IndexDev

Indexes

Indexing Is Removed since 3.0

There are alternate options which might work similarily to indexing:

- Materialized views with automatic rewriting can result in very similar results. Hive 2.3.0 adds support for materialzed views.
- Using columnar file formats (Parquet, ORC) – they can do selective scanning; they may even skip entire files/blocks.

Indexing has been **removed** in version 3.0 (HIVE-18448).

Introduction

This document explains the proposed design for adding index support to Hive (HIVE-417). Indexing is a standard database technique, but with many possible variations. Rather than trying to provide a "one-size-fits-all" index implementation, the approach we are taking is to define indexing in a pluggable manner (related to StorageHandlers) and provide one concrete indexing implementation as a reference, leaving it open for contributors to plug in other indexing schemes as time goes by. No index support will be available until Hive 0.7.

Scope

Only single-table indexes are supported. Others (such as join indexes) may be more appropriately expressed as materialized views once Hive has support for those.

This document currently only covers index creation and maintenance. A follow-on will explain how indexes are used to optimize queries (building on FilterPushdownDev).

CREATE INDEX

```
CREATE INDEX index_name
ON TABLE base_table_name (col_name, ...)
AS 'index.handler.class.name'
[WITH DEFERRED REBUILD]
[IDXPROPERTIES (property_name=property_value, ...)]
[IN TABLE index_table_name]
[PARTITIONED BY (col_name, ...)]
[
   [ ROW FORMAT ...] STORED AS ...
   | STORED BY ...
]
[LOCATION hdfs_path]
[TBLPROPERTIES (...)]
[COMMENT "index comment"]
```

For the details of the various clauses such as ROW FORMAT, see Create Table.

By default, index partitioning matches the partitioning of the base table. The PARTITIONED BY clause may be used to specify a subset of the table's partitioning columns (this column list may be empty to indicate that the index spans all partitions of the table). For example, a table may be partitioned by date+region even though the index is partitioned by date alone (each index partition spanning all regions).

Indexes cannot be created on views. We will (eventually) support them on non-native tables (in cases where the corresponding storage handler indicates that it supports them).

Index handlers may require that the base table being indexed have a particular format.

Question: should we allow indexes on EXTERNAL tables? What does this mean for implicit DROP when the table is dropped? Is there a concept of an EXTERNAL index?

If the index handler stores its representation in tabular form, then index_table_name can be used to control the name of the "index table" automatically created for this purpose. The index table storage format can be controlled using STORED AS (e.g. RCFILE or SEQUENCFILE) or STORED BY (e.g. to store the index table in a non-native table such as HBase), although some index handlers may require usage of a specific storage format. Not all index handlers store their representation in tabular form; some may use non-table files, and others may use structures maintained completely outside of Hive (e. g. a persistent key/value store).

Metastore Model

The diagram below shows the new metastore schema with index support:

http://issues.apache.org/jira/secure/attachment/12449601/idx2.png

The new IDXS table in the metastore schema contains one entry per index created. It has two relationships with the TBLS table:

- ORIG_TBL_ID is a mandatory foreign key referencing the ID of the base table containing the data to be indexed.
- IDX_TBL_ID is an optional foreign key referencing the ID of a table containing the index representation. It is optional because not all index implementations use a table for storage. For indexes which do use a table for storage, the implicitly created table will have its TBL_TYPE set to INDEX_TABLE.

So, given the following DDL:

```
CREATE TABLE t(i int, j int);
CREATE INDEX x ON TABLE t(j)
AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';
```

The TBLS table in the metastore will have two entries:

- one for base table t
- one for the index table, automatically named as `default+t_x+`

In the IDXS entry for x, ORIG_TBL_ID will reference the TBL_ID of x, and IDX_TBL_ID will reference the TBL_ID of `default+t_x+`.

To avoid the generated name, a user-specified name such as t_x can be supplied instead:

```
CREATE INDEX x ON TABLE t(j)
AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler'
IN TABLE t_x;
```

Note that index names are qualified by the containing base table (like partitions), so the same index name can be used across two different tables. However, names of index tables are in the same namespace as all other tables and views, so they must be unique within the same database.

An index has a storage descriptor which includes the subset of columns from the original table covered by the index. If the index representation is stored in a table, most of the other fields in the index's own storage descriptor (e.g. LOCATION) will be irrelevant.

