- Feature 1. Add permissions to an existing user

1. auth/`permission.java`

   permission is a class that simply contains possible permissions a user can have on a resource. For example, user can create, alter, drop, etc.

```
public enum Permission
{
    CREATE,
    ALTER,
    DROP,

    // data access
    SELECT, // required for SELECT on a table
    MODIFY, // required for INSERT, UPDATE, DELETE, TRUNCATE on a DataResource.
    // permission management
    AUTHORIZE, // required for GRANT and REVOKE of permissions or roles.

    DESCRIBE, // required on the root-level RoleResource to list all Roles

    // UDF permissions
    EXECUTE;
```

2. auth/`RoleResource.java`

   This class implements "IResource" interface and represents database roles. It contains information for defining the root user and any sub user (roles) along with permissions.

```
// permissions which may be granted on the root level resource
private static final Set<Permission> ROOT_LEVEL_PERMISSIONS = Sets.immutableEnumSet(Permission.CREATE,
                                  Permission.ALTER,
                                             Permission.DROP,

Permission.AUTHORIZE,
            Permission.DESCRIBE);
// permissions which may be granted on role level resources
private static final Set<Permission> ROLE_LEVEL_PERMISSIONS = Sets.immutableEnumSet(Permission.ALTER,
                                    Permission.DROP,
                                             Permission.AUTHORIZE);

```

   The critical method in this class for applying permissions is `applicablePermissions()`.

```
public Set<Permission> applicablePermissions()
{
 return level == Level.ROOT ? ROOT_LEVEL_PERMISSIONS :
ROLE_LEVEL_PERMISSIONS;
}
```

```

   When this method is called, it checks whether user is a root or not, and
returns the appropriate permissions that **could** be granted later.
Specifically, if the user is a root, it could be given all 5 permissions.
If it is not a root user, it can only be given 3 instead (no permissions on
`CREATE` and `DESCRIBE`)

3.  cql3/statements/`CreateRoleStatement`

   This class calls `applicablePermission` inside its method:
`grandPermissionsToCreator`. This method grants all applicable permissions
on the newly created role to the user performing the request. If the user
is anonymous, and the configured IAuthorizer supports granting of
permissions, the creator of a Role automatically gets ALTER/DROP/AUTHORIZE
permissions. Else, the `grant` method is called to give certain
permissions.

   ```java
       private void grantPermissionsToCreator(ClientState state)
       {
           // The creator of a Role automatically gets ALTER/DROP/AUTHORIZE
permissions on it if:
           // * the user is not anonymous
           // * the configured IAuthorizer supports granting of permissions
(not all do, AllowAllAuthorizer doesn't and
           //   custom external implementations may not)
           if (!state.getUser().isAnonymous())
           {
               try
               {
DatabaseDescriptor.getAuthorizer().grant(AuthenticatedUser.SYSTEM_USER,
   role.applicablePermissions(),
   role,
RoleResource.role(state.getUser().getName()));
               }
               catch (UnsupportedOperationException e)
               {
                   // not a problem, grant is an optional method on
IAuthorizer
               }
           }
       }
```

4. auth/`CassandraAuthorizer.java`
```

```
public void grant(AuthenticatedUser performer, Set<Permission>
permissions, IResource resource, RoleResource grantee)
    throws RequestValidationException, RequestExecutionException
    {
        modifyRolePermissions(permissions, resource, grantee, "+");
        addLookupEntry(resource, grantee);
    }
```

This grant method gives the "grantee" permissions from permission argument. It also calls the `modifiedRolePermissions` method.

