Essential Features

What is changed:

Changed parts are in **bold**.

- Elaboration of why permissions to an user is essential
- Additional graph with explanation
- Replace the part related to class files, and change with java files which can be edited.

And some other minor changes which are not shown specifically.

Feature 1. Add permissions to an existing user

The two main components of a completed database system is firstly, the database itself, where all the data is stored. In this case, Cassandra uses a distributed system that pieces of data can be retrieved from any node or data center that are connected to each other.

Secondly, user's reads and writes of this database. As the database itself alone does not create any value until it is being used. Permissions to users is the connection between these two components where the database has the function of granting users different permissions of either just reading data, or being able to change data as well.

Granting different permission is also highly related to the security level of Cassandra, as not all users will have the permission to make changes to database. Even though the database will still work, the server will be still running if this feature is removed, the database itself does not mean anything at that point where there is no one being able to read or write any data.

To learn about how Cassandra gives permissions to an existing user, we start with the Permission.java file.

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.

```
1  public enum Permission
2  {
3      CREATE,
4      ALTER,
5      DROP,
6
7      // data access
8      SELECT, // required for SELECT on a table
9      MODIFY, // required for INSERT, UPDATE, DELETE, TRUNCATE on a
   DataResource.
10     // permission management
11     AUTHORIZE, // required for GRANT and REVOKE of permissions or
   roles.
12
13     DESCRIBE, // required on the root-level RoleResource to list all
   Roles
```

```
14
15      // UDF permissions
16      EXECUTE;
```

2. auth/`RoleResource.java`

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

```
1  // permissions which may be granted on the root level resource
2      private static final Set<Permission> ROOT_LEVEL_PERMISSIONS =
   Sets.immutableEnumSet(Permission.CREATE,
3
                         Permission.ALTER,
4
                         Permission.DROP,
5
                         Permission.AUTHORIZE,
6
                         Permission.DESCRIBE);
7      // permissions which may be granted on role level resources
8      private static final Set<Permission> ROLE_LEVEL_PERMISSIONS =
   Sets.immutableEnumSet(Permission.ALTER,
9
                         Permission.DROP,
10
                         Permission.AUTHORIZE);
11
```

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

```
1  public Set<Permission> applicablePermissions()
2      {
3          return level == Level.ROOT ? ROOT_LEVEL_PERMISSIONS :
   ROLE_LEVEL_PERMISSIONS;
4      }
```

   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() function inside its method called grandPermissionsToCreator():

```
1      /**
2       * Grant all applicable permissions on the newly created role to
   the user performing the request
3       * see also: AlterTableStatement#createdResources() and the
   overridden implementations
```

```
     * of it in subclasses CreateKeyspaceStatement &
CreateTableStatement.
     * @param state
     */
    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
            }
        }
    }
```

This class is about how a user create a new role. The method above grants permissions of the newly created role to the user, if the user is not anonymous. The permissions are granted with the "grant" method.