TBD:

- change IDX_TYPE to IDX_HANDLER
- what does LAST_ACCESS_TIME mean? last time the optimizer used this index?
- need LAST_REBUILD_TIME? how do we track it at partition-level? it should be in the metastore (not just HDFS)
- in the case where the index partitioning is a subset of the base table partitioning, we need a way to model this in the metastore

Metastore Upgrades

Here are the MySQL metastore upgrade statements.

```
DROP TABLE IF EXISTS {{IDXS}};
CREATE TABLE {{IDXS}} (
  {{INDEX_ID}} bigint(20) NOT NULL,
  {{CREATE_TIME}} int(11) NOT NULL,
  {{DEFERRED_REBUILD}} bit(1) NOT NULL,
  {{INDEX_HANDLER_CLASS}} varchar(256) DEFAULT NULL,
  {{INDEX_NAME}} varchar(128) DEFAULT NULL,
  {{INDEX_TBL_ID}} bigint(20) DEFAULT NULL,
  {{LAST_ACCESS_TIME}} int(11) NOT NULL,
  {{ORIG_TBL_ID}} bigint(20) DEFAULT NULL,
  {{SD_ID}} bigint(20) DEFAULT NULL,
  PRIMARY KEY ({{INDEX_ID}}),
  UNIQUE KEY {{UNIQUEINDEX}} ({{INDEX_NAME}},{{ORIG_TBL_ID}}),
  KEY {{IDXS_FK1}} ({{SD_ID}}),
  KEY {{IDXS_FK2}} ({{INDEX_TBL_ID}}),
  KEY {{IDXS_FK3}} ({{ORIG_TBL_ID}}),
  CONSTRAINT {{IDXS_FK3}} FOREIGN KEY ({{ORIG_TBL_ID}}) REFERENCES {{TBLS}} ({{TBL_ID}}),
  CONSTRAINT {{IDXS_FK1}} FOREIGN KEY ({{SD_ID}}) REFERENCES {{SDS}} ({{SD_ID}}),
  CONSTRAINT {{IDXS_FK2}} FOREIGN KEY ({{INDEX_TBL_ID}}) REFERENCES {{TBLS}} ({{TBL_ID}})
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Table structure for table {{INDEX_PARAMS}}
--

DROP TABLE IF EXISTS {{INDEX_PARAMS}};
CREATE TABLE {{INDEX_PARAMS}} (
  {{INDEX_ID}} bigint(20) NOT NULL,
  {{PARAM_KEY}} varchar(256) NOT NULL,
  {{PARAM_VALUE}} varchar(767) DEFAULT NULL,
  PRIMARY KEY ({{INDEX_ID}},{{PARAM_KEY}}),
  CONSTRAINT {{INDEX_PARAMS_FK1}} FOREIGN KEY ({{INDEX_ID}}) REFERENCES {{IDXS}} ({{INDEX_ID}})
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

REBUILD

```
ALTER INDEX index_name ON table_name [PARTITION (...)] REBUILD
```

For the PARTITION clause syntax, see LanguageManual DDL#Add_Partitions.

If WITH DEFERRED REBUILD is specified on CREATE INDEX, then the newly created index is initially empty (regardless of whether the table contains any data). The ALTER INDEX ... REBUILD command can be used to build the index structure for all partitions or a single partition.

If data in the base table changes, then the REBUILD command must be used to bring the index up to date. This is an atomic operation, so if the table was previously indexed, and a rebuild fails, then the stale index remains untouched.

DROP INDEX

```
DROP INDEX index_name ON table_name
```

An index can be dropped at any time with DROP INDEX. This will also cascade to the index table (if one exists).

Attempting to drop an index table directly with DROP TABLE will fail.

When an indexed base table is dropped, the DROP implicitly cascades to all indexes (and their corresponding index tables if any).

When an indexed base table has one of its partitions dropped, this implicitly cascades to drop corresponding partitions from all indexes.

Question: what do we do if the index partitioning granularity is not the same as the table partitioning granularity? Probably just ignore the drop, and let the user clean up manually with a new ALTER INDEX DROP PARTITION statement.

Plugin Interface

An index handler has these main responsibilities:

- During CREATE INDEX, validating the format of the base table and then generating the structure of the index table (if any) and filling any additional information into the index's storage descriptor
- During REBUILD, producing a plan for reading the base table's data and writing to the index storage and/or index table
- During DROP, deleting any index-specific storage (index tables are dropped automatically by Hive)
- During queries, participating in optimization in order to convert operators such as filters into index access plans (this part is out of scope for the moment)

The corresponding Java inerface is defined below, together with a companion abstract base class which handlers should extend.

