NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CZ4031: Database System Principles

Project 1 Report: Querying Databases Efficiently

Name	Matriculation Number	Contribution
Aashika Treesa Saju	U1423018H	Schema Design & Data Acquisition; Queries and Optimization; Indexing; Caching; Report
Shantanu Arun Kamath	U1422577F	Schema Design & Data Acquisition; Queries and Optimization; Indexing; Caching; Report
Khare Simran	U1423025D	Schema Design & Data Acquisition; Queries and Optimization; Indexing; Caching; Report
Priyanshu Singh	U1422744C	Schema Design & Data Acquisition; Queries and Optimization; Indexing; Caching; Report

October 16th, 2017

Index

Introduction

In this project we are using PostgreSQL. The database for this project was tested on a machine running Mac OS with the following configuration.

1. Schema Design & Data Acquisition

1.1 Schema Design

There are 3 tables created: *Author, Publication* and *PublicationAuthor*.

Table	Owner	Row Count
Author	Postgres	1988710
Publication	Postgres	3820611
PublicationAuthor	Postgres	11119265

Table 1: Information regarding tables in database

The Table Schema can be found in Appendix A. The ER diagram is as shown in the following page.

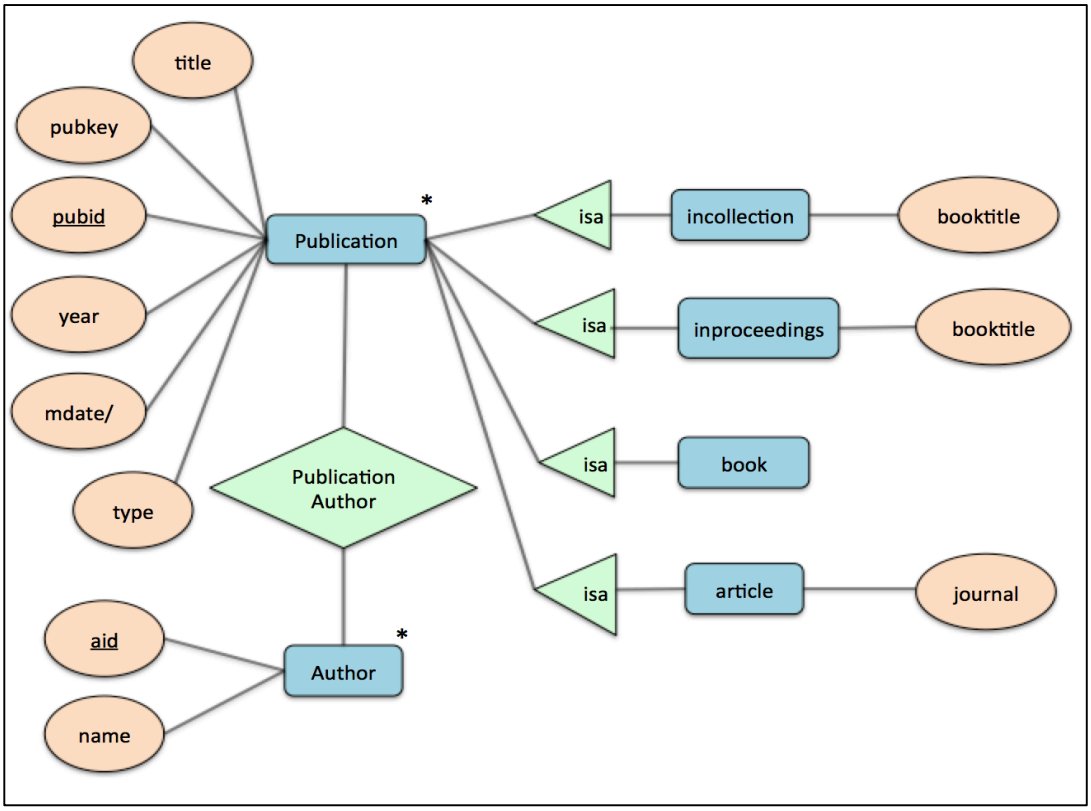

Figure 1: ER Diagram

1.2 Data Acquisition

Python and SAX xml Parser was used to populate the databases from DBLP's file. The SQL files were executed in PostgreSQL so that the data is imported. We extracted the conference name and the parent type of the publication from the publication key and we are using *mdate* to calculate the month of publication.

Using the ER diagram, we created our relation schemas following the NULL approach to deal with *isa* relationships. We parse the publications in the DBLP XML and put them in a CSV file *publications.csv.* Similarly, we extract the author for each publication into a separate file *author.csv.* We copy the data from these files into temporary staging tables *PublicationsCSV* and *AuthorCSV* and use them to populate the final tables. The schemas can be found in Appendix A.

2. Queries

The file *query.sql* is used to run all the given queries and process the output, and to record the execution time.

In Query 3(a), X (author) was set to 'Ursula Goltz'.
In Query 3(b), X (author) was set to 'Peter Mowforth', Y (year) to 1990 and Z (conference) to 'bmvc'
In Query 3(c), Y (year) to 1990 and Z (conference) to 'bmvc'

The custom query is Query 10, which is designed to return the top five most common first names of authors who have published papers in the last two years.

Code snippets of all the queries including their optimized and non-optimized versions are included in Appendix B. In general, we observed that queries are faster when we avoid multiple sub-queries and instead perform table joins.

Effect of Database Size on Query Time

We can observe from the values below that the queries' running time decreases with the decrease in the size of the database.

Query	Full Size (ms)	Half Size (ms)	Quarter Size (ms)
1	6147.494	3595.239	1518.427
2(a)	1739.136	433.025	212.302
3(a)	2155.030	2731.326	772.219
3(b)	1580.799	1631.829	729.059
3(c)	1752.829	1390.090	161.147
4(a)	2371.616	1139.750	188.715
4(b)	3488.864	1206.488	627.943
5	7256.297	5471.023	1797.104
6	65296.033	34279.392	15319.209
7	8804.659	4276.846	1807.463
8	827.198	438.141	217.017
9(a)	9380.156	5558.421	1895.229
9(b)	8160.867	6276.421	3398.859
10	103249.539	11325.331	4057.391

Table 2: Effect of change in database size

When we compared the decrease in running time from full size to half size and half size to quarter size, we found that it doesn't change the same percentage that the size of the database changes. This can be shown in the following table.

