZSET
Built by merging last 10 minute buckets
TTL: 120 seconds
```

Latency Guarantee:
Redis updates arrive within < 5 seconds of event creation.

Sink C — External System (HTTP API)

Tool: FastAPI (mock external service)

Endpoint:

```
POST /events
```

---

5. Failure Handling

Spark Checkpointing

- Offsets and state stored in checkpoint directories
- Job resumes safely after failure

Idempotency (Non-Transactional Sinks)

- Redis keys include `event_id`
- Duplicate events are ignored
- Provides exactly-once effect

External API Failures

Two strategies:

- Fail-fast: Spark retries the batch
- Best-effort: Log failures and continue

(Current implementation demonstrates best-effort behavior.)

---

6. Backfill (Historical Reprocessing)

Tool: `backfill.py`

What it does:

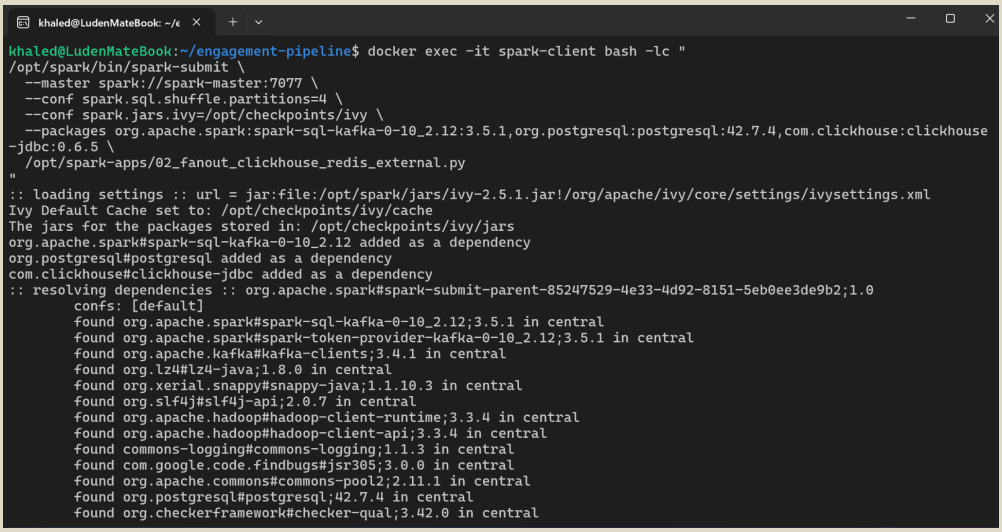
- Reads historical data from Postgres by time range
- Applies the same enrichment logic
- Writes to ClickHouse + Redis
- Uses idempotency keys to avoid duplicates

Usage:

```
python backfill.py --start 2026-01-08T09:00:00 --end 2026-01-08T10:00:00
```

4) تجربة البايبلاين

1.الكونكتر

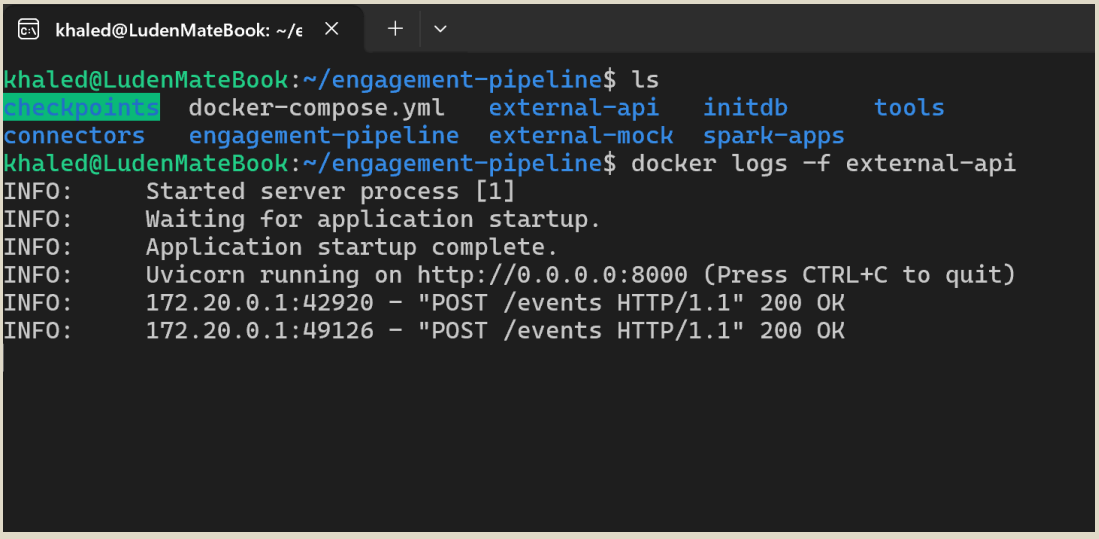


بعد تنفيذ هذا الأمر، يبدأ Spark في:

- تحميل الاعتمادات
- تهيئة بيئة التشغيل
- تشغيل برديوسر مستمر (Streaming)

بمجرد مايصير اي Hit لافنت جيديد راح يتحول للتحويلات من ثم فان اوت ل3 كنسيومرز

2.ال External API



في هذه الخطوة:

- شغّلنا النظام الخارجي باستخدام حاوية external-api.
- تابعنا السجلات عبر docker logs -f للتأكد أن الخدمة تعمل بشكل صحيح.

النتيجة:

- الـ API اشتغل على المنفذ 8000.
- Spark بدأ يرسل له الأحداث المُثرّاة.
- ظهور رسائل POST /events 200 OK يؤكد أن البيانات وصلت بنجاح للنظام الخارجي.

3.الـ INSERT

