Principles of Data Management

Course Code – 5516

Spring 2017, Section: 01

[Project: Phase 1]

Submitted To:

Dr. Eduard C. Dragut
Assistant Professor

Submitted By:

Ashis Kumar Chanda

(TU ID: 915364305)

Procedure:

- At first, I started to study on Google scholar pages to know about the process of finding each Google scholar profile and necessary link, Html id, classes and so on.
- Then, I planned to start with Python to retrieve data form Google scholar pages using **beautiful soup** package.
- However, I found Java has a library, (**HtmlUnit**) that is easy to use and extract data form Html Tag.
- So, I worked on HtmlUnit. At first, I took the URL structure of Google Scholar search page and call this URL from my code.
- After getting first 10 authors for a single keyword search, I called next button URL to find next author list.
- At the same time, I also run code to find individual profile link and load the page.
- From the individual profile link, I collected author name, position, email, homepage, h-index.
- However, the information is not stored in proper Html structure. Thus, I need to use many string parsing operation to find position, email, and homepage.
- Sometimes, the data are missing. So, I stored the missing data as null.
- Moreover, we have to store the publication list. But, it was easy to get the list, because each publication is stored in <Table> tag and there is a good function in HtmlUnit to read data from <Table> tag.

Observation:

- If we take the profile name as a single keyword, then the result is different when we consider the profile name in two keywords, such as: **First name, last name**.
  Example: If we search with "Krishna Kant", then we get 17 Google scholar profiles. However, if we search with first name="Krishna", last name ="Kant"; then we get 12 Google scholar profiles.
- Google scholar searching option is **not case sensitive**. We get same result from Keyword = "Krishna Kant" and Keyword = "krishna kant".

Result:

The total number of authors for a specific name in Google scholar is given at Table 1. Additionally, the average H-index of the same author list is shown at Table 2.

Table: 1

Author Name	**Total number of authors**
Jiewu	22
Wei Zhang	113
Slobodan Vucetic	1
Zhang Qiang	32
Krishna Kant	17

Table: 2

Author Name	**Total number of authors**	**Average of H-index**
Jiewu	22	26.59
Wei Zhang	113	13.51
Slobodan Vucetic	1	28
Zhang Qiang	32	16.56
Krishna Kant	17	4

Discussion:

From this experiment, we learn how to automatically browse web pages by writing code and extract data automatically from the web pages following Html structure. This experience would be useful in many practical experiments.

Principles of Data Management

Course Code – 5516

Spring 2017, Section: 01

[Project: Phase 2]

Submitted To:

Dr. Eduard C. Dragut
Assistant Professor

Submitted By:

Ashis Kumar Chanda

(TU ID: 915364305)

Part 1:

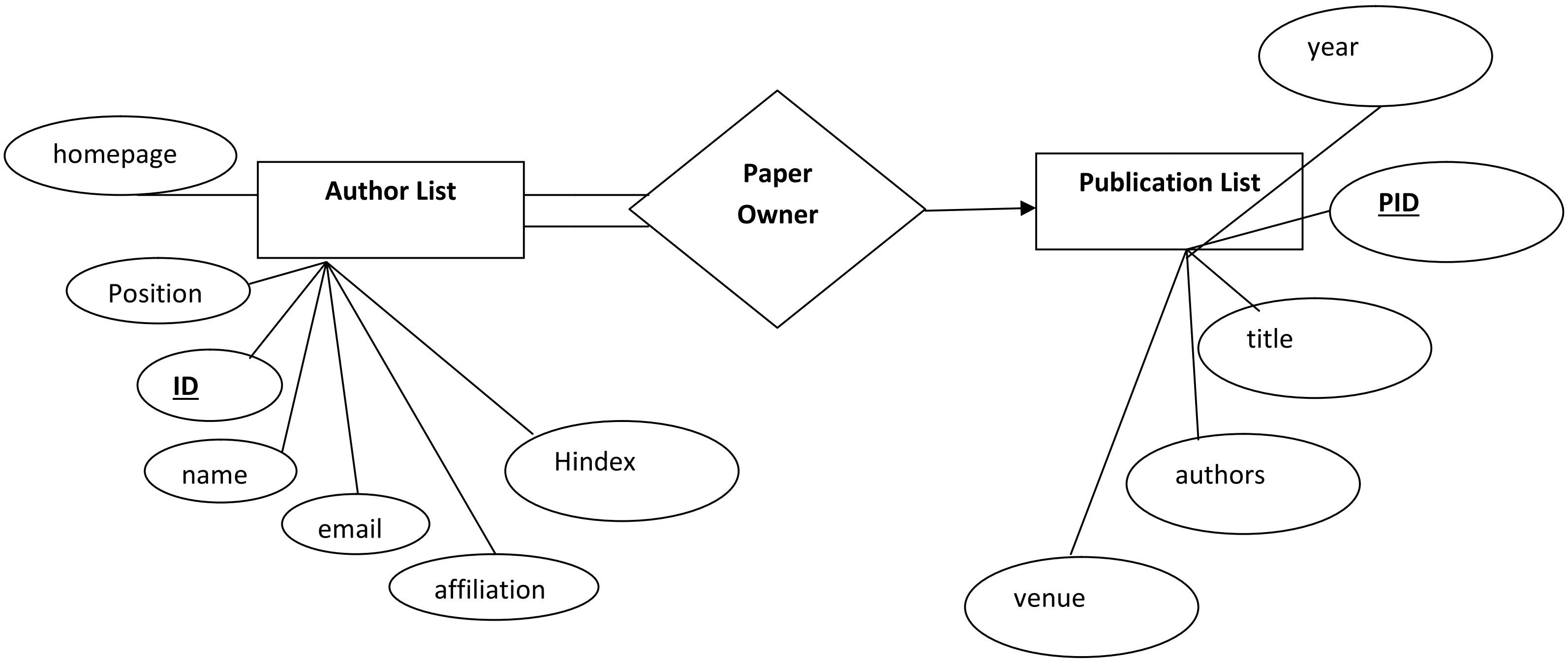


Figure: **ER** diagram of Google Scholar Database

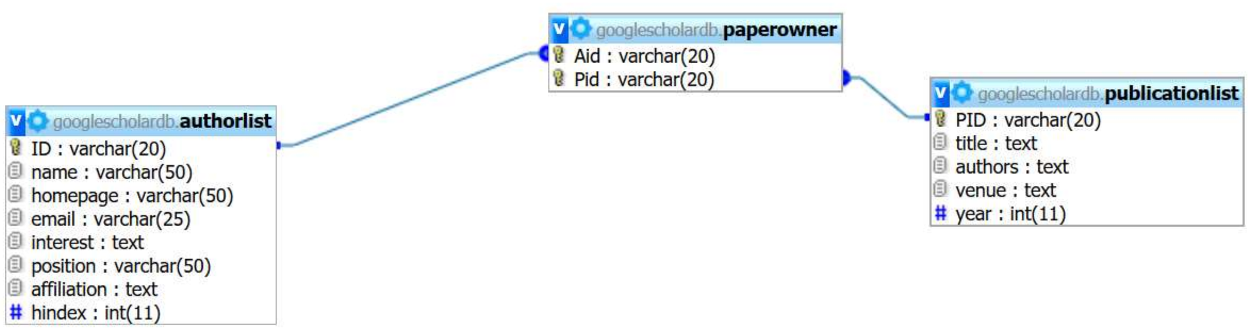


Figure: Table Structure of Google Scholar Database

Part 2:

Create Table Query for AUTHORS:

```
CREATE TABLE authorlist (
`ID` VARCHAR( 20 ) NOT NULL ,
`name` VARCHAR( 50 ) NOT NULL ,
`homepage` VARCHAR( 50 ),
`email` VARCHAR( 25 ),
`interest` TEXT,
`position` VARCHAR( 50 ),
`affiliation` TEXT,
`hindex` INT NOT NULL ,
PRIMARY KEY ( `ID` )
) ENGINE = InnoDB;
```

Create Table Query for PUBLICATIONS:

```
CREATE TABLE publicationlist (
`PID` VARCHAR( 20 ) NOT NULL ,
`title` TEXT NOT NULL ,
`authors` TEXT NOT NULL ,
`venue` TEXT,
`year` INT NOT NULL ,
PRIMARY KEY ( `PID` )
) ENGINE = InnoDB;
```

Another table:

```
CREATE TABLE paperowner(
`a_ID` VARCHAR( 20 ) NOT NULL ,
`p_ID` VARCHAR( 20 ) NOT NULL ,
PRIMARY KEY ( `a_ID` , `p_ID` ),
FOREIGN KEY (a_ID) REFERENCES authorlist on delete cascade on update cascade,
FOREIGN KEY (p_ID) REFERENCES publicationlist on delete cascade on update
cascade
) ENGINE = InnoDB;
```

Part 3:

Number of authors: 178

Number of publications: 68,662

Min, max and avg h-index: 0, 84, and 14.2528

The top-5 institutions by the number of authors:

unknown	7
the chinese university of hong kong	2
ucsf	2
virginia tech	2
??????	2

Used SQL:

```
SELECT affiliation, count( * )

FROM `authorlist`

GROUP BY affiliation

order by count( * ) DESC

LIMIT 0 , 163
```

The top-5 institutions by the avg h-index:

research scientist marintek	84.0000
the comprehensive cancer center of wake forest bap...	81.0000
Temple University	80.0000
Tongji university	69.0000
Physical therapy college of...	68.0000

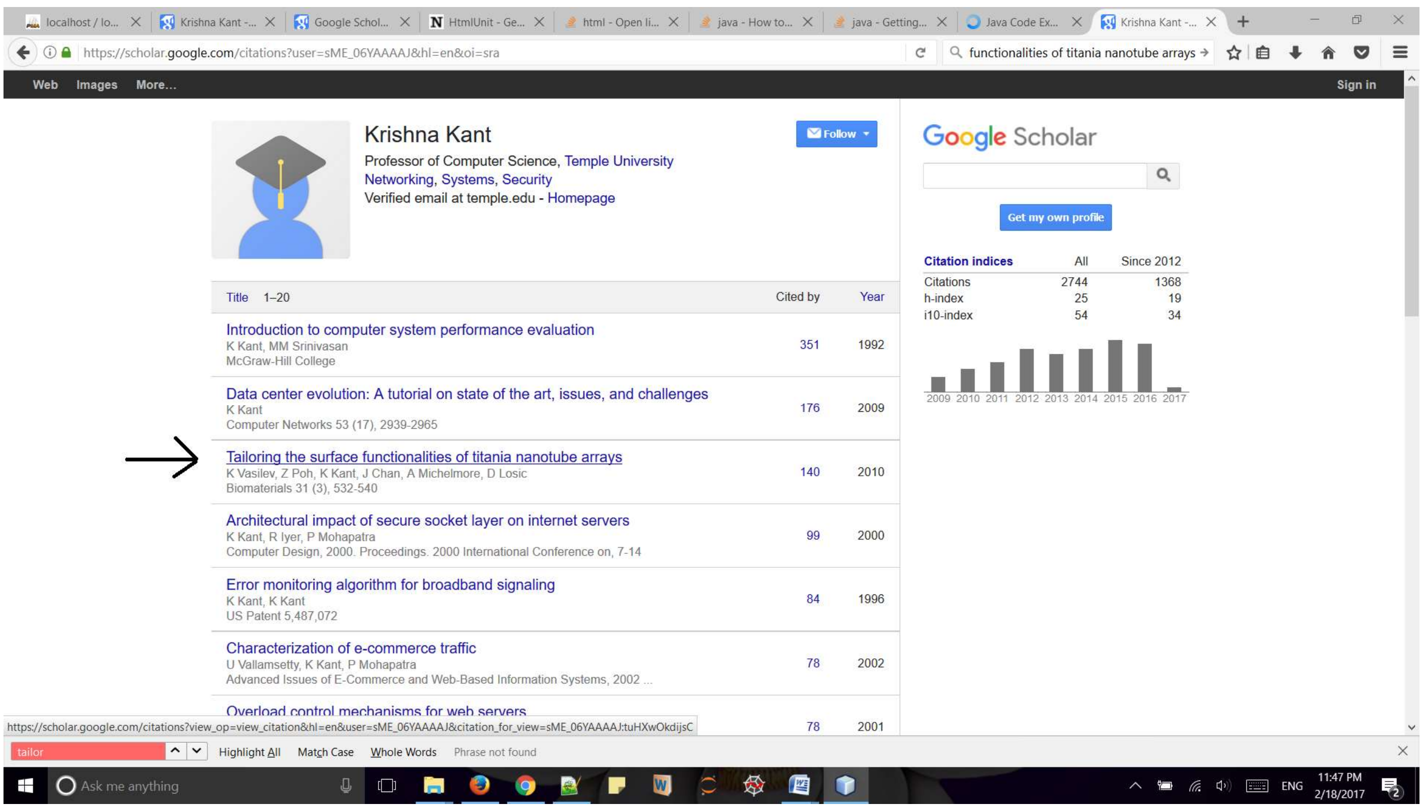
Used SQL:

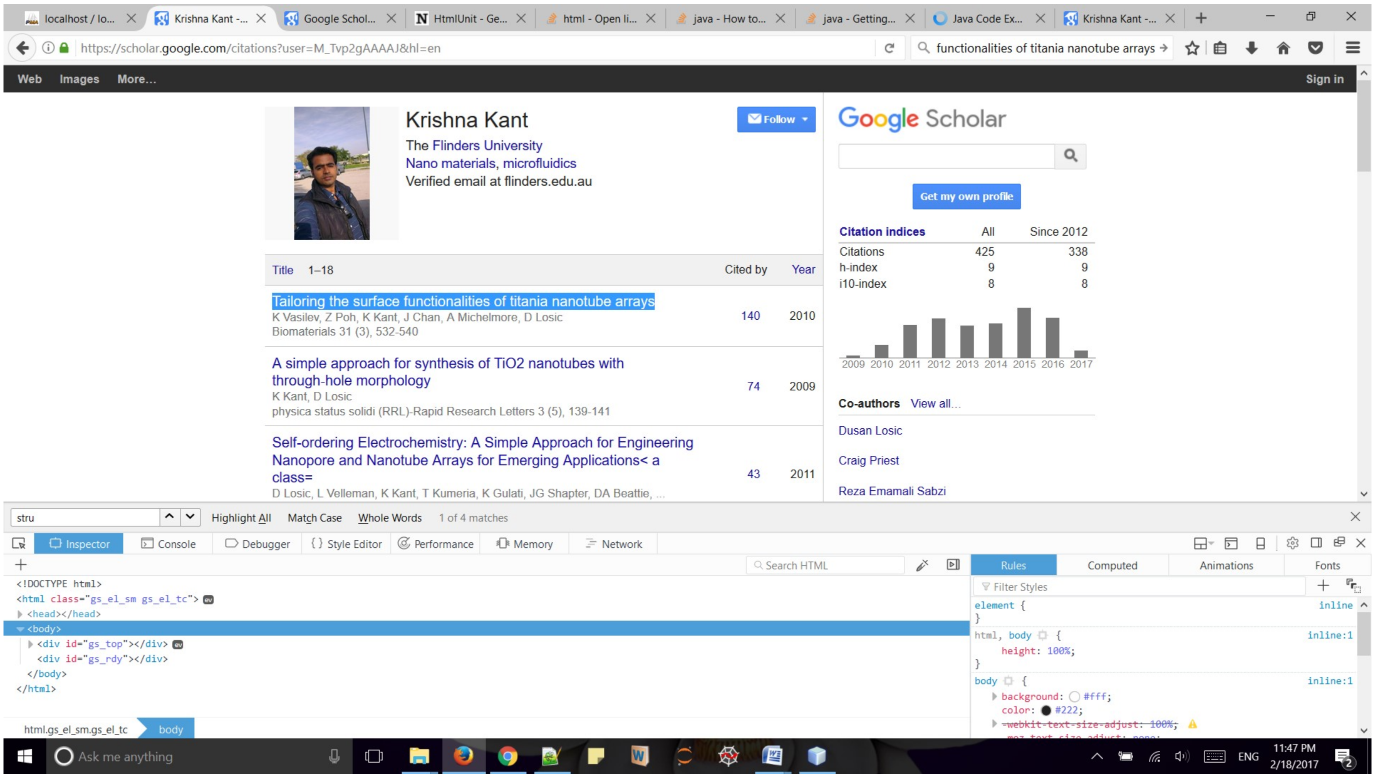
```
SELECT affiliation, avg( hindex )

FROM `authorlist`

GROUP BY affiliation

order by avg( hindex ) DESC
```

LIMIT 0 , 163

Observation:

- At the time of schema drawing, I considered many to many relationship. So, it is possible to find a single paper with co-authors where all authors are included in my author list table. Then, all author ID will point to the single paper, rather than creating multiple tuples of same paper.
- In google scholar, a single publication is listed with multiple authors even if they are not the author of the paper. The problem is caused for same user name.
- Example: At the figures, we can find two different authors with same name for a single publication.




