Iceberg REST Scan API Support

Authors: Rahil Chertara (rchertar@amazon.com), Jack Ye (yzhaoqin@amazon.com)
WIP PR: https://github.com/apache/iceberg/pull/9252

In Iceberg, when a user submits a query to an engine, the engine will invoke one of Iceberg's catalog implementations (such as AWS Glue, HiveMetaStore, REST, etc.) to load the table's metadata. From the table metadata, iceberg can obtain the table schema, and from the engine Iceberg can obtain the selected columns and filters, to initialize an iceberg table scan.

Once an Iceberg scan has been created, it will initiate a planning phase. During the planning, an iceberg scan will read the table's snapshot to find the relevant data at client side. A table's snapshot contains a list of manifest files with each manifest file containing pointers to the underlying data files in local or cloud storage. Once Iceberg has filtered through the data and delete files that are relevant for this query it will start converting those files into file scan tasks. The file scan tasks are then returned back to the engine in order for the engine to execute the tasks distributedly.

In this doc, we propose moving this planning logic to server side, by introducing scan APIs to Iceberg REST catalog spec.

Motivations

1. Improve read performance

With a scan API, we can power better read query performance by improving the scan planning. For example:

Caching

By allowing the server to handle creation of Scans and FileScanTasks, we can implement other optimizations on the server side for further performance gains. One such example can be **scan plan result caching**. If the user provides an identical scan, the REST catalog can immediately provide the cached scan tasks.

If a new scan is a subset of a cached scan (e.g. first run `select * from orders where user='xxx'` then run `select * from orders where user='xxx' and order_status='PENDING'`), we can plan tasks based on the previous scan result to further reduce the planning time. In addition to inter-query usage, this approach can be used intra-query during rule-based optimizers and runtime filters.

Secondary indexes

Today the adoption of secondary indexes and puffin spec is relatively low in the Iceberg community. However, based on internal testing, we see significant gains for using secondary indexes during planning, some sketches show over 30% performance gain. While our main direction is to contribute those sketches to Puffin spec, the scan API allows us to apply those indexes behind the API, thus iterating faster to verify the effectiveness of those indexes and drive better adoption of Puffin across different compute engines.

2. Integration with new languages

By providing a scan API that conforms to the OpenAPI spec, engines that are implemented in languages that are non-JVM based can now integrate with iceberg table scans in an easier way.

There will be no dependencies needed for processing Avro files, and reduced complexity for performing scan planning. This will be beneficial for the ongoing Python, Rust and Go library implementations.

3. Reading non-Iceberg tables

With the scan API, it becomes feasible to read non-Iceberg tables such as Delta and Hive-Parquet tables as Iceberg tables, by exposing them through Iceberg table model (e.g. expose Hive table as an Iceberg table with 1 snapshot plus name-mapping), and supply the right scan tasks through the scan API.

Based on our collaboration experience with internal teams, we see a huge advantage for them to adopt Iceberg through this approach. The organization can adopt REST catalog right away, and start to convert all their ML & AI pipelines to use Iceberg connectors and supported engines, while the data infrastructure team can have more time to gradually migrate non-Iceberg tables to Iceberg without impacting downstream users.

Proposal

We would like to introduce 2 APIs, CreateScan and GetScanTasks to the REST specification.

The main reason for introducing 2 APIs instead of just 1 API of GetScanTasks is that: in query engines there is typically a period of time between when the engine knows the table scan to perform, and when the engine starts to fetch scan tasks. By separating these 2 steps, the REST service can asynchronously start to do scan planning and apply optimizations like caching and indexes before the engine starts to fetch scan tasks.