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 permissons of 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())));
    }
```

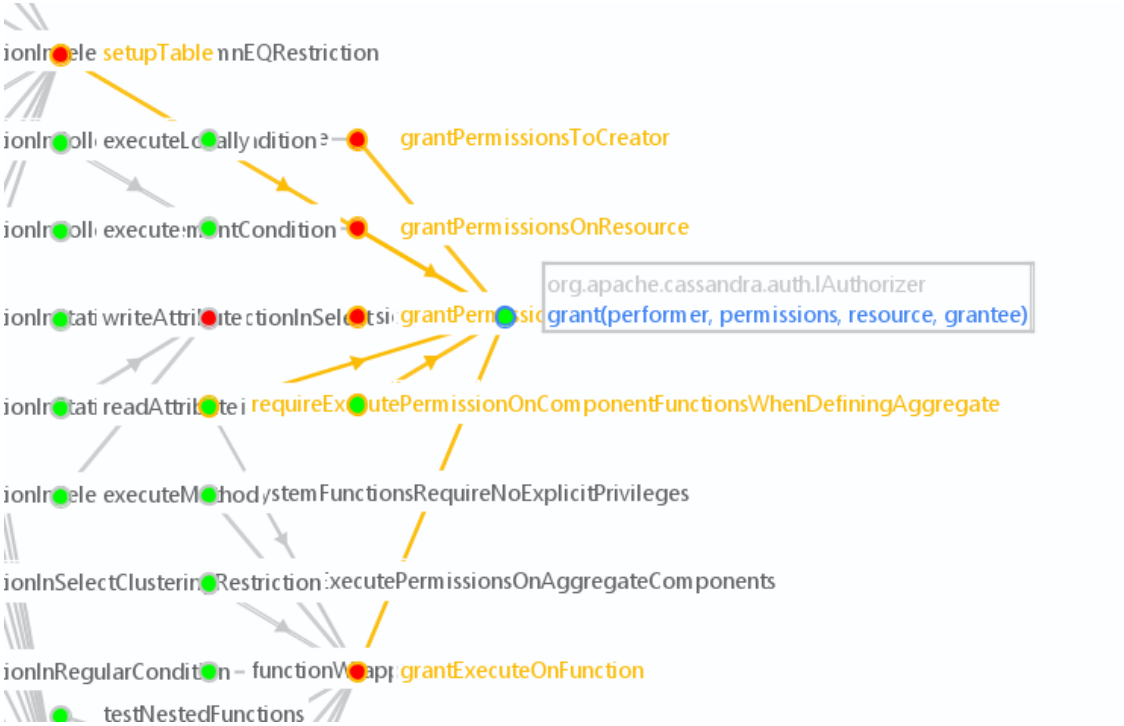
modifyRolePermissions() is a private methd that is called by a public method called grant, which means in order to make permission changes, the performer needs to be a "authenticaltedUser". If the performer has the permission, on the side of the modifyRolePermission, the new role source will be added to a inverted index that (which is deployed on the start of the program) for the updated role permissions.

6. auth/`AuthKeyspace.java`

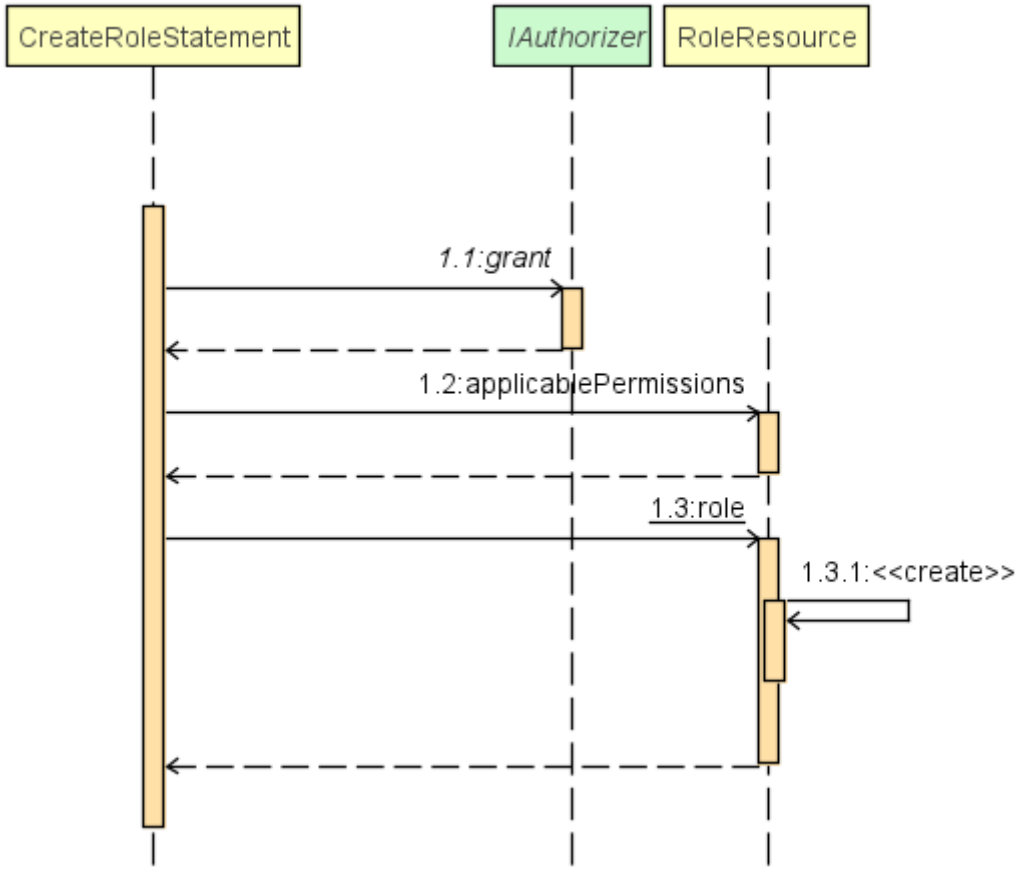
   Eventually, the previous `addLookupEntry` method will modiy the the "ResourceRoleIndex" table that stores index, and "RolePermissions" table (index --> table). Both of which are automatically created on startup.

