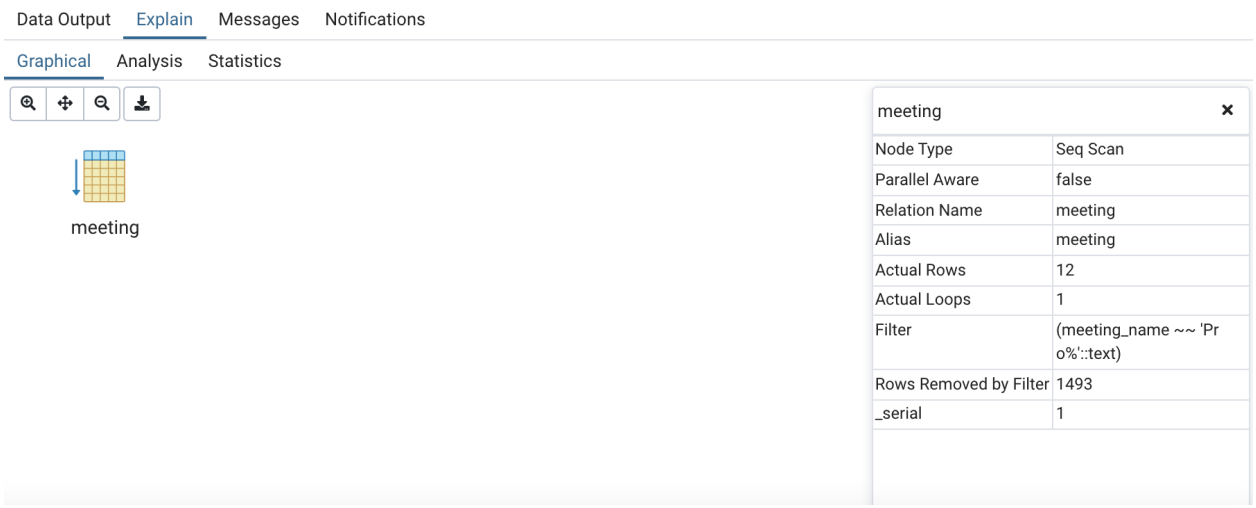
Last Name: Ren First Name:Yangkaiqi Student ID:93672321

1. [10 pts] To start down the path of exploring physical database designs (indexing) and query plans, start by checking the query plans for each of the following queries against the database without any indexes using the graphical EXPLAIN ANALYZE functionality in PgAdmin4 (see instructions). For each query, take snapshots of the EXPLAIN results and paste them into your copy of the HW7 template file. (Just take a snapshot of the query plan, not the whole screen, but do be sure to include the additional details window discussed in the instructions..)

a) [2pts]

```
select meeting_name FROM meeting WHERE meeting_name LIKE 'Pro%';
```

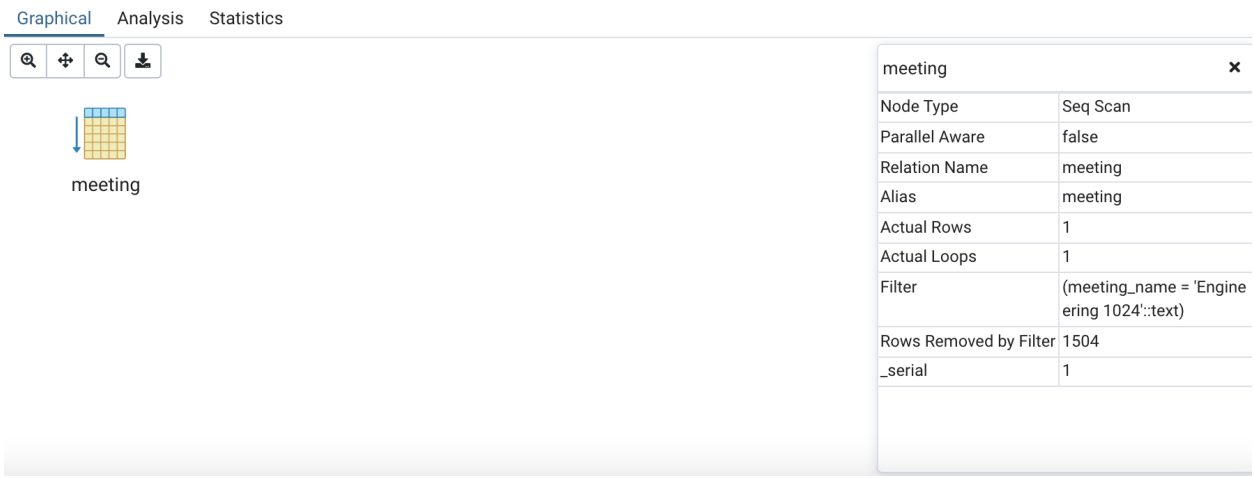
Results:



b) [2pts]

```
SELECT * FROM meeting WHERE meeting_name = 'Engineering 1024';
```

