Writing and optimizing queries over nested arrays

In this blog we show how to write and optimize queries over data that contain nested arrays. We do so via a set of example queries and indexes over a table that models the users of an imaginary TV streaming service. In addition to presenting these examples in text here, a java application program is also provided that creates the “users” table, populates it with some sample rows, creates some indexes and executes the queries displaying their results.

Let’s start with creating the “users” table:

```
create table users(
    acct_id integer,
    user_id integer,
    info json,
    primary key(acct_id, user_id))
```

The “info” column stores the information about the shows watched by each user. The information is stored as a json document. Four sample rows are given in the attached .json files. The java program populates the table with these rows.

Next, let’s create some indexes over the table:

```
create index idx_country_showid_date on users(
    info.country as string,
    info.shows[].showId as integer,
    info.shows[].seriesInfo[].episodes[].date as string)

create index idx_country_genre on users(
    info.country as string,
    info.shows[].genres[] as string)

create index idx_showid on users(
    info.shows[].showId as integer)
    with unique keys per row

create index idx_showid_minWatched on users(
    info.shows[].showId as integer,
    info.shows[].seriesInfo[].episodes[].minWatched as integer,
    info.shows[].seriesInfo[].episodes[].episodeID as integer)
    with unique keys per row

create index idx_showid_seasonNum_minWatched on users(
    info.shows[].showId as integer,
    info.shows[].seriesInfo[].seasonNum as integer,
    info.shows[].seriesInfo[].episodes[].minWatched as integer,
    info.shows[].seriesInfo[].episodes[].episodeID as integer)
    with unique keys per row
```

The contents of these indexes for the 4 sample rows are shown in the attached .idx files. As shown in these create index statements, the indexes index fields that appear at different levels of data paths that contain nested arrays. For example, the idx_country_showid_date index indexes the showId field inside the top-level shows array and also the date field inside the episodes arrays, which are nested 2 levels deep under the show array. See <link> for more details about creating indexes on arrays.

And now, for the fun part, let’s look at some queries over the streaming data.

1. Don’t unnest that array!

People familiar with more traditional SQL on flat data may think that in order to do interesting things with arrays, the arrays have to be unnested and/or subqueries have to be employed (and indeed some other commercial NoSQL systems require unnesting and subqueries to work with arrays). Unnesting, employed in the FROM clause, is like a row-level self-join operation: assuming only one array is to be unnested, each table row is combined with each element of that array, which is contained in the row itself, to produce N tuples, where N is the size of the array. Although Oracle NoSQL supports unnesting (but not subqueries), it also includes various other features that make unnesting and subqueries unnecessary in most cases. In this section we showcase these features. Avoiding unnesting makes queries on arrays easier to write and more efficient to execute. However, some times unnesting is indeed unavoidable. We show examples of unnesting queries in the next section.

1.1. Query 1

```
select count(*) as cnt
from users u
where u.info.country = "USA" and
      u.info.shows.showId =any 16
```

The query returns the number of users in USA that have shown an interest in the show with id 16.

Notice the use of the “=any” operator here (see <link>). Using the simple “=” operator would cause a runtime error to be raised, because “=” expects each of its operands to be at most one value, but the u.info.shows.showId path expression returns all the show ids in the shows array. Instead, the “=any” acts like a “contains” here, i.e., it returns true if the shows array contains a show id with value 16.

The query could also have been written equivalently like this:

```
select count(*) as cnt
from users u
where u.info.country = "USA" and
      exists u.info.shows[$element.showId = 16]
```

Here, the $element variable iterates over the elements of the shows array and the condition $element .showId = 16 selects the shows whose id is 16. Since the path expression u.info.shows[$element.showId = 16] returns a set of shows, rather than a boolean value, the exists operator is needed to convert this set to a boolean value (returning true if the set is not empty). See <link> for the full specification of path expressions.

Both of the above (equivalent) queries uses the idx_country_showid_date index. Both of the query conditions are pushed to the index (see <link> for more examples and details about how indexes are used by queries). In fact, the index is “covering” each query, i.e., it contains all the info needed by the query, and as a result, no table rows are retrieved during execution. To confirm the use of the index and to see what conditions are pushed to it, you can display the query execution plan by running the attached java program with the -showPlan option.

Notice that this query does not access any data in arrays nested under the shows array. We included it in this blog to serve as an introduction/reminder of some of the query features (e.g., the "any" comparison operators, the exists operator, and the filtering conditions inside path expression) that we will use in the more complex queries that follow.