https://scholar.google.com/citations?user=M_Tvp2gAAAAJ&hl=en
functionalities of titania nanotube arrays
Web Images More... Sign in
Krishna Kant
The Flinders University
Nano materials, microfluidics
Verified email at flinders.edu.au
Follow
Google Scholar
Get my own profile
Citation indices All Since 2012
Citations 425 338
h-index 9 9
i10-index 8 8
2009 2010 2011 2012 2013 2014 2015 2016 2017
Co-authors View all...
Dusan Losic
Craig Priest
Reza Emamali Sabzi
Title 1–18 Cited by Year
Tailoring the surface functionalities of titania nanotube arrays
K Vasilev, Z Poh, K Kant, J Chan, A Michelmore, D Losic
Biomaterials 31 (3), 532-540
140 2010
A simple approach for synthesis of TiO2 nanotubes with through-hole morphology
K Kant, D Losic
physica status solidi (RRL)-Rapid Research Letters 3 (5), 139-141
74 2009
Self-ordering Electrochemistry: A Simple Approach for Engineering Nanopore and Nanotube Arrays for Emerging Applications< a class=
D Losic, L Velleman, K Kant, T Kumeria, K Gulati, JG Shapter, DA Beattie, ...
43 2011
stru
Highlight All Match Case Whole Words 1 of 4 matches
Inspector Console Debugger Style Editor Performance Memory Network
<!DOCTYPE html>
<html class="gs_el_sm gs_el_tc">
<head></head>
<body>
<div id="gs_top"></div>
<div id="gs_rdy"></div>
</body>
</html>
html.gs_el_sm.gs_el_tc body
Rules Computed Animations Fonts
element { }
html, body { height: 100%; }
body { background: #fff; color: #222; -webkit-text-size-adjust: 100%;
Ask me anything
11:47 PM
2/18/2017

Principles of Data Management

Course Code – 5516

Spring 2017, Section: 01

[Project: Phase 3]

Submitted To:

Dr. Eduard C. Dragut
Assistant Professor

Submitted By:

Ashis Kumar Chanda

(TU ID: 915364305)

Description of Experimental Steps:

- At first, two DBMS systems (MYSQL, Oracle) have installed in a same machine to perform different experiments.
- For study 1, we searched for author, selected publication, extracted author names from publication list and saved in a file. Then, we did the exactly same process (phase 1) for the list of authors.
- We also set a counter to track 100,000 records of database.
- To compute the time of our program, a java method (System.currentTimeMillis()) is used that provide the time in milliseconds. Then, we converted the time into minute and second.
- prepareStatement() method is used to avoid SQL injection problems.
- We followed the instruction of our project and keep record of time for various studies. The time of each study is given below, at **Table 1**.
- In oracle database, we found there is an operation limit in a connection. It means if we connect with database then we can run 500 queries and then the connection will close automatically. However, the limit depends on oracle version and we can set the value. Hence, we set the limit with 1000 using the following command:
  ALTER SYSTEM SET open_cursors = 1000 SCOPE=BOTH;
- To run queries in a batch mode, we used preparedStmt.addBatch() method to add 100 queries and then, preparedStmt.executeBatch() is used to execute all 100 queries.

Environment settings:

Operating system: Windows 7

Processor: Intel core i-7, 64 bits, 2.7 GHz

Memory: 8 GB

DBMS 1: MySQL 5 (XAMPP)

DBMS 2: Oracle 11g

Results:

Experiment	**Time in MYSQL Database**	**Time in Oracle Database**
Study 1:	58 min 28 sec	3 min 8 sec
Study 2: (index on author name)	58 min 36 sec	3 min 37 sec
Study 3: (index on author, publication)	59 min 23 sec	3 min 44 sec
Study 4: (batch mode, no index)	53 min 38 sec	11 sec

Table 1: Experimental result

Discussion:

If we look at the two columns of Table 1, then we can find a good comparison. Oracle database is faster than MySQL database. In each study, oracle database required less time than the other. However, we can find a time change in each study.

Suppose, when we use index on author name, there is little change in time. For MySQL and Oracle, both took more time than the study 1. The reason is that indexing strategy provides better performance for searching operation or select query. As there is no 'where' clause in our insert operation, we could not find benefit of indexing.

Moreover, both systems have taken more time in study 3, where we applied indexing on two keys.

Apparently, each database required less time for batch mode. As we added 100 records in a batch and performed execute operation, it became faster than previous studies. It is point to note that oracle also performed better in batch mode. For MySQL, it needed less time than study 1, but the difference is small.

From this study, we can infer that if we use indexing, then we cannot find benefit in inserting operation. However, if we use batch mode, then the performance depends on the database system. In our analysis, oracle database system is better than MySQL.

Appendix: **(**For the reader's convenient, this part is added from Project Phase 2**)**