```
^Ckhaled@LudenMateBook:~/engagement-pipelinedocker exec -it pg psql -U app -d engagement -c "
INSERT INTO engagement_events (content_id, user_id, event_type, duration_ms, device)
VALUES (1, 999, 'play', 7000, 'web');
"
INSERT 0 1
khaled@LudenMateBook:~/engagement-pipeline$
```

- قمنا بإدخال حدث جديد يدويًا في جدول engagement_events داخل PostgreSQL.
- هذا الإدخال يمثّل مصدر البيانات (الـ input) للنظام كامل.

النتيجة:

- السطر تم إدخاله بنجاح (INSERT 0 1).
- بمجرد الإدخال، يبدأ البايبلاين تلقائيًا بالتقاط الحدث وبثّه لبقية المراحل (Spark → Kafka → باقي الأنظمة).

4.التأكد من وصول البيانات لأول كونسيومر (ClickHouse) قاعدة البيانات العمودية

```
khaled@LudenMateBook:~/engagement-pipeline$ docker exec -it clickhouse clickho
use-client -q "
SELECT event_id, content_id, user_id, event_type, engagement_seconds, engageme
nt_pct
FROM engagement.events_enriched
ORDER BY event_id DESC
LIMIT 5;"
19      1       777     play    7       0.02
18      1       999     play    7       0.02
17      1       12345   play    7       0.02
16      1       999     play    7       0.02
15      1       999     play    7       0.02
khaled@LudenMateBook:~/engagement-pipeline$
```

في هذه الخطوة:

- استعلمنا من جدول events_enriched داخل ClickHouse.
- تأكدنا أن الحدث اللي دخلناه تم إثراؤه بنجاح (تحويل المدة للثواني + حساب نسبة التفاعل).

النتيجة:

- ظهور السجلات الجديدة مباشرة.
- القيم مثل engagement_seconds = 7 و engagement_pct = 0.02 محسوبة صح.
- هذا يثبت أن مسار Kafka → Spark → ClickHouse شغال بشكل صحيح.

5.التأكد من وصول بيانات التحديث الفورية Redis

```
khaled@LudenMateBook:~/engagement-pipeline$ docker exec -it redis redis-cli GET content:l
ast:1
docker exec -it redis redis-cli GET content:count:1
"{\"event_id\": 19, \"content_id\": 1, \"user_id\": 777, \"event_type\": \"play\", \"even
t_ts\": \"2026-01-09 14:04:31.163702\", \"duration_ms\": 7000, \"device\": \"web\", \"con
tent_type\": \"video\", \"length_seconds\": 300, \"engagement_seconds\": 7.0, \"engagemen
t_pct\": 0.02}"
"13"
khaled@LudenMateBook:~/engagement-pipeline$
```

في هذه الخطوة:

- استعلمنا من جدول events_enriched داخل ClickHouse.
- تأكدنا أن الحدث اللي دخلناه تم إثراؤه بنجاح (تحويل المدة للثواني + حساب نسبة التفاعل).

النتيجة:

- ظهور السجلات الجديدة مباشرة.
- القيم مثل engagement_seconds = 7 و engagement_pct = 0.02 محسوبة صح.
- هذا يثبت أن مسار Kafka → Spark → ClickHouse شغال بشكل صحيح.

6.عرض المحتويات الأكثر تفاعل اخر 10 دقائق