1.2. Query 2

```
select count(*) as cnt
from users u
where u.info.country = "USA" and
      exists u.info.shows[$element.showId = 16].
             seriesInfo.episodes[$element.date > "2021-04-01"]
```

The query returns the number of users in USA that have watched at least one episode of show 16 after 2021-04-01.

In this example, the path expression after the "exists" operator drills down to the deepest arrays and applies filtering at 2 levels. The first $element variable iterates over the shows of each user and for each show, the second $element variable iterates over the episodes of that show.

The query uses the idx_country_showid_date index. All of the query conditions are pushed to the index and the index is covering the query.

1.3. Query 3: A common mistake!

```
select count(*) as cnt
from users u
where u.info.country = "USA" and
      u.info.shows.showId =any 16 and
      u.info.shows.seriesInfo.episodes.date >any "2021-04-01"
```

Sometimes, users who wish to write Query 2, end up writing Query 3 instead. However the 2 queries are **not** equivalent! Query 3 returns the number of users in USA that have watched at least one episode of show 16 and have also watched an episode of some show (not necessarily show 16) after 2021-04-01. Basically, the 2 "any" predicates in this query are applied independently from each other.

Query 3 uses the idx_country_showid_date index. However, only one of the two "any" conditions can be pushed to the index. In this case, the showId condition is pushed because it's an equality condition whereas the date condition is a range one. The date condition must be applied on the table rows, which must be retrieved. Therefore, the index is not covering this query.

1.4. Query 4

```
select count(*) as cnt
from users u
```

```
where exists u.info.shows[$element.showId = 15].
             seriesInfo.episodes[$element.date > "2021-04-01"]
```

The query returns the number of users that have watched at least one episode of show 15 after 2021-04-01.

The “exists” condition here is the same as in Query 2, but there is no condition on the country. In this case, the query can use either idx_country_showid_date or idx_showId, and it’s not clear which index is the better choice. If it chooses idx_country_showid_date, it will do a full scan of the whole index applying the 2 conditions on each index key. As a result, it will access and filter-out many non-qualifying index keys. In contrast, all the index keys scanned by Query 2 are keys that satisfy the query conditions. If it chooses idx_showId, it will scan only the index keys with showId = 15, but it will then have to retrieve the associated table rows in order to apply the date conditions on them. So, idx_showid scans far fewer index keys than idx_country_showid_date, but idx_showId is not covering whereas idx_country_showid_date is.

Whether idx_showId or idx_country_showid_date is better depends on how selective the show id condition is. Oracle NoSQL does not currently collect statistics on index keys distribution, so it relies on a simple heuristic to choose among multiple applicable indexes. In this case, the heuristic chooses idx_showId on the assumption that the showId = 15 predicate must be a highly selective one because showId is the complete key of an index. If this turns out to be the wrong choice, users can force the use of idx_country_showid_date by using an index hint, as shown in the next example.

1.5. Query 5

```
select /*+ FORCE_INDEX(users idx_country_showid_date) */
       count(*) as cnt
from users u
where exists u.info.shows[$element.showId = 15].
             seriesInfo.episodes[$element.date > "2021-04-01"]
```

Same as Q4, but forcing the use of index idx_country_showid_date.

1.6. Query 6

```
select count(*) as cnt
from users u
where u.info.country = "USA" and
      exists u.info.shows[
        exists $element.genres[$element in ("french", "danish")] and
        exists $element.seriesInfo.episodes["2021-01-01" <= $element.date and
                                            $element.date <= "2021-12-31"]
      ]
```

The query returns the number of users in USA that have watched a French or Danish show in 2021.

The query uses the idx_country_genre index. The country and genres conditions are pushed to the index. In fact, 2 index scans will be performed: one scanning the keys with value (“USA”, “french”) and another scanning the keys with value (“USA”, “danish”). The date conditions will be applied on the table rows associated with the qualifying index keys.

Two more things are worth mentioning here.

First, the query could have benefited by an index on all three of country, genre, and episode date. If such an index existed, all the query conditions could be pushed to the index. However, it’s not possible to create such an index: in the current implementation, all the arrays indexed by an index must be nested into each other. Obviously, the genres arrays and the episodes arrays do not satisfy this constraint.

Second, the expression `exists $element.genres[$element in ("french", "danish")]` could have been written in one of following two equivalent ways:

exists $element.genres[$element = "french" or $element = "danish")]
or
$element.genres[] =any seq_concat(“french”, “danish”)

However, either of these two forms would cause the condition to be pushed to the index as a “filtering predicate” (similarly to the conditions of Query 5), not as a “start/stop” predicate that establishes the boundaries of the index scan (only the country predicate would be pushed as a start/stop predicate in this case). In other words, the IN operator is optimizable, whereas the OR operator and an “any” operator whose right operand returns more than one values are not optimizable.