API Contract for CreateScan

```
/v1/{prefix}/namespaces/{namespace}/tables/{table}/scans:
    parameters:
      - $ref: '#/components/parameters/prefix'
      - $ref: '#/components/parameters/namespace'
      - $ref: '#/components/parameters/table'
    post:
      tags:
        - Catalog API
      summary: Create a table scan for the table
      description:
        Creates a table scan for the table.
      operationId: createScan
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateScanRequest'
      responses:
        200:
          $ref: '#/components/responses/CreateScanResponse'
        400:
          $ref: '#/components/responses/BadRequestErrorResponse'
        401:
          $ref: '#/components/responses/UnauthorizedResponse'
```

```
        403:
          $ref: '#/components/responses/ForbiddenResponse'
        404:
          description:
            Not Found
            - NoSuchTableException, the table does not exist
            - NoSuchNamespaceException, the namespace does not exist
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ErrorModel'
              examples:
                TableDoesNotExist:
                  $ref: '#/components/examples/NoSuchTableError'
                NamespaceDoesNotExist:
                  $ref: '#/components/examples/NoSuchNamespaceError'
        419:
          $ref: '#/components/responses/AuthenticationTimeoutResponse'
        503:
          $ref: '#/components/responses/ServiceUnavailableResponse'
        5XX:
          $ref: '#/components/responses/ServerErrorResponse'
```

Request

In order to create scans on the service side, clients would need to send a HTTP request providing the `select` and `filter` specified by the user to the service.

```
CreateScanRequest:
  type: object
  required:
    - select
  properties:
    select:
      type: array
      items:
        type: string
    filter:
      $ref: '#/components/schemas/Expression'
    snapshot-id:
      type: integer
      Format: int64
    timestamp-ms:
      type: integer
      format: int64
```

Responses

The service will respond back to the client's request with two parameters, a `scan ID and` a list of `shards`. A shard is some data returned by the service that allows us to parallelize the work required to process a scan.

```
CreateScanResult:
      type: object
      description: create scan response
      required:
        - scan
        - shards
      properties:
        scan:
          type: string
        shards:
          type: array
          items:
            type: string
```

Here we introduce the concept of a `shard`. When creating a scan, it tells the client a list of shards that can be used to fetch different parts of the scan tasks in parallel. This simulates the experience we have today at client side for parallelizing the Iceberg manifest reading process.

API Contract for GetScanTasks

```
/v1/{prefix}/namespaces/{namespace}/tables/{table}/scans/{scan}:
    parameters:
      - $ref: '#/components/parameters/prefix'
      - $ref: '#/components/parameters/namespace'
      - $ref: '#/components/parameters/table'
      - $ref: '#/components/parameters/scan'
    get:
      tags:
        - Catalog API
      summary: Gets a list of FileScanTasks
      operationId: getScanTasks
      description:
        Gets a list of FileScanTasks
      parameters:
        - in: query
          name: shard
          description:
            Reads a single shard of a scan to obtain a sub-list of file scan tasks.
          required: true
          schema:
            type: string
        - in: query
          name: next
          description:
            Used for obtaining the next batch of fileScanTasks
          required: false
          allowEmptyValue: true
          schema:
            type: string
      responses:
        200:
          $ref: '#/components/responses/GetScanTasksResponse'
        400:
          $ref: '#/components/responses/BadRequestErrorResponse'
```

```
      401:
        $ref: '#/components/responses/UnauthorizedResponse'
      403:
        $ref: '#/components/responses/ForbiddenResponse'
      404:
        description:
          Not Found - NoSuchTableException, table to load does not exist
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/IcebergErrorResponse'
            examples:
              TableToLoadDoesNotExist:
                $ref: '#/components/examples/NoSuchTableError'
      419:
        $ref: '#/components/responses/AuthenticationTimeoutResponse'
      503:
        $ref: '#/components/responses/ServiceUnavailableResponse'
      5XX:
        $ref: '#/components/responses/ServerErrorResponse'
```

Request

In order to retrieve scan tasks from the service, clients would send a HTTP request providing the following parameters: a `scan ID` which is returned from `createScan`, an optional `shard` query parameter (which can be obtained from the shard list from `createScan`), and a `nextToken` query parameter which is used for pagination when we are retrieving large amounts of data and can only batch a certain number of `FileScanTask` over the network.