```
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 ("
              + "resource text,"
              + "role text,"
              + "PRIMARY KEY(resource, role))");

```

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 is created and granted perspective permissions.



Feature 2 How Cassandra parse query and update database

Parsing query is a very important and seemingly straight forward 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 to query language works and this article will analyze how Cassandra Query Language actually get parsed to its server. We will use one of its function

"delete a column" to see what files are involved and how they are connected to each other.

To discover the source code where how Cassandra reads a query and process it, we started with looking into how the query data is read and processed.

We find it in the `QueryMessage` java file. The main line of code doing this is the following:

```
public Message.Response execute(QueryState state){
 ...
 Message.Response response = ClientState.getCQLQueryHandler().process(query, state, options, getCustomPayload());
    ...
}
```

It creates a Response instance by calling "process" method with 3 parameters. The process method prepares an object of `ParsedStatement.Prepared` in the `QueryProcessor.java` .

```
public ResultMessage process(String queryString, QueryState queryState, QueryOptions options)
 throws RequestExecutionException, RequestValidationException
 {
     ParsedStatement.Prepared p = getStatement(queryString, queryState.getClientState());
     options.prepare(p.boundNames);
     CQLStatement prepared = p.statement;
     if (prepared.getBoundTerms() != options.getValues().size())
         throw new InvalidRequestException("Invalid amount of bind variables");

     if (!queryState.getClientState().isInternal)
         metrics.regularStatementsExecuted.inc();

     return processStatement(prepared, queryState, options);
 }
```

To dive deeper of how the code prepare this object, we looked into `processStatement` method.

```
 public ResultMessage processStatement(CQLStatement statement, QueryState queryState, QueryOptions options)
 throws RequestExecutionException, RequestValidationException
 {
     logger.trace("Process {} @CL.{}", statement, options.getConsistency());
     ClientState clientState = queryState.getClientState();
     statement.checkAccess(clientState);
     statement.validate(clientState);