1.7. Query 7

```
select u.acct_id, u.user_id,
       seq_sum(u.info.shows[$element.showId =
                            16].seriesInfo.episodes.minWatched) as time,
       [ seq_transform(u.info.shows[$element.showId = 16],
                       seq_transform($sq1.seriesInfo[],
                                     seq_transform($sq2.episodes[],
                                     { "showName" : $sq1.showName,
                                       "seasonNum" : $sq2.seasonNum,
                                       "episodeId" : $sq3.episodeID,
                                       "dateWatched" : $sq3.date
                                     }))) ] as episodes
from users u
where u.info.country = "USA" and
      exists u.info.shows[$element.showId = 16].
             seriesInfo.episodes[$element.date > "2021-04-01"]
```

For each user in USA that has watched at least one episode of show 16 after 2021-4-01, the query returns the user’s account and user ids, the total time the user has spent watching show 16, and an array containing information about all the episodes of show 16 that the user has watched. Specifically, each element of this array is a json document containing the show name, the season number, the episode id, and the date the episode was watched.

When run over the sample rows, the query returns one result that looks like this:

```
{ "acct_id":2,"user_id":1,
  "time":220,
  "episodes":[
    {"dateWatched":"2021-03-18","episodeId":20,"seasonNum":1,"showName":"Rita"},
    {"dateWatched":"2021-03-19","episodeId":30,"seasonNum":1,"showName":"Rita"},
    {"dateWatched":"2021-05-05","episodeId":40,"seasonNum":2,"showName":"Rita"},
    {"dateWatched":"2021-05-06","episodeId":50,"seasonNum":2,"showName":"Rita"}
  ]
}
```

The query illustrates how the seq_transform expression can be used, together with json object and array constructors, to transform the shape of the stored data. In this case, information from arrays in 3 different levels is combined into a new single flat array. See <link> for the definition of the seq_transform expression.

The query also illustrates the use of a sequence aggregation function (seq_sum) to sum up the time spent by a user watching episodes that satisfy a condition (the episodes of show 16). See <link> for details on sequence aggregation functions.

1.8. Query 8

```
select count(*) as cnt
from users u
where u.info.shows.showId =any 15 and
      size(u.info.shows[$element.showId = 15].seriesInfo) =
      u.info.shows[$element.showId = 15].numSeasons and
      not seq_transform(u.info.shows[$element.showId = 15].seriesInfo[],
                        $sq1.numEpisodes = size($sq1.episodes)) =any false and
      not seq_transform(u.info.shows[$element.showId=15].seriesInfo.episodes[],
                        $sq1.lengthMin = $sq1.minWatched) =any false
```

The query returns the number of users who have fully watched show 15 (all seasons and all episodes to their full length).

Let’s look closer at the WHERE clause of this query. The first condition selects show 15. The second condition requires that the size of the seriesInfo array for show 15 is equal to the number of seasons for show 15, that is, the user has watched all the seasons. This (and the following) condition is required because the data contain only the seasons and episodes that a user has actually watched, not all the available seasons/episodes of a show. The third condition requires that the user has watched all the episodes of each season of show 15. It does so by using a seq_transform expression to iterate over the seasons of show 15 and for each season, check if its number or episodes is equal to the size of its episodes array. The seq_transform returns the result of these checks, i.e., a sequence of true/false values. The rest of the condition checks that the sequence contains only true values. The fourth condition requires that the user has fully watched each episode of show 15. It does so in the same way as the third condition.

The query uses idx_showId, pushing the showId condition to it. The other conditions cannot be pushed to any index.

2. But unnest if you really have to!

In the previous section we showed how path expressions with filtering conditions on array elements, the seq_transform expression, and the sequence aggregation functions can be used to avoid the need for unnesting and subqueries. In fact, probably the only case where unnesting is really needed is when we want to group by fields that are contained in arrays. Nevertheless, we start with an example that does not involve grouping, just to illustrate the concept of unnesting. The rest of the queries in this section will use grouping. See <link> for more details on unnesting queries.