```
package org.apache.hadoop.hive.ql.metadata;

import java.util.List;

import org.apache.hadoop.conf.Configurable;

import org.apache.hadoop.hive.ql.plan.api.Task;

/**
 * HiveIndexHandler defines a pluggable interface for adding new
 * index handlers to Hive.
 */
public interface HiveIndexHandler extends Configurable
{
  /**
 * Determines whether this handler implements indexes by creating
 * an index table.
   *
 * @return true if index creation implies creation of an index table in Hive;
 * false if the index representation is not stored in a Hive table
   */
  boolean usesIndexTable();

  /**
 * Requests that the handler validate an index definition and
 * fill in additional information about its stored representation.
 *
 * @param baseTable the definition of the table being indexed
   *
 * @param index the definition of the index being created
   *
 * @param indexTable a partial definition of the index table to be used for
 * storing the index representation, or null if usesIndexTable() returns
 * false; the handler can augment the index's storage descriptor
 * (e.g. with information about input/output format)
 * and/or the index table's definition (typically with additional
 * columns containing the index representation, e.g. pointers into HDFS)
   *
 * @throw HiveException if the index definition is invalid with
 * respect to either the base table or the supplied index table definition
   */
  void analyzeIndexDefinition(
    org.apache.hadoop.hive.metastore.api.Table baseTable,
    org.apache.hadoop.hive.metastore.api.Index index,
    org.apache.hadoop.hive.metastore.api.Table indexTable)
      throws HiveException;

  /**
 * Requests that the handler generate a plan for building the index;
 * the plan should read the base table and write out the index representation.
   *
 * @param baseTable the definition of the table being indexed
   *
 * @param index the definition of the index
   *
 * @param partitions a list of specific partitions of the base
 * table for which the index should be built, or null if
 * an index for the entire table should be rebuilt
   *
 * @param indexTable the definition of the index table, or
 * null if usesIndexTable() returns null
   *
```

```
 * @return list of tasks to be executed in parallel for building
 * the index
   *
 * @throw HiveException if plan generation fails
   */
  List<Task<?>> generateIndexBuildTaskList(
    org.apache.hadoop.hive.metastore.api.Table baseTable,
    org.apache.hadoop.hive.metastore.api.Index index,
    List<org.apache.hadoop.hive.metastore.api.Partition> partitions,
    org.apache.hadoop.hive.metastore.api.Table indexTable)
      throws HiveException;
}

/**
 * Abstract base class for index handlers.  This is provided as insulation
 * so that as HiveIndexHandler evolves, default implementations of new
 * methods can be added here in order to avoid breaking existing
 * plugin implementations.
 */
public abstract class AbstractIndexHandler implements HiveIndexHandler
{
}
```

For CREATE INDEX, Hive first calls usesIndexTable() on the handler to determine whether an index table will be created. If this returns false, the statement fails immediately if the user specified any table storage options for the index. However, if usesIndexTable() returns true, then Hive creates a partial table definition for the index table based on the index definition (such as the covered columns) combined with any table storage options supplied by the user. Next, Hive calls analyzeIndexDefinition (passing either null or the partial index table definition for the indexTable parameter). The handler responds by validating the definitions (throwing an exception if any unsupported combination is detected) and then filling in additional information on the index and indexTable parameters as output. Hive then stores these results in the metastore.

TBD: we will be adding methods for calling the handler when an index is dropped (e.g. to give a cleanup opportunity to a handler which stores the index representation in an external system such as HBase)

Reference Implementation

The reference implementation creates what is referred to as a "compact" index. This means that rather than storing the HDFS location of each occurrence of a particular value, it only stores the addresses of HDFS blocks containing that value. This is optimized for point-lookups in the case where a value typically occurs more than once in nearby rows; the index size is kept small since there are many fewer blocks than rows. The tradeoff is that extra work is required during queries in order to filter out the other rows from the indexed blocks.

The compact index is stored in an index table. The index table columns consist of the indexed columns from the base table followed by a _bucketname string column (indicating the name of the file containing the indexed block) followed by an _offsets array<string> column (indicating the block offsets within the corresponding file). The index table is stored as sorted on the indexed columns (but not on the generated columns).

The reference implementation can be plugged in with

