

SOEN 363 - Project Phase 2
Section S

Group ID: 4

Group Name: Group B

Zain Khan - 40110693 - zmkhan025@gmail.com

Noor Hammodi - 40061760 - noor.hammodi@gmail.com

Nicolas Towa Kamgne - 40154685 - nicolastowakamgne@gmail.com

Ashiqur Rahman - 40096265 - ashlegitimate@outlook.com

Presented to Professor Essam Mansour
December 14, 2022

3 Analyzing Big Data Using SQL and RDBMS Systems

3a)

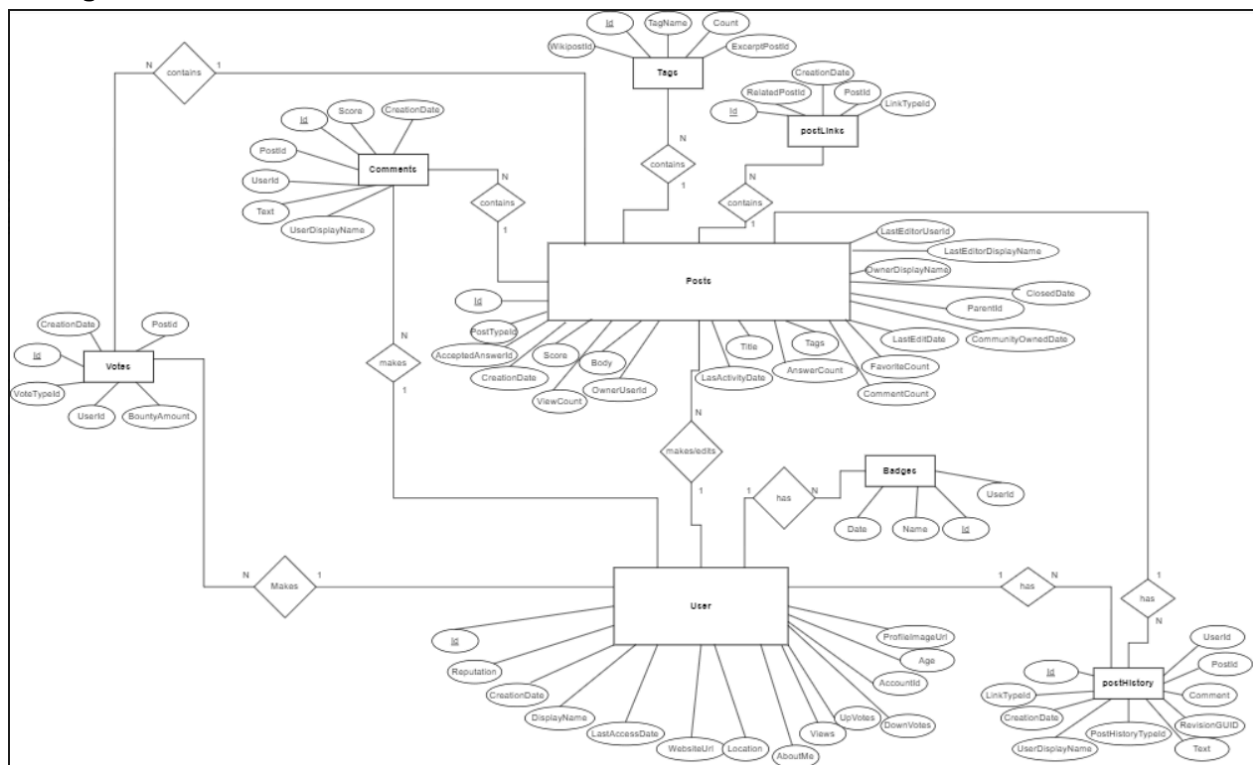
We have picked a Stack Overflow dataset (<https://archive.org/details/stackexchange>) measuring 658.4MB and containing 8 files.

The data can be found

here: https://drive.google.com/drive/folders/14F_w2MAAx-sFTOXGRC4dAt188pdhZRnr?usp=share_link

3b)

ErDiagram



DDL Statements

--CREATING TABLES (with postgresql)

CREATE TABLE users (

Id INT NOT NULL,

Reputation INT DEFAULT NULL,

CreationDate TIMESTAMP DEFAULT NULL,

```

    DisplayName VARCHAR(255) DEFAULT NULL,
    LastAccessDate TIMESTAMP DEFAULT NULL,
    WebsiteUrl VARCHAR(255) DEFAULT NULL,
    Location VARCHAR(255) DEFAULT NULL,
    AboutMe TEXT DEFAULT NULL,
    Views INT DEFAULT NULL,
    UpVotes INT DEFAULT NULL,
    DownVotes INT DEFAULT NULL,
    AccountId INT DEFAULT NULL,
    Age INT DEFAULT NULL,
    ProfileImageUrl VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY (Id)
);

CREATE TABLE posts (
    Id INT NOT NULL,
    PostTypeId INT DEFAULT NULL,
    AcceptedAnswerId INT DEFAULT NULL,
    CreationDate TIMESTAMP DEFAULT NULL,
    Score INT DEFAULT NULL,
    ViewCount INT DEFAULT NULL,
    Body TEXT DEFAULT NULL,
    OwnerUserId INT DEFAULT NULL,
    LastActivityDate TIMESTAMP DEFAULT NULL,
    Title VARCHAR(255) DEFAULT NULL,
    Tags VARCHAR(255) DEFAULT NULL,
    AnswerCount INT DEFAULT NULL,
    CommentCount INT DEFAULT NULL,
    FavoriteCount INT DEFAULT NULL,
    LastEditorUserId INT DEFAULT NULL,
    LastEditDate TIMESTAMP DEFAULT NULL,
    CommunityOwnedDate TIMESTAMP DEFAULT NULL,
    ParentId INT DEFAULT NULL,
    ClosedDate TIMESTAMP DEFAULT NULL,
    OwnerDisplayName VARCHAR(255) DEFAULT NULL,
    LastEditorDisplayName VARCHAR(255) DEFAULT NULL,
    PRIMARY KEY (Id),
    CONSTRAINT posts_LastEditorUserId_fkey FOREIGN KEY (LastEditorUserId)
        REFERENCES users (Id),
    CONSTRAINT posts_OwnerUserId_fkey FOREIGN KEY (OwnerUserId)
        REFERENCES users (Id),
    CONSTRAINT posts_ParentId_fkey FOREIGN KEY (ParentId)
        REFERENCES posts (Id)
);