Response

The response is a list of iceberg fileScanTasks, which has been modeled after iceberg's `BaseFileScanTask` as well as a `nextToken` for pagination as explained in the request section. Within the `FileScanTask`, there is a reference to a `DataFile`, which we have modeled after iceberg's `ContentFile`.

```
GetScanTasksResult:
      description: Result of when attempting to get file scan tasks from a
scan.
      type: object
      required:
        - file-scan-tasks
      properties:
        file-scan-tasks:
          type: array
          items:
            $ref: '#/components/schemas/FileScanTask'
        next:
          type: string

 FileScanTask:
      type: object
      required:
        - data-file
```

```
    properties:
      data-file:
        $ref: '#/components/schemas/ContentFile'
      partition:
          type: object
          additionalProperties:
            type: string
      size-bytes:
        type: number
      start:
        type: number
      length:
        type: number
      estimated-rows-count:
        type: number
      delete-files:
        type: array
        items:
          $ref: '#/components/schemas/ContentFile'
      schema:
        $ref: '#/components/schemas/Schema'
      spec:
        $ref: '#/components/schemas/PartitionSpec'
      residual-filter:
        $ref: '#/components/schemas/Expression'

  ContentFile:
    type: object
    required:
      - file-path
    properties:
      spec-id:
        type: string
      content:
        $ref: '#/components/schemas/FileContent'
      file-path:
        type: string
      file-format:
        type: string
      partition:
        type: object
        additionalProperties:
          type: string
      record-count:
        type: number
      file-size-in-bytes:
        type: number
      column-sizes:
        type: object
        additionalProperties:
          type: object
          properties:
```

```
        keys:
          type: integer
        values:
          type: number
    value-counts:
      type: object
      additionalProperties:
        type: object
        properties:
          keys:
            type: integer
          values:
            type: number
    null-value-counts:
      type: object
      additionalProperties:
        type: object
        properties:
          keys:
            type: integer
          values:
            type: number
    nan-value-counts:
      type: object
      additionalProperties:
        type: object
        properties:
          keys:
            type: integer
          values:
            type: number
    lower-bounds:
      type: object
      additionalProperties:
        type: object
        properties:
          keys:
            type: integer
          values:
            type: string
            format: binary
    upper-bounds:
      type: object
      additionalProperties:
        type: object
        properties:
          keys:
            type: integer
          values:
            type: string
            format: binary
    key-metadata:
```

```
        type: string
        format: binary
      split-offsets:
        type: array
        items:
          type: number
      equality-ids:
        type: array
        items:
          type: integer
      sort-order-id:
        type: integer
      data-sequence-number:
        type: number
      file-sequence-number:
        type: number

  FileContent:
    type: string
    enum:
      - DATA
      - POSITION_DELETES
      - EQUALITY_DELETES
```

Client Side Changes Required

`RestSessionCatalog Property`

- When client calls the iceberg `getConfig` endpoint, a service will return back a property `rest.table-scan-enabled` letting the client know that the service supports the rest scan api.
- When client calls `loadTable` endpoint, a service will return back the same property letting the client know that this table allows for rest scan api support.

`RestTable`

- Introduce a new table class called `RestTable` which will extend the `BaseTable` iceberg class. The `RestTable` will be the entry point for the above workflows, and will be called by the already existing `RestSessionCatalog` during the `loadTable` call.
- Since `RestTable` extends `BaseTable` we can override its implementation of `newScan` which is expecting to retrieve a iceberg scan class with a new scan class.

`RestTableScan`

- Introduce a new Scan class called the `RestTableScan` which will extend the iceberg `DataTableScan` class.
- Since the `RestTableScan` extends the `DataTableScan` we can override its implementation of `planFiles` and write the logic in here for sending createScan requests.

`RestScanTasks`

- Introduce an inner class within the `RestTableScan` called `RestScanTasks` which will implement a **`CloseableIterable<FileScanTask>`** .
- The `RestScanTasks` will be responsible for sending the getScanTasks request and getting the `FileScanTask` from the service.

`Models, Serializers, Parsers`

- Introduce models for the CreateScanRequest, and GetScanTasksRequest, as well for the CreateScanResponse and GetScanTasksResponse
- Introduce additional serializers for `GetScanTasksResponse` and a new parser `GetFileScanTasksParser` , in order to send data and receive data from the server.

Error Handling

We will need to add new error handling around the concept of a “Scan” and a “Shard”.

For example, when we do a `GetScanTasksRequest` we pass a scanID and a shard. If the user passes an invalid scanID then we should return a `NoSuchScan`(same should apply for if no shard exists then `NoSuchShard` Error should be returned). In `GetScanTasks` we also have the `next` parameter which indicates the index of the next file scan task to obtain from the service.

During a `CreateScansRequest` if a user is passing a scan that contains selects for columns that they do have access to we should return back to the user `UnauthorizedResponse.`