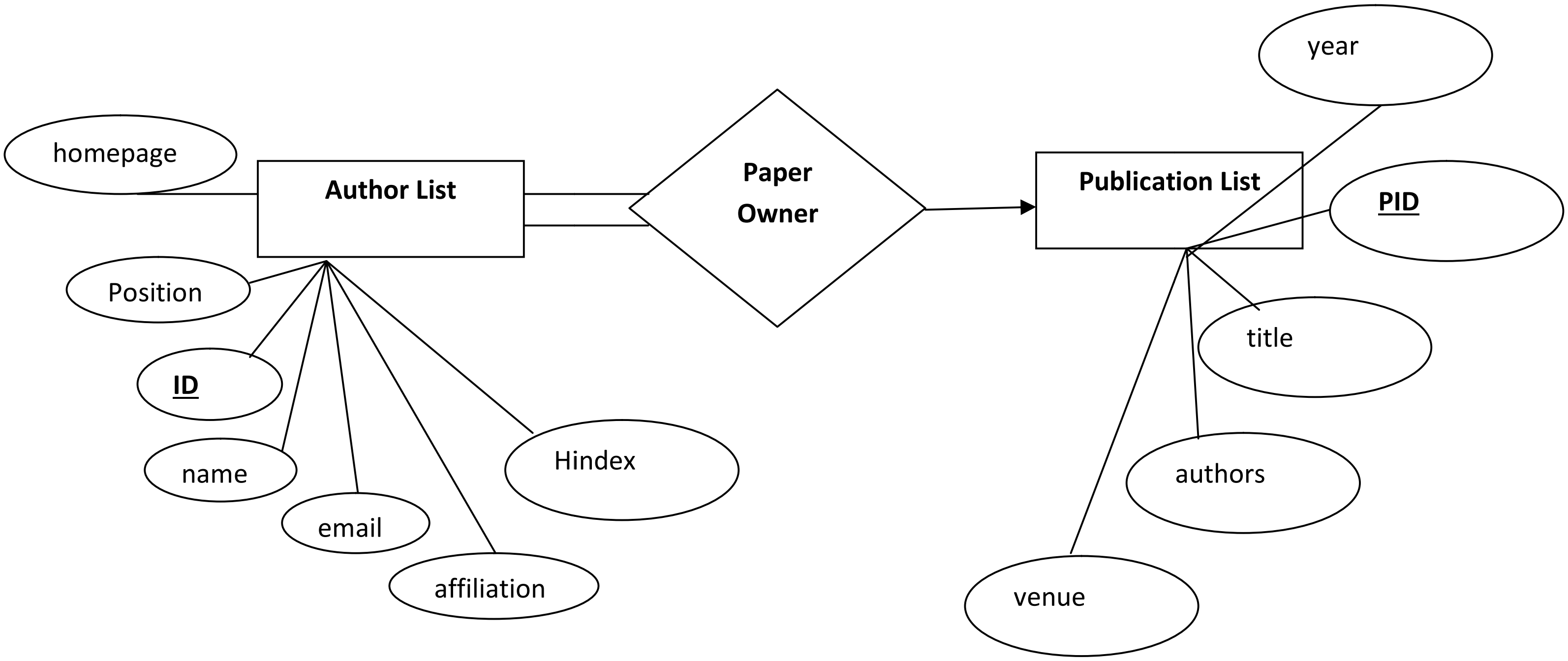


Figure: **ER** diagram of Google Scholar Database

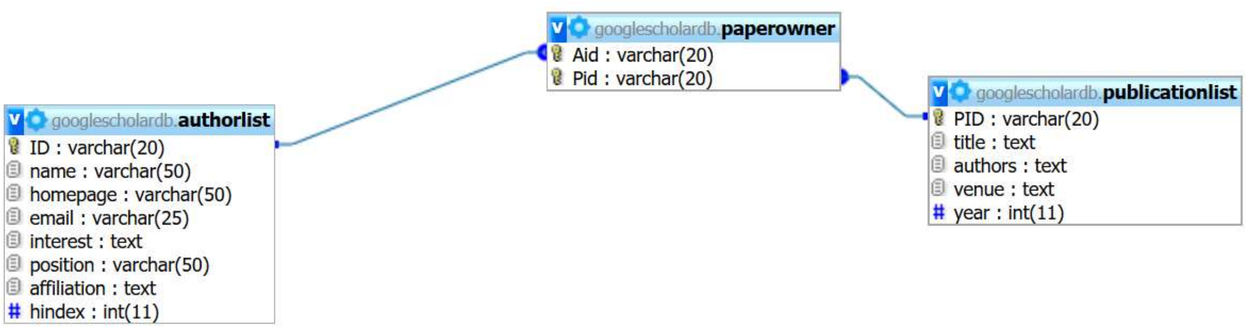


Figure: Table Structure of Google Scholar Database

Principles of Data Management

Course Code – 5516

Spring 2017, Section: 01

[Project: Phase 4]

Submitted To:

Dr. Eduard C. Dragut
Assistant Professor

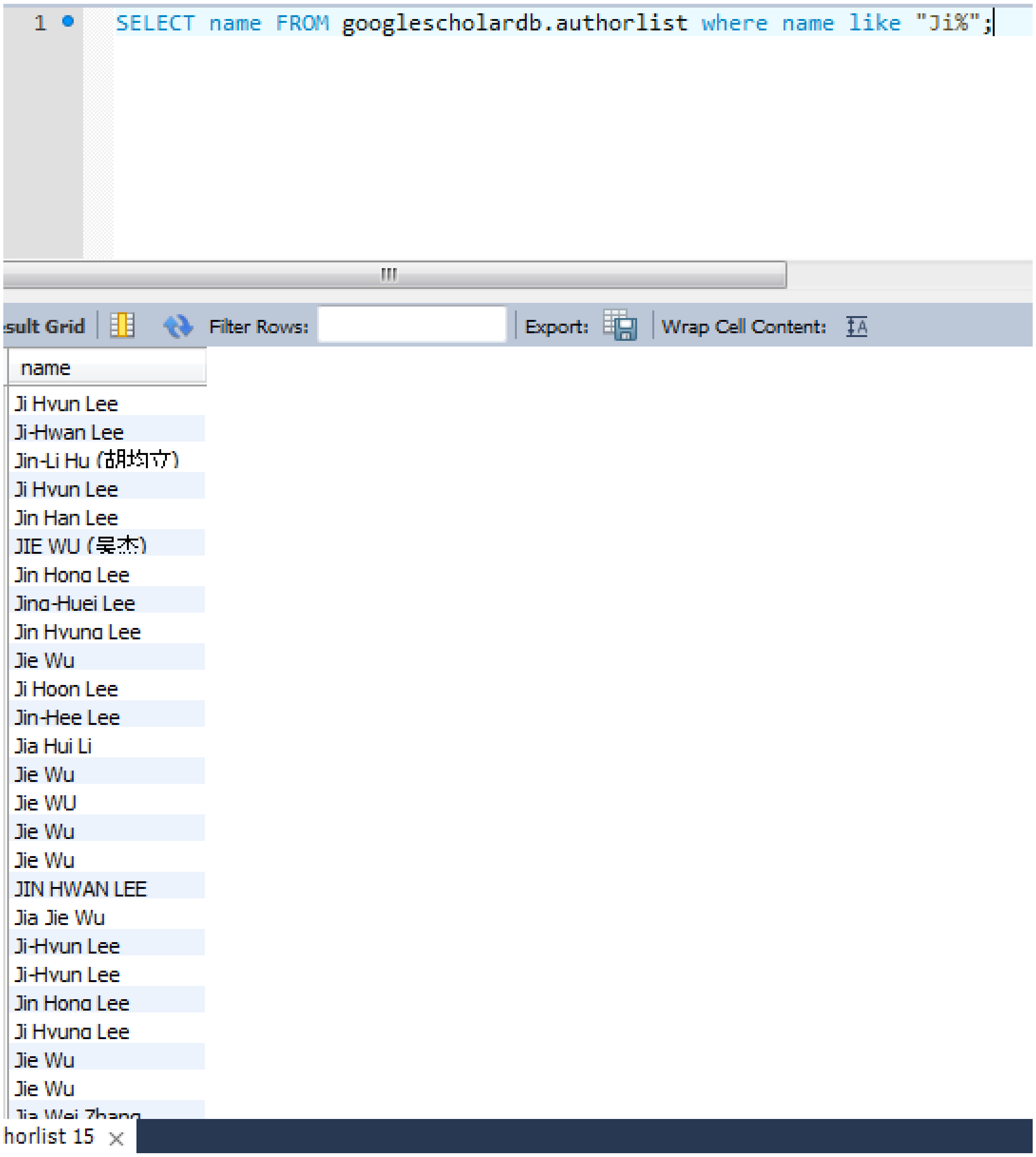
Submitted By:

Ashis Kumar Chanda

(TU ID: 915364305)

1. **Q1:** Find all authors whose names start with a user defined substring.