```

```
CREATE TABLE postLinks (  
    Id INT NOT NULL,  
    CreationDate TIMESTAMP DEFAULT NULL,  
    PostId INT DEFAULT NULL,  
    RelatedPostId INT DEFAULT NULL,  
    LinkTypeId INT DEFAULT NULL,  
    PRIMARY KEY (Id),  
    CONSTRAINT postlinks_stripped_PostId_fkey FOREIGN KEY (PostId)  
        REFERENCES posts (Id),  
    CONSTRAINT postlinks_stripped_RelatedPostId_fkey FOREIGN KEY (RelatedPostId)  
        REFERENCES posts (Id)  
);
```

```
CREATE TABLE tags (  
    Id INT NOT NULL,  
    TagName VARCHAR(255) DEFAULT NULL,  
    Count INT DEFAULT NULL,  
    ExcerptPostId INT DEFAULT NULL,  
    WikiPostId INT DEFAULT NULL,  
    PRIMARY KEY (Id),  
    FOREIGN KEY (ExcerptPostId)  
        REFERENCES posts (Id)  
);
```

```
CREATE TABLE badges (  
    Id INT NOT NULL,  
    UserId INT DEFAULT NULL,  
    Name VARCHAR(255) DEFAULT NULL,  
    Date timestamp DEFAULT NULL,  
    PRIMARY KEY (Id),  
    FOREIGN KEY (UserId)  
        REFERENCES users (Id)  
);
```

```
CREATE TABLE comments (  
    Id INT NOT NULL,  
    PostId INT DEFAULT NULL,  
    Score INT DEFAULT NULL,  
    Text TEXT DEFAULT NULL,  
    CreationDate timestamp DEFAULT NULL,  
    UserId INT DEFAULT NULL,  
    UserDisplayName VARCHAR(255) DEFAULT NULL,  
    PRIMARY KEY (Id),
```

```

FOREIGN KEY (PostId)
  REFERENCES posts (Id),
FOREIGN KEY (UserId)
  REFERENCES users (Id)
);

CREATE TABLE postHistory (
  Id INT NOT NULL,
  PostHistoryTypeId INT DEFAULT NULL,
  PostId INT DEFAULT NULL,
  RevisionGUID VARCHAR(255) DEFAULT NULL,
  CreationDate timestamp DEFAULT NULL,
  UserId INT DEFAULT NULL,
  Text TEXT DEFAULT NULL,
  Comment TEXT DEFAULT NULL,
  UserDisplayName VARCHAR(255) DEFAULT NULL,
  PRIMARY KEY (Id),
  FOREIGN KEY (PostId)
    REFERENCES posts (Id),
  FOREIGN KEY (UserId)
    REFERENCES users (Id)
);

```

```

CREATE TABLE votes (
  Id INT NOT NULL,
  PostId INT DEFAULT NULL,
  VoteTypeId INT DEFAULT NULL,
  CreationDate timestamp DEFAULT NULL,
  UserId INT DEFAULT NULL,
  BountyAmount INT DEFAULT NULL,
  PRIMARY KEY (Id),
  FOREIGN KEY (PostId)
    REFERENCES posts (Id),
  FOREIGN KEY (UserId)
    REFERENCES users (Id)
);

```

3c)

--IMPORTING DATA (substitute for personalized csv directories)

-Using 'Import/Export data' in PGAdmin4 GUI, or alternatively by running the following queries:

```

copy public.users (id, reputation, creationdate, displayname, lastaccessdate, websiteurl,
location, aboutme, views, upvotes, downvotes, accountid, age, profileimageurl) FROM

```

```
'C:/Users/Desktop/Tables/users.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8' QUOTE
\" NULL 'null' ESCAPE \";
```

```
copy public.posts (id, posttypeid, acceptedanswerid, creationdate, score, viewcount, body,
owneruserid, lastactivitydate, title, tags, answercount, commentcount, favoritecount,
lasteditoruserid, lasteditdate, communityowneddate, parentid, closeddate, ownerdisplayname,
lasteditordisplayname) FROM 'C:/Users/Desktop/Tables/posts.csv' DELIMITER ',' CSV
HEADER ENCODING 'UTF8' QUOTE \" NULL 'null' ESCAPE \"
```

```
copy public.votes (id, postid, votetypeid, creationdate, userid, bountyamount) FROM
'C:/Users/Desktop/Tables/votes.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8' QUOTE
\" NULL 'null' ESCAPE \"
```

```
copy public.tags (id, tagname, count, excerptpostid, wikipostid) FROM
'C:/Users/Desktop/Tables/tags.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8' QUOTE
\" NULL 'null' ESCAPE \"
```

```
copy public.postlinks (id, creationdate, postid, relatedpostid, linktypeid) FROM
'C:/Users/Desktop/Tables/postLinks.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8'
QUOTE \" NULL 'null' ESCAPE \";
```

```
copy public.posthistory (id, posthistorytypeid, postid, revisionguid, creationdate, userid, text,
comment, userdisplayname) FROM 'C:/Users/Desktop/Tables/postHistory.csv' DELIMITER ','
CSV HEADER ENCODING 'UTF8' QUOTE \" NULL 'null' ESCAPE \";
```

```
copy public.comments (id, postid, score, text, creationdate, userid, userdisplayname) FROM
'C:/Users/Tables/comments.csv' DELIMITER ',' CSV HEADER ENCODING 'UTF8' QUOTE \"
NULL 'null' ESCAPE \";
```

```
copy public.badges (id, userid, name, date) FROM 'C:/Users/Desktop/Tables/badges.csv'
DELIMITER ',' CSV HEADER ENCODING 'UTF8' QUOTE \" NULL 'null' ESCAPE \";
```

3d)

--10 Reports (Queries)

-- 1) Showing top 10 users with the most badges (based on the 'UserId' column in the 'badges' table)

```
SELECT u.DisplayName, COUNT(*) as TotalBadges
FROM users u, badges b
WHERE b.UserId = u.Id
GROUP BY u.DisplayName
ORDER BY TotalBadges DESC
LIMIT 10;
```

	displayname character varying (255) 🔒	totalbadges bigint 🔒
1	whuber	456
2	Glen_b	318
3	chl	282
4	gung	256
5	Jeromy Anglim	222
6	John	182
7	Macro	176
8	Tal Galili	159
9	Rob Hyndman	156
10	Peter Flom	152

-- 2) Showing top 10 posts with the most comments (based on the 'PostId' column in the 'comments' table)

```
SELECT p.Title, COUNT(*) as TotalComments
FROM posts p, comments c
WHERE c.PostId = p.id
GROUP BY p.title
ORDER BY TotalComments DESC
LIMIT 10;
```

	title character varying (255) 🔒	totalcomments bigint 🔒
1	[null]	83628
2	Does the presence of an outlier increase the probability that another outlier will also be present on the same observa...	37
3	Amazon interview question—probability of 2nd interview	35
4	Which test is better suited to compare averaged versus single data sets	33
5	How to fit a mixture of Gamma distributions to the PMF of a discrete distribution?	33
6	How do I incorporate an innovative outlier at observation 48 in my ARIMA model?	33
7	Binomial GLM - Predicting the time of buying of a product	32
8	Rejection sampling from a normal distribution	31
9	Composition of probability density	29
10	What are the chances my wife has lupus?	29