	Half to Full	Quarter to Half
1	58.48300137	42.23438275
2(a)	24.89885782	49.02765429
3(a)	126.7419015	28.27267781
3(b)	103.2281144	44.67741412
3(c)	79.30551126	11.59255876
4(a)	48.05794867	16.55757842
4(b)	34.58111294	52.04718157
5	75.3969001	32.84767766
6	52.49842973	44.68926695
7	48.57480568	42.26158716
8	52.96688338	49.53131526
9(a)	59.2572341	34.09653569
9(b)	76.90875246	54.15282053
10	10.96889256	35.825805

Table 3: Percentage of change in database size

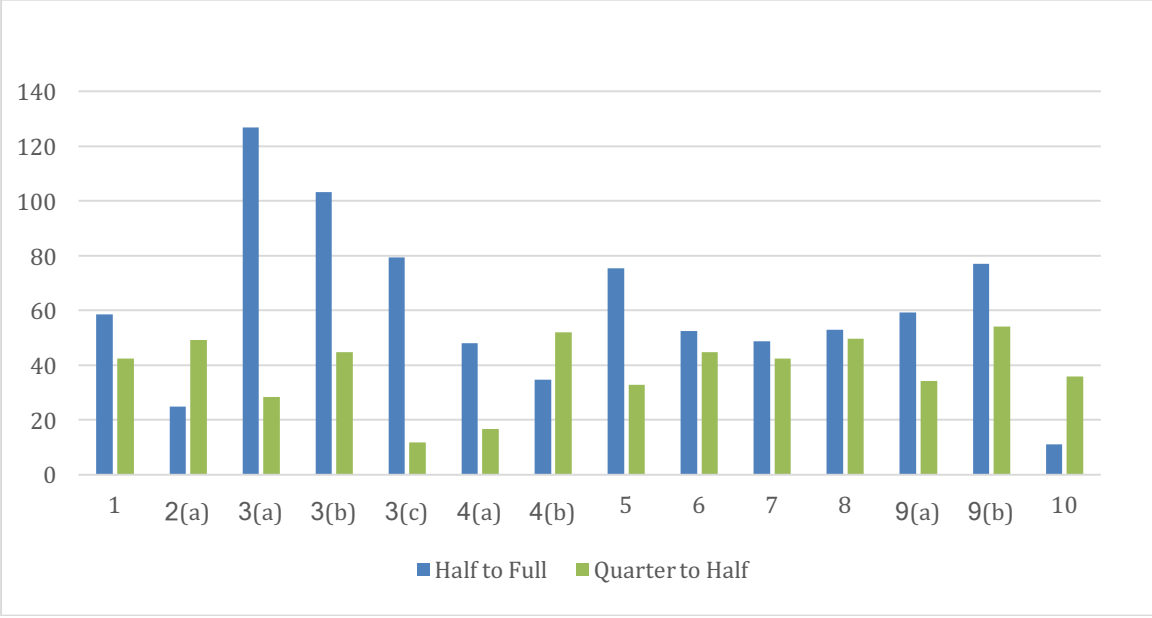

Fig2: Percentage of change in time with respect to database size

3. Build Index and Effect of Index

PostgreSQL creates an index automatically for any attribute that is declared as a PRIMARY KEY in a particular relation. The PRIMARY KEY constraint will become invalid if we remove these indexes. The automatic indexes are:

- Author (author_id)
- Publication (publication_id)

Apart from these, we added other indexes in order to speed up the queries. These indexes include:

- Publication (year)
- Publication (parent_type)
- Publication (category)
- Publication (conf_name)
- Author (name)

These attributes are used WHERE and/or JOIN statements with ORDER BY, GROUP BY, range or equality.

There are several index methods available (hash, btree, gist etc). For this project, we decided to use btree, which is it default method in PostgreSQL. We chose btree as it can be used for both range and value based queries, which we require. Btrees are more useful for the purpose of range and inequality queries compared to other types of indexes.

Effect of Indexes:

Below we have shown our attempts to incorporate various indexes in our database. We have logged our running times with and without these indexes along with our reasoning for the selection of specific indexes by the PSQL optimizer and their effect on the performance.

We have used the EXPLAIN statement to generate the query plan for each query that provides a step-by-step view on the execution of that query. Using this query plan we can determine how and where indexes were used if used at all.

Query 1:

Runtime without index: **6147.494 ms.**

Index 1:
```
CREATE INDEX PubCategoryIndex ON Publication(category);
```
Runtime: **6046.094 ms.**

Index 2:
```
CREATE INDEX PubYearIndex ON Publication(year);
```
Runtime: **6259.863 ms.**

Query Plan:

HashAggregate (cost=99267.56..99267.60 rows=4 width=19)
Group Key: category
-> Bitmap Heap Scan on publication (cost=3192.77..98516.17 rows=150277 width=11)
Recheck Cond: ((year >= '2000'::text) AND (year <= '2001'::text))
-> Bitmap Index Scan on pubyearindex (cost=0.00..3155.20 rows=150277 width=0)
Index Cond: ((year >= '2000'::text) AND (year <= '2001'::text))

Conclusion: If a SELECT query returns more than approximately 5-10% of all the rows in a table, a sequential scan is much faster than an index scan as can be seen from the runtimes above.

Query 2:

Runtime without index: **1739.136 ms**.

Due to the same reasoning as Query 1, no attribute was found suitable to be used as the index.

Query 3(a):

Runtime without index**: 2155.030 ms.**

Index:
CREATE INDEX PubYearIndex ON Publication(year);
Runtime: **2164.342 ms.**

Query Plan:

Nested Loop (cost=188.00..189794.37 rows=4 width=169)
 -> Hash Semi Join (cost=187.57..189768.57 rows=52 width=4)
Hash Cond: (ap.author_id = author.author_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=187.56..187.56 rows=1 width=4)
-> Limit (cost=0.00..187.55 rows=1 width=4)
-> Seq Scan on author (cost=0.00..37322.88 rows=199 width=4)
Filter: (name ~~* '%Ursula Goltz%'::text)
-> Index Scan using publication_pkey on publication p (cost=0.43..0.49 rows=1 width=169)
Index Cond: (publication_id = ap.publication_id)
Filter: (year = '2015'::text)

Conclusion: According to the Query Plan, the PSQL optimizer found the PRIMARY KEY – Publication (publication_id) to be more suitable during the execution, due to the JOIN clauses.