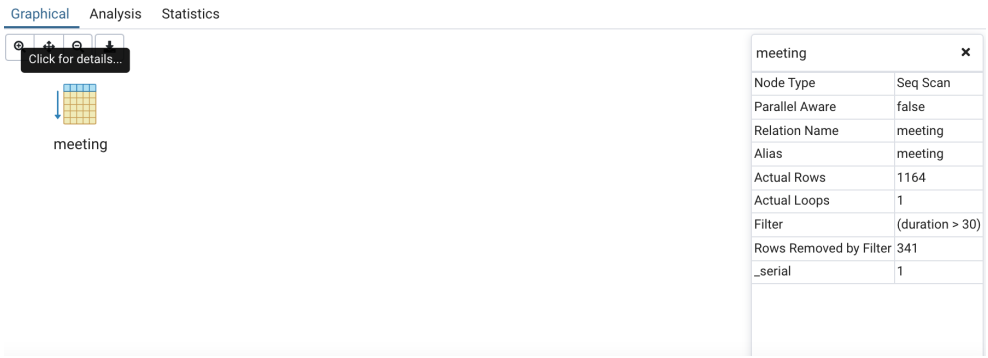
Results:



c) [2pts]

```
SELECT * FROM meeting WHERE duration > 30;
```

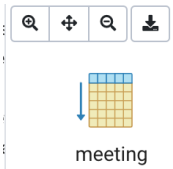
Results:



d) [2pts]

```
SELECT * FROM meeting WHERE duration = 60;
```

Results:

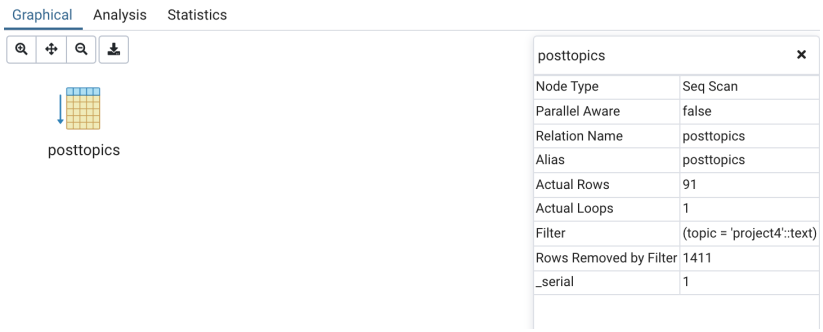


meeting	
Node Type	Seq Scan
Parallel Aware	false
Relation Name	meeting
Alias	meeting
Actual Rows	315
Actual Loops	1
Filter	(duration = 60)
Rows Removed by Filter	1190
_serial	1

e) [2pts]

```
SELECT * FROM posttopics WHERE topic = 'project4';
```

Results:



posttopics	
Node Type	Seq Scan
Parallel Aware	false
Relation Name	posttopics
Alias	posttopics
Actual Rows	91
Actual Loops	1
Filter	(topic = 'project4'::text)
Rows Removed by Filter	1411
_serial	1

2. [10 pts] Now create secondary indexes on the meeting.meeting_name, recording.start_time, and posttopics.topic attributes separately. (I.e., create three indexes, one per table.) Paste your CREATE INDEX statements below.

```
CREATE INDEX meetingnameIdx on Meeting (meeting_name);
CREATE INDEX starttimeIdx on recording (start_time);
CREATE INDEX topic on posttopics (topic);
```

3. [10 pts] Re-"explain" the queries in Q1. Indicate whether the indexes you created in Q2 are used (e.g. by highlighting where on your screenshot the indexes are used), and if so, whether the uses are index-only plans or not. Copy and paste the query plan after each query, as before.

