Oracle NoSQL Database

Creating Index Views

12c Release 1

(Library Version 12.1.4.0)

Legal Notice

6/3/2016

Table of Contents

Index Views

Index views are a design pattern you use to create auxiliary records that are reflective of information contained in your primary records. There are many ways you can create index views. This document describes two of them.

Note

This article assumes that you are using Oracle NoSQL Database's Key/Value API and have read and understood the *Oracle NoSQL Database Getting Started with the Key/Value API* guide. If you have not read that manual, you should do so before reading this document.

Users of the Tables API have a built-in indexing mechanism available, and so this article is not meant for them.

As described in Reading Records in the *Oracle NoSQL Database Getting Started with the Key/Value API* guide, records are generally retrieved from the store using their key major and minor paths. You can either retrieve a single record using its key, or you can retrieve multiple records using part of a major path and then iterate over the result.

For example, suppose your store contains records related to users. The key might contain user organization information and other identifying information such as a user ID. Each record's data, however, would likely contain additional details about people such as names, addresses, phone numbers, and so forth. While your application may frequently want to query a person by user ID (that is, by the information stored as a part of the key path), it may also on occasion want to locate people by, say, their name.

Rather than iterating through all of the records in your store, examining each in turn for a given person's name, you can instead create application-managed index views. There are multiple ways to implement index views, but in general they are simply key/value pairs where the key relates to some information within your primary record, and the value identifies the primary record where that information can be found.

That is, if you had a record which contained the name `Peter`, then the key for its index view would contain `Peter` and the value would contain the major and minor key paths to that record.

Using Traditional Key/Data Pairs

This method of creating index views is, intuitively, the way many developers familiar with key/value stores will think to implement views.

For a different approach to building index views, see Using Key-Only Records (page 5).

When you use traditional key/data pair records to build index views, you create records where:

- The record's key path is some information in your primary data records that you want to quickly query.

- The record's data is the key path to a record which has the information contained in the key path.

For example, suppose you had records that used the following schema:

```
{
    "type": "record",
    "name": "PrimaryDBValue",
    "namespace": "oracle.kv.indexView",
    "fields": [
        {"name": "name", "type": "string", "default": ""},
        {"name": "email", "type": "string", "default": ""},
        {"name": "phone", "type": "string", "default": ""},
        {"name": "date", "type": "string", "default": ""},
        {"name": "org", "type": "string", "default": ""},
        {"name": "cost", "type": "long", "default": 0}
    ]
}
```

Further, suppose these records are stored using the employee's unique identifier. For example, these records might use key paths which end with an employee unique identifier, like this:

```
/Corporate/people/10012
/Corporate/people/10013
/Corporate/people/10014
```

In this case, in order to find all people who belong to the organization called "Support," you would have to iterate over every record whose key begins with `/Corporate/people`, examine each in turn for the proper organization name, and construct a list of those people who belong to that organization. Depending on the number of people contained in your store, this could be a lengthy operation.

The alternative is to create an index view that is keyed by the organization name. For example:

```
/IndexView/People/Organization/Engineering
/IndexView/People/Organization/Sales
/IndexView/People/Organization/Support
```

There are two ways to handle the data portion of these records. One way is for each record to contain a list of keys corresponding to the people records belonging to that organization. That is, the key:

```
/IndexView/People/Organization/Support
```

would return a data item with was a list of major keys for all those people entries containing an 'org' of 'Support'. As an Avro schema, you would represent the data item in the following way:

```
{
    "type": "record",
    "name": "SecondaryDBValue",
```

```
    "namespace": "oracle.kv.indexView",
    "fields": [
        {"name": "arrays",
         "type": {"type" : "array", "items" : "string"},
         "default" : []}
  ]
}
```

While this approach will work for small-to-medium sized indexes, it ultimately suffers from an inability to scale. It would be far too easy to create a view whose list of primary keys is too large to be easily handled by your code. In fact, it could easily grow so large that it could not fit into available memory. Given the size of the datasets for which Oracle NoSQL Database is designed, this is a very real consideration.

