Madison Data Model in PostgreSQL

Design Decisions

The database model as designed conforms to the relational model, rather than the NoSQL-like JSON based design using documents and collections. In the following sections, I will prove with evidence why traditional relational design is a better choice for *this* system.

But first the data model description.

Data Model

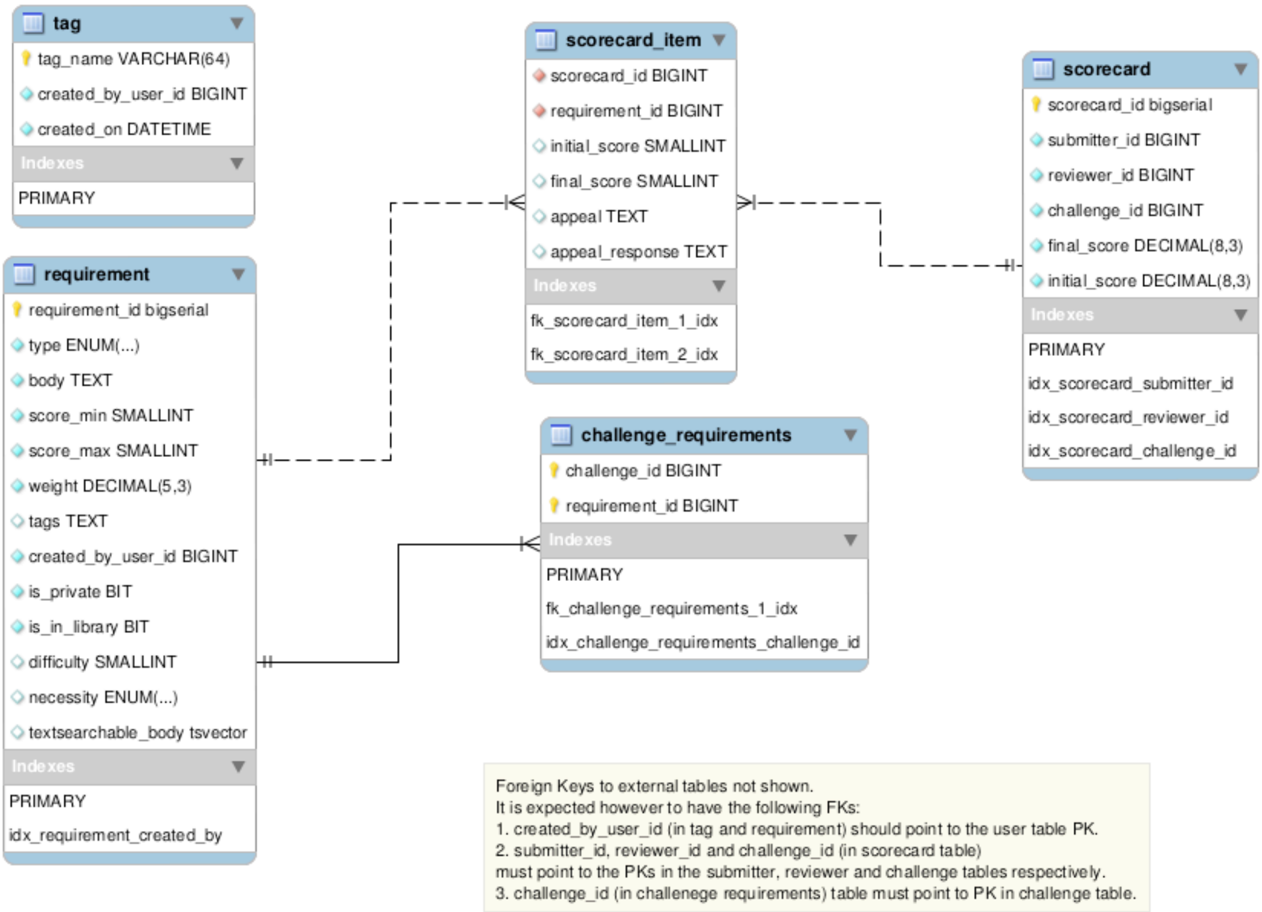

TABLE requirement

This table holds one row for each requirement.

- requirement_id is an auto-generated primary key.
- type is the requirement type and can be one of {'Functional', 'Technical', 'Other', 'Informational'}
- body is the user-enetered text of the requirement.