   NOTE: If you see any types of scans that don't make sense to you, you might want to refer to the following short blog:

   https://www.cybertec-postgresql.com/en/postgresql-indexing-index-scan-vs-bitmap-scan-vs-sequential-scan-basics/?gclid=Cj0KCQiAhMOMBhDhARIsAPVml-FkwGU5Ya8qX2JJF_Yph4oZA_OwCscOBs8Bot7lZb-tGRzc0UfqOpMaAp86EALw_wcB

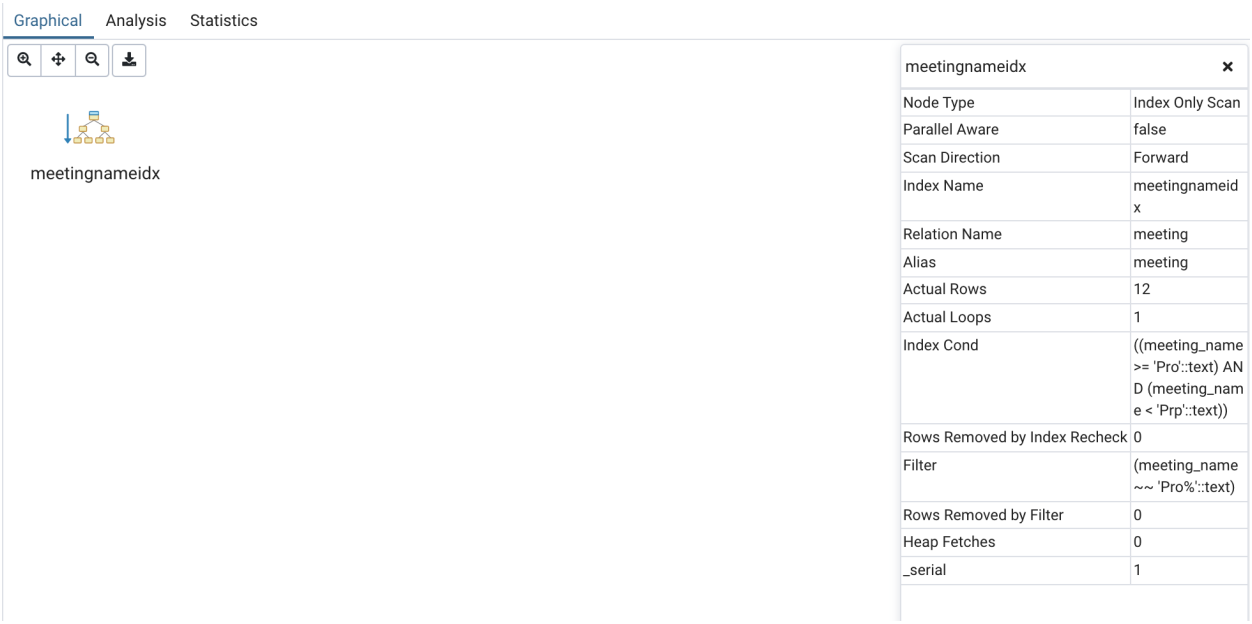
a) [2pts]

```
SELECT meeting_name FROM meeting WHERE meeting_name like 'Pro%';
```

Results:

Index used: Y

Index-only plans: Y



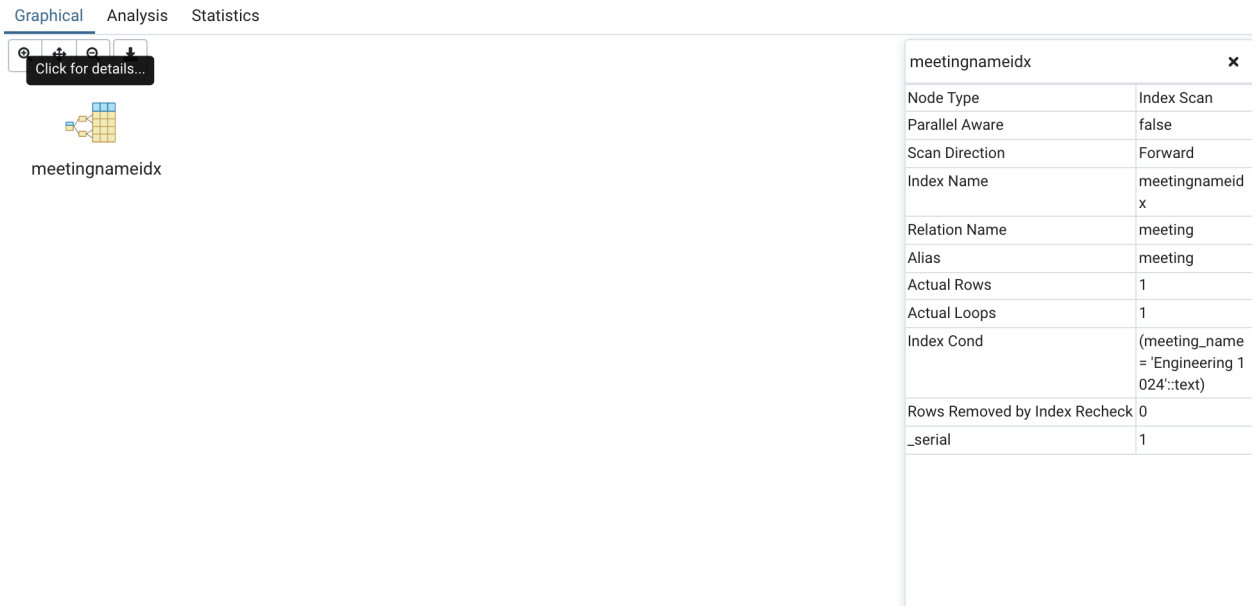
b) [2pts]

```
SELECT * FROM meeting WHERE meeting_name = 'Engineering 1024';
```

Results:

Index used: Y

Index-only plans: N



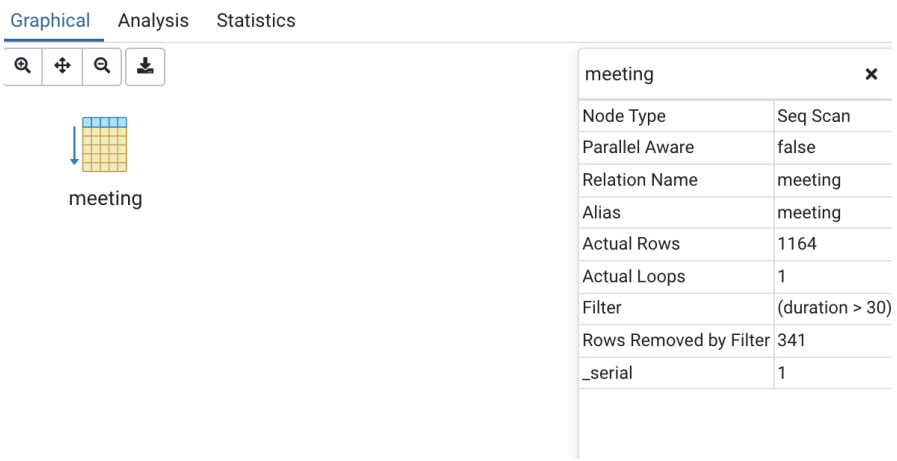
c) [2pts]

```
SELECT * FROM meeting WHERE duration > 30;
```

Results:

Index used: N

Index-only plans: N



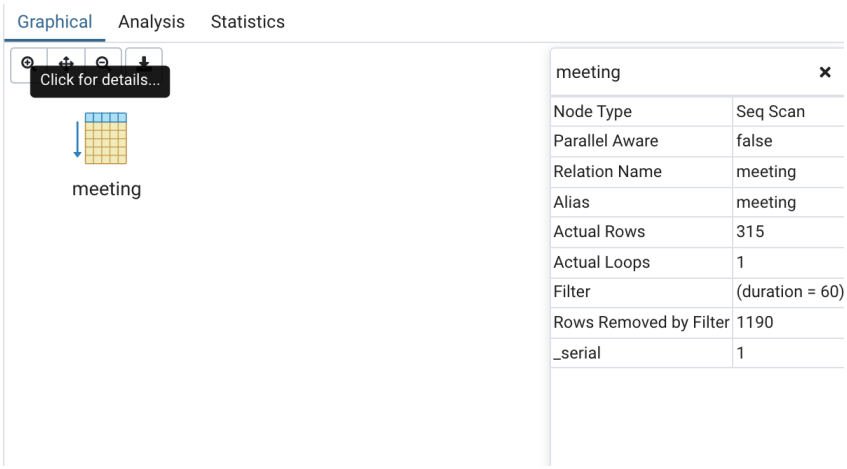
d) [2pts]

```
SELECT * FROM meeting WHERE duration = 60;
```