SELECT name FROM googlescholardb.authorlist where name like "Ji%";



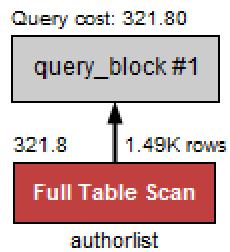


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00319736
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

After applying optimization or indexing:

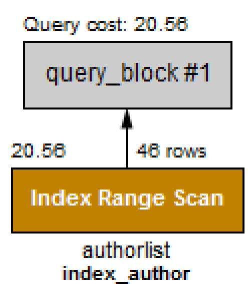


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00041869
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

2. **Q2:** Find all authors who have published in a given venue, say ICDE.

SELECT name FROM googlescholardb.authorlist Natural Join googlescholardb.paperowner Natural Join googlescholardb.publicationlist where venue like "IEEE%";

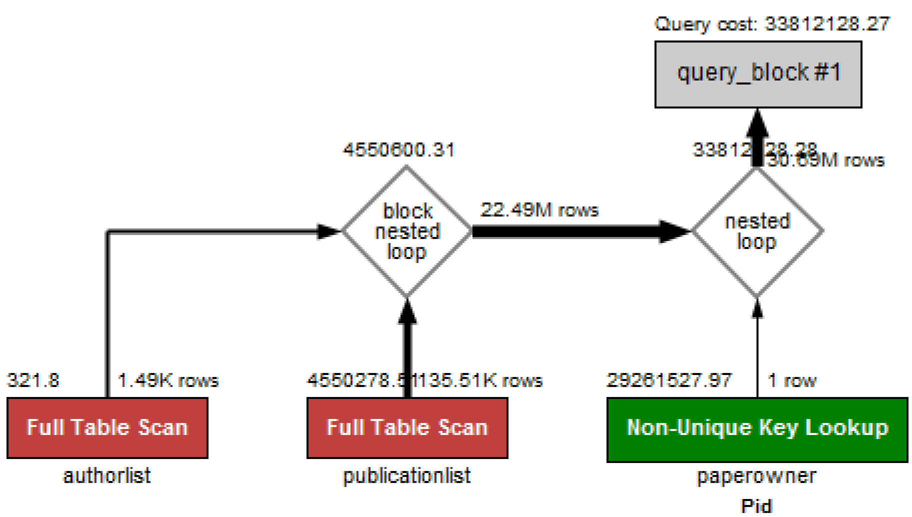


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.56200000

Timing (as measured by the server):
Execution time: 0:00:0.55688153
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

After applying optimization or indexing:

At first, I tried to add indexing on the previous query. But, the result was not good. Thus, I wrote the query in different way, which is given below:

SELECT name FROM googlescholardb.authorlist where ID IN(
SELECT Aid from googlescholardb.paperowner WHERE Pid IN(
SELECT PID from googlescholardb.publicationlist where venue like "IEEE%"))

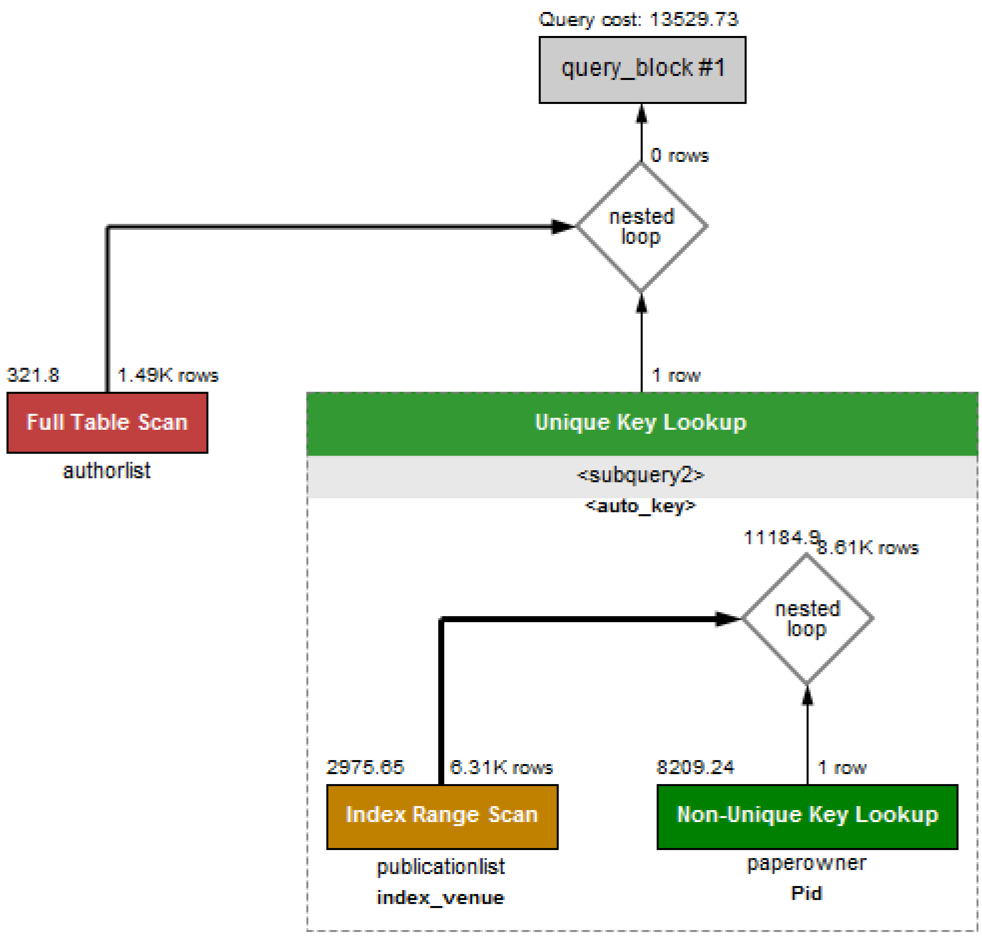


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.01486310
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