Query 3(b):

Runtime without index**: 1580.799 ms.**

Index:
CREATE INDEX PubConfnameIndex ON Publication(conf_name);
Runtime: **1589.302 ms.**

Query Plan:

Nested Loop (cost=188.00..189794.50 rows=1 width=169)
-> Hash Semi Join (cost=187.57..189768.57 rows=52 width=4)
Hash Cond: (ap.author_id = author.author_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=187.56..187.56 rows=1 width=4)
-> Limit (cost=0.00..187.55 rows=1 width=4)
-> Seq Scan on author (cost=0.00..37322.88 rows=199 width=4)
-Filter: (name ~~* '%Peter Mowforth%'::text)
-> Index Scan using publication_pkey on publication p (cost=0.43..0.49 rows=1 width=169)
Index Cond: (publication_id = ap.publication_id)
Filter: ((year = '1990'::text) AND (conf_name = 'bmvc'::text))

Conclusion: According to the Query Plan, the PSQL optimizer found the PRIMARY KEY – Publication (publication_id) to be more suitable during the execution, due to the JOIN clauses.

Query 3(c):

Runtime without index**: 1752.829 ms.**

Index:
CREATE INDEX PubYearIndex ON Publication(year);
Runtime: **1210.786 ms.**

CREATE INDEX PubConfnameIndex ON Publication(conf_name);
Runtime: **1152.320 ms.**

With year, conf_name, runtime: **1161.550 ms.**

Query Plan:

GroupAggregate (cost=189727.48..189727.59 rows=6 width=15)
Group Key: a.name
Filter: (count(*) > 1)
-> Sort (cost=189727.48..189727.50 rows=6 width=15)

Sort Key: a.name
-> Nested Loop (cost=144.58..189727.41 rows=6 width=15)
-> Hash Semi Join (cost=144.15..189724.64 rows=6 width=4)
Hash Cond: (ap.publication_id = p.publication_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=144.14..144.14 rows=1 width=4)
-> Bitmap Heap Scan on publication p (cost=140.12..144.14 rows=1 width=4)
Recheck Cond: ((conf_name = 'bmvc'::text) AND (year = '1990'::text))
-> BitmapAnd (cost=140.12..140.12 rows=1 width=0)
-> Bitmap Index Scan on pubconfnameindex (cost=0.00..18.25 rows=776 width=0)
Index Cond: (conf_name = 'bmvc'::text)
-> Bitmap Index Scan on pubyearindex (cost=0.00..121.62 rows=6559 width=0)
Index Cond: (year = '1990'::text)
-> Index Scan using author_pkey on author a (cost=0.43..0.45 rows=1 width=19)
Index Cond: (author_id = ap.author_id)

Conclusion: The optimizer could use both indexes on *year* and *conf_name*, but the performance was better while using the *conf_name* index.

Query 4(a):

Runtime without index**: 2327.277ms.**

Index:
CREATE INDEX AuthorIdIndex ON Author(author_id);
CREATE INDEX PubConfnameIndex ON Publication(conf_name);
Runtime: **1629.400 ms**

Query Plan:

GroupAggregate (cost=225528.30..295791.99 rows=20422 width=19)
Group Key: a.author_id
Filter: (count(*) = 2)
-> Merge Join (cost=225528.30..295485.66 rows=20422 width=19)
Merge Cond: (ap.author_id = a.author_id)
-> GroupAggregate (cost=225527.87..225936.31 rows=20422 width=17)
Group Key: ap.author_id, p.conf_name
Filter: (count(*) >= 10)
-> Sort (cost=225527.87..225578.93 rows=20422 width=9)
Sort Key: ap.author_id, p.conf_name
-> Hash Join (cost=21772.18..224065.88 rows=20422 width=9)

Hash Cond: (ap.publication_id = p.publication_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=21684.47..21684.47 rows=7017 width=9)
-> Bitmap Heap Scan on publication p (cost=135.23..21684.47 rows=7017 width=9)
Recheck Cond: (conf_name = ANY ('{pvldb,sigmod}'::text[]))
-> Bitmap Index Scan on pubconfnameindex (cost=0.00..133.48 rows=7017 width=0)
Index Cond: (conf_name = ANY ('{pvldb,sigmod}'::text[]))
-> Index Scan using author_pkey on author a (cost=0.43..64118.08 rows=1988710 width=19)

Conclusion: We found that the optimizer was able to use both *author_id* and *conf_name* indexes to get a considerable performance gain.

Query 4(b):

Runtime without index**: 3488.864 ms.**

Index:
CREATE INDEX PubConfnameIndex ON Publication(conf_name);
Runtime: **2470.448 ms**

Query Plan:

Nested Loop (cost=205153.04..429345.82 rows=994355 width=15)
-> Subquery Scan on "ANY_subquery" (cost=205152.62..410435.07 rows=2258 width=4)
-> HashSetOp Except (cost=205152.62..410412.49 rows=2258 width=8)
-> Append (cost=205152.62..410401.20 rows=4516 width=8)
-> Subquery Scan on "*SELECT* 1" (cost=205152.62..205214.71 rows=2258 width=8)
-> GroupAggregate (cost=205152.62..205192.13 rows=2258 width=4)
Group Key: ap.author_id
Filter: (count(*) >= 15)
-> Sort (cost=205152.62..205158.26 rows=2258 width=4)
Sort Key: ap.author_id
-> Hash Join (cost=2914.78..205026.84 rows=2258 width=4)
Hash Cond: (ap.publication_id = p.publication_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=2905.08..2905.08 rows=776 width=4)
-> Bitmap Heap Scan on publication p (cost=18.44..2905.08 rows=776 width=4)
Recheck Cond: (conf_name = 'pvldb'::text)
-> Bitmap Index Scan on conf (cost=0.00..18.25 rows=776 width=0)
Index Cond: (conf_name = 'pvldb'::text)
-> Subquery Scan on "*SELECT* 2" (cost=205152.62..205186.49 rows=2258 width=8)
-> Group (cost=205152.62..205163.91 rows=2258 width=4)
Group Key: ap_1.author_id

-> Sort (cost=205152.62..205158.26 rows=2258 width=4)
Sort Key: ap_1.author_id
-> Hash Join (cost=2914.78..205026.84 rows=2258 width=4)
Hash Cond: (ap_1.publication_id = p_1.publication_id)
-> Seq Scan on publicationauthor ap_1 (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=2905.08..2905.08 rows=776 width=4)
-> Bitmap Heap Scan on publication p_1 (cost=18.44..2905.08 rows=776 width=4)
Recheck Cond: (conf_name = 'kdd'::text)
-> Bitmap Index Scan on conf (cost=0.00..18.25 rows=776 width=0)
Index Cond: (conf_name = 'kdd'::text)
-> Index Scan using author_pkey on author a (cost=0.43..8.37 rows=1 width=19)
Index Cond: (author_id = "ANY_subquery".author_id)

Conclusion: We found that the optimizer was able to use the *conf_name* index to get a significant performance gain.

