Vectorized Query Execution

# Introduction

Vectorized query execution is a Hive feature that greatly reduces the CPU usage for typical query operations like scans, filters, aggregates, and joins. A standard query execution system processes one row at a time. This involves long code paths and significant metadata interpretation in the inner loop of execution. Vectorized query execution streamlines operations by processing a block of 1024 rows at a time. Within the block, each column is stored as a vector (an array of a primitive data type). Simple operations like arithmetic and comparisons are done by quickly iterating through the vectors in a tight loop, with no or very few function calls or conditional branches inside the loop. These loops compile in a streamlined way that uses relatively few instructions and finishes each instruction in fewer clock cycles, on average, by effectively using the processor pipeline and cache memory. A detailed design document is attached to the vectorized query execution JIRA, at <https://issues.apache.org/jira/browse/HIVE-4160>.

# Using Vectorized Query Execution

## **Enabling vectorized execution**

To use vectorized query execution, you must store your data in [ORC](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC) format, and set the following variable as shown in Hive SQL (see [Configuring Hive](https://cwiki.apache.org/confluence/display/Hive/AdminManual+Configuration" \l "AdminManualConfiguration-ConfiguringHive)):

set hive.vectorized.execution.enabled = true;

Vectorized execution is off by default, so your queries only utilize it if this variable is turned on. To disable vectorized execution and go back to standard execution, do the following:

set hive.vectorized.execution.enabled = false;

Additional configuration variables for vectorized execution are documented in [Configuration Properties – Vectorization](https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties" \l "ConfigurationProperties-Vectorization).

## **Supported data types and operations**

The following data types are currently supported for vectorized execution:

* tinyint
* smallint
* int
* bigint
* boolean
* float
* double
* decimal
* date
* timestamp (see [Limitations](https://cwiki.apache.org/confluence/display/Hive/Vectorized+Query+Execution" \l "VectorizedQueryExecution-Limitations) below)
* string

Using other data types will cause your query to execute using standard, row-at-a-time execution.

The following expressions can be vectorized when used on supported types:

* arithmetic: +, -, \*, /, %
* AND, OR, NOT
* comparisons <, >, <=, >=, =, !=, BETWEEN, IN ( list-of-constants ) as filters
* Boolean-valued expressions (non-filters) using AND, OR, NOT, <, >, <=, >=, =, !=
* IS [NOT] NULL
* all math functions (SIN, LOG, etc.)
* string functions SUBSTR, CONCAT, TRIM, LTRIM, RTRIM, LOWER, UPPER, LENGTH
* type casts
* Hive user-defined functions, including standard and generic UDFs
* date functions (YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, UNIX\_TIMESTAMP)
* the IF conditional expression

User-defined functions are supported using a backward compatibility bridge, so although they do run vectorized, they don't run as fast as optimized vector implementations of built-in operators and functions. Vectorized filter operations are evaluated left-to-right, so for best performance, put UDFs on the right in an ANDed list of expressions in the WHERE clause. E.g., use

column1 = 10 and myUDF(column2) = "x"

instead of

myUDF(column2) = "x" and column1 = 10

This will allow the optimized filter to run first, potentially eliminating many rows from consideration, before running the UDF via the bridge. The UDF will only be run for rows that pass the filter on the left hand side of the AND operation.

Using a built-in operator or function that is not supported for vectorization will cause your query to run in standard row-at-a-time mode. If a compile time or run time error occurs that appears related to vectorization, please file a [Hive JIRA](https://issues.