Description of Queries for ReCache Evaluation

For presentation purposes only, when dealing with JSON data, we use the PostgreSQL JSON extensions (https://www.postgresql.org/docs/9.5/static/functions-json.html).
PostgreSQL treats JSON as an explicit data type, therefore field manipulation of a JSON object requires overloaded constructs. In the following query, `(obj->>'x')::int` accesses field x of a JSON object, and treats it as an integer.

Performing unnesting of a JSON array with PostgreSQL requires using the function json_array_elements, as well as a nested query to continue manipulation of the results. In contrast, Proteus and ReCache express unnesting using a single flat query.

Spam Analysis Queries

For legal reasons, we are not allowed to disclose the exact workloads run on the Symantec spam dataset. Nevertheless, this section presents some indicative SQL queries used for which table and field names have been anonymized. In the FROM clause of the following SQL queries, "csv" and "json" indicates that the query operates over JSON data and CSV data respectively.

Queries over CSV data

```
SELECT COUNT(*)
FROM csv
WHERE f1 < val1 AND f2 < val2;

SELECT MAX(f1), MAX(f2)
FROM csv
WHERE f2 < val1 AND f3 < val2;
```

Queries over JSON data

An example of a query accessing only non-nested fields:

```
SELECT MAX(json->>'f1'::int),  MAX(json->>'f2'::int)
FROM json
WHERE (json->>'f2')::int < val1 AND (json->>'f3')::int < val2;
```

An example of a query that accesses both nested and non-nested fields:

```
SELECT count(*)
FROM (
      SELECT json_array_elements((json->>'x')::json) as x1,
      json_array_elements((json->>'y')::json) as x2)
        FROM json
        WHERE (json->>'z')::int < val1 ) internal;
```

Joins across CSV and JSON data

```
SELECT count(*)
FROM csv
JOIN (
      SELECT (json->>'z')::int as f,
            json_array_elements((json->>'x')::json) as x1,
            json_array_elements((json->>'y')::json) as x2
        FROM json
        WHERE (json->>'z')::int < val1 ) internal
ON (csv.f1 = internal.f)
;
```

Queries on the Yelp dataset

We run queries on the **business**, **review** and **user** datasets included in the Yelp dataset challenge (https://www.yelp.com/dataset/documentation/json). Only the **business** and **user** contain nested collections. In query workloads that specify exact percentages of nested attributes, we typically include all three files with a probability of 33%. However, if a 33% probability makes it impossible to achieve the required percentage of nested attributes, we do not include the **review** file and instead include the other two files with a probability of 50%.

The queries for each of these files are described below.

Queries on the business dataset

For the business dataset, our queries touch the **stars**, **review_count**, **city**, **attributes**, **hours** and **categories** fields. We use the following subset of possible cities and categories in equality predicates involving **city** and **categories**:

Cities: Edinburgh, Phoenix, Toronto, Montreal, Waterloo, Pittsburgh, Charlotte, Cleveland, Urbana, Champaign
Categories: Hotels, Food, Bakeries, Shopping, Restaurants, Pets, Churches, Indian, Italian, Mediterranean

Queries on the business dataset have one of the following two templates:

```
1)    SELECT MAX(business->>'stars'), MAX(business->>'review_count')
      FROM (SELECT
                  json_array_elements((business->>'attributes')::json) as
            attributes,
                  json_array_elements((business->>'categories')::json) as
            categories,
                  json_array_elements((business->>'hours')::json) as hours
            FROM business
            WHERE predicate1 AND predicate2) internal;


2)    SELECT internal.attributes, internal.categories, internal.hours,
business->>'stars', business->>'review_count', business->>'city'
```

```
FROM (SELECT
            json_array_elements((business->>'attributes')::json) as
      attributes,
            json_array_elements((business->>'categories')::json) as
      categories,
            json_array_elements((business->>'hours')::json) as hours
      FROM business
      WHERE predicate1 AND predicate2) internal;
```

The second template requires the query engine to process both nested and non-nested columns of all records that would be generated if the data was flattened into relational form and which satisfied the two predicates. The first query, on the other hand, only unnests the nested collections to allow further query processing. If the predicates or aggregates do not reference a nested collection, the collection is not subsequently accessed.