Query 5:

Runtime without index**: 7256.297 ms.**

Index:
CREATE INDEX PubSubYearIndex ON Publication(substring(year, 1, 3));
Runtime: **7256.305 ms**

Query Plan:

HashAggregate (cost=175738.08..175738.93 rows=68 width=40)
Group Key: "substring"(year, 1, 3)
-> Seq Scan on publication (cost=0.00..156702.14 rows=3807189 width=32)
Filter: ((year >= '1970'::text) AND (year <= '2019'::text))

Conclusion: The indexes made using substrings made no difference to the runtime.

Query 6:

Runtime without indexes: **65296.033** ms

Not much difference was made to the runtime using indexes

Query Plan:

Limit (cost=27579842.08..27579842.11 rows=10 width=23)
-> Sort (cost=27579842.08..27579859.78 rows=7078 width=23)
Sort Key: q2.collaborators_count DESC
-> Hash Join (cost=26689420.81..27579689.13 rows=7078 width=23)
Hash Cond: (q2.author_id = author.author_id)
-> Subquery Scan on q2 (cost=26278099.46..27165030.38 rows=212964 width=12)
-> GroupAggregate (cost=26278099.46..27162900.74 rows=212964 width=12)
Group Key: pa1.author_id
-> Sort (cost=26278099.46..26572323.34 rows=117689552 width=8)
Sort Key: pa1.author_id
-> Merge Join (cost=3531486.87..5674460.66 rows=117689552 width=8)
Merge Cond: (pa1.publication_id = pa2.publication_id)
Join Filter: (pa1.author_id <> pa2.author_id)
-> Sort (cost=1765743.43..1793541.34 rows=11119162 width=8)
Sort Key: pa1.publication_id
-> Seq Scan on publicationauthor pa1 (cost=0.00..160392.62 rows=11119162 width=8)
-> Materialize (cost=1765743.43..1821339.24 rows=11119162 width=8)
-> Sort (cost=1765743.43..1793541.34 rows=11119162 width=8)
Sort Key: pa2.publication_id
-> Seq Scan on publicationauthor pa2 (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=410107.12..410107.12 rows=66099 width=23)
-> Hash Semi Join (cost=370147.82..410107.12 rows=66099 width=23)
Hash Cond: (author.author_id = ap.author_id)
-> Seq Scan on author (cost=0.00..32351.10 rows=1988710 width=19)
-> Hash (cost=369321.59..369321.59 rows=66099 width=4)
-> Hash Join (cost=166571.12..369321.59 rows=66099 width=4)
Hash Cond: (ap.publication_id = p.publication_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=166287.22..166287.22 rows=22712 width=4)
-> Seq Scan on publication p (cost=0.00..166287.22 rows=22712 width=4)
Filter: (((parent_type = 'journals'::text) OR (parent_type = 'conf'::text)) AND (lower(title) ~~ '%data%'::text))

Conclusion: The SQL optimizer that the primary keys (*pub_id, author_id)* were more suitable than any of the custom indexes as seen in the JOIN clause.

Query 7:

Runtime without indexes: **8804.659** ms

Index
CREATE INDEX PubTypeIndex ON Publication(parent_type);
Runtime: **8804.650 ms**

CREATE INDEX PubYearIndex ON Publication(year);
Runtime: **8808.159** ms

Query Plan

Limit (cost=360790.60..360790.62 rows=10 width=27)
-> Sort (cost=360790.60..360843.64 rows=21216 width=27)
Sort Key: (count(*)) DESC
-> GroupAggregate (cost=359960.85..360332.13 rows=21216 width=27)
Group Key: a.author_id
-> Sort (cost=359960.85..360013.89 rows=21216 width=19)
Sort Key: a.author_id
-> Nested Loop (cost=146344.92..358436.17 rows=21216 width=19)
-> Hash Join (cost=146344.49..348646.13 rows=21216 width=4)
Hash Cond: (pa.publication_id = p.publication_id)
-> Seq Scan on publicationauthor pa (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=146253.37..146253.37 rows=7290 width=4)
-> Bitmap Heap Scan on publication p (cost=25721.14..146253.37 rows=7290 width=4)
Recheck Cond: ((year >= '2013'::text) AND (year <= '2017'::text))
Filter: (((parent_type = 'journals'::text) OR (parent_type = 'conf'::text)) AND (lower(title) ~~ '%data%'::text))
-> Bitmap Index Scan on year (cost=0.00..25719.32 rows=1226289 width=0)
Index Cond: ((year >= '2013'::text) AND (year <= '2017'::text))
-> Index Scan using author_pkey on author a (cost=0.43..0.45 rows=1 width=19)
Index Cond: (author_id = pa.author_id)
Conclusion: We created an index on the attributes *parent_type* and *year* but found no significant performance gain.

Query 8:

Runtime without indexes: **827.198** ms

Index:
CREATE INDEX PubCategoryIndex ON Publication(category);
Runtime: **696.109 ms.**

Query Plan:

HashAggregate (cost=127276.90..127613.31 rows=33641 width=10)

Group Key: conf_name
-> HashAggregate (cost=126856.39..127192.80 rows=33641 width=10)
Group Key: conf_name, year
Filter: (count(*) > 100)
-> Index Scan using cat on publication (cost=0.43..126586.19 rows=36026 width=10)
Index Cond: (category = 'inproceedings'::text)
Filter: ((mdate ~~ '%-07-%'::text) AND (parent_type = 'conf'::text))

Conclusion: We created an index on the attribute *category* and found a considerable gain in performance.

Query 9(a):

Runtime without indexes: **9380.156** ms

Index
CREATE INDEX PubIdIndex ON Publication(publication_id);
Runtime: **9241.23 ms**

Query Plan:

GroupAggregate (cost=306830.40..307903.30 rows=53645 width=19)
Group Key: ap.author_id, a.name
Filter: (count(DISTINCT p.year) = 30)
-> Sort (cost=306830.40..306964.51 rows=53645 width=24)
Sort Key: ap.author_id, a.name
-> Nested Loop (cost=72250.03..302616.27 rows=53645 width=24)
-> Hash Join (cost=72249.60..274895.06 rows=55598 width=23)
Hash Cond: (ap.author_id = a.author_id)
-> Seq Scan on publicationauthor ap (cost=0.00..160392.62 rows=11119162 width=8)
-> Hash (cost=72125.30..72125.30 rows=9944 width=19)
-> Seq Scan on author a (cost=0.00..72125.30 rows=9944 width=19)
Filter: ("substring"(name, ((length(name) - strpos(reverse(name), ' '::text)) + 2), length(name)) ~~ 'H%'::text)
-> Index Scan using publication_pkey on publication p (cost=0.43..0.49 rows=1 width=9)
Index Cond: (publication_id = ap.publication_id)
Filter: ((year >= '1988'::text) AND (year <= '2017'::text))

Conclusion: There was no difference in the runtime when year and author were used as indexes. Hence only used the existing Publication PRIMARY KEY.