3. Q3: Give the average paper count by year. Retain only those years where there are at least 1000 publications.

```
SELECT avg(Rough.paper_count)
FROM (
    Select count(*) as paper_count, year
    from googlescholardb.publicationlist
    group by year
    having count(*) >1000
) as Rough;
```

```
SELECT avg(Rough.paper_count)
FROM (
    Select count(*) as paper_count, year
    from googlescholardb.publicationlist
    group by year
    having count(*) >1000
) as Rough;
```

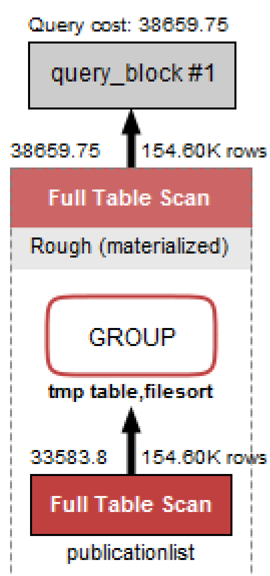


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.17100000

Timing (as measured by the server):
Execution time: 0:00:0.16591860
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

After applying optimization or indexing:

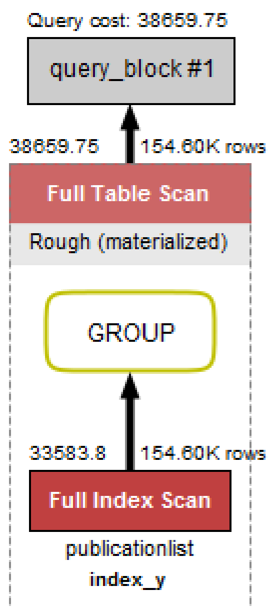


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.07800000

Timing (as measured by the server):
Execution time: 0:00:0.06524622
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

4. **Q4:** Find the minimum and maximum h-index.

```
SELECT max(hindex), min(hindex) FROM googlescholardb.authorlist;
```

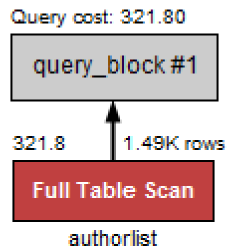


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00089641
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

After applying optimization or indexing:

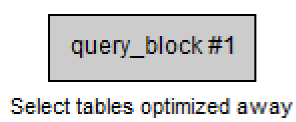


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00022731
Table lock wait time: 0:00:0.00000000

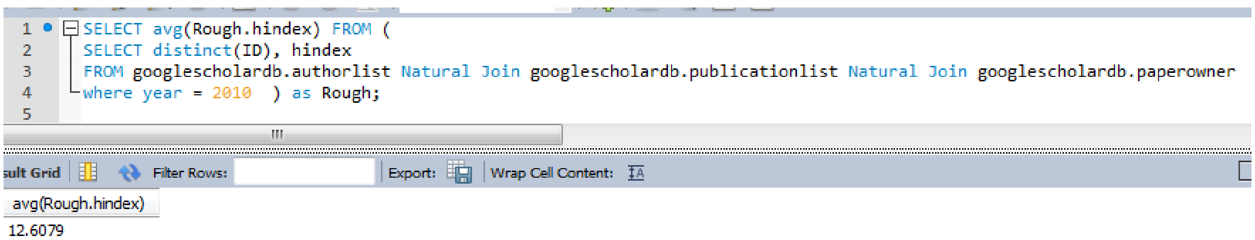
Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

5. **Q5:** Find the average h-index for the authors who have published at least one paper in 2010.

As we need to find authors who have at least one paper in 2010, we do not need to count the papers in 2010. Rather, we can select an author if we find any publication in 2010. In this way, we can improve result.

```
SELECT avg(Rough.hindex) FROM (
        SELECT distinct(ID), hindex
        FROM googlescholardb.authorlist Natural Join googlescholardb.publicationlist Natural Join
        googlescholardb.paperowner
        where year = 2010  ) as Rough;
```