5. auth/`CassandraAuthorizer.java`

   This method allows user to make a input query that modifies permissions of a certain role.

```
private void modifyRolePermissions(Set<Permission> permissions,
IResource resource, RoleResource role, String op)
            throws RequestExecutionException
    {
        process(String.format("UPDATE %s.%s SET permissions =
permissions %s {%s} WHERE role = '%s' AND resource = '%s'",
                                SchemaConstants.AUTH_KEYSPACE_NAME,
                                AuthKeyspace.ROLE_PERMISSIONS,
                                op,
                                "'" + StringUtils.join(permissions,
"','") + "'",
                                escape(role.getRoleName()),
                                escape(resource.getName())));
    }
```

Note: `modifyRolePermissions` is a private method, which means in order to make permission changes, it's always called together with `addLookupEntry` in the `grant` method. When the permission is changed, the new role source will be added to a inverted index that (which is deployed on the start of the program) for the updated role permissions. This strategy avoids the mismatch between the permissions and the table recording the permissions. (The details of `addLookupEntry` are shown below.)

6. auth/`AuthKeyspace.java`

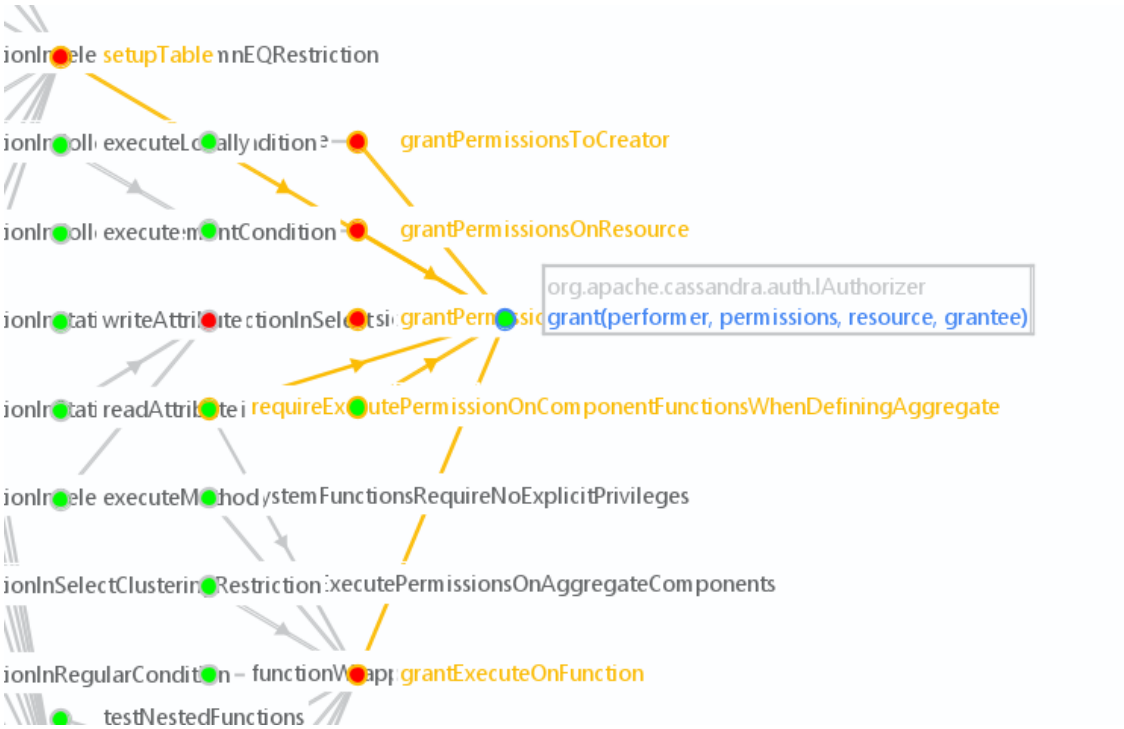
   Eventually, the `addLookupEntry` method will modify the the "ResourceRoleIndex" table that stores index, and "RolePermissions" table (index --> table). Both of which are automatically created on startup.

```
private static final TableMetadata RolePermissions =
        parse(ROLE_PERMISSIONS,
              "permissions granted to db roles",
              "CREATE TABLE %s ("
              + "role text,"
              + "resource text,"
              + "permissions set<text>,"
              + "PRIMARY KEY(role, resource))");

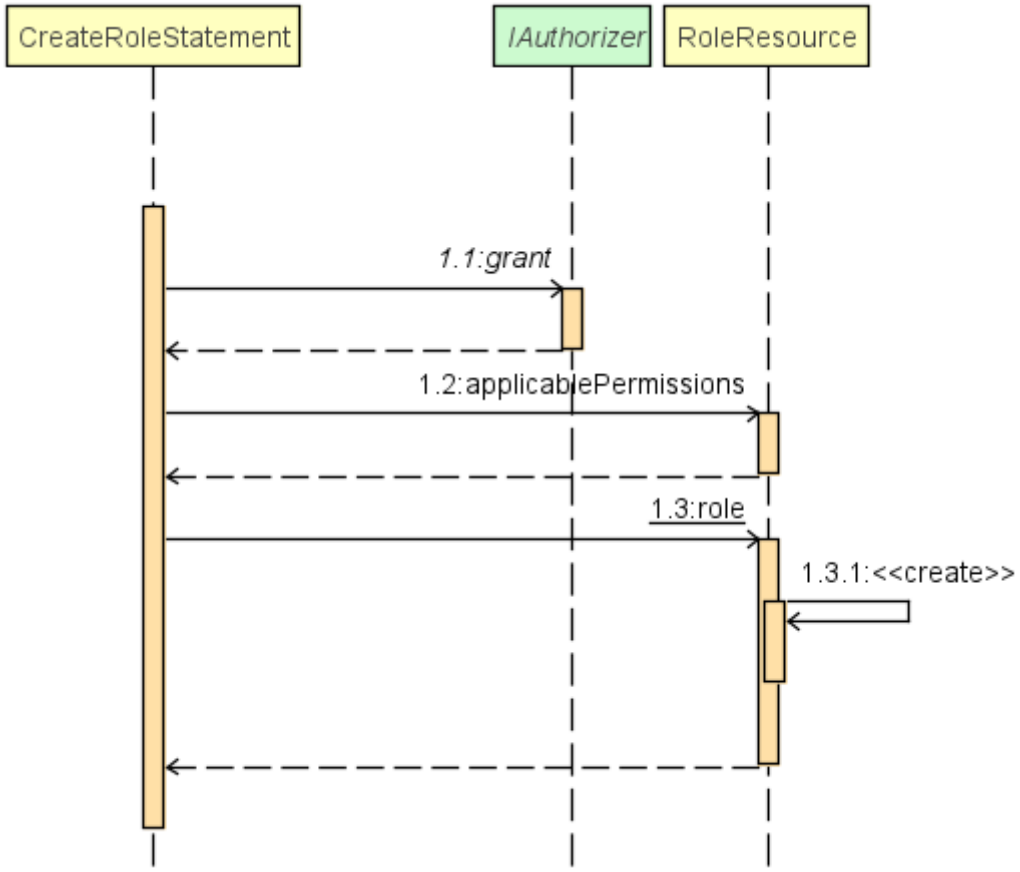
    private static final TableMetadata ResourceRoleIndex =
        parse(RESOURCE_ROLE_INDEX,
              "index of db roles with permissions granted on a
resource",
              "CREATE TABLE %s ("
```

```
14                 + "resource text,"
15                 + "role text,"
16                 + "PRIMARY KEY(resource, role))");
17
```