A different approach would be to create index views where each record referred to one and only one primary record. That is, the data portion of the record contains a simple string representing the key path to a primary record. (You could also carry this information as an array of key path components.) However, you cannot duplicate keys in Oracle NoSQL Database, so in this case the key needs to somehow be unique, based on the information found in the primary record. As an example, you could create keys that contains both the organization name, as well as the user's UID:

```
/IndexView/People/Organization/Support/-/10012
```

refers to the primary record:

```
/Corporate/People/10012
```

Using Key-Only Records

Key-only index view records carry all of the record's information in the key; the data portion of the record is set to an empty value. In this scheme, each index view record represents a single pairing between the secondary key and the primary record key to which it refers. Because Oracle NoSQL Database is good at scaling up to large numbers of records, this eliminates the scalability problem described in the previous section.

Note

The following examples use fairly long key paths. This is done for the purpose of clarity. However, in general, shorter key paths are desirable and so the paths shown here should not be taken as advice for how to construct the keys for your records.

Essentially, key-only index view records carry the index view's key in the major portion of the key path, and the corresponding primary record's key in the minor portion of the key path. That is:

```
/Secondary/Key/Path/-/Primary/Key/Path
```

The minor path component here is the key path for a primary record. For example, building on the example presented in the previous section, this might be:

```
/Secondary/Key/Path/-/Corporate/people/10012
```

The major key path portion of the record needs to carry more information:

- Index key prefix

  This is simply a prefix value used to indicate that the record is an index view record. The prefix can be anything so long as it is unique within your store; for example, `IDX`.

- Index name

  This is used to differentiate this index view from other types of index views. You could use something fairly simple here that is indicative of the information indexed by this record, such as `EMPLOYEE_NAME` or `EMPLOYEE_LOCATION`. However, it is possible to carry more complex information if you set up your code correctly. We discuss this further in Complex Index Names (page 6).

- Field value(s)

  The remainder of the major key path is a sequence of one or more field values that are obtained from the associated primary record. This is the actual information that you are indexing.

  In the simplest case, this portion of the key contains only one field value; for example, an organization name if what you are doing is indexing all employees by organization. For example:

  ```
  /IDX/ORGANIZATION/Engineering/-/Corporate/people/10012
  ```

  is a view entry that indicates employee record 10012 belongs to the Engineering organization.

  However, this portion of the key path can contain multiple field values, which gives you multi-column views. An example of this is indexing by employee common and family name, both of which would be individual fields in the primary record:

  ```
  /IDX/EMP_NAME/Smith/Robert/-/Corporate/people/10012
  /IDX/EMP_NAME/Smith/Patricia/-/Corporate/people/40288
  /IDX/EMP_NAME/Smyth/Don/-/Corporate/people/7893
  ```

Complex Index Names

As described above, an index name can be a simple text label, especially if you have fairly simple indexing requirements. However, it is possible to carry more information about the view record in the index name. You can construct the index name so that it identifies:

- The Avro schema name used by the primary record.
- A list of the field names that this view is indexing. This information is useful for generalizing your Avro binding code, especially as the number of fields you are indexing grows large, and/or as the number of types of index views grows large.

One way to construct an index name that carries this information is to create a list object that holds all the information you want in your index name, then create a one-way hash of the

information using `java.security.MessageDigest`. Converting the list to a byte array can be accomplished using the `Key.createKey()` method. For example:

```
    /**
     * Construct and return an index name representing an index type.
     */
    private String getIndexName(String schemaName,
                                List<String> indexFieldNames) {

        MessageDigest md = null;
        try {
            /*
             * The implementation for digestCache is omitted
             * for brevity.
             */
            md = digestCache.get();
            List<String> minorPath = new ArrayList<String>();
            minorPath.add(schemaName);
            minorPath.addAll(indexFieldNames);
            byte[] bytes =
                Key.createKey("", minorPath).toString().getBytes();
            md.update(bytes);
            return new String(md.digest());
        } finally {
            digestCache.free(md);
        }
    }
```

This means that the information you are carrying in your index name is locked up in a one-way hash. There is no way to retrieve that information from the hash, so you need to store it somewhere. You need a separate set of records to record index view metadata.