     ResultMessage result = statement.execute(queryState, options);
     return result == null ? new ResultMessage.Void() : result;
 }
```

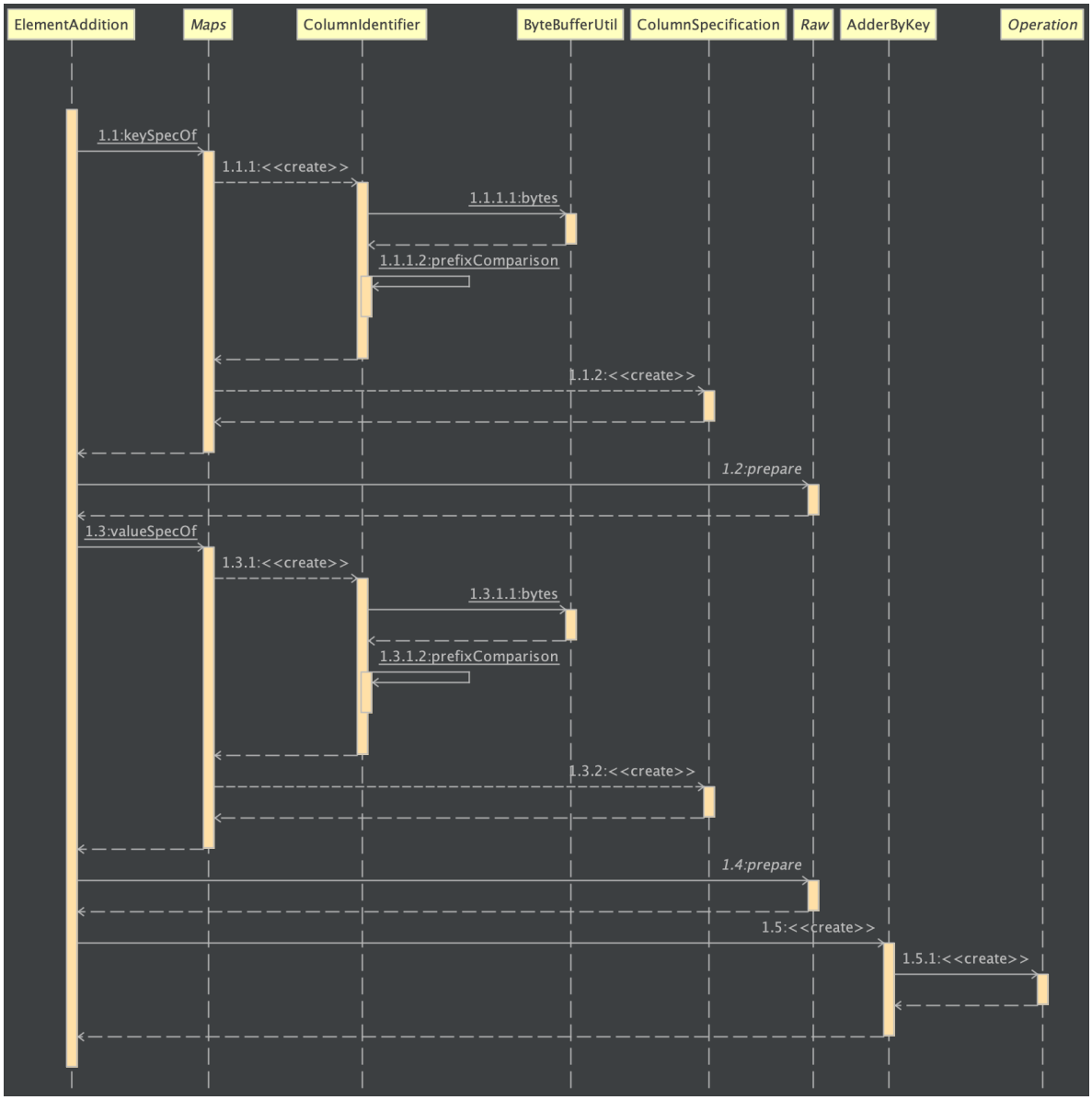
This method returns a resultMessage based on the query that user input by calling statement.execute method. Let's use select statment as an example here. In the `SelectStatement.java` the object process query by calling execute method:

```
public ResultMessage.Rows execute(QueryState state, QueryOptions options)
throws RequestExecutionException, RequestValidationException
 {
     ConsistencyLevel cl = options.getConsistency();
     checkNotNull(cl, "Invalid empty consistency level");

     cl.validateForRead(keyspace());

     int nowInSec = FBUtilities.nowInSeconds();
     int userLimit = getLimit(options);
     ReadQuery query = getQuery(options, nowInSec, userLimit);

     int pageSize = getPageSize(options);

     if (pageSize <= 0 || query.limits().count() <= pageSize)
         return execute(query, options, state, nowInSec, userLimit);

     QueryPager pager = query.getPager(options.getPagingState(),
options.getProtocolVersion());
     return execute(Pager.forDistributedQuery(pager, cl,
state.getClientState()), options, pageSize, nowInSec, userLimit);
 }
```

Then it will create a msg by searching through the current partition section ()