```
khaled@LudenMateBook:~/engagement-pipeline$ docker exec -it redis redis-cli ZREVRANGE top:content:10m 0 10 WITHSCORES
1) "1"
2) "4"
khaled@LudenMateBook:~/engagement-pipeline$
```

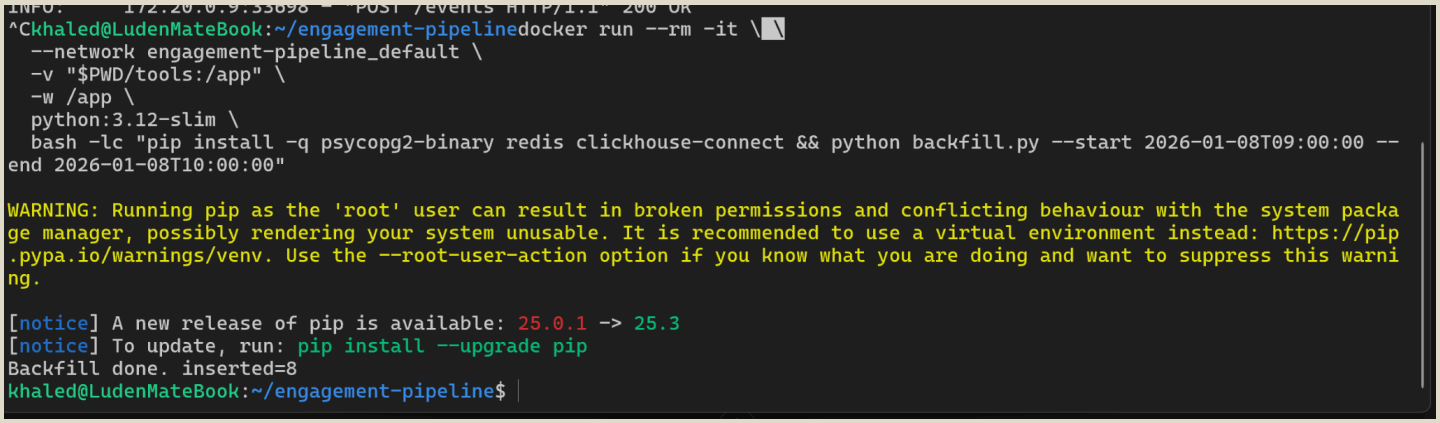
في هذه الخطوة:

- استخدمنا ZREVRANGE على المفتاح top:content:10m في Redis.
- هذا المفتاح يمثل تجميع آخر 10 دقائق مبني على Buckets زمنية.

النتيجة:

- الرقم "1" هو content_id.
- الرقم "4" هو مجموع التفاعلات (score) خلال آخر 10 دقائق.
- الترتيب تنازلي، يعني هذا المحتوى هو الأعلى تفاعلًا حاليًا.

7.تنفيذ ال Backfill



في هذه الخطوة:

- شغّلنا حاوية Python مؤقتة (docker run --rm) داخل نفس شبكة المشروع.
- ثبّتنا المكتبات المطلوبة (psycopg2, redis, clickhouse-connect).
- شغّلنا سكربت backfill.py مع تحديد فترة زمنية معيّنة.

النتيجة:

- تمّت إعادة معالجة البيانات التاريخية بين الوقتين المحددين.
- السطر Backfill done. inserted=8 يعني أنه تم إدخال 8 أحداث قديمة بنجاح إلى النظام.

5)الخلاصة

العمل هذا المشروع تجربة عملية ممتعة بالذات انها تقنيات لاول مره اتعامل معها مثل CDC باستخدام Debezium، والتجميع الزمني في Redis وفايدته الفعلية، والتعامل مع Multi-sink fan-out في Spark Structured Streaming.
التكليف كان تحدّيًا مفيدًا وساعدني على تعميق فهمي لكيفية تصميم أنظمة Streaming بشكل عام

6)تحسينات ممكن اضافتها

- إضافة مراقبة (Monitoring Dashboard)
- تحسين إدارة الأخطاء وإعادة المحاولة (Retry & DLQ) خاصة مع النظام الخارجي.
- إضافة اختبارات تحميل (Load Testing) لمحاكاة ضغط إنتاجي أعلى.
- تصميم Front end للنظام

7) ملف الكود بـ Git

https://github.com/KhaledAlruwita/engagement-pipeline.git