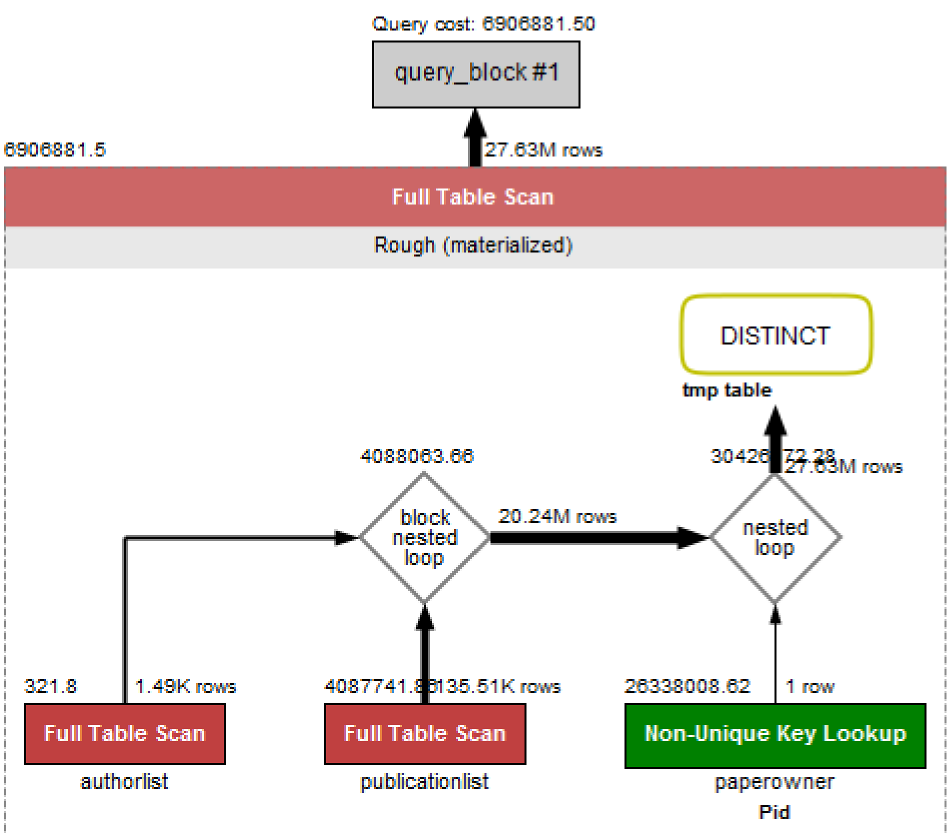


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:01:112.94400000

Timing (as measured by the server):
Execution time: 0:01:112.93599893
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

After applying optimization or indexing:

At first, I tried to add indexing on the previous query. But, the result was not good. Thus, I wrote the query in different way, which is given below:

```
SELECT avg(hindex)
FROM googlescholardb.authorlist WHERE ID IN
(       Select distinct(Aid) from googlescholardb.paperowner WHERE Pid IN
        (SELECT PID
        from googlescholardb.publicationlist
        where year = 2010  ) )
```

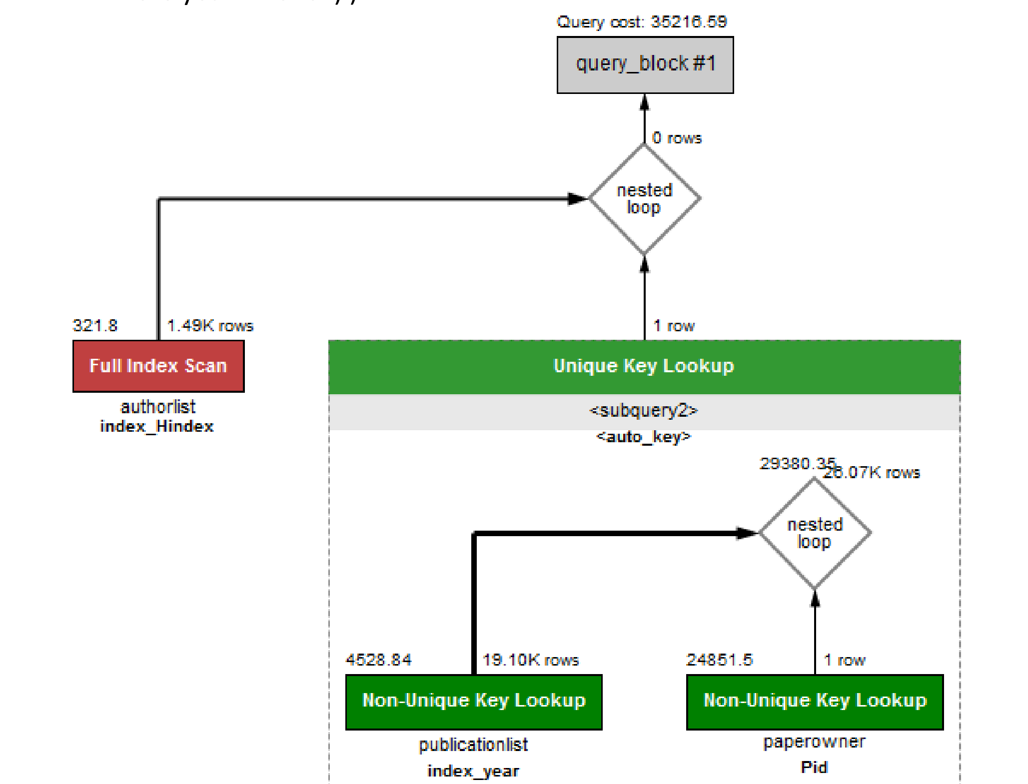


Figure: Parse tree with query cost

Timing (as measured at client side):
Execution time: 0:00:0.29600000

Timing (as measured by the server):
Execution time: 0:00:0.29066048
Table lock wait time: 0:00:0.00000000

Errors:
Had Errors: NO
Warnings: 0

Figure: Execution time

Table:

	Initial execution time	**Best execution time**	**Difference on both time**	**Average execution time**	**Comment**
Query 1	0.0039	0.0004	0.0035	0.0009	As there is a 'where' query, indexing used on author name.
Query 2	0.56	0.016	0.544	0.131	As there is a 'where' query, indexing used on venue. However, there was no big change on performance. So, I rewrite the sql and it became faster. I also used indexing for author ID. It might be helpful in checking equality of ID in two tables.
Query 3	0.171	0.078	0.098	0.081	As there is a 'where' query on year, indexing used on year.
Query 4	0.00089	0.00022	0.00067	0.00045	It is better to use index for finding max and min.
Query 5	1.112	0.296	0.816	0.58	As there is a 'where' query, indexing used on year. However, there was no big change. So, I rewrite the sql and it became faster.

(* Here, time is given in seconds)

Summarization:

From this experiment, we learnt how to get better performance by applying query optimization. Sometimes, we can easily optimize by using index. However, we can find better result by rewriting our query. For example, I wrote the query 2 in different way to avoid joining operation. I first checked the where condition and then, merged two tables for the selected rows. As we checked ‘where’ condition first, it pruned many rows. Hence, we got better result. Even, if we need joining operation, then we would select small table as inner table of the joining operation. It returns good result. The same strategy is used for query 5. Moreover, the parse tree generation helps us to understand what steps are followed in the database system and how a query is executed. It presents a flow of database system.