```
    ResultMessage.Rows msg;
    try (PartitionIterator page = pager.fetchPage(pageSize))
    {
        msg = processResults(page, options, nowInSec, userLimit);
    }
```

To sum up, the sequence diagram below shows the flow from user input a query and then parsed to the system and finally returned a response. The QueryProcessor first check the permission of this user and validate its role. If the user is authorized to perform such query, processor will execute Cassandra Query Language and create ResultMessage, ResultMessage creates a response and response will generate the final message to user.



Then, to focus on the exact execution of parsing, we go from Parser.

1. cql/`Parser.java`

   This program will read a query from user input and parse this query to make changes to the database. We found a class file called `Parser`, after building the project that does the initial analysis.

   ```
   1 public class Parser extends BaseRecognizer {
   2     public TokenStream input;
   ```

   The parse will initially have a input variable the read from user's input query.

2. cql/`CqlParser.java`

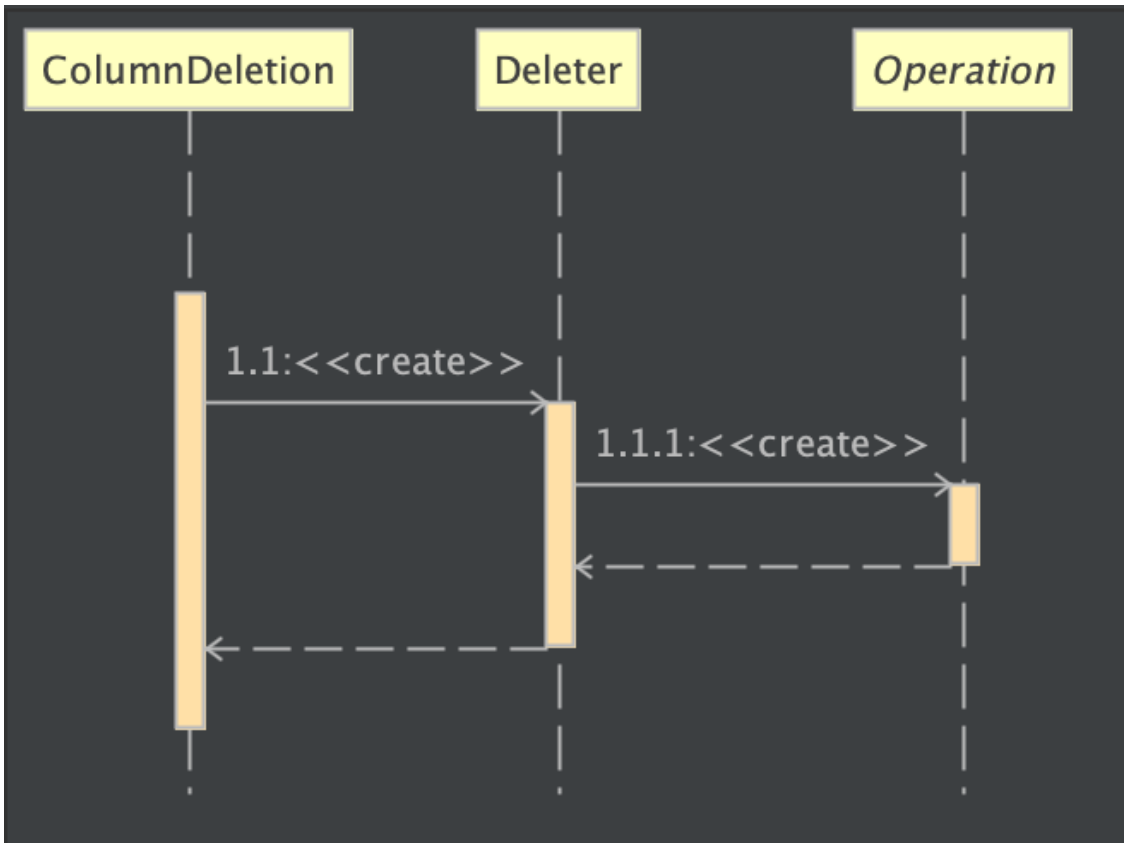
   Based on user input, the deleteOp method inside `CqlParser` class will analyse this input by using a predict function and assign this value to "alt41"

   ```
   1 alt41 = dfa41.predict(input);
   ```

   Then based on the value of this variable and only in the case when alt41 == 1, will a column delection operation be created and will be executed later.

```
switch (alt41) {
                case 1 :
                    {
                    pushFollow(FOLLOW_cident_in_deleteOp2559);
                    c=cident();
                    state._fsp--;

                     op = new Operation.ColumnDeletion(c);
                    }
                    break;
```

When the deletion starts, from a high level, it follows the steps shown in the following graph. The ColumnDeletion creates a Deleter, and the Deleter creates an Operation to deal with the removal (set the column to null).



But the more detailed version is shown below.

3. cql3/`Operation.java`

   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 a interface in ColumnIdentifier class, which represent a indentifier 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 futher understand the sytem, we digged 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;
```