Managing Index View Metadata

Index view metadata is information you want to record about each index type. Mostly, this is information you use to construct your index names (if you use complex index names). You can also record your index state as a part of your metadata.

You can collect your index view metadata as a series of key-only records. In this case, the keys are constructed like this:

```
/PREFIX/INDEX_NAME/-/SCHEMA_NAME/FNAME1/FNAME2/.../STATE
```

where:

- `PREFIX` is a unique identifier that you use to indicate this record is an index view metadata record. For example: `META`.
- `INDEX_NAME` is the name you are using for the type of index for which you are collecting metadata. If you are using a simple name (for example, `ORGANIZATION` or `EMP_NAME`), then use that. If you are using a hashed complex name, such as is described in the previous section, then use that here.

- SCHEMA_NAME is the name of the Avro schema used by the primary record. This must be the same schema name as you used to construct your complex index name.
- FNAME1, FNAME2, and so forth, are the primary record field names this view type is using. Again, these must be identical to the field names you used to construct your complex index name. They must also appear in the same order as the field values used to construct your index view record keys.
- STATE is the current state of the index type represented by this metadata record. Examples of view STATE are:
  - BUILDING to indicate that the index view is currently being built.
  - DELETING to indicate that the index view is currently being deleted.
  - READY to indicate that the index view is ready for use.

  These are just some suggestions. STATE can really indicate anything that is useful to your code. But in the example given here, your code would only use the view if its state was READY.

Using Index View Records and Metadata Together

Putting it all together, to create an index view that uses complex index names, you would:

1. Create the index name, using the schema and field names that you are working with.
2. Create the metadata record, as described in the previous section, setting its state to BUILDING.
3. Iterate over your store, creating a view record for each primary record that you want to index. Use the index name you created in step 1 as part of the view record's major key path. See Using Key-Only Records (page 5) and Complex Index Names (page 6) for more information.
4. When you are done creating the view, change the status for the metadata record to READY. (To do this, you delete the old record and create a new one.)

To use (read) index views, you:

1. Check the corresponding metadata record to make sure the index view is in a READY state. If it is not, you can abort the read, or pause until the state has changed to READY.
2. Iterate over the index view records that interest you for the search.
3. For each such record, use it to retrieve the corresponding primary record.
4. For each primary record, use the schema and field names, contained in the corresponding metadata record, along with your Avro binding, to serialize/deserialize the primary record's data.

To update an existing record, you:

1. Retrieve the primary record.
2. Retrieve the index view record.
3. Modify the primary record as needed.
4. Modify the index view record to reflect the changes to the primary record.
5. Check the status of the index view to ensure that it is in a READY state. If it is, then write the index view record back to the store.

   If the index view status is not READY, then either wait for the status to change to READY before writing the index view record, or fail the operation.
6. Write the modified primary record back to the store.

An example of performing all these operations is available in your Oracle NoSQL Database distribution. See Example (page 11) for details.

Key Size Consideration

The longer your keys, the more memory you are using at your nodes. Keys can therefore grow so large that they harm your system's overall read/write throughput due to an inability to maintain enough records in cache.

The key-only design pattern described here will probably result in very long keys. Whether those key sizes are so large that they cause you a performance problem is a function of how long your keys actually are, how many keys you need to manage, and how much memory is available on your nodes.

If your keys are so large that they will cause an I/O throughput issue, then you need to implement some other design approach.

General Index Views Considerations

While creating index views can vastly improve your stores read performance (depending on the size of your data set, and the kinds of questions you want to ask of it), there are some limitations of which you need to be aware.

Additional Write Activity

Maintaining an index view necessarily requires additional read and write activity over and above what is required just to maintain a primary record. Whether this additional activity will measurably affect your write throughput depends on the size of the dataset you are indexing, and the size of your views.

For small datasets and small views, this additional activity will not be noticeable. But as the size of the data to be indexed grows, and so your views themselves grow larger, you might

notice a reduction in throughput, particularly in write throughput. Given this, when planning your store size, be sure to consider overhead to maintain index views.

