1. Take a look at the load script you ran in the instructions (`checkedtweets_schema .sqlpp`) Which of the dataset(s) in AsterixDB can be classified as ***not*** being in 1NF based on the DDL that you've been given? For this question only, we will consider the schema'ed data, i.e., the data stored in the AsterixDB dataverse and datasets that have predeclared ADM data types. Identify and list the 1NF and non-1NF datasets' name(s) and briefly explain your answer(s). (*Reminder*: 1NF = Flat atomic-valued relational data.)

1NF: EvidenceFrom

non-1NF: Verification (because of list of evidences in it), RawTweet (because of list of hashtags in it), User (because of list of expertizes and phones in it), Evidence (because list of about)

2. Repeat **step 0** (the setup/loading step) from the instructions document, but this time use the **schemaless** version of CheckedTweets.org. The schemaless version is the load script named `checkedtweets _schemaless.sqlpp`. The data file is the same one as in step 0, but you will need to change the 'USE' statement at the top of the file to say `USE checkedtweets_schemaless;`. Examine and compare the DDL of both schema versions. How is the DDL for the schemaless version different from the DDL for the schema version in terms of its number and types of attributes? Is there any difference in the way the data appears in the two versions? (**Note**: Again, the dataverse that contains the schema version is named `checkedtweets_schema` while the schemaless one is named `checkedtweets _schemaless`.)

In schemaless version, all the fields have only one attributes which is their primary keys.

3. Looking back at the E-R diagram from the CheckedTweets.org conceptual design, compare and contrast the relational (MySQL) version of the schema from the past SQL HW assignments with the AsterixDB schema here (the one given in the schema DDL version). You will find that we have made some rather different design decisions here in the NoSQL database case. To help make it clear how things have been mapped for SQL++/AsterixDB, the schema DDL version includes comments that indicate which E-R attributes correspond to which data in the raw JSON tweets. (And of course you may also recall that information from your trigger-writing work a few HW's ago.) Very briefly explain, after looking at the AsterixDB schema and after also exploring the data (e.g., by looking at the DDL statements in the script and running exploratory queries like "`SELECT VALUE u FROM User u LIMIT 20;`" and similarly for Verification), how we have captured the information from each of the following E-R entities differently in AsterixDB and what the benefit(s) of this new design probably are:

User:
AsterixDB: We have the details of a checker user inside of User and and a User can be identified as Checker with a single field 'kind'. The expertise and phone of Users are in the same table and they are multivalued. It has a composite field named address.
MySQL version: We don't have any multi-value field or composite one. The checker has a separate table.

***Verification**:*
The only difference here is that in Asterix DB, we have muti-valued attribute 'Evidence' in Verification.
4. [7 pts] *Use the dataverse* `checkedtweets_schema` *for problems 4 and beyond.* For checkers with the first name "Christopher" and the last name "Smith", list their user ids, their addresses, and the dates they started being a checker. (Hint: A checker is a user of the *kind* "CHECKER"). [Result size: 2]
Sample output:

```
{ "user_id": 2401, "address": { "country": "United States of America",
"state": "WI", "city": "Larsen" }, "checker_since": "2020-01-
20T20:43:20.000Z" }
```

Query:

```
USE checkedtweets_schema;
SELECT U.user_id AS id, U.address AS Address,
U.checker_since
From User U
WHERE U.name.first="Christopher" AND U.name.last="Smith";
```

Result:

```
{ "id": 2401, "Address": { "country": "United States of America", "state": "W
I", "city": "Larsen" }, "checker_since": "2020-01-20T20:43:20.000Z" }
{ "id": 181, "Address": { "country": "United States of America", "state": "AR
", "city": "Horseshoe Bend" }, "checker_since": "2020-08-14T21:26:42.000Z" }
```

5. For users that live in the city "Upham" or "East Megan", write a query to get the users' user_id, full name, email, and the evidence URLs they have submitted. [Result size: 2]
Sample output:

```
{ "user_id": 136, "first": "Nathaniel", "last": "Schneider", "email":
"schneider.nat@gmail.com", "urls": [ "https://www.healthy-habits.fr" ] }
```

Query:

```
SELECT U.user_id AS user_id, U.name.first AS first, U.name.last
AS last, U.email AS email, (SELECT VALUE  E.url
FROM Evidence AS E, EvidenceFrom EF
WHERE EF.user_id=U.user_id AND EF.ev_id=E.ev_id) AS urls
From User U
WHERE U.address.city="Upham" OR U.address.city="East Megan";
```