Query 9(b):

Runtime without indexes**: 8160.867** ms

Index:
CREATE INDEX PubYearIndex ON Publication(year);
Runtime: **4519.729** ms

Query Plan:

```
GroupAggregate  (cost=784984.99..800984.21 rows=914241 width=27)
Group Key: a.author_id
 ->  Sort  (cost=784984.99..787270.59 rows=914241 width=19)
Sort Key: a.author_id
 ->  Hash Join  (cost=376654.62..675715.51 rows=914241 width=19)
 Hash Cond: (ap.author_id = a.author_id)
 ->  Seq Scan on publicationauthor ap  (cost=0.00..160392.62 rows=11119162 width=4)
 ->  Hash  (cost=373651.67..373651.67 rows=163516 width=23)
 ->  Hash Semi Join  (cost=310316.09..373651.67 rows=163516 width=23)
 Hash Cond: (a.author_id = ap_1.author_id)
 ->  Seq Scan on author a  (cost=0.00..32351.10 rows=1988710 width=19)
 ->  Hash  (cost=307633.14..307633.14 rows=163516 width=4)
 ->  Unique  (cost=305180.40..305997.98 rows=163516 width=4)
   InitPlan 2 (returns $1)
 ->  Result  (cost=0.55..0.56 rows=1 width=32)
    InitPlan 1 (returns $0)
 ->  Limit  (cost=0.43..0.55 rows=1 width=5)
 ->  Index Only Scan using year on publication  (cost=0.43..467969.26 rows=3820611 width=5)
    Index Cond: (year IS NOT NULL)
 ->  Sort  (cost=305179.84..305588.63 rows=163516 width=4)
  Sort Key: ap_1.author_id
 ->  Hash Join  (cost=85058.97..288783.61 rows=163516 width=4)
   Hash Cond: (ap_1.publication_id = p.publication_id)
   ->  Seq Scan on publicationauthor ap_1  (cost=0.00..160392.62 rows=11119162 width=8)
 ->  Hash  (cost=84356.66..84356.66 rows=56185 width=4)
   ->  Bitmap Heap Scan on publication p  (cost=1055.86..84356.66 rows=56185 width=4)
 Recheck Cond: (year = $1)
->  Bitmap Index Scan on year  (cost=0.00..1041.82 rows=56185 width=0)
 Index Cond: (year = $1)
```

Conclusion: We created an index on the attribute *year* and found almost a 2x gain in performance. This was due to the fact that the index created has the structure of btree which is suitable for equality/inequality queries.

Query 10

Runtime without indexes: **24511.882** ms

Index:
CREATE INDEX PubYearIndex ON Publication(year);
Runtime: **20040.7**72 ms

Query Plan:

```
Limit  (cost=774034.65..774034.67 rows=5 width=40)
->  Sort  (cost=774034.65..777301.83 rows=1306872 width=40)
 Sort Key: (count(*)) DESC
->  GroupAggregate  (cost=726190.54..752327.98 rows=1306872  width=40)
 Group Key: (split_part(a.name, ' '::text, 1))
 ->  Sort  (cost=726190.54..729457.72 rows=1306872 width=32)
Sort Key: (split_part(a.name, ' '::text, 1))
 ->  Hash Join  (cost=182373.00..530889.46 rows=1306872 width=32)
  Hash Cond: (ap.author_id = a.author_id)
->  Hash Join  (cost=113510.03..417293.23 rows=1306872 width=4)
  Hash Cond: (ap.publication_id = p.publication_id)
->  Seq Scan on publicationauthor ap  (cost=0.00..160392.62 rows=11119162 width=8)
->  Hash  (cost=106141.92..106141.92 rows=449049 width=4)
->  Bitmap Heap Scan on publication p  (cost=9531.18..106141.92 rows=449049 width=4)
  Recheck Cond: ((year >= '2016'::text) AND (year <= '2017'::text))
->  Bitmap Index Scan on year  (cost=0.00..9418.92 rows=449049 width=0)
 Index Cond: ((year >= '2016'::text) AND (year <= '2017'::text))
->  Hash  (cost=32351.10..32351.10 rows=1988710 width=19)
```

Conclusion: We created an index on the attribute *year* and found significant gain in performance. This was because the index created has the structure of btree which is suitable for range queries.

4. Advanced Part: Effect of Cache

We used the shared buffer attribute in the Postgres configuration settings to define the size of the cache. According to the Postgres documentation, the Postgres cache relies on the OS cache and is significantly slower in comparison. Even after doubling the shared buffer, we could not see a considerable gain in performance. This was because the Postgres cache increments the shared buffer by 32 cache blocks (each being 8 kilobytes) resulting in only additional 32 bits on every run as illustrated in the query plan in the example below.

```
db4031=# EXPLAIN (analyze,buffers) SELECT category, COUNT(*) count
db4031-# FROM Publication
db4031-# WHERE year BETWEEN '2000' AND '2017'
db4031-# GROUP BY category;
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=163411.57..163411.61 rows=4 width=19) (actual time=6046.907..6046.908 rows=4
loops=1)
   Group Key: category
   Buffers: shared hit=64 read=89811
         ……
         ……ignored extra……
         ……
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=163411.57..163411.61 rows=4 width=19) (actual time=5997.291..5997.292 rows=4
loops=1)
   Group Key: category
   Buffers: shared hit=96 read=89779
   ->  Seq Scan on publication  (cost=0.00..147184.17 rows=3245482 width=11) (actual
time=0.053..5264.555 rows=3243428 loops=1)
         ……
         ……ignored extra……
         ……
 Execution time: 5997.332 ms
(9 rows)
```