-- 3) Showing top 10 users who have made the most edits to posts (based on the 'UserId' column in the 'postHistory' table)

```
SELECT u.DisplayName, COUNT(*) as TotalEdits
FROM users u, postHistory p
WHERE p.userId = u.Id
```

```

GROUP BY u.displayName
ORDER BY TotalEdits DESC
LIMIT 10;

```

	displayname character varying (255) 🔒	totalEdits bigint 🔒
1	gung	7321
2	whuber	7097
3	Glen_b	6900
4	mbq	6570
5	chl	6005
6	Community	4511
7	Jeromy Anglim	3107
8	Nick Cox	3034
9	Peter Flom	2739
10	Nick Stauner	2175

-- 4) Showing the total number of new posts per year, followed by the number of new users per year

```

SELECT EXTRACT(YEAR FROM p.CreationDate) AS Year,
       COUNT(*) AS NumberOfNewPosts
FROM posts p
GROUP BY Year
ORDER BY Year ASC;

```

	year numeric 🔒	numberofnewposts bigint 🔒
1	2009	18
2	2010	5450
3	2011	13163
4	2012	20113
5	2013	26771
6	2014	26461

```

SELECT EXTRACT(YEAR FROM u.CreationDate) AS Year,
       COUNT(*) AS NumberOfNewUsers
FROM users u
GROUP BY Year
ORDER BY Year ASC;

```


	year numeric	numberofnewusers bigint
1	2010	1678
2	2011	4430
3	2012	7544
4	2013	12231
5	2014	14442

We may note that, oddly enough, this dataset had no new users in the year 2009 yet had 18 posts that year. This may be due to the fact that StackOverFlow started as a company in 2008 and was still in its early stages. We may also notice that as the number of users increases, so does the number of posts, with the slight exception of the number of posts remaining more or less the same between 2013 and 2014.

-- 5) Showing the 10 lowest reputable users followed by the 10 highest reputable users along with their account creation dates

```
(SELECT u1.DisplayName, u1.Reputation, u1.UpVotes, u1.DownVotes
FROM users u1
WHERE u1.UpVotes > 10
AND u1.DownVotes > 10
ORDER BY u1.Reputation ASC
LIMIT 10)
UNION ALL
(SELECT u2.DisplayName, u2.Reputation, u2.UpVotes, u2.DownVotes
FROM users u2
ORDER BY u2.Reputation DESC
LIMIT 10);
```

	displayname character varying (255) 🔒	reputation integer 🔒	upvotes integer 🔒	downvotes integer 🔒
1	Community	1	5007	1920
2	hadley	111	59	15
3	Deer Hunter	224	96	12
4	Joel Reyes Noche	245	41	11
5	Mike John	292	12	16
6	subhash c. davar	298	64	12
7	Bill the Lizard	379	163	49
8	garciaj	393	150	19
9	Gschneider	406	15	13
10	RioRaider	551	418	18
11	whuber	87393	11273	779
12	Glen_b	65272	7035	143
13	Peter Flom	44152	2156	82
14	gung	37083	8641	125
15	chl	31170	10523	214
16	Greg Snow	25123	582	4
17	Jeromy Anglim	22625	2496	45
18	Michael Chernick	22275	2619	42
19	Frank Harrell	19585	155	56
20	Rob Hyndman	18283	1014	59

In this case, the lowest reputable users with low upvotes/downvotes were omitted as they all had a reputation of 1. Having a count of upvotes/downvotes greater than or equal to 10 immediately filtered out these users, as it represents users who interact more with StackOverFlow. Therefore, with this filter criteria, it may be noted that user reputation may range from about 100 to 87000. There also seems to be no correlation between the number of upvotes/downvotes and the reputation among the 10 least and most reputable users individually.

```
-- 6) Selecting top 10 users with the most badges
SELECT u.DisplayName, COUNT(b.Id) AS num_badges
FROM users u, badges b
WHERE u.id = b.userid
GROUP BY u.DisplayName
ORDER BY num_badges DESC
```

	displayname character varying (255) 🔒	num_badges bigint 🔒
1	whuber	456
2	Glen_b	318
3	chl	282
4	gung	256
5	Jeromy Anglim	222
6	John	182
7	Macro	176
8	Tal Galili	159
9	Rob Hyndman	156
10	Peter Flom	152

Interestingly enough, among the top 10 most reputable users (as seen in query #5), 7 of them are among the top 10 badge earners. This indicates a sound correlation between reputation and badge number.

-- 7) Selecting Title, ViewCount and Score of top 10 lowest and highest scored posts with a significant viewcount

```
(SELECT p.Title, p.ViewCount, p.Score
FROM posts p
WHERE p.ViewCount IS NOT NULL
AND p.ViewCount > 10000
ORDER BY p.Score ASC
LIMIT 10)
UNION ALL
(SELECT p.Title, p.ViewCount, p.Score
FROM posts p
WHERE p.ViewCount IS NOT NULL
ORDER BY p.Score DESC
LIMIT 10);
```

	title character varying (255)	viewcount integer	score integer
1	What is the difference between normal distribution and standard normal distributi...	16957	-4
2	How do I interpret the results from the F-test in excel	14046	0
3	What is the "root mse" in stata?	12116	1
4	How do I interpret a probit model in Stata?	16604	1
5	How do you interpret results from unit root tests?	12859	1
6	Working with Likert scales in SPSS	17323	1
7	How to stop excel from changing a range when you drag a formula down?	25583	1
8	How to determine which variables are statistically significant in multiple regressi...	16587	1
9	How to make a forest plot with Excel?	23434	2
10	How to compute standard deviation of difference between two data sets?	18883	2
11	Python as a statistics workbench	60964	192
12	Making sense of principal component analysis, eigenvectors & eigenvalues	66071	184
13	What is your favorite "data analysis" cartoon?	64481	156
14	The Two Cultures: statistics vs. machine learning?	29229	152
15	Famous statistician quotes	34780	124
16	Why square the difference instead of taking the absolute value in standard deviati...	39118	122
17	What are common statistical sins?	10402	121
18	Is Facebook coming to an end?	25744	110
19	Is $\$R^2\$$ useful or dangerous?	8688	106
20	Bayesian and frequentist reasoning in plain English	21916	102

To be considered “significant” an arbitrary viewcount of 10000 views was selected to filter out posts with low scores and viewcounts. To answer the question of whether a post’s increased viewcount corresponded to a higher score, this query shows that there are posts with up to 25000 views (index 7 in this table) that receive scores of 1, while there are posts with 25000 views that receive scores of 110 (index 18 in this table).