Result:

```
{ "user_id": 136, "first": "Nathaniel", "last": "Schneider", "email": "schnei
der.nat@gmail.com", "urls": [ "https://www.healthy-habits.fr" ] }
{ "user_id": 21, "first": "Cynthia", "last": "Stein", "email": "Stein.cynthia
47@gmail.com", "urls": [ "https://health.kh" ] }
```

Now try the same query on the **schemaless** version of the CheckedTweets.org dataverse (**'checkedtweets_schemaless'**). Did it work? Were the results different? What does this tell you about querying typed versus untyped data in SQL++?

It works and there is no change in the result of the query. We can query for non-defined attributes the same way that we would the predefined attributes.

6. Write a query to print the tweet_ids and their corresponding hashtag texts for tweets where the number of hashtags used in the tweet is greater than 23. [Result size: 1]
Hints:

- See the documentation (and lecture notes) for array_count().
- Check out the LET-clause in AsterixDB's documentation, as it may help simplify your query. You do not need to repeat yourself with SQL++ :-)
- Remember that the result of a subquery in SELECT is always an array! (Life gets easier once you remove the straight-jacket of 1NF :-))

A similar output:

```
{ "id": "1321194418041458688", "ht_list": [ "voting", "vote", …, "COVID19"
]}
** That's not the output, but it's what your output should look like :-)
```

Query:

```
SELECT T.id AS id, ht_list
From RawTweet T
LET ht_list = (SELECT VALUE ht.text
FROM T.extended_tweet.entities.hashtags ht)
WHERE ARRAY_COUNT(T.extended_tweet.entities.hashtags)>23;
```

Is your query result in 1NF? Why or why not?
No, because it has multivalued items.

Result:

```
{ "ht_list": [ "Trump", "GOP", "Georgia", "Florida", "Texas", "Alaska", "Arizona", "Pennsylvania", "Michigan", "Ohio", "Maine", "Louisiana", "Mississippi", "SouthCarolina", "NorthCarolina", "Kansas", "Tennessee", "Kentucky", "Wisconsin", "Minnesota", "Colorado", "Missouri", "Oklahoma", "Idaho", "VoteBlue" ], "id": "1321199733621379076" }
```


7. Write a query that prints the tweet_ids and a list of ver_ids for those tweets that have been posted after "2020-08-10 00:00:00", that have been verified more than once (i.e., the size of ver_ids is greater than 1), ***and*** that also contain the hashtag COVID19 (all uppercase letters). [Result size: 4]

Hints:

- Check out the lecture and docs about the existential SOME clause. You might find it useful. :-)

Sample output:


```
{ "id": "1321210168722468864", "ver_ids": [ 660, 80 ] }
```


Query:

```
SELECT T.id AS tweet_id, ver_ids
From RawTweet T
LET ver_ids = (SELECT VALUE V.ver_id  FROM Verification V WHERE V.tweet_id=T.id)
WHERE T.created_at >= ('2020-08-10 00:00:00')
AND 'COVID19' IN (SELECT VALUE ht.text
FROM T.extended_tweet.entities.hashtags ht) AND
ARRAY_COUNT(ver_ids)>1;
```

Result:


```
{ "tweet_id": "1321202970898251776", "ver_ids": [ 46, 401 ] }
{ "tweet_id": "1321205651348082693", "ver_ids": [ 61, 490 ] }
{ "tweet_id": "1321207379313250304", "ver_ids": [ 66, 546 ] }
{ "tweet_id": "1321210168722468864", "ver_ids": [ 80, 660 ] }
```


8. Write a query that analyzes hashtag popularity. It should print all of the ***normalized*** hashtags (e.g., Covid19, CoViD19 and coviD19 should be normalized to covid19, and similarly for other hashtags) and the number of distinct Tweeters who used each hashtag. Order your result in descending order and limit the number of results to the top five. [Result size: 5, of course]

Hints:

- Check the UNNEST clause here and/or the shorthand for UNNEST as a join clause here.