7. attached is a call graph that represent where the essential method "grand" is called



8. below is a sequence graph that shows how a role grants an applicable permissions.



- Feature 2 How Cassandra parse query and update database (use *delete a column* as an example)

Parsing query is a very important and seemingly straightforward feature for a database. It takes the query from a user, analyze it and either pull date from or update the selected database. Every database server has little difference on how query language works and this part will analyze how Cassandra Query Language actually get parsed to its server. We will use "delete a column" as an example to find out what files are involved and how they are connected to each other.

1. `Parser.class`

   First of all, the program will read a query from user input through the `antlr-runtime` library, in `Parser`, and save the input as a `TokenStream`.

```
public Parser(TokenStream input, RecognizerSharedState state) {
    super(state);
    this.input = input;
}
```

2. cql3/`Cql_Parser.java`

   Based on user input, `Cql_Parser`, which extends `Parser` analyzes the query (using the ANTLR library mentioned before) and directs us to the `deleteOp` method.

   This method deals with different kinds of delete operations, including `ColumnDeletion` , `ElementDeletion` and `FieldDeletion`.

   In our case, we go with the `ColumnDeletion` method in `Operation` class.

```
public final Operation.RawDeletion deleteOp() throws
RecognitionException {
    ...

        // Parser.g:597:5: (c= cident |c= cident '[' t= term ']' |c=
cident '.' field= fident )
        int alt61=3;
        alt61 = dfa61.predict(input);
        switch (alt61) {
            case 1 :
                // Parser.g:597:7: c= cident
                {
                    // ColumnDeletion
                        break;
            case 2 :
                // Parser.g:598:7: c= cident '[' t= term ']'
                {
                    // ElementDeletion
                }
                break;
             case 3 :
                // Parser.g:599:7: c= cident '.' field= fident
                {
                    // FieldDeletion
                }
                break;
        }
        ...
    }
```

3. cql3/`Operation.java`

   Under `Operation` class, in the following code, the `ColumnDeletion` class will have a constructor that assigns an ID of the column to be deleted, and then it can create an operation class that will be executed later.

```
public static class ColumnDeletion implements RawDeletion
    {
        private final ColumnIdentifier.Raw id;

        public ColumnDeletion(ColumnIdentifier.Raw id)
        {
            this.id = id;
        }

        public ColumnIdentifier.Raw affectedColumn()
        {
            return id;
        }

        public Operation prepare(String keyspace, ColumnDefinition
receiver) throws InvalidRequestException
        {
            // No validation, deleting a column is always "well typed"
            return new Constants.Deleter(receiver);
        }
    }
```

4. cql3/`ColumnIdentifier.java`

```
private final ColumnIdentifier.Raw id;
```

   Raw is an interface in `ColumnIdentifier` class, which represent a identifier for a CQL column definition. `ColumnIdentifier.Raw` is a placeholder that can be converted to a real `ColumnIdentifier` once it is associated with a prepare() function. the interface Raw will return a ColumnIdentifier object that can be used later when performing column deletion operation.

5. cql3/selection/`Selectable.java`

   Further down to see what the parent class of ColumnIdentifier does helps better understand the whole system. One important interface is `Raw` that check if a column, row or any other data is performed before on this column to determine if it is still raw or not. It does not affect our investigation this time as it seems like this raw status does not affect column deletion process but may affect other column related performance.

```
public Selectable prepare(CFMetaData cfm);

        /**
         * Returns true if any processing is performed on the selected
column.
         **/
        public boolean processesSelection();
```

6. cql3/`Operation.java`

   Back to the `Operation` class file, when Deleter is called from the previous "prepare" method, when initialized the constructor, its parent method of initializing constructor is called. A null value is assigned to the selected column, by far the column deletion task is finished.

```
public static class Deleter extends Operation
    {
        public Deleter(ColumnDefinition column)
        {
            super(column, null);
        }

        public void execute(DecoratedKey partitionKey, UpdateParameters
params) throws InvalidRequestException
        {
            if (column.type.isMultiCell())
                params.setComplexDeletionTime(column);
            else
                params.addTombstone(column);
        }
    }
```

```
protected Operation(ColumnDefinition column, Term t)
    {
        assert column != null;
        this.column = column;
        this.t = t;
    }
```