Results:

Index used: N

Index-only plans: N



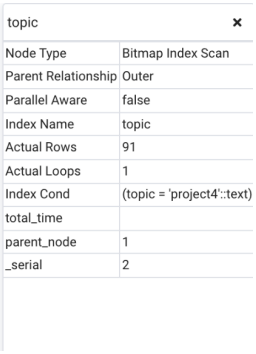
e) [2pts]

```
SELECT * FROM posttopics WHERE topic = 'project4';
```

Results:

Index used: Y

Index-only plans: N



posttopics

Node Type	Bitmap Heap Scan
Parallel Aware	false
Relation Name	posttopics
Alias	posttopics
Actual Rows	91
Actual Loops	1
Recheck Cond	(topic = 'project 4'::text)
Rows Removed by Index Recheck	0
Exact Heap Blocks	9
Lossy Heap Blocks	0
_serial	1

4. [24pts] Examine the queries above and answer the following questions:
   a. [8pts] For queries 3(b) and 3 (c):

      i. Explain the difference between their plans.

      3b used meetingnameIdx to answer the query while 3c used the full table scan to filter duration > 30 meetings

      ii. Why are the queries using different plans?

      It is because 3b was searching for meeting_ame which is related to meetingnameIdx where PostgreSQL is able to run an index scan which costs less than full table scan.

      On the other hand 3c is looking for duration > 30 meeting which does not have a related index. Therefore PostgreSQL ran sequential scan to filter out the unuseful meetings.

   b. [8pts] For the LIKE query in 3(a).

      i. Explain whether the index is helpful and why.

      It is answered by Index only scan. It is helpful that the original table has 1505 rows, and the actual row is only 12. Which cost less than full table scan.

ii. What if the string is changed to '%Pro%'? Will the index be helpful then, and why?

The query (LIKE '%Pro%') still needs a full table scan, because it isn't a range query that can be handled by a B+ tree index since the condition is not specifically on the prefix but any part of the key.

c. [8pts] For equality query 3(b), is the index helpful and why?

The index is helpful for query 3b because meeting_name can be found by the index meetingnameIdx which helps to run Index scan which cost less than a full table scan.

5. [22pts] It's time to go one step further and explore the notion of a "Composite Index," which is an index that covers several fields together.

a. [4pts] Create a composite index on the attributes first_name and last_name (in that order!) of the Users table. Paste your CREATE INDEX statement below:

```
CREATE INDEX UsersFirstLastName ON Users(first_name, last_name);
```

b. [6 pts] 'Explain' the queries below. Copy and paste the query plan of each query into your response, as before. Be sure to look carefully at each plan.

i. [2pts]

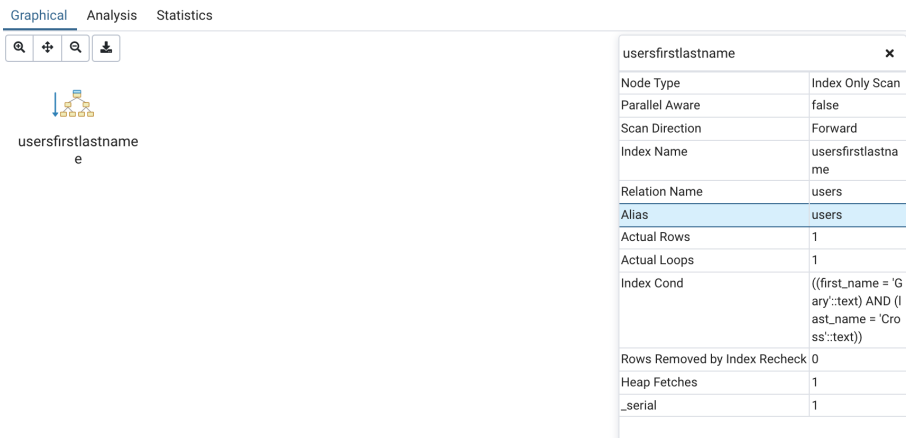
```
SELECT * FROM users WHERE last_name = 'Cross';
```