- score_min and score_max are the range of the scores possible. For boolean (yes-no) requirements this range can simply be 0,1.
- weight is the weightage of the requirement when calculating the entire score for the scorecard. It defaults to 1. Note that it is different from the difficulty field which is optional and is also not used for calculating score.
- tags is a space-separated list of tags that are applicable to the requirement. In relational terms, it would be better to have a separate table to model the 1:N requirement-tag relation. However the tags stored in this way are more amenable to full-text search (more on this later).
- create_by_user_id is the id of the user who created the requirement.
- is_private is whether the requirement is private to the user who created the requirement. The requirement will only show up in this particular user's search results.
- is_in_library is whether the requirement belongs to the searchable requirements library i.e. only requirement with is_library = true will show up in the search results.
- difficulty is the difficult level of the contest. Can be from 1 to 5.
- necessity is the requirement necessity and can be one of {'Must', 'Should', 'Nice', 'Optional'}
- textsearchable_body is a Postgres tsvector type. It is the body split into its constituent words (after lexing and stemming), so that full-text search can be run on the body in very efficient manner.

TABLE challenge_requirement

This table holds the requirements for a given challenge. There is a 1:N relation from challenge to requirement. It stores the equivalent of 'scorecard templates'.

TABLE scorecard

This table is an actual scorecard instance. It contains the challenge for which the scorecard exists, the reviewer who has performed the review and the submitter who has been reviewed.

scorecard_id is an auto-generated primary key.

initial_score is the total score before appeals. final_score is total score post appeal response. These fields are convenient fields so that we do not have calculate the score every time using the individual scores for requirements.

TABLE scorecard_item

This table is a contains each requirement for a scorecard and the score for that requirement. There is a 1:N relation for scorecard_id to requirement_id here.

- scorecard_id is the scorecard.
- requirement_id is the requirement for which the score exists.
- initial_score is the initial score given in this scorecard for this requirement (pre appeals)
- final_score is the final score given in this scorecard for this requirement (post appeal responses)
- appeal is the appeal text.

- appeal_response is the appeal response text.

TABLE tag

This table contains tags, their description and the user who created them.

Notes:

- It is expected that when user creates a new requirement, or takes an existing requirement and modifies it, then in either case, a new requirement will be created in the database.
- Tags cannot contain space they are stored in space-separated manner. This is quite common for example on stackoverflow.
- When a user creates a requirement and explicitly saves to library then the is_in_library field becomes true. However if user creates a requirement and just uses it for one challenge, then this field is false (this is assumed to be default behavior).

Design Decision Reasoning

The database model as designed conforms to the relational model, rather than the NoSQL-like JSON based design using documents and collections.
The main reason here is that the database is rather small (< 100M rows). JSON based designs are favorable when the database is much bigger (at least 100M rows or higher) or when it I supposed to scale horizontally infinitely.

The trade-off between relational and NoSQL/JSON designs is mainly amongst the following:

- Indexes on JSON fields are slower than normal column indexes. Accessor functions on JSON can be quite slow (see sources at the end)
- Joins on huge tables can be quite slow

In the case of small databases, the former affects performance more because joins in smaller tables are not that slow. And so relational is a better choice.
In the case of large databases, the latter affects performance more. And so JSON is a better choice.

One other minor drawback of using JSON is that it cannot be migrated easily (as is noted in the requirements regarding Salesforce).

Actual Evidence

I have generated test data based on the numbers quoted in the requirements.

20K challenges per year for 10 years = 200K challenges

10 requirements per challenge = 200K * 10 = 2M requirements

3 reviewers and 3 submitters per challenge = 200K * 9 = 1.8M scorecards

10 requirements per scorecard = 1.8M * 10 = 18M scorecard items

```
postgres=# select count(*) from requirement;
  count
---------
 2000000
(1 row)

postgres=# select count(*) from challenge_requirements;
  count
---------
 2000000
(1 row)

postgres=# select count(*) from scorecard;
  count
---------
 1800000
(1 row)

postgres=# select count(*) from scorecard_item;
  count
----------
 18000000
(1 row)

postgres=#
```