Sample output:


```
{ "hashtag": "vote", "cnt": 225 }
```


Query:

```
SELECT lower(ht.text) AS hashtag, COUNT(DISTINCT T.user.id_str)
AS CNT
From RawTweet T, T.extended_tweet.entities.hashtags AS ht
GROUP BY lower(ht.text)
ORDER BY COUNT(DISTINCT T.user.id_str) DESC
LIMIT 5;
```

Result:

```
{ "hashtag": "trump", "CNT": 253 }
{ "hashtag": "vote", "CNT": 225 }
{ "hashtag": "bidenharris2020", "CNT": 158 }
{ "hashtag": "election2020", "CNT": 142 }
{ "hashtag": "biden", "CNT": 117 }
```

Is the result in 1NF? Why?

Yes, because all tuples are atomic.

9. Write a query that returns the handles of tweeters and their number of Covid-tagged tweets for those tweeters who've used the hashtag "**covid19**" more than 3 times. Your query should normalize the hashtags to lowercase (e.g., **Covid19** should be converted to **covid19**) in order to properly consider all Covid-tagged tweets. [Result size: 2]
Sample output:

```
{ "cnt": 5, "handle": "ppl4justice" }
```

Query:

```
SELECT T.user.screen_name AS handle, COUNT(*) AS cnt
FROM RawTweet T
WHERE SOME ht IN T.extended_tweet.entities.hashtags SATISFIES
lower(ht.text)='covid19'
GROUP BY T.user.screen_name
HAVING COUNT(*)>3;
```

Result:

```
{ "handle": "ppl4justice", "cnt": 5 }

{ "handle": "CupofJoeintheD2", "cnt": 5 }
```

If you wanted to answer this question using the HW6 tables in MySQL, what are the tables that you would need? Do you find the SQL++ query easier or harder, and why? (Answer in <= 3 sentences.) Note that the last part of the question has no right answer. (We just want to get your opinion :-))

We would then need Tweet, Tweeter, hashtags tables. SQL++ is easier because we don't need to join and we can take advantage of the fact that all information is stored in RawTweet dataset.

10. Write a query that, for those tweets that have been verified using more than 12 URLS, prints the tweets' texts and the URLs used to verify those tweets. Your result could have the same URL appear multiple times, i.e., not distinct (e.g., *"http://www.fact-checked.hr"* could appear twice in *urls*) and that is expected. [Result size: 2]
Sample output:

```
{ "urls": [ "https://minnesota-ballots.gov", … "https://www.rhode-island-
ballots.gov" ], "text": "@Nigel_Farage China Will Attack Russia In The
2020s   Article ➡ https://t.co/1xWWXIZDoR\n\n#Nxivm #COVID19...
https://t.co/r6aeldNd6d" }
```

[7 pts] Query:

```
SELECT T.text AS Text,  urls
FROM RawTweet T
LET urls = (SELECT VALUE E.url
FROM Verification V, V.evidence ev, Evidence E
WHERE T.id = V.tweet_id
AND ev = E.ev_id)
WHERE ARRAY_COUNT(urls)>12;
```

[3 pts] Result:

```
{ "Text": "#Biden Says #Trump pick Barrett Supreme Court Confirmation Threate
ns #ACA - Newsweek 😁 #election2020 #Texas… https://t.co/wcxIwD6P0M", "urls":
[ "https://www.georgia-election.org", "https://vermont-vote.gov", "http://ala
ska-ballots.gov", "https://www.colorado-vote.org", "https://www.maryland-elec
tion.org", "http://www.pennsylvania-ballots.gov", "https://www.montana-electi
on.gov", "https://alabama-election.gov", "https://louisiana-ballots.gov", "ht
```

tp://www.new-mexico-election.gov", "https://minnesota-ballots.gov", "http://www.ohio-election.gov", "https://utah-vote.org"] }

{ "Text": "@Nigel_Farage China Will Attack Russia In The 2020s Article ➡ https://t.co/1xWWXIZDoR\n\n#Nxivm #COVID19… https://t.co/r6aeldNd6d", "urls": ["https://vermont-vote.gov", "https://health.cu", "https://www.rhode-island-ballots.gov", "https://www.washington-ballots.gov", "https://vermont-vote.gov", "https://www.montana-election.gov", "https://health.cu", "https://www.rhode-island-ballots.gov", "https://minnesota-ballots.gov", "http://www.ohio-election.gov", "http://oklahoma-vote.gov", "http://www.fact-checked.es", "https://minnesota-ballots.gov"] }