-- 8) Selecting display name, number of comments, average post score and average view count of the top 10 users with the most comments on their posts

```
SELECT u.DisplayName,
       COUNT(c.Id) AS num_comments,
       AVG(p.Score) AS avg_score,
       AVG(p.ViewCount) AS avg_views
FROM users u, posts p, comments c
WHERE u.id = p.OwnerUserId
AND p.Id = c.PostId
AND p.viewcount IS NOT NULL
GROUP BY u.DisplayName
```



ORDER BY num_comments DESC
LIMIT 10;

	displayname character varying (255) 🔒	num_comments bigint 🔒	avg_score numeric 🔒	avg_views numeric 🔒
1	Tim	551	2.372050816	508.1197822
2	JohnK	287	3.125435540	156.2543554
3	Stéphane Laurent	269	4.200743494	426.7806691
4	Chris	242	2.161157024	429.4834710
5	Tal Galili	238	6.962184873	1480.840336
6	David	217	5.562211981	763.9677419
7	user34790	214	1.775700934	263.1822429
8	luciano	204	2.416666666	366.9068627
9	shabbychef	200	8.575000000	1236.910000
10	Luca	196	1.612244897	1046.673469

Note that these top 10 users with the most number of comments on their posts do not have a high average post view compared to the viewcounts of the posts in query 7. These users are also not among the top 10 most reputable users or top 10 users with the most badges found in queries 5 and 5, respectively. The randomness of these users with low average viewcounts and scores on their posts indicates that posts made on StackOverFlow all have equal reach, opposed to posts on other platforms like Twitter where users with more followers have more comments.

-- 9) Selecting the top 5 most upvoted posts and least upvoted posts by a user based on the user badge status,

```
(SELECT b."name", COUNT(u.upvotes) AS upv
FROM badges b, users u
WHERE b.id=u.id
GROUP BY b."name",u.upvotes
ORDER BY upv DESC
LIMIT 5)
Union ALL
(SELECT b."name", COUNT(u.upvotes) AS upv
FROM badges b, users u
WHERE b.id=u.id
GROUP BY b."name",u.upvotes
ORDER BY upv ASC
LIMIT 5)
```

	name character varying (255) 	upv bigint 
1	Student	5391
2	Supporter	3090
3	Editor	2846
4	Scholar	2376
5	Teacher	1722
6	Notable Question	1
7	Talkative	1
8	Critic	1
9	Editor	1
10	Scholar	1

-- 10) Selecting the top 10 most viewed tagnames containing the word “data” in their tag name, followed by the top 10 most viewed tagnames.

```
(SELECT t.tagname, p.viewcount
FROM tags t, posts p
WHERE t.id=p.id and t.tagname LIKE '%data%'
AND p.viewcount IS NOT NULL
GROUP BY t.tagname,p.viewcount
ORDER BY p.viewcount DESC
LIMIT 10)
UNION ALL
(SELECT t.tagname, p.viewcount
FROM tags t, posts p
WHERE t.id=p.id
AND p.viewcount IS NOT NULL
GROUP BY t.tagname,p.viewcount
ORDER BY p.viewcount DESC
LIMIT 10)
```

	tagname character varying (255) 🔒	viewcount integer 🔒
1	dataset	21925
2	collecting-data	11533
3	binary-data	6700
4	data-acquisition	5593
5	compositional-data	3221
6	data-imputation	811
7	data-visualization	497
8	bootstrap	70255
9	kalman-filter	64481
10	kde	39118
11	optimal-scaling	36801
12	delta-method	34780
13	contingency-tables	29261
14	distributions	29229
15	value-of-information	28878
16	computational-statistics	25597
17	clinical-trials	23985

Note that in this case, there have only been 7 returned tagnames containing the word “data”, meaning that there are only 7 tagnames with that word with a viewcount that is not null. The top tagname turned out to be “bootstrap”, with a viewcount of 70255, while the tag name containing the word “data” with the most views is “dataset”, with a viewcount of 21925.

3e) Indexing

The following queries (queries 1, 2, 3 & 5 from part 3d) were picked to demonstrate the effectiveness of indexing:

Query 1 without indexing:

	QUERY PLAN text
1	Limit (cost=4855.70..4855.73 rows=10 width=17) (actual time=134.331..134.335 rows=10 loops=1)
2	-> Sort (cost=4855.70..4941.67 rows=34388 width=17) (actual time=134.328..134.331 rows=10 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=3768.71..4112.59 rows=34388 width=17) (actual time=123.674..130.391 rows=22071 ...)
6	Group Key: u.displayname
7	Batches: 1 Memory Usage: 3345kB
8	-> Hash Join (cost=1804.31..3369.45 rows=79851 width=9) (actual time=21.899..78.612 rows=79851 loops=1)
9	Hash Cond: (b.userid = u.id)
10	-> Seq Scan on badges b (cost=0.00..1355.51 rows=79851 width=4) (actual time=0.021..11.402 rows=79851 lo...)
11	-> Hash (cost=1300.25..1300.25 rows=40325 width=13) (actual time=21.535..21.536 rows=40325 loops=1)
12	Buckets: 65536 Batches: 1 Memory Usage: 2365kB
13	-> Seq Scan on users u (cost=0.00..1300.25 rows=40325 width=13) (actual time=0.014..9.821 rows=40325 loo...)
14	Planning Time: 0.338 ms
15	Execution Time: 135.712 ms
Total rows: 15 of 15 Query complete 00:00:00.208	

Query 1 with indexing:

	QUERY PLAN text
1	Limit (cost=4855.70..4855.73 rows=10 width=17) (actual time=97.364..97.368 rows=10 loops=1)
2	-> Sort (cost=4855.70..4941.67 rows=34388 width=17) (actual time=97.361..97.364 rows=10 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=3768.71..4112.59 rows=34388 width=17) (actual time=85.573..92.978 rows=2...)
6	Group Key: u.displayname
7	Batches: 1 Memory Usage: 3345kB
8	-> Hash Join (cost=1804.31..3369.45 rows=79851 width=9) (actual time=16.982..53.834 rows=79851 l...)
9	Hash Cond: (b.userid = u.id)
10	-> Seq Scan on badges b (cost=0.00..1355.51 rows=79851 width=4) (actual time=0.021..6.969 rows=7 ...)
11	-> Hash (cost=1300.25..1300.25 rows=40325 width=13) (actual time=16.619..16.620 rows=40325 loop...)
12	Buckets: 65536 Batches: 1 Memory Usage: 2365kB
13	-> Seq Scan on users u (cost=0.00..1300.25 rows=40325 width=13) (actual time=0.017..7.237 rows=40...)
14	Planning Time: 0.666 ms
15	Execution Time: 98.866 ms
Total rows: 15 of 15 Query complete 00:00:00.151	