With the proper indexes, the expected queries to be run on these tables are quite fast.
Here is the query for getting all requirements for a challenge i.e. the scorecard template.

```
postgres=# select r.* from requirement r inner join challenge_requirements cr on r.requirement_id = cr.requirement_id where cr.challenge_id = 149111;
 requirement_id |    type    |                                                                                                    body
                                                            | score_min | score_max | tags | created_by_user_id | is_private | difficulty | necessity |
                                                                     textsearchable_body

----------------+------------+---------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------+-----------+-----------+------+--------------------+------------+------------+-----------+-------
----------------------------------------------------------------------------------------------------------------------------------------------------------------
-------
        1491101 | Functional | Just solved my own problem, apparently I downloaded it wrong... Thanks for the answer.
                                                            |         0 |         0 |      |                 20 | 1          |            |           | 'answe
ar':6 'download':8 'problem':5 'solv':2 'thank':11 'wrong':10
        1491102 | Functional | Could you post the xml code for the animation?
                                                            |         0 |         0 |      |                943 | 1          |            |           | 'anim'
6 'could':1 'post':3 'xml':5
        1491103 | Functional | I usually end up iterating trough the list of values and constructing the IN part kinda by hand in that way... not sure how to
HP though                                                   |         0 |         0 |      |               1160 | 1          |            |           | 'const
end':3 'hand':18 'iter':5 'kinda':16 'list':8 'part':15 'php':29 'sure':23 'though':30 'trough':6 'usual':2 'valu':10 'way':21
        1491104 | Functional | Agreed. This way lay madness.
                                                            |         0 |         0 |      |               1199 | 1          |            |           | 'agre'
 'mad':5 'way':3
        1491105 | Functional | why is a UIWebView no option? you can load it with data generated locally (use data: uri scheme for the images if you like), an
disable user interaction. What problem would then be left?  |         0 |         0 |      |               1793 | 1          |            |           | 'data'
sabl':28 'generat':13 'imag':21 'interact':30 'left':36 'like':24 'load':9 'local':14 'option':6 'problem':32 'scheme':18 'uiwebview':4 'uri':17 'use':15 'use
ld':33
        1491106 | Functional | @saarpa: See my updated answer
                                                            |         0 |         0 |      |                955 | 1          |            |           | 'answe
pa':1 'see':2 'updat':4
        1491107 | Functional | this is awesome, thanks !
                                                            |         1 |         1 |      |                200 | 1          |            |           | 'aweso
```

```
k':4
       1491108 | Functional | This is crazy, I am sure I tried this!!
                                                              +|         0 |         0 |      |              1886 | 1          |            |
:14 'sure':6 'thank':13 'tri':8 'work':12
                |            | But it works, thanks dude.
                                                               |           |           |      |                   |            |            |
       1491109 | Functional | I suspect this is not a function of the auto-complete feature itself, but rather due to the fact that chrome:// URLs are
story. So perhaps there is a way to override that behavior. |         1 |         1 |      |              1977 | 1          |            |
complet':10 'behavior':37 'complet':12 'due':17 'fact':20 'featur':13 'function':7 'histori':27 'overrid':35 'perhap':29 'rather':16 'store':25 'suspec
y':33
       1491110 | Functional | Just tried that, still no parsing of the embedded XHTML in FF 3.6.6. Will keep trying in case I misunderstood you, thoug
                                                               |         0 |         0 |      |              1032 | 1          |            |
':18 'embed':9 'ff':12 'keep':15 'misunderstood':20 'pars':6 'still':4 'though':22 'tri':2,16 'xhtml':10
(10 rows)

Time: 63.540 ms
postgres=#
```

The time for the query reported above is 63 ms. In general the time reported for this query is 50-250ms range.

Similarly here is the query which gets an entire scorecard (with scores).

```
postgres=# select * from scorecard_item si inner join requirement r on si.requirement_id = r.requirement_id where si.scorecard_id = 765432;
 scorecard_id | requirement_id | score | appeal | appeal_response | requirement_id |    type    |                                                   body
          | score_min | score_max | tags | created_by_user_id | is_private | difficulty | necessity |                          textsearchable_body

--------------+----------------+-------+--------+-----------------+----------------+------------+----------------------------------------------------
----------+-----------+-----------+------+--------------------+------------+------------+-----------+----------------------------------------------------
-----
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850480 |     1 | appeal | appeal response |         850480 | Functional | Well it depends if you count Mars twice.
          |         0 |         0 |      |                1916 | 1          |            |           | 'count':6 'depend':3 'mar':7 'twice':8 'well':1
      765432 |         850479 |     1 | appeal | appeal response |         850479 | Functional | `eval` is evil :) Use `json2.js` from http://json.org for more pea
n mind ;) |         0 |         0 |      |                1177 | 1          |            |           | 'eval':1 'evil':3 'json.org':7 'json2.js':5 'mind':12 'peac':1
se':4
(10 rows)

Time: 99.470 ms
```

The time for the query reported above is 99 ms. In general the time reported for this query is 50-150ms range.