2.1. Query 9

```
select u.acct_id, u.user_id,
       $show.showName, $season.seasonNum,
       $episode.episodeID, $episode.date
from users u, u.info.shows[] as $show,
              $show.seriesInfo[] as $season,
              $season.episodes[] as $episode
where u.info.country = "USA" and
      $show.showId = 16 and
      $show.seriesInfo.episodes.date >any "2021-04-01"
```

For each user in USA that has watched at least one episode of show 16 after 2021-04-01, return one result for every episode of show 16 that the user has watched. Each such result contains the user's account and user ids, the show name, the season number, the episode id, and the date the episode was watched. The result of the query is the following:

```
{"acct_id":2,"user_id":1,"showName":"Rita","seasonNum":1,"episodeID":20,"date":"
2021-03-18"}
{"acct_id":2,"user_id":1,"showName":"Rita","seasonNum":1,"episodeID":30,"date":"
2021-03-19"}
{"acct_id":2,"user_id":1,"showName":"Rita","seasonNum":2,"episodeID":40,"date":"
2021-05-05"}
{"acct_id":2,"user_id":1,"showName":"Rita","seasonNum":2,"episodeID":50,"date":"
2021-05-06"}
```

Although a single table row satisfies the WHERE clause of this query, the result set of the query consists of 4 rows. Compare this query with query 7 above. The two queries are essentially equivalent, except from the way they present their results. Query 7 produces a single result with an array containing 4 elements. Query 9 flattens this array to produce 4 separate results.

2.2. Query 10

```
select $show.showId, count(*) as cnt
from users u, unnest(u.info.shows[] as $show)
group by $show.showId
order by count(*) desc
```

For each show, the query returns the number of users who have shown an interest in that show. The results are returned ordered by the number of users in descending order. Essentially the query orders the shows according to a measure of their popularity.

The query uses the idx_showid index and the index is a covering one for this query. To make the use of this index possible, two Oracle NoSQL features have been used.

First, the index was created with the “with unique keys per row” property. This informs the query processor that for any streaming user, the shows array cannot contain two or more shows with the same showId. The restriction is necessary because if duplicate show ids existed, they wouldn’t be included in the index, and as a result, the index would contain fewer entries than the number of elements in the shows arrays. So, use of such an index by the query would yield fewer results from the FROM clause than if the index was not used.

Second, the UNNEST clause was used in the query to wrap the unnesting expression. Semantically, the UNNEST clause is a noop (for example, query 9 did not use it). However, if an index exists on the array(s) that are being unnested by a query, use of UNNEST is necessary for the index to be considered by the query. The UNNEST clause places some restrictions on what kind of expressions can appear in it (see <link>), and these restrictions make it easier for the query optimizer to “match” the index and the query.

2.3. Query 11

```
select $show.showId, sum($show.seriesInfo.episodes.minWatched) as totalTime
from users u, unnest(u.info.shows[] as $show)
group by $show.showId
order by sum($show.seriesInfo.episodes.minWatched) desc
```

For each show, the query returns the total time users have spent watching that show. The results are returned ordered by that time in descending order. Query is 11 is similar to Query 10; it just sorts the shows according to a different measure of popularity.

The query uses the idx_showid_minWatched index, and the index is a covering one for this query. Notice that dx_showid_minWatched index indexes the episodeID field as well. It may appear that this is not necessary, but the field is actually needed so that the index satisfies the “with unique keys per row” property.

What if we didn’t have idx_showid_minWatched? Could this query have used idx_showid? The answer is no. This is because of the argument to the sum() function: The idx_showid index contains just the showid (and the primary key). So, if the query were evaluated by scanning that index, we wouldn’t be able to evaluate the `$show.seriesInfo.episodes.minWatched` expression, because there wouldn’t be any $show variable anymore (there would exist only an internal $showid var that ranges over the index entries). Therefore, in order to compute the argument of sum(), the full row would have to be retrieved. But what is the exact expression that should be computed as the argument of sum()? One might say that the expression is info.shows. seriesInfo.episodes.minWatched. However, this is not correct because the expression returns the minutes watched for **all** the show ids in the row, instead of just the show id of the current group. This analysis leads to the following equivalent query, which does use the idx_showid index. However, the index is not covering for Query 12.

2.4. Query 12

```
select $show.showId,
       sum(u.info.shows[$element.showId = $show.showId].
           seriesInfo.episodes.minWatched) as totalTime
from users u, unnest(u.info.shows[] as $show)
group by $show.showId
order by sum(u.info.shows[$element.showId = $show.showId].
             seriesInfo.episodes.minWatched) desc
```

The query is equivalent to Q11, but uses index idx_showid.

2.5. Query 13

```
select $show.showId,
       $seriesInfo.seasonNum,
       sum($seriesInfo.episodes.minWatched) as totalTime
from users u, unnest(u.info.shows[] as $show,
                       $show.seriesInfo[] as $seriesInfo)
group by $show.showId, $seriesInfo.seasonNum
order by sum($seriesInfo.episodes.minWatched) desc
```

For each show and associated season, the query returns the total time users have spent watching that show and season. The results are returned ordered by that time in descending order.

The query uses idx_showid_seasonNum_minWatched as a covering index.