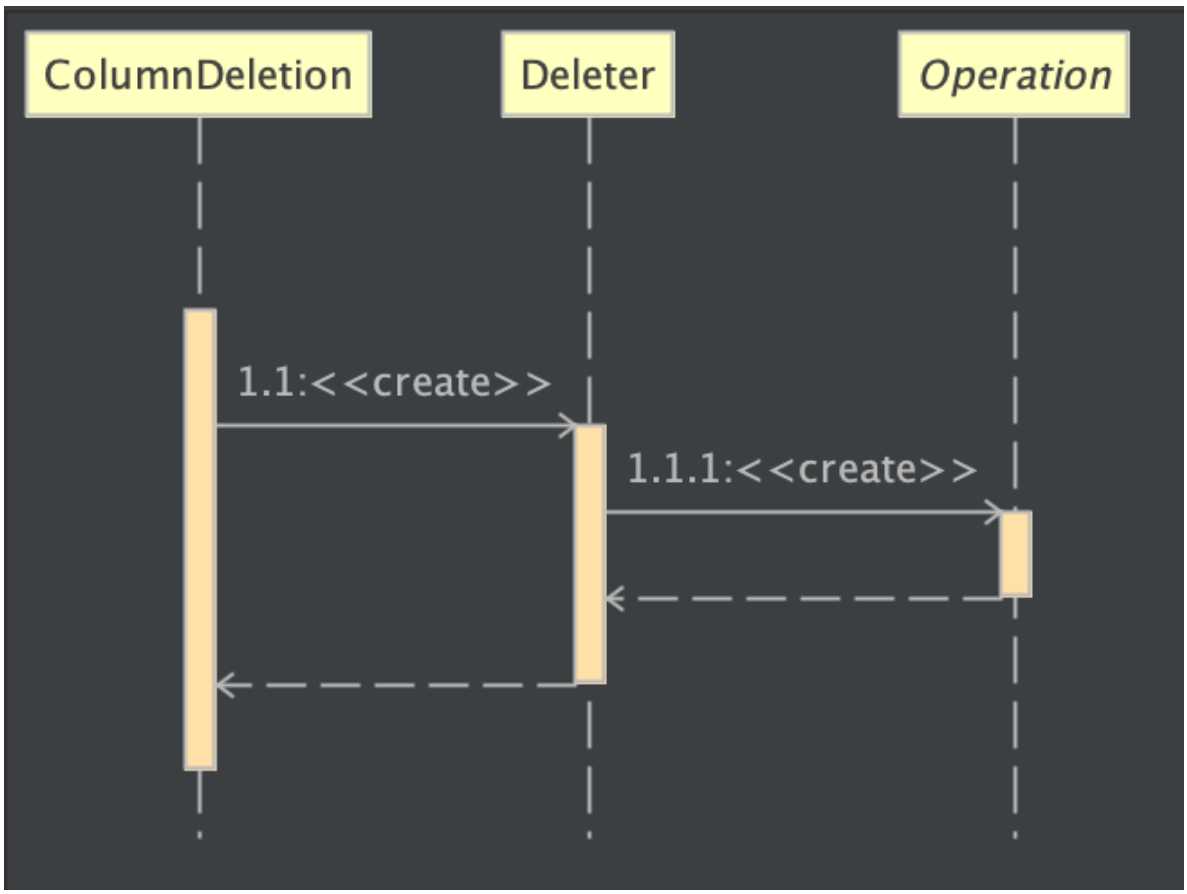
7. config/`ColumnDefinition.java`

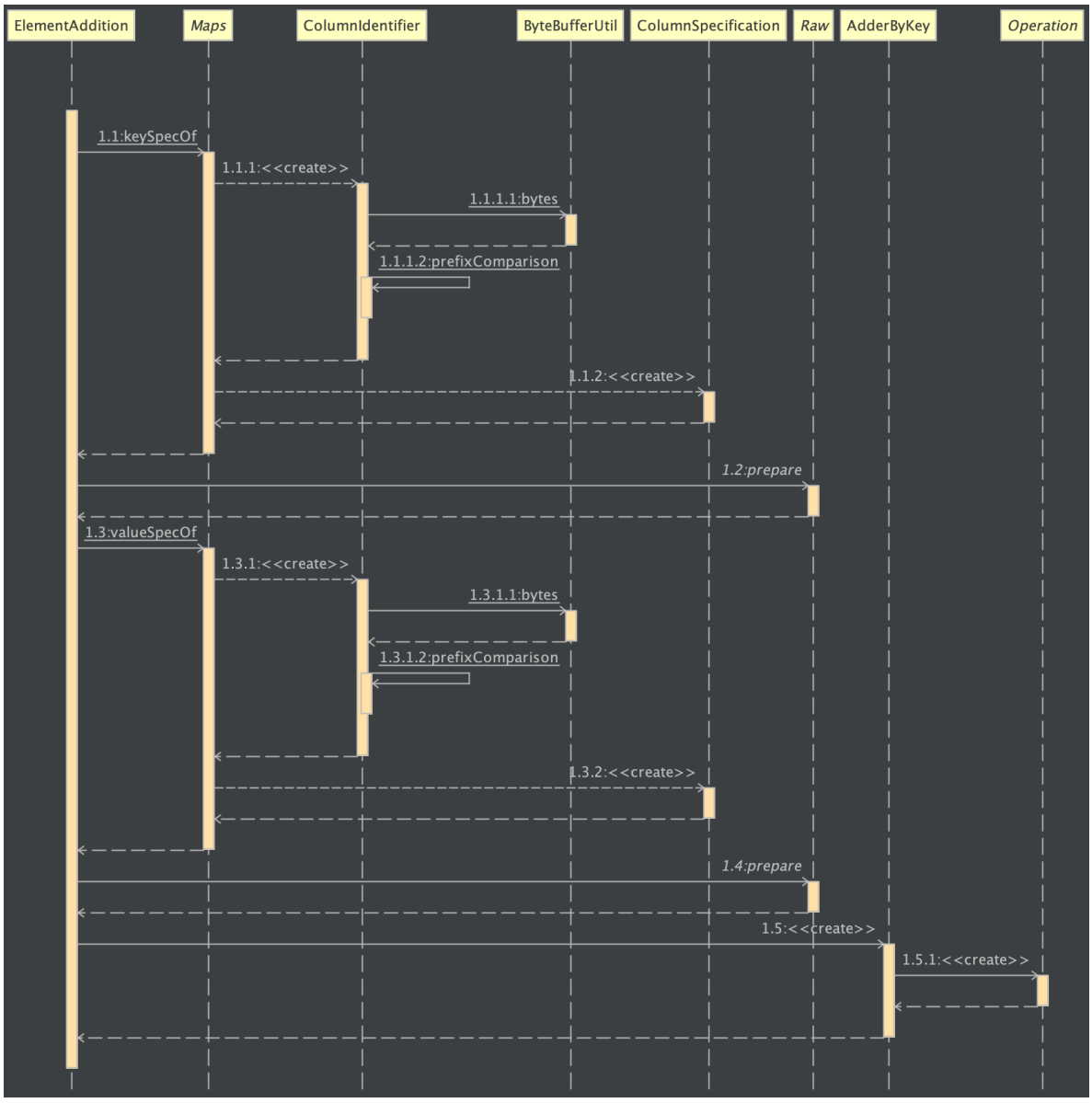
   To further understand the system, we dug deeper to see what kinds of column can Cassandra has, we explored the `ColumnDefinition` class. Essentially a column is a map of name/value pair. When deleting a column by assigning a null value to the specified column, it goes to to the ColumnDefinition class that defines what is a column in Cassandra. A column can be four types:

   - Partition key: in Cassandra, partition key can consist of multiple columns
   - Clustering key columns: The clustering columns are used to control how data is sorted for storage within a partition
   - Regular
   - Static

```
public enum Kind
    {
        // NOTE: if adding a new type, must modify comparisonOrder
        PARTITION_KEY,
        CLUSTERING,
        REGULAR,
        STATIC;
```

8. The following are some graphs showing the related methods and classes.


ColumnDeletion
Deleter
Operation
1.1:<<create>>
1.1.1:<<create>>


ElementAddition
Maps
ColumnIdentifier
ByteBufferUtil
ColumnSpecification
Raw
AdderByKey
Operation
1.1:keySpecOf
1.1.1:<<create>>
1.1.1.1:bytes
1.1.1.2:prefixComparison
1.1.2:<<create>>
1.2:prepare
1.3:valueSpecOf
1.3.1:<<create>>
1.3.1.1:bytes
1.3.1.2:prefixComparison
1.3.2:<<create>>
1.4:prepare
1.5:<<create>>
1.5.1:<<create>>