In query workloads that specify exact percentages of nested attributes, we pick the predicates only from the non-nested attributes. This allows us to pick template 1 for queries that access only non-nested attributes and template 2 for queries that access only nested attributes.

Predicates are of the form: `field < integer or field = 'string'`, where the integer is a random number less than the maximum possible value for that field. The first form is chosen either for **stars** (maximum value 5) or **review_count** (maximum value 20000). The second form is used for **city** and **categories** and a string is chosen randomly from the list of options given above**.**

In the final experiment on the Yelp dataset, 33% of queries are on the business dataset. For each template, there is a 50% probability it will be used in a query. For each of **stars**, **review_count**, **categories** and **city**, there is a 25% probability it will be used as a predicate.

Queries on the review dataset

For the review dataset, our queries touch the **stars**, **useful**, **cool** and **funny** fields. The review dataset does not contain any nested fields.

Queries on the business dataset have one of the following two templates:

```
1)    SELECT MAX(review->>'stars'), MAX(business->>'useful')
      FROM review
      WHERE predicate1 AND predicate2;

2)    SELECT COUNT(*)
      FROM review
      WHERE predicate1 AND predicate2;
```

Predicates are of the form: `field < integer`, where the integer is a random number less than the maximum possible value for that field. For **stars,** the maximum value is 5. For the remaining fields, we use 1000.

In the final experiment on the Yelp dataset, 33% of queries are on the review dataset. The probability of using the first or second template in a query is 10% and 90% respectively. For each of the **stars**, **useful**, **cool** and **funny** fields above, there is a 25% probability it will be used as a predicate.

Queries on the user dataset

For the user dataset, our queries touch the **review_count**, **fans**, **compliment_funny**, **compliment_cool** and **elite** fields. For the **elite** field, we use randomly chosen years between 2010 and 2016 inclusive.

Queries on the business dataset have one of the following two templates:

```
1)    SELECT MAX(user->>'stars'), MAX(user->>'review_count')
      FROM (SELECT
                  json_array_elements((user->>'elite')::json) as elite
            FROM user
            WHERE predicate1 AND predicate2) internal;


2)    SELECT internal.elite, user->>'fans', user->>'review_count',
user->>'compliment_funny', user->>'compliment_cool'
      FROM (SELECT
                  json_array_elements((user->>'elite')::json) as elite
            FROM user
            WHERE predicate1 AND predicate2) internal;
```

As with the **business** dataset, the second template requires the query engine to process both nested and non-nested columns of all records that would be generated if the data was flattened into relational form and which satisfied the two predicates. The first query, on the other hand, only unnests the nested collections to allow further query processing. If the predicates or aggregates do not reference a nested collection, the collection is not subsequently accessed.

In query workloads that specify exact percentages of nested attributes, we pick the predicates only from the non-nested attributes. This allows us to pick template 1 for queries that access only non-nested attributes and template 2 for queries that access only nested attributes.

Predicates are of the form: `field < integer or field = 'year'`, where the integer is a random number less than the maximum possible value for that field. The first form is chosen for **fans** (maximum value 1000), **review_count** (maximum value 11000), **compliment_funny** (maximum value 100) or **compliment_cool** (maximum value 1000). The second form is used for **elite** and the string representation of a random year from 2010-2016 is chosen as the equality predicate**.**

In the final experiment on the Yelp dataset, 33% of queries are on the user dataset. The probability of using the first or second template in a query is 10% and 90% respectively. For each of the **review_count**, **fans**, **compliment_funny**, **compliment_cool** and **elite** fields, there is a 20% probability it will be used as a predicate.