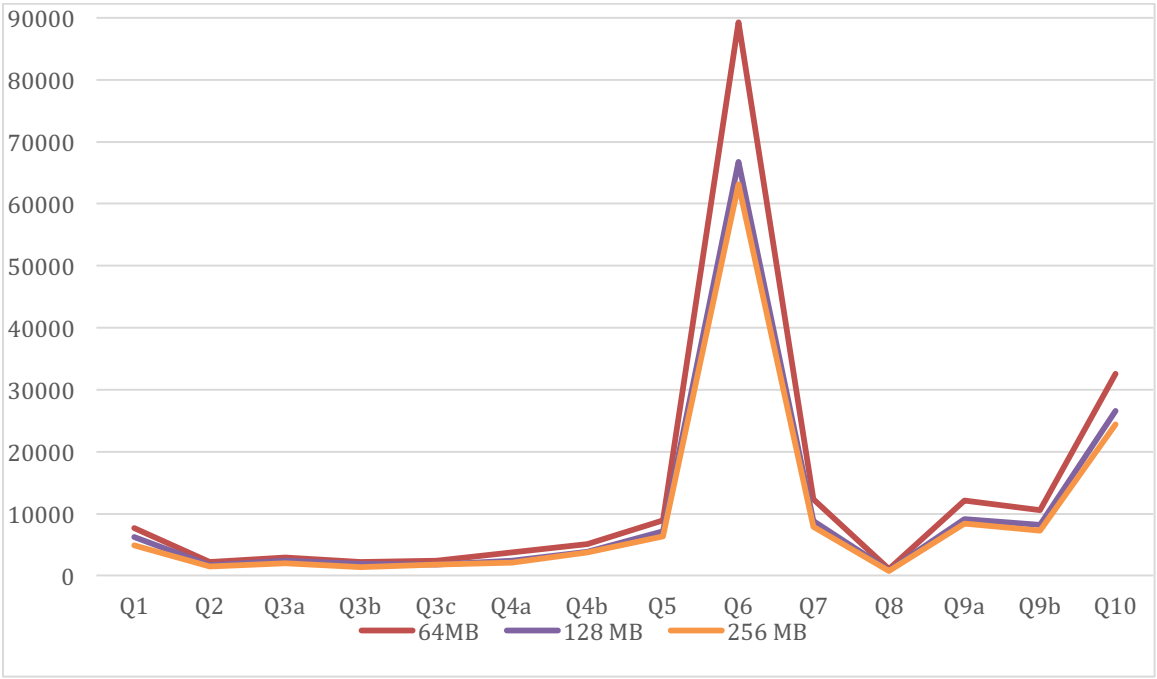

Fig 3: Query time for different cache sizes

	64MB	128 MB	256 MB
Q1	7641.8485	6249.313	4939.8885
Q2	2190.2309	1712.598	1510.7634
Q3a	2908.867	2375.848	2013.831
Q3b	2198.1505	1904.835	1397.6145
Q3c	2385.7626	1790.029	1831.7556
Q4a	3764.1201	2426.006	2073.5091
Q4b	5096.3692	3827.355	3762.4032
Q5	8895.9338	7193.291	6348.1554
Q6	89245.9386	66716.921	63143.3367
Q7	12364.0049	8773.529	7850.5551
Q8	1060.7012	840.897	751.0221
Q9a	12080.8805	9086.304	8407.9125
Q9b	10621.4381	8224.788	7277.8914
Q10	32553.8915	26601.173	24372.5787

Conclusion: Shown above are the results from 4 consecutive query executions to ensure that Postgres is able to cache at least some of the query results. We couldn't observe any material difference in the query performance by changing the cache size. According to the QUERY PLANS most of our queries use sequential scans and as stated above it was limited to an increase of only 32 blocks per query execution.

Appendix A: Table Schema

1. Publication Table

```
DROP TABLE IF EXISTS Publication;

CREATE TABLE Publication (
        publication_id SERIAL PRIMARY KEY,
        category TEXT NOT NULL,
        key TEXT NOT NULL,
        conf_name TEXT NOT NULL,
        parent_type TEXT NOT NULL,
        mdate TEXT NOT NULL,
        title TEXT,
        year TEXT,
        journal TEXT,
        month TEXT
);
```

2. Author Table

```
DROP TABLE IF EXISTS Author;
CREATE TABLE Author (
        author_id SERIAL PRIMARY KEY,
        name TEXT NOT NULL
);
```

3. PublicationAuthor Table

```
DROP TABLE IF EXISTS PublicationAuthor;
CREATE TABLE PublicationAuthor (
        publication_id INT NOT NULL,
        author_id INT NOT NULL
);
```

Temporary Staging Tables

```
    DROP TABLE IF EXISTS AuthorCSV;
    CREATE TEMP TABLE AuthorCSV (
            publication_key TEXT NOT NULL,
            author_name TEXT NOT NULL
    );
    COPY AuthorCSV FROM :path_to_author_csv CSV;

DROP TABLE IF EXISTS PublicationCSV;
CREATE TABLE PublicationCSV (
        category TEXT NOT NULL,
        key TEXT NOT NULL,
        mdate TEXT NOT NULL,
        publtype TEXT,
        reviewid TEXT,
        rating TEXT,
        title TEXT,
        booktitle TEXT,
        pages TEXT,
        year TEXT,
        address TEXT,
        journal TEXT,
        volume TEXT,
        number TEXT,
        month TEXT,
        publisher_id INT,
        school TEXT,
        chapter INT
);
```

Appendix B: Queries

Query 1

```
SELECT category, COUNT(*) count
FROM Publication
WHERE year BETWEEN '2000' AND '2017'
GROUP BY category;
```

Query 2

```
SELECT DISTINCT conf_name
FROM (
        SELECT conf_name, year, COUNT(*) as conf_count
        FROM Publication
        WHERE mdate LIKE '%-07-%' AND parent_type = 'conf'
        -- Assuming mdate is in the format '2017-07-14'
        GROUP BY conf_name, year) temp
WHERE temp.conf_count > 200;
```

Query 3(a)

```
SELECT P.* -- Need to show all Publication details
FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
WHERE AP.author_id IN (
        SELECT author_id
        FROM Author
        WHERE name ILIKE '%Ursula Goltz%' --- X
        LIMIT 1 -- select just one id, in case there are multiple authors with the same
) AND P.year = '2015';
```

Query 3(b)

```
SELECT P.* -- Need to show all publication details
FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
WHERE AP.author_id IN (
        SELECT author_id
        FROM Author
        WHERE name ILIKE '%Peter Mowforth%' --- X
        LIMIT 1 -- select just one id, in case there are multiple authors with the same name
) AND P.year = '1990' AND conf_name = 'bmvc';
```

Query 3(c)

```
SELECT A.name
FROM PublicationAuthor AP JOIN author A ON AP.author_id = A.author_id
WHERE AP.publication_id IN (
        SELECT P.publication_id
        FROM Publication P
        WHERE conf_name = 'bmvc' AND year = '1990') -- Y AND Z
GROUP BY A.name
HAVING COUNT(*) > 1;
```

