DSV2 Column Index Support

Supposed we have a query like `SELECT * FROM t WHERE col = constant;` the data source has to scan row by row to find all the matching records. This is very inefficient if there are many rows in the table and only a few rows match the searching criteria. Many data sources (e.g. postgres, iceberg, etc.) support indexes to improve performance. In order to take advantage of the data source indexes support, the following APIs are proposed:

```
package org.apache.spark.sql.connector.catalog.index;

import org.apache.spark.annotation.Experimental;

/**
 * An Index Type.
 *
 * @since 3.3.0
 */
@Experimental
public enum IndexStatus {
  ACTIVE, DELETED;  //Todo: do we need other status???

  @Override
  public String toString() {
    switch (this) {
      case ACTIVE:
        return "ACTIVE";
      case DELETED:
        return "DELETED";
      default:
        throw new IllegalArgumentException("Unexpected index status: " +
          this);
    }
  }
}
```

```
package org.apache.spark.sql.connector.catalog.index;

import org.apache.spark.annotation.Experimental;

/**
 * An Index Type.
 *
 * @since 3.3.0
 */
@Experimental
public enum IndexType {
  BLOOM_FILTER_INDEX, Z_ORDERING_INDEX, BTREE_INDEX;  //Todo: other types???
```

```
  @Override
  public String toString() {
    switch (this) {
      case BLOOM_FILTER_INDEX:
        return "BLOOM_FILTER_INDEX";
      case Z_ORDERING_INDEX:
        return "Z_ORDERING_INDEX";
      case BTREE_INDEX:
        return "BTREE_INDEX";
      default:
        throw new IllegalArgumentException("Unexpected index type: " + this);
    }
  }
}
```

```
package org.apache.spark.sql.connector.catalog.index;

import java.util.Collections;
import java.util.Map;

import org.apache.spark.annotation.Evolving;
import org.apache.spark.sql.connector.catalog.Identifier;

/**
 * An Index in a table
 *
 * @since 3.3.0
 */
@Evolving
public class Index {
  private String name;
  private Identifier[] columns;
  private Identifier table;
  private String indexType;
  private Map<String, String> properties = Collections.emptyMap();

  public Index(
      String name,
      Identifier[] columns,
      Identifier table,
      String indexType,
      Map<String, String> properties) {
    this.name = name;
    this.columns = columns;
    this.table = table;
    this.indexType = indexType;
```

```
    this.properties = properties;
  }

  /**
   * @return the Index name
   */
  String name() { return name; }

  /**
   * @return the column(s) this Index is on. Could be multi-column index
   */
  Identifier[] columns() { return columns; }

  /**
   * @return the table this Index is on.
   */
  Identifier table() { return table; }

  /**
   * @return the indexType of this Index.
   */
  String indexType() { return indexType; }


  /**
   * Returns the string map of index properties.
   */
  Map<String, String> properties() {
    return Collections.emptyMap();
  }
}

package org.apache.spark.sql.connector.catalog.index;

import java.util.Map;

import org.apache.spark.annotation.Evolving;
import org.apache.spark.sql.connector.catalog.Identifier;

/**
 * Catalog methods for working with index
 *
 * @since 3.3.0
 */
@Evolving
public interface supportsIndex extends CatalogPlugin
 {
```

```
/**
 * Creates index.
 *
 * @param indexName the name of the index to be created.
 * @param table the table on which index to be created.
 * @param columns the columns on which index to be created.
 * @param type the IndexType of the index to be created.
 * @param properties the properties of the index to be created.
 * @return index object that stores information about index created.
 */
Index createIndex(String indexName,
                  Identifier table,
                  FieldReference columns,
                  String type,
                  Map<String, String> properties) throws Exception;

/**
 * Soft deletes the index with the given name.
 * Dropped index can be restored by calling restoreIndex
 * @param indexName the name of the index to be deleted.
 * @return true if the index is dropped, false if no index exists for the given name
 * @throws Exception
 */
boolean dropIndex(String indexName) throws Exception;

/**
 * Checks whether an index exists.
 *
 * @param indexName the name of the index
 * @return true if the index exists, false otherwise
 */
boolean indexExists(String indexName);

/**
 * Lists indexes in a table.
 *
 * @param table the table to be checked on for indexes.
 * @throws Exception
 */
Index[] listIndexes(Identifier table) throws Exception;

/**
 * Hard deletes the index with the given name.
 * The Index can't be restored once hard deleted
 * @param indexName the name of the index to be deleted.
 * @return true if the index is purged, false if no index exists for the given name
 * @throws UnsupportedOperationException
 */
default boolean purgeIndex(String indexName) throws UnsupportedOperationException {
  throw new UnsupportedOperationException("Purge index is not supported.");
```

```
 }

 /**
  * Restores the index with the given name.
  * Dropped index can be restored by calling restoreIndex, but purged index can't be
restored.
  * @param indexName the name of the index to be restored.
  * @return true if the index is restored, false if no index exists for the given name
or index
  *         can't be restored.
  * @throws UnsupportedOperationException
  */
 default boolean restoreIndex(String indexName) throws UnsupportedOperationException {
   throw new UnsupportedOperationException("Restore index is not supported.");
 }

 /**
  * Refreshes index using the latest data. This causes the index to be rebuilt.
  * @param indexName the name of the index to be rebuilt.
  * @return true if the index is rebuilt, false if no index exists for the given name.
  * @throws UnsupportedOperationException
  */
 default boolean refreshIndex(String indexName) throws UnsupportedOperationException {
   throw new UnsupportedOperationException("Refresh index is not supported.");
 }
}
```

When an index is defined on a column, it will be shown in the query plan as following:

```
== Physical Plan ==
*(1) Filter (isnotnull(_1#42) AND (_1#42 > 5))
+- *(1) ColumnarToRow
   +- BatchScan[_1#42] ParquetScan DataFilters: [isnotnull(_1#42), (_1#42 > 5)],
Format: parquet, Location: InMemoryFileIndex(1
paths)[file:/private/var/folders/pt/_5f4sxy56x70dv9zpz032f0m0000gn/T/spark-f9...,
ColumnIndexes: [......], PartitionFilters: [], PushedFilters: [IsNotNull(_1),
GreaterThan(_1,5)], ReadSchema: struct<_1:int>, PushedFilters: [IsNotNull(_1),
GreaterThan(_1,5)] RuntimeFilters: []
```

The new SQL syntax we need to support are:

```
CREATE [index_type] INDEX index_name ON [TABLE] table_name (column [OPTIONS
indexPropertyList] [ , ... ])
DROP INDEX index_name ON [TABLE] table_name [FOR COLUMNS (column [ , ... ])
PURGE INDEX index_name ON [TABLE] table_name [FOR COLUMNS (column [ , ... ])
RESTORE [index_type] INDEX index_name ON [TABLE] table_name (column [OPTIONS
indexPropertyList] [ , ... ])
```

Use Create Index as an example, if Spark application calls

```
sql(“CREATE BLOOM_FILTER INDEX index1 ON bloom_test (col1 OPTIONS (fpp=0.1, numItems=50000000)”)
```

Spark parses the `CREATE INDEX` statement, creates a corresponding `CREATE INDEX` command, this will be later converted to a physical plan and get executed in `CreateIndexExec`. Inside `CreateIndexExec`, `SupportsIndex.createIndex` will be called. This `SupportsIndex interface` and `createIndex` method need to be implemented by data source and are used to create indexes on data source side.