CREATE INDEX idx_badges_user_id ON badges (UserId);

It is believed that the costs for query 1 with and without indexing are very similar as badges is a relatively smaller table

Query 2 without indexing:

	QUERY PLAN text	
1	Limit (cost=41483.71..41483.74 rows=10 width=66) (actual time=621.861..642.935 rows=10 loops=1)	
2	-> Sort (cost=41483.71..41592.10 rows=43356 width=66) (actual time=621.856..642.929 rows=10 loops=1)	
3	Sort Key: (count(*)) DESC	
4	Sort Method: top-N heapsort Memory: 26kB	
5	-> Finalize GroupAggregate (cost=28692.71..40546.80 rows=43356 width=66) (actual time=505.449..637.049 rows=...	
6	Group Key: p.title	
7	-> Gather Merge (cost=28692.71..39679.68 rows=86712 width=66) (actual time=505.439..618.796 rows=37793 lo...	
8	Workers Planned: 2	
9	Workers Launched: 2	
10	-> Partial GroupAggregate (cost=27692.68..28670.94 rows=43356 width=66) (actual time=446.552..486.442 rows=...	
11	Group Key: p.title	
12	-> Sort (cost=27692.68..27874.25 rows=72627 width=58) (actual time=446.540..471.514 rows=58102 loops=3)	
13	Sort Key: p.title	
14	Sort Method: external merge Disk: 2240kB	
15	Worker 0: Sort Method: external merge Disk: 2272kB	
16	Worker 1: Sort Method: external merge Disk: 2392kB	
17	-> Parallel Hash Join (cost=11894.27..19095.20 rows=72627 width=58) (actual time=87.174..158.288 rows=58102...	
18	Hash Cond: (c.postid = p.id)	
19	-> Parallel Seq Scan on comments c (cost=0.00..7010.27 rows=72627 width=4) (actual time=0.207..36.749 rows=5...	
20	-> Parallel Hash (cost=11415.23..11415.23 rows=38323 width=62) (actual time=86.456..86.457 rows=30659 loops=...	
21	Buckets: 131072 Batches: 1 Memory Usage: 7104kB	
22	-> Parallel Seq Scan on posts p (cost=0.00..11415.23 rows=38323 width=62) (actual time=0.179..68.105 rows=306...	
23	Planning Time: 0.642 ms	
24	Execution Time: 649.319 ms	
Total rows: 24 of 24		Query complete 00:00:00.710

Query 2 with indexing:

	QUERY PLAN	
	text	🔒
1	Limit (cost=37267.66..37267.68 rows=10 width=66) (actual time=519.275..534.454 rows=10 loops=1)	
2	-> Sort (cost=37267.66..37376.05 rows=43356 width=66) (actual time=519.270..534.448 rows=10 loops=1)	
3	Sort Key: (count(*)) DESC	
4	Sort Method: top-N heapsort Memory: 26kB	
5	-> Finalize GroupAggregate (cost=24476.65..36330.75 rows=43356 width=66) (actual time=405.857..528.398 rows=26560 loops=1)	
6	Group Key: p.title	
7	-> Gather Merge (cost=24476.65..35463.63 rows=86712 width=66) (actual time=405.844..512.257 rows=26565 loops=1)	
8	Workers Planned: 2	
9	Workers Launched: 2	
10	-> Partial GroupAggregate (cost=23476.63..24454.89 rows=43356 width=66) (actual time=333.272..375.090 rows=8855 loops=3)	
11	Group Key: p.title	
12	-> Sort (cost=23476.63..23658.19 rows=72627 width=58) (actual time=333.258..359.533 rows=58102 loops=3)	
13	Sort Key: p.title	
14	Sort Method: external merge Disk: 2600kB	
15	Worker 0: Sort Method: external merge Disk: 1856kB	
16	Worker 1: Sort Method: external merge Disk: 2456kB	
17	-> Parallel Hash Join (cost=11894.69..14879.14 rows=72627 width=58) (actual time=68.951..102.317 rows=58102 loops=3)	
18	Hash Cond: (c.postid = p.id)	
19	-> Parallel Index Only Scan using idx_comments_post_id on comments c (cost=0.42..2794.22 rows=72627 width=4) (actual time=0.167..10.076 rows=58102 loops=3)	
20	Heap Fetches: 0	
21	-> Parallel Hash (cost=11415.23..11415.23 rows=38323 width=62) (actual time=68.411..68.413 rows=30659 loops=3)	
22	Buckets: 131072 Batches: 1 Memory Usage: 7136kB	
23	-> Parallel Seq Scan on posts p (cost=0.00..11415.23 rows=38323 width=62) (actual time=0.242..53.417 rows=30659 loops=3)	
24	Planning Time: 0.499 ms	
25	Execution Time: 538.730 ms	
Total rows: 25 of 25		Query complete 00:00:00.563

```
CREATE INDEX idx_posts_title ON posts (title);
CREATE INDEX idx_comments_post_id ON comments (PostId);
```

Query 3 without indexing:

	QUERY PLAN text
1	Limit (cost=29815.05..29815.08 rows=10 width=17) (actual time=609.012..609.017 rows=10 loops=1)
2	-> Sort (cost=29815.05..29901.02 rows=34388 width=17) (actual time=609.009..609.012 rows=10 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=28728.06..29071.94 rows=34388 width=17) (actual time=596.710..604.634 rows=19103 l...
6	Group Key: u.displayname
7	Batches: 1 Memory Usage: 3089kB
8	-> Hash Join (cost=1804.31..27212.13 rows=303187 width=9) (actual time=18.174..452.086 rows=281859 loops=1)
9	Hash Cond: (p.userid = u.id)
10	-> Seq Scan on posthistory p (cost=0.00..24611.87 rows=303187 width=4) (actual time=0.477..274.777 rows=303...
11	-> Hash (cost=1300.25..1300.25 rows=40325 width=13) (actual time=17.337..17.338 rows=40325 loops=1)
12	Buckets: 65536 Batches: 1 Memory Usage: 2365kB
13	-> Seq Scan on users u (cost=0.00..1300.25 rows=40325 width=13) (actual time=0.026..7.659 rows=40325 loops=1)
14	Planning Time: 2.380 ms
15	Execution Time: 610.515 ms
Total rows: 15 of 15 Query complete 00:00:00.639	

Query 3 with indexing:

	QUERY PLAN text
1	Limit (cost=11055.41..11055.44 rows=10 width=17) (actual time=228.085..228.089 rows=10 loops=1)
2	-> Sort (cost=11055.41..11141.38 rows=34388 width=17) (actual time=228.083..228.086 rows=10 loops=1)
3	Sort Key: (count(*)) DESC
4	Sort Method: top-N heapsort Memory: 25kB
5	-> HashAggregate (cost=9968.42..10312.30 rows=34388 width=17) (actual time=220.329..224.978 rows=19103 loops=1)
6	Group Key: u.displayname
7	Batches: 1 Memory Usage: 3089kB
8	-> Hash Join (cost=1804.73..8452.48 rows=303187 width=9) (actual time=18.027..134.481 rows=281859 loops=1)
9	Hash Cond: (p.userid = u.id)
10	-> Index Only Scan using idx_posthistory_user_id on posthistory p (cost=0.42..5852.23 rows=303187 width=4) (actual time=...
11	Heap Fetches: 0
12	-> Hash (cost=1300.25..1300.25 rows=40325 width=13) (actual time=17.629..17.630 rows=40325 loops=1)
13	Buckets: 65536 Batches: 1 Memory Usage: 2365kB
14	-> Seq Scan on users u (cost=0.00..1300.25 rows=40325 width=13) (actual time=0.025..7.505 rows=40325 loops=1)
15	Planning Time: 1.470 ms
16	Execution Time: 229.579 ms
Total rows: 16 of 16 Query complete 00:00:00.295	

```
CREATE INDEX idx_posthistory_user_id ON posthistory (userId);
CREATE INDEX idx_users_display_name ON users (displayname);
```

In this case, the cost and execution time was cut drastically with indexing

Query 5 without indexing:

	QUERY PLAN text	
1	Append (cost=1501.95..3673.89 rows=16 width=21) (actual time=9.631..35.292 rows=20 loops=1)	
2	-> Limit (cost=1501.95..1501.97 rows=6 width=21) (actual time=9.630..9.633 rows=10 loops=1)	
3	-> Sort (cost=1501.95..1501.97 rows=6 width=21) (actual time=9.628..9.630 rows=10 loops=1)	
4	Sort Key: u1.reputation	
5	Sort Method: top-N heapsort Memory: 26kB	
6	-> Seq Scan on users u1 (cost=0.00..1501.88 rows=6 width=21) (actual time=0.016..9.556 rows=123 loops=1)	
7	Filter: ((upvotes > 10) AND (downvotes > 10))	
8	Rows Removed by Filter: 40202	
9	-> Limit (cost=2171.66..2171.68 rows=10 width=21) (actual time=25.649..25.653 rows=10 loops=1)	
10	-> Sort (cost=2171.66..2272.47 rows=40325 width=21) (actual time=25.647..25.649 rows=10 loops=1)	
11	Sort Key: u2.reputation DESC	
12	Sort Method: top-N heapsort Memory: 26kB	
13	-> Seq Scan on users u2 (cost=0.00..1300.25 rows=40325 width=21) (actual time=0.024..14.770 rows=4032...)	
14	Planning Time: 0.279 ms	
15	Execution Time: 35.338 ms	
Total rows: 15 of 15		Query complete 00:00:00.148

Query 5 with indexing:

	QUERY PLAN text	
1	Append (cost=50.32..51.91 rows=16 width=21) (actual time=0.537..0.583 rows=20 loops=1)	
2	-> Limit (cost=50.32..50.33 rows=6 width=21) (actual time=0.536..0.539 rows=10 loops=1)	
3	-> Sort (cost=50.32..50.33 rows=6 width=21) (actual time=0.535..0.536 rows=10 loops=1)	
4	Sort Key: u1.reputation	
5	Sort Method: top-N heapsort Memory: 26kB	
6	-> Bitmap Heap Scan on users u1 (cost=27.62..50.24 rows=6 width=21) (actual time=0.375..0.495 rows=123 loops=1)	
7	Recheck Cond: ((downvotes > 10) AND (upvotes > 10))	
8	Heap Blocks: exact=106	
9	-> BitmapAnd (cost=27.62..27.62 rows=6 width=0) (actual time=0.356..0.357 rows=0 loops=1)	
10	-> Bitmap Index Scan on idx_users_downvotes (cost=0.00..5.24 rows=127 width=0) (actual time=0.098..0.098 rows=124 ...)	
11	Index Cond: (downvotes > 10)	
12	-> Bitmap Index Scan on idx_users_upvotes (cost=0.00..22.13 rows=1845 width=0) (actual time=0.250..0.250 rows=1842...)	
13	Index Cond: (upvotes > 10)	
14	-> Limit (cost=0.29..1.34 rows=10 width=21) (actual time=0.035..0.041 rows=10 loops=1)	
15	-> Index Scan Backward using idx_users_reputation on users u2 (cost=0.29..4223.71 rows=40325 width=21) (actual time...)	
16	Planning Time: 1.123 ms	
17	Execution Time: 0.652 ms	
Total rows: 17 of 17		Query complete 00:00:00.157

```
CREATE INDEX idx_users_upvotes ON users (UpVotes);
CREATE INDEX idx_users_downvotes ON users (DownVotes);
CREATE INDEX idx_users_reputation ON users (Reputation);
```

The cost in this case was cut drastically by about $\frac{1}{3}$, with the execution time being 0.6 ms vs 35 ms.

4 NoSQL Databases: We have chosen couchbase as our NoSQL system

5 Analyzing Big Data Using NoSQL Systems:

5a) The data set used for NoSQL is the same as the one used for the SQL part.

5b)

Data model(document based):

5c)