Results:

ii. [2pts]

```
SELECT last_name FROM users WHERE first_name='Gary' AND last_name =
'Cross';
```

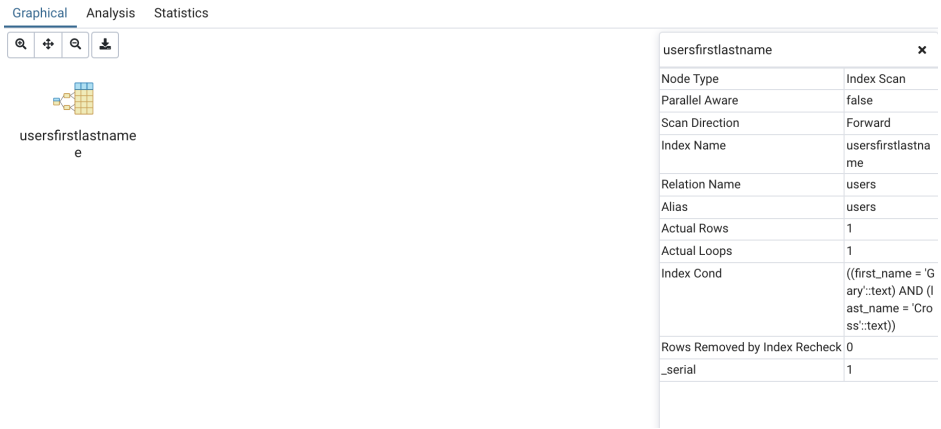
Results:



iii. [2pts]

```
SELECT * FROM users WHERE first_name='Gary' AND last_name = 'Cross';
```

Results:



c. [12pts] For each query above (i, ii, iii), explain whether the composite index is used or not and why. Also indicate whether the plan is index-only.

i. The composite index is not used because values in the additional composite index columns are grouped by the Users of the first column. In order to find values in the second column without having an index on the first column, it would have to scan the entire index.

ii. The composite index is used and every field is equal to a specific value. Index only scan is used which cost less than full table scan

iii. The composite index is used because it's an equality search such that every field is equal to a specific value, hence using the index is faster than a full table scan.

6. [24pts] In Postgres, you can ask for a table to be clustered according to an index.

   Eg: To create a clustered index on Users (first_name), one would first create the index:

```
CREATE INDEX userFirstNameIdx ON Users(first_name);
```

   Then, the following statement clusters the Users table based on the userFirstNameIdx index:

```
CLUSTER Users USING userFirstNameIdx;
```

   For each of the queries below, specify the most appropriate CREATE INDEX statement that would build a helpful index to accelerate that query. If an index wouldn't be useful for a given query, then write 'No index'. If clustering the table will help accelerate the queries, write the CLUSTER statement too. For all of the questions, give brief reasons for your answer.

   In each case where you do decide to employ clustering, also see if you can "prove" that your decision is sound based on the resulting query plan costs - which you can try to do by CREATEing the index, running the query and paying attention to time, then CLUSTERing the index, and then running the query once more. See if you notice a difference. (It will be okay if you don't notice one, as this is a pretty small sample database, but you should at least try!)

   a. [8pts]

```
SELECT * FROM PostTopics WHERE post_id = '3';
```

   Results:

   CREATE INDEX posidIdx on PostTopics (post_id);
   Cluster PostTopics Using postidIdx
   Index is helpful for this query because index can cost less than sequential scan. Cluster helps to run faster because post_id is multi_valued attributes with duplicate values.Therefor cluster helps but it will not make an improvement in runtime because there are not that many of rows in the table

   b. [8pts] (**Note:** All of the SWOOSH teachers were born after 1950...)

```
SELECT * FROM InstructorEducation WHERE graduation_year > 1800;
```

   Results:

   No index is the best choice for this query because all teachers were born after 1950. Therefore, no matter what sequential scan will be used.

   c. [8pts]

```
SELECT meeting_id, count(*) AS cnt FROM Attended GROUP BY meeting_id;
```

   Results:

   CREATE INDEX AttendedmeetingidIdx ON swoosh.Attended(meeting_id);
   An index should be created here for grouping query, and in fact it can be processed as an index-only query. The index should be an unclustered index on table Attended and attribute kind because index-only queries do not touch the data (so clustering is irrelevant and it should thus

be used for something else where it can help).