Conclusion:

As we can see, these queries are very quick on even large number of rows, so there is no need to make it faster by using JSON to store the entire scorecard.
Also these queries are relatively simple (just one join for both), so development is easy as well.

Text Search

PostgreSQL provides full text search which is quite feature rich. It also provides the GIN (general inverted) and GIST (general search tree) indexes on TEXT fields which make the index very quick.

The GIN index is applied to the body and tags field of the requirement table. Note that the tags are stored as space-separated values, so the index can be applied to them as well.

```
CREATE INDEX pgreq_idx1 ON requirement USING gin(to_tsvector('english', body));
CREATE INDEX pgreq_idx2 ON requirement USING gin(to_tsvector('english', tags));
```

Now this column can be free-text searched. For example, we can search for all requirements with the 'postgres' in their body, using:

```
select count(*) from requirement where to_tsvector('english', body) @@ to_tsquery('english', 'postgres');
```

One improvement that can be applied to this search is to already create the tsvector of each body value and store it beforehand. For this we create a new column called textsearchable_body of type tsvector. This column will be populated with the tsvector of the body text using a trigger whenever the body is inserted or updated. And the index will now be created on this column rather than on body. (This technique is dicussed at http://www.postgresql.org/docs/9.3/static/textsearch-tables.html)

```
CREATE TRIGGER trg_req_body BEFORE INSERT OR UPDATE of body ON requirement
FOR EACH ROW EXECUTE PROCEDURE create_tsv_body();

CREATE FUNCTION create_tsv_body() RETURNS TRIGGER AS $_$
BEGIN
   NEW.textsearchable_body = to_tsvector('english', NEW.body);
   RETURN NEW;
END $_$ LANGUAGE 'plpgsql';

CREATE INDEX pgreq_idx ON requirement USING gin(textsearchable_body);
```

To test the performance of the text search, I downloaded the dump of all stackoverflow comments from https://archive.org/details/stackexchange. I used these stackoverflow comments to insert into body of requirements. As noted above, I generated 2M rows in requirement table. Here are the results:

```
postgres=# select count(*) from requirement where textsearchable_body @@ to_tsquery('english', 'delphi');
 count
-------
  2302
(1 row)

Time: 69.373 ms
postgres=# select count(*) from requirement where textsearchable_body @@ to_tsquery('english', 'haskell');
 count
-------
  1043
(1 row)

Time: 33.559 ms
postgres=# select count(*) from requirement where textsearchable_body @@ to_tsquery('english', 'postgres');
 count
-------
   412
(1 row)

Time: 13.379 ms
postgres=# select count(*) from requirement where textsearchable_body @@ to_tsquery('english', 'mysql');
 count
-------
  8511
(1 row)

Time: 307.897 ms
postgres=# select count(*) from requirement where textsearchable_body @@ to_tsquery('english', 'simple');
 count
-------
 19420
(1 row)

Time: 1285.570 ms
postgres=#
```

The time taken is proportional to the number of matching rows. Anything less than 15000 matched rows will return in less than a second.
This performance, in my opinion, is good enough on a database of 2M requirements. (at least for now).
Please also note that these results are on a machine with just 1GB RAM, so the psql process has only limited memory available. Performance on dedicated servers with 8GB RAM or higher is likely to be much better. Also the database startup parameters can also be tuned to improve on these numbers further.
In case in the future, this becomes a bottleneck, we can always use tools like Solr or Lucene, which

will give < 50ms performance for all queries.

Sources:

Full Text Search

http://blog.lostpropertyhq.com/postgres-full-text-search-is-good-enough/

http://www.slideshare.net/billkarwin/practical-full-text-search-with-my-sql

http://www.postgresql.org/docs/9.3/static/textsearch.html

Relational vs JSON

http://wiki.postgresql.org/images/b/b4/Pg-as-nosql-pgday-fosdem-2013.pdf

http://stackoverflow.com/a/18801020/354448