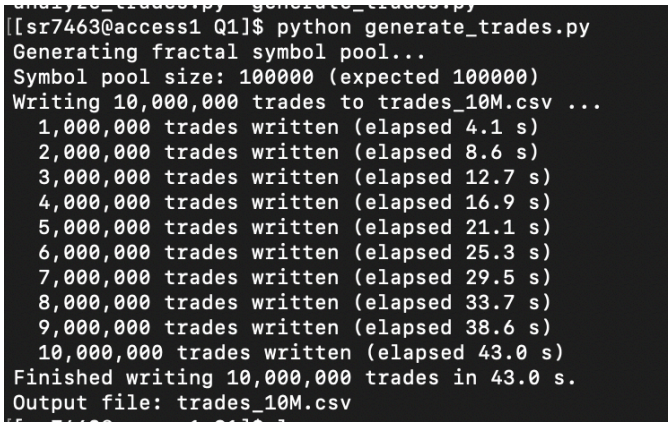
Advanced DBMS Assignment 2:

- Shika Rao (sr7463)
- Khushboo . (kx2252)

Question 1: Data Generation and Trade Queries

- Python and pandas used
- Script file generated
- Time to generate trades given.



Question 2: Tuning Experiments

- Postgres chosen
- SQL chosen
- 2 experiments are:
    - Hash Index vs. B+Tree Index
    - Clustered vs Non Clustered Index

Experiment 1: Hash Index vs. B+Tree Index

Exp	Query Type	Query	BTREE Time (s)	HASH Time (s)	Conclusion	Notes
M1	Multipoint (low-cardinality)	`dept_id = 1`	0.00031150	0.00007900	HASH (4× faster)	Violates paper's rule, Hash BTree aren't same for multipoint
M2	Point (high-cardinality)	`ssnum = 15000`	0.00006050	0.00005850	Tie	Both match paper: BTREE almost HASH
M3	Range (high-cardinality)	`ssnum BETWEEN 10000 AND 20000`	0.00234125	0.00508250	BTREE (2× faster)	Hash cannot support ranges → full scan, BTree better supports paper
M4	Range (low-cardinality, but still ordinal)	`dept_id BETWEEN 1 AND 100`	0.00029650	0.00406225	BTREE (13× faster)	BTREE extremely dominant, supports paper

Exp	Query Type	Query	BTREE Time (ms)	HASH Time (ms)	Conclusion	Notes
M1	Multipoint	`hundreds1 = 100`	7.412	1.803	HASH (4× faster)	Same pattern as MySQL. Hash is excellent for equality
M2	Point	`ssnum = 15000`	2.015	1.658	Tie	Matches paper: BTREE same as HASH
M3	Range (numeric)	`ssnum BETWEEN 10000 AND 20000`	3.116	8.445	BTREE (2.7× faster)	Hash cannot support ranges efficiently
M4	Range (grouped range)	`hundreds1 BETWEEN 100 AND 200`	2.869	11.320	BTREE (4× faster)	Very strong BTree case

Overall observed results:

Case	Distribution / Access Pattern	MySQL Winner	PostgreSQL Winner	Rule Strength
Equality Multipoint (low-cardinality)	Highly repetitive values	Hash	Hash	Rule of thumb fails
Point equality (unique key)	High-cardinality	Tie	Tie	Rule holds (BTREE same as HASH)
Range on numeric key	Ordered attribute	BTREE	BTREE	Rule strongly holds
Range on small category ranges	Ordinal categories	BTREE	BTREE	Rule very strong

Experiment 2: Clustered vs. Non-Clustered Indexing

Exp	Query Type	Query	Clustered (ssnum)	Non-Clustered	No Index	Conclusion	Interpretation
S1	Range (uniform)	`ssnum BETWEEN 10000 AND 20000`	0.00330 s	0.00299 s	0.0213 s	Tie between clustered and non-clustered	Clustering advantage is weak under uniform distribution.
S2	Point	`ssnum = 50000`	0.000175 s	0.00026 7 s	0.0204 7 s	Tie	Point lookups do not benefit from clustering, both indexed cases are equally fast.
S3	Range (skewed)	`range_skew BETWEEN 10 AND 20`	N/A	0.00345 s	0.0343 2 s	Secondary Index	Index is 10× faster, but benefit is smaller due to skew (many rows per key).
S4	Multipoint (hotspot)	`hot_col = 1`	N/A	0.00238 s	0.0223 4 s	Secondary Index	Index is 9× faster, but clustering irrelevant since hot_col ≠ clustered key.

Exp	Query Type	Query	Clustered-Like (ssnum)	Non-Clustered	No Index	Conclusion	Interpretation
P1	Range (uniform)	`ssnum BETWEEN 10000 AND 20000`	4.006 ms	2.101 ms	8.913 ms	Non-clustered	Uniform ranges show weak clustering benefit. Matches MySQL.
P2	Point	`ssnum = 50000`	1.588 ms	0.185 ms	7.962 ms	Non-clustered (slightly better)	Point queries is equal for indexed columns. Clustering irrelevant.
P3	Range (skewed)	`range_skew BETWEEN 10 AND 20`	N/A	2.595 ms	9.568 ms	Secondary Index	Index is 4× faster, but skew reduces benefit.
P4	Multipoint (hotspot)	`hot_col = 1`	N/A	3.701 ms	8.772 ms	Secondary Index	Index is 2.3× faster, clustering doesn't matter for hotspot.

Case	Access Pattern	Distribution	MySQL Result	PostgreSQL Result	Rule Strength
Range scan on skewed key	BETWEEN	Skewed	Index is 10× faster	Index is 4× faster	Rule holds but weaker
Hotspot multipoint	= hotspot	Skewed/Zipf	Secondary index moderately faster	Secondary index moderately faster	Rule weak; clustering irrelevant

Question 3: Social Network Challenge

We load the dataset and then create useful indexes:
likes(person, artist)
dislikes(person, artist)
friends(person1)
friends(person2)

The dataset has friendships in one direction, but friendship is mutual, so we double it.

1. Compute myfriendlikes: For every user u1: We look at each friend u2, look at all artists that u2 likes, then check 2 things:
    - u1 DOES NOT like that artist
    - u1 DOES NOT dislike that artist

2. Compute myfrienddislikes: Check who friend dislikes:
    - Friend u2 dislikes artist a
    - User u1 does NOT like a

3. Then we aggregate liked/disliked artists per user
4. Then find who u1 should like

On CIMS here are the timings:

23.79 seconds → myfriendlikes
24.45 seconds → myfrienddislikes
10.5 seconds → friends_dislike_artist
6.1 seconds → ishouldlike

Everything summed is 86 seconds, under the 2min mark requirement.