Non-Atomic Updates

Because index views are managed by the application, Oracle NoSQL Database cannot insure that operations performed on the primary record are atomic with operations performed on the corresponding view records. This means that it is possible to change your primary records, but have the corresponding operation(s) on your index view(s) fail thereby causing them to be out of sync with the primary data. The reverse can also happen, where the index view update operation is successful, but the update to the primary record fails.

Note that for some workloads, non-atomic updates to primary records and their index views can be desirable. This is sometimes the case for workloads that require the maximum read and write throughput possible. For these types of applications, consistency between the index view and the primary data is not nearly as important as overall performance.

That said, you should still make an attempt to determine whether your indexes are out of sync relative to your primary data, so that you can perform compensating transactions if your code detects a problem. You may also need to flag your index views as being in an unsafe state if some aspect of the update operations fail. The safest way (not necessarily the fastest way) to update a primary record for which you are maintaining an index view is:

1. Check whether your view is in a READY state. If it is, proceed with the update operation. If it is not, either pause and wait for the state to change, or abort the update entirely.

2. Update your primary record as necessary, but *do not write the results back to the store yet*.

3. Update your index view to be reflective of the changes you have made to the primary record.

4. Write the primary record to the store. If the write fails, perform a compensating transaction to fix the problem. Either retry the write operation with the updated record, or check to ensure that the record which is currently in the store is not corrupted or altered in any way.

5. If the update to the primary record succeeds, then write the changes to the index view to the store. If this succeeds, then you are done with your update.

6. If the update to the index view record fails, then immediately mark your index view as being in a non-READY state. How you do this depends on how you are storing index view state flags, but assuming you are using metadata records, that needs to be updated before you take steps to fix your index view.

A similar algorithm is required for the creation and deletion of primary records.

Of course, this means that before you perform a read with your index view, you need to check the view's state before you proceed. If the view's state is not READY, then you need to either pause until the state is READY, or you need to abandon the read entirely. In addition to this

check, you also need to ensure that your index views are in a state that is consistent with the primary records. This is described next.

Decoupled Consistency

As described above, index views can be out of sync with your primary data due to some generic failure in the update operation. Your code needs to be robust enough to recognize when this has happened, and take corrective action (including rebuilding the index view, if necessary). A related, but temporary, problem is that for any given node, changes to your views may not have caught up to changes to your primary records due to replication delays. Note that it is also possible for views on the local node to have been updated when the corresponding primary data modifications have not yet arrived.

Again, for some workloads, it might not be critically important that your views are in sync with your primary data. However, if your workload is such that you need assurance your views accurately reflect your primary data, you need to make use of Oracle NoSQL Database's built-in consistency guarantees.

One way to do this is to use an absolute consistency guarantee for any reads that you perform using your views. But this can ultimately harm your read and write performance because absolute consistency requires the read operation to be performed on a master node. (See Using Predefined Consistency in the *Oracle NoSQL Database Getting Started with the Key/Value API* guide for details.) For this reason, you should use absolute consistency only when it is truly critical that your views are completely up-to-date relative to your primary data.

A better way to resolve the problem is to use a version-based consistency guarantee when using your index views. You will need to check the version information on both the primary data and the views when performing your reads in order to ensure that they are in sync. You may also need to create a way to transfer version information between different processes in your application, if one process is responsible for performing store writes and other processes are performing store reads. For details on using version-based consistency, see Using Version-Based Consistency in the *Oracle NoSQL Database Getting Started with the Key/Value API* guide.

Example

An example of creating and managing index views is included in the Oracle NoSQL Database distribution. It can be found here:

```
<KVROOT>/examples/secondaryindex
```

The example exposes a command line interface that allows you to create and delete index views, retrieve the primary data referred to by an index view record, and insert, delete, and update new primary records. The application is a very simple application that allows you to create views against customer billing records.

The example uses key-only index view records, with complex index names and associated metadata records. The code that is responsible for managing the views and associated metadata is contained in this class:

```
<KVROOT>/examples/secondaryindex/IndexViewService.java
```

Note that the example is one expression of the index view design pattern. Its operations may not be a match for the way your code operates, but it should serve as good design guide. Feel free to adapt, expand, or simplify the example code to match your own design needs and goals.