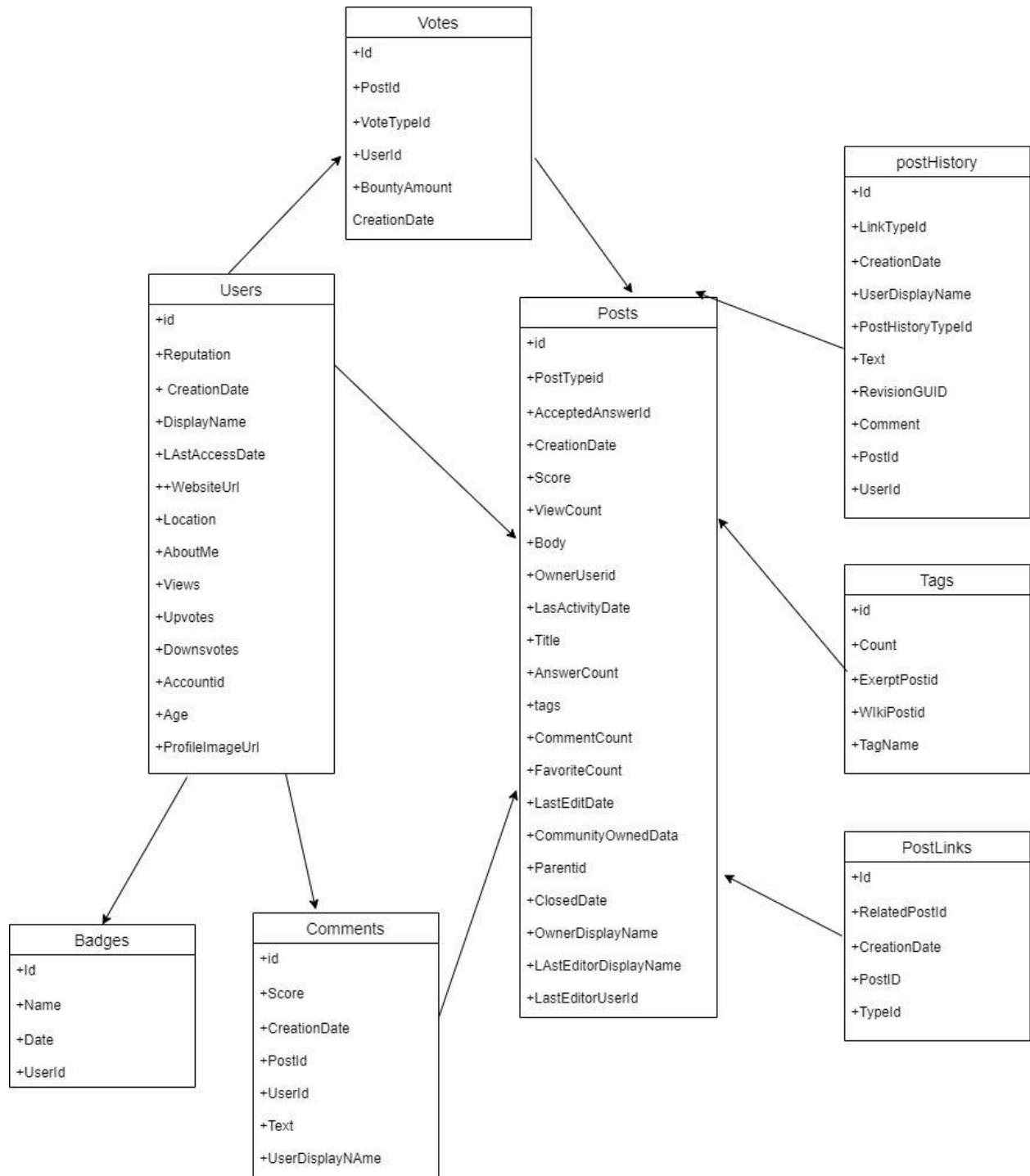
The NoSQL database was created using Couchbase cloud.

To create the database based on Couchbase's Bucket → Scope → Collection model, a 'stackoverflow' bucket was first created. Since there was no concern for accessibility, one '_default' scope was created (scopes are similar to 'views' in SQL - meaning certain scopes have certain data accessibilities). The importing of the data through collections is explained in the following question.

5d)

The data set was imported to Couchbase through their online GUI. Each csv file had a corresponding collection that was created beforehand with the same name. Following this, each csv file had the following import procedure: 'Import' → 'Choose a bucket' (stackoverflow) → 'Next' → 'CSV' → 'Using your web browser' → dragging and dropping CSV file to dedicated drop location → 'Next' → Selecting '_default' scope & collection corresponding to CSV name.

5-Couchbase



5e) 10 different queries

1) Top 10 users who have received the most badges

```

SELECT u.Id, u.DisplayName, COUNT(b.Id) AS BadgeCount
FROM users AS u
JOIN badges AS b ON u.Id = b.UserId
GROUP BY u.Id, u.DisplayName
ORDER BY BadgeCount DESC
LIMIT 10

```

BadgeCount	DisplayName	Id
456	whuber	919
318	Glen_b	805
282	chl	930
256	gung	7290
222	Jeromy Anglim	183
176	Macro	4856
159	Tal Galili	253
156	Rob Hyndman	159
152	Peter Flom	686
139	cardinal	2970

Execution: 5.77 minutes

2) Retrieve 10 comments made by users on posts with a score of more than 50

```

SELECT c.* FROM comments c
JOIN posts as p ON p.Id = c.PostId
WHERE p.Score > 50
LIMIT 10;

```

CreationDate	Id	PostId
2010-07-27 15:22	669	1
2010-07-20 8:03	129	10
2010-07-20 15:06	183	10
2010-07-21 0:07	229	10
2010-07-21 0:26	235	10
2014-05-25 15:48	193651	100000
2014-05-25 14:51	193644	100000
2014-05-25 14:10	193635	100000
2014-05-25 14:00	193631	100000
2014-05-26 22:08	193833	100001

Execution: 25ms

3) Top 5 posts with their id that have the most votes

```

SELECT p.Id, p.DisplayName, COUNT(v.Id) AS vote

```

```

FROM posts AS p
JOIN votes AS v ON p.Id = v.UserId
GROUP BY p.Id, p.DisplayName
ORDER BY vote DESC
LIMIT 5

```

Id	vote
14730	712
805	509
253	459
930	457
5003	404

Execution: 41 seconds

4) Select 10 users who are from Canada

```

SELECT u.*
FROM users AS u
WHERE u.Location LIKE '%Canada%'
LIMIT 10;

```

AboutMe	AccountId	Age	CreationDate	DisplayName	DownVotes	Id	LastAccessDate	Location	ProfileImageUrl	Reputation	UpVotes	Views	WebsiteUrl
Currently goir	8194	25	2010-08-01 9:11	WalterJ89	0	640	2014-04-20 19:45	Abbotsford, t	null	101	0	1	http://www.gys
null	509459	36	2010-07-19 19:42	DaRob	0	92	2011-12-06 23:10	Acton Vale, t	null	31	3	5	null
null	509940	null	2011-05-06 19:32	Anne	0	4498	2011-08-24 1:00	Alberta, Can	null	242	0	36	null
<p>(my <em	39355	26	2010-09-02 23:41	AASoft	0	1161	2010-09-16 4:09	British Colur	null	101	0	0	null
<p>I am a Ge	1025180	null	2014-08-28 17:02	MyCarta	0	54871	2014-09-02 1:47	Calgary, Car	https://www.gravata	1	0	1	http://mycarta.v
null	154985	34	2013-09-04 14:34	Calgary Coder	0	29914	2014-06-28 3:13	Calgary, Car	null	63	8	4	http://cbaxter.gi
I am a websit	3746	32	2010-08-02 21:40	Darryl Hein	0	669	2010-08-02 21:40	Calgary, Car	null	101	0	1	http://www.xmr
<p>Tweet m	68837	30	2013-07-08 21:33	Pent	0	27778	2014-05-10 16:52	Calgary, Car	null	101	0	0	http://www.emc
null	1949203	17	2013-11-02 5:01	Lowerison	0	32206	2014-08-08 16:24	Calgary, Car	null	1	0	0	null
<p><a href="	306012	null	2014-07-17 20:15	Sebastian Patten	0	52296	2014-08-12 22:01	Calgary, Car	https://www.gravata	101	1	0	http://reverseec

Execution: 76 milliseconds

5)
Top 5 users who commented the most

```

SELECT u.Id, u.DisplayName, COUNT(c.Id) AS Comments
FROM users AS u
JOIN comments AS c ON u.Id = c.UserId
GROUP BY u.Id, u.DisplayName
ORDER BY Comments DESC
LIMIT 5

```

Comments	DisplayName	Id
13220	whuber	919
8818	Glen b	805
3870	gung	7290
3108	Peter Flom	686
3069	cardinal	2970

Execution: 32 seconds

6)top score comments

```
SELECT c.Score, c.Text, c.PostId
FROM comments AS c
ORDER BY c.Score DESC
LIMIT 5;
```

PostId	Score	Text
7224	90	+1 for filtering out justin bieber
103345	9	How do you know it's normal? It doesn't look normal to me. It looks left-skew
44445	9	This is an interesting and maddening question. Any student that has taken a
11446	9	Actually we could do better than this ... using AB testing we could pick to disp
455	9	Statistical voyeurism? And there we were wondering what to call the site...

Execution: 7.5 seconds

7) FIND OLDEST USERS

```
SELECT u.DisplayName, u.Age
FROM users as u
WHERE Age <> "null"
ORDER BY Age DESC
LIMIT 10
```

Age	DisplayName
94	PherricOxide
94	blochwave
94	Scott
94	sharoz
94	Jay
94	hb20007
94	dannyla
94	1.01pm
94	David K
94	Pio

Execution: 1.6 seconds

8)

Most commented posts

```
SELECT p.Title,p.Id, COUNT(c.Id) AS Comments
FROM posts AS p
JOIN comments AS c ON p.Id = c.PostId
GROUP BY p.Title,p.Id
ORDER BY Comments DESC
```

LIMIT 10

Comments	Id	Title
45	31038	null
41	92246	null
41	2365	null
37	6605	null
37	73613	Does the presence of an outlier increase the probability that another outlier will also be present on the same observation?
35	30160	null
35	86015	Amazon interview question—probability of 2nd interview
34	88453	null
33	62237	How do I incorporate an innovative outlier at observation 48 in my ARIMA model?
33	93530	Which test is better suited to compare averaged versus single data sets

Execution: 65 seconds

9) Find the content of the comment with the lowest score

```
SELECT c.Score, c.Text
FROM comments AS c
ORDER BY c.Score ASC
LIMIT 1
```

10) Most used badges

```
SELECT badges.Name, COUNT(users.Id) as NumberofUsers
FROM badges
JOIN users ON badges.UserId=users.Id
GROUP BY badges.Name
ORDER BY NumberofUsers DESC
LIMIT 20
```

Name ▼	NumberOfUsers ▼
Student	14847
Supporter	8828
Editor	8453
Scholar	6523
Teacher	4679
Popular Question	4536
Autobiographer	4480
Tumbleweed	3268
Nice Answer	2570
Yearling	2293
Commentator	2047
Notable Question	1725
Nice Question	1654
Informed	1537
Revival	1018
Custodian	1001
Curious	950
Critic	858
Enlightened	733
Citizen Patrol	656

Execution: 36 seconds

5f)

Investigate the balance between the consistency and availability in your NoSQL system. When it comes to distributed databases there are some limitations like consistency and availability. Consistency refers to the fact that every node needs to always see the same data value at any given time. In our NoSQL system, Couchbase, we can see that everytime we run our queries our outputs are always consistent. Availability is another important aspect. In the realm of databases, availability means that the system needs to continue to operate even if its nodes get into a cluster crash. Since our NoSQL system, Couchbase, uses cloud services, we are sure that it will be available even after a hardware/software crash, since everything is backed up. Document based datasytem, couchbase, allows us to have great balance between these 2 important aspects: availability and consistency.

5g)

Indexing is vital for Couchbase as it is the only way to link collections to one another. For this reason, the primary and foreign keys in the table creation queries for the SQL counterpart of this dataset found above may be referred to by their primary and foreign key constraints when creating indexes for Couchbase. This translates to the following queries:

```
//users
```

```
CREATE INDEX idx_id ON users(Id);
```

```
//posts
```

```
CREATE INDEX idx_id ON posts(Id);
```

```
CREATE INDEX idx_lastEditorUserId ON posts(LastEditorUserId);
```

```
CREATE INDEX idx_ownerUserId ON posts(OwnerUserId);
```

```
CREATE INDEX idx_parentId ON posts(ParentId);
```

```
//postLinks
```

```
CREATE INDEX idx_id ON postLinks(Id);
```

```
CREATE INDEX idx_postId ON postLinks(PostId);
```

```
CREATE INDEX idx_relatedPostId ON postLinks(RelatedPostId);
```

```
//tags
```

```
CREATE INDEX idx_id ON tags(Id)
```

```
CREATE INDEX idx_excerptPostId ON tags(ExcerptPostId)
```

```
//badges
```

```
CREATE INDEX idx_id ON badges(Id)
```

```
CREATE INDEX idx_userId ON badges(UserId)
```

```
//comments
```

```
CREATE INDEX idx_id ON comments(Id)
```

```
CREATE INDEX idx_postId ON comments(PostId)
```

```
CREATE INDEX idx_userId ON comments(UserId)
```

```
//postHistory
```

```
CREATE INDEX idx_id ON postHistory(Id)
```

```
CREATE INDEX idx_postId ON postHistory(PostId)
```

```
CREATE INDEX idx_userId ON postHistory(UserId)
```

```
//votes
```

```
CREATE INDEX idx_id ON votes(Id)
```

```
CREATE INDEX idx_postId ON votes(PostId)
```

```
CREATE INDEX idx_userId ON votes(UserId)
```

Additionally, when querying collections on attributes like users.Location found in query 4, an index on users.Location needs to be created, otherwise an error may be thrown. The index is as follows:

```
CREATE INDEX idx_location ON users(Location)
```

This need to create an index for many queries highlights a potential drawback of Couchbase: since each index requires more storage space, having too many may consume a lot more storage.