Query 4(a)

```
SELECT name, A.author_id
FROM Author A JOIN
(
        SELECT AP.author_id, conf_name, COUNT(*)
        FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
        WHERE conf_name IN (LOWER('pvldb'), LOWER('SIGMOD'))
        GROUP BY AP.author_id, conf_name
        HAVING COUNT(*) >= 10
) x
ON A.author_id = x.author_id
GROUP BY name, A.author_id
HAVING COUNT(*) = 2;
```

Query 4(b)

```
SELECT A.name
FROM AUTHOR A
WHERE A.author_id IN(
  (
        SELECT AP.author_id
        FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
        WHERE conf_name = LOWER('PVLDB')
        GROUP BY AP.author_id
        HAVING COUNT(*) >= 15)
  EXCEPT
  ( SELECT AP.author_id
        FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
        WHERE conf_name = LOWER('KDD')
        GROUP BY AP.author_id)
);
```

Query 5

```
SELECT substring(year from 1 for 3) as y3, COUNT(*)
FROM Publication
WHERE year BETWEEN '1970' AND '2019'
GROUP BY substring(year from 1 for 3);
-- Tried this on some sample data, and it works without having to form
-- intermediate tables for each ten year interval
```

Query 6 (not optimized)

```
-- The following gives us all the author-co author pairs
CREATE VIEW coauthors AS
SELECT X.author_id AS authorID, Y.author_id AS coauthorID
FROM PublicationAuthor X JOIN PublicationAuthor Y
ON (X.author_id != Y.author_id) and (X.publication_id = Y.publication_id)
GROUP BY X.author_id, Y.author_id;

CREATE VIEW datacoauthors AS
SELECT authorID, count(*) as count -- Get the IDs of authors with the maximum number of co authors
FROM coauthors
WHERE authorID IN (
        SELECT AP.author_id
        FROM PublicationAuthor AP JOIN Publication P
        ON P.publication_id = AP.publication_id
        WHERE (P.key LIKE 'journals%' OR P.key LIKE 'conf%')
        AND LOWER(P.title) LIKE '%data%'
)
GROUP BY authorID;

SELECT authorID
FROM datacoauthors
WHERE count = (SELECT MAX(count) FROM datacoauthors);
```

Query 6 (optimized)

```
SELECT q1.name AS author, q2.collaborators_count
FROM (SELECT * FROM Author) as q1, (
        SELECT PA1.author_id AS author_id,COUNT(DISTINCT PA2.author_id) AS collaborators_count
        FROM PublicationAuthor PA1, PublicationAuthor PA2
        WHERE PA1.author_id!=PA2.author_id AND PA1.publication_id=PA2.publication_id
        GROUP BY PA1.author_id) as q2
WHERE q1.author_id=q2.author_id
AND q1.author_id IN (
        SELECT AP.author_id
        FROM PublicationAuthor AP JOIN Publication P
        ON P.publication_id = AP.publication_id
        WHERE (P.parent_type = 'journals' OR P.parent_type = 'conf')
        AND LOWER(P.title) LIKE '%data%'
)
ORDER BY collaborators_count DESC LIMIT 10;
```

Query 7

```
SELECT A.name, COUNT(*) AS publication_count
FROM Author A, Publication P, PublicationAuthor PA
WHERE P.publication_id=PA.publication_id AND A.author_id=PA.author_id
AND (P.parent_type = 'journals' OR P.parent_type = 'conf')
AND P.year BETWEEN '2013' AND '2017'
AND LOWER(P.title) LIKE '%data%'
GROUP BY A.author_id
ORDER BY count(*) DESC LIMIT 10;
```

Query 8

```
SELECT DISTINCT conf_name
FROM Publication
WHERE mdate LIKE '%-07-%'
AND category = 'inproceedings'
AND parent_type = 'conf'
GROUP BY conf_name, year
HAVING COUNT(*) > 100;
```

Query 9(a)

```
SELECT A.name, AP.author_id
FROM Author A JOIN PublicationAuthor AP ON A.author_id = AP.author_id
JOIN Publication P ON AP.publication_id = P.publication_id
WHERE P.year BETWEEN '1988' AND '2017'
AND SUBSTRING(A.name , LENGTH(A.name)-STRPOS(REVERSE(A.name),' ')+2, LENGTH(A.name)) LIKE 'H%'
GROUP BY AP.author_id, A.name
HAVING COUNT(DISTINCT P.year) = 30;
```

Query 9(b)

```
SELECT A.author_id, A.name, COUNT(*)
FROM Author A JOIN PublicationAuthor AP
ON A.author_id = AP.author_id
WHERE A.author_id IN (
        -- Get the authors of Publication with the earliest Publication date
        SELECT DISTINCT AP.author_id
        FROM PublicationAuthor AP JOIN Publication P ON AP.publication_id = P.publication_id
        WHERE P.year = (SELECT MIN(year) FROM Publication)
        )
GROUP BY A.author_id , A.name;
```

Query 10

```
SELECT SPLIT_PART(A.name, ' ', 1) as firstname, COUNT(*)
FROM Author A JOIN PublicationAuthor AP ON A.author_id = AP.author_id
JOIN Publication P ON AP.publication_id = P.publication_id
WHERE P.year BETWEEN '2016' AND '2017'
GROUP BY SPLIT_PART(A.name, ' ', 1)
ORDER BY COUNT(*) DESC
LIMIT 5;
```

Appendix C: Query Results

Query 1:

category	Count
incollection	44075
article	1417310
inproceedings	1772684
book	9359

(4 rows)
Time: 6147.494 ms

Query 2:

conf_name
aaai
acl
amcc
amcis
amia
atal
ccnc
cec
chi
dagstuhl
gecco
hci

hicss
icc
iccsa
icdcs
iceis
icic
icinco
icml
icphs
icpp
icra
icse
ijcai
ijcnn
ipps
iwcmc
mipro
naacl
ni
pdpta
sac
siu
www

(35 rows)

Time: 1739.136 ms

Query 3(a):

publication_id	category	key	conf_name	parent_type	mdate	title	year	journal
37	article	journals/acta/GlabbeekG015	acta	journals	2017-05-28	Special issue on Combining Compositionality and Concurrency: part 1	2015	Acta Inf.
1443	article	journals/acta/GlabbeekG015A	acta	journals	2017-05-28	Special issue on Combining Compositionality and Concurrency: part 2	2015	Acta Inf.
1115948	article	journals/ife/GoltzRGHMV15	ife	journals	2017-09-16	Design for future: managed software evolution	2015	Computer Science – R&D