```
ADD JAR /path/to/hive_idx-compact.jar;
CREATE INDEX ...
AS 'org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler';
```

TBD: algorithm for building the index

TBD: mechanism for searching the index

TBD: validation on base table (can be any managed table?)

TBD: validation on index table format (can be any managed table format?)

TBD

- specs for SHOW/DESCRIBE INDEX (HIVE-1497)
- ALTER INDEX DROP PARTITION?
- ALTER INDEX SET IDXPROPERTIES, change tableformat, etc
- what happens when the structure of a table or partition changes after it has already been indexed
- automatic indexing as part of INSERT when WITH DEFERRED REBUILD is not specified
- prevent creation of an index on an index table?
- metastore upgrade script
- stats collection for index tables
- intersection with new Hive concurrency control feature (what locks do we take for various index operations?)

Current Status (JIRA)

T	Key	Summary	Assignee	Reporter	P	Status	Resolution	Created	Updated	Due
	HIVE-21792	Hive Indexes... Again	Unassigned	David Mollitor		OPEN	Unresolved	May 24, 2019	May 24, 2019	
	HIVE-18448	Drop Support For Indexes From Apache Hive	Zoltan Haindrich	David Mollitor		CLOSED	Fixed	Jan 12, 2018	Jul 09, 2019	
	HIVE-18035	NullPointerException on querying a table with a compact index	Unassigned	Brecht Machiels		OPEN	Unresolved	Nov 09, 2017	Nov 13, 2017	
	HIVE-15282	Different modification times are used when an index is built and when its staleness is checked	Marta Kuczora	Marta Kuczora		RESOLVED	Fixed	Nov 24, 2016	Jul 21, 2017	
	HIVE-13844	Invalid index handler in org.apache.hadoop.hive.ql.index.HiveIndex class	Svetozar Ivanov	Svetozar Ivanov		CLOSED	Fixed	May 25, 2016	Jun 21, 2016	
	HIVE-13377	Lost rows when using compact index on parquet table	Unassigned	Gabriel C Balan		OPEN	Unresolved	Mar 29, 2016	Mar 29, 2016	
	HIVE-12877	Hive use index for queries will lose some data if the Query file is compressed.	Unassigned	yangfang		PATCH AVAILABLE	Unresolved	Jan 15, 2016	Feb 01, 2016	Jan 15, 2016
	HIVE-11227	Kryo exception during table creation in Hive	Unassigned	Akamai		OPEN	Unresolved	Jul 10, 2015	Jul 17, 2015	Jul 12, 2015
	HIVE-11154	Indexing not activated with left outer join and where clause	Bennie Can	Bennie Can		OPEN	Unresolved	Jun 30, 2015	Jun 30, 2015	Jul 11, 2015
	HIVE-10021	"Alter index rebuild" statements submitted through HiveServer2 fail when Sentry is enabled	Aihua Xu	Richard Williams		CLOSED	Fixed	Mar 19, 2015	Feb 16, 2016	
	HIVE-9656	Create Index Failed without WITH DEFERRED REBUILD	Chaoyu Tang	Will Du		OPEN	Unresolved	Feb 11, 2015	Sep 28, 2015	
	HIVE-9639	Create Index failed in Multiple version of Hive running	Unassigned	Will Du		OPEN	Unresolved	Feb 10, 2015	Mar 14, 2015	
	HIVE-8475	add test case for use of index from not-current database	Thejas M Nair	Thejas M Nair		CLOSED	Fixed	Oct 15, 2014	Nov 13, 2014	
	HIVE-7692	when table is dropped associated indexes also should be dropped	Thejas M Nair	Thejas M Nair		RESOLVED	Not A Problem	Aug 12, 2014	Nov 04, 2014	
	HIVE-7239	Fix bug in HiveIndexedInputFormat implementation that causes incorrect query result when input backed by Sequence/RC files	Illya Yalovyy	Sumit Kumar		CLOSED	Fixed	Jun 16, 2014	Jul 26, 2017	
	HIVE-6996	FS based stats broken with indexed tables	Ashutosh Chauhan	Ashutosh Chauhan		CLOSED	Fixed	Apr 30, 2014	Jun 09, 2014	
	HIVE-6921	index creation fails with sql std auth turned on	Ashutosh Chauhan	Ashutosh Chauhan		CLOSED	Fixed	Apr 16, 2014	Jun 09, 2014	
	HIVE-5902	Cannot create INDEX on TABLE in HIVE 0.12	Unassigned	Juraj Volentier		OPEN	Unresolved	Nov 27, 2013	Mar 14, 2015	

	HIVE-5664	Drop cascade database fails when the db has any tables with indexes	Venki Korukanti	Venki Korukanti		CLOSED	Fixed	Oct 28, 2013	Feb 19, 2015
	HIVE-5631	Index creation on a skew table fails	Venki Korukanti	Venki Korukanti		CLOSED	Fixed	Oct 23, 2013	Feb 19, 2015

Showing 20 out of 57 issues