(3 rows)
Time: 2155.030 ms

Query 3(b):

publication_id	category	key	conf_name	parent_type	mdate	title	year
1828377	inproceedings	conf/bmvc/MowforthSJU90	bmvc	conf	2017-05-21	A head called Richard.	1990

(1 row)

Time: 1580.799 ms

Query 3(c):

name
Andrew Blake 0001
Andrew Zisserman
Chris Harris
Christopher J. Taylor
David A. Forsyth
David W. Murray
Geoffrey D. Sullivan
J. Michael Brady
John E. W, Mayhew
John P. Frisby
John Porrill
Joseph L. Mundy
Keith D. Baker
Neil A. Thacker
Robert B. Fisher
Stephen Pollard

(16 rows)

Time: 1752.829 ms

Query 4(a):

name	author_id
Mourad Ouzzani	7651
Laks V.S. Lakshmanan	112287
Johannes Gehrke	115356
Ugur Cetintemel	143198
Dan Suciu	174577
Shivnath Babu	200394
Wolfgang Lehner	229547
David Maier	233723
Haixun Wang	238177
M. Tamer Ozsu	273848
Sihem Amer-Yahia	402174
Tim Kraska	437088
Jeffrey F. Naughton	466908
Jiawei Han 0001	476940
Stanley B. Zdonik	528462
Andrew Pavlo	556336
Jignesh M. Patel	563368
Samuel Madden	595841
Hector Garcia-Molina	607475
Nick Koudas	615126
Eugene Wu 0002	619539
Tova Milo	621928
Gao Cong	621990
Stefan Manegold	690433
Michael Stonebraker	710822
Xiaokui Xiao	739443
H.V. Jagadish	774268
Michael J. Franklin	776291

Thomas Neumann 0001	792363
Guoling Li	832926
Renee J. Miller	852396
Neoklis Polyzotis	858320
Paolo Papotti	872421
Christian S. Jensen	920479
Jianzhong Li	935689
Xin Luna Dong	1050931
Philip S. Yu	1088681
Anthony K. H. Tung	1111524
Gustavo Alonso	1150755
Beng Chin Ooi	1166115
Donald Kossman	1171809
Jun Yang 0001	1189325
Xuemin Lin	1209410
Magdalena Balazinska	1223240
Xifeng Yan	1226161
Alon Y. Halevy	1238221
Joseph M. Hellerstein	1239869
Lei Chen 0002	1244851
Yanlei Diao	1245635
Divyakant Agrawal	1356114
Christoph Koch 0001	1364732
Divesh Srivastava	1371376
Anastasia Ailamaki	1383413
Surajit Chaudhuri	1405498
Amol Deshpande	1425362
Vivek R. Narasayya	1437647
Ashwin Machanavajjhala	1454397
Alfons Kemper	1501613
Amr El Abbadi	1502605

(58 rows)

Time: 2371.616 ms

Query 4(b):

name
Wenfei Fan
Jignesh M. Patel
Magdalena Balazinska
Michael J. Franklin
Mohamed F. Mokbel
Dan Suciu
Samuel Madden
Tova Milo
Anastasia Ailamaki
Ihab F. Ilyas
Jens Dittrich
Tim Kraska
Thomas Neumann 0001
Christian S. Jensen
Christopher Ré

(15 rows)

Time: 3488.864 ms

Query 5:

y3	count
200	1331086
201	1912862
198	117634
199	407435
197	38947

(5 rows)

Time: 7256.297 ms

Query 6:

name	collaborators_count
Wei Wang	2326
Wei Li	2260
Yang Liu	1973
Wei Zhang	1942
Lei Wang	1909
Jing Li	1894
Jun Wang	1881
Li Zhang	1840
Lei Zhang	1837
Yu Zhang	1768

(10 rows)

Time: 65296.033 ms

Query 7:

name	publication_count
Alfredo Cuzzocrea	92
Sören Auer	87
Wolfgang Lehner	73
Erik Mannens	67
Francisco Herrera	65

Ruben Verborgh	64
Jun Wang	64
Rajiv Ranjan	62
Albert Y. Zomaya	59
Fan Zhang	58

(10 rows)
Time: 8804.659 ms

Query 8:

conf_name
Sigir
edm
interact
gmds
ijcai
mmvr
iassist
mva
ieaaie
iccsa
amia
sac
cse
mipro
amcc
isvlsi
aaai
amcis
ms

icic
mobisys
appinf
atal
wscg
iwcmc
mie
chi
icra
ipps
icc
cicc
cgiv
eucnc
dac
iccc
icdcs
siu
hci
icse
icalp
www
ijcnn
pics
cec
um
icml
ACISicis
esann
ccgrid
iticse
pdpta
icphs
gecko
visapp
naacl

apcc
ccnc
fgr
icinco

(58 rows)

Time: 827.198 ms

Query 9(a):

name	author_id
Rolf Hennicker	51341
Pascal Van Hentenryck	96270
Mark Horowitz	129809
Seth Hutchinson	141854
Geoffrey E. Hinton	279157
Thomas S. Huan	370588
Johan Håstad	493321
Joseph Y. Halpern	557281
Wen-mei W. Hwu	573355
Richard I. Hartley	616566
David Hutchison	665068
Pierre Hansen	678851
Michael Hanus	727270
Frank van Harmelen	739432
Lenwood S. Heath	790207
Vincent Hayward	839773
Ali R. Hurson	1044790
Theo Härder	1214753
Alan R. Hevner	1228319
Jenq-Neng Hwang	128994
Juraj Hromkovic	1318627

Maurice Herlihy	1482693
David Harel	1529416
Richard Hull	1593795
John P. Hayes	1709531
Manuel V. Hermenegildo	1805493
James A. Hendler	1811366
Scott E. Hudson	184857
David Haussler	1888135
Nicholas J. Higham	1906306
Matthew Hennessy	1969159

(31 rows)

Time: 9380.156 ms

Query 9(b):

author_id	name	count
259865	J. Barkley Rosser	18
466390	Alonzo Church	6
544831	Emil L. Post	3
788422	C. I. Lewis	1
986737	W. V. Quine	28
1266027	Frederic Brenton Fitch	30
1436589	Arnold F. Emch	3
1824470	C. J. Ducasse	4

(8 rows)

Time: 8160.867 ms

Query 10:

firstname	count
David	56876
Michael	55810
Daniel	36486
Thomas	34894
Peter	32626

(5 rows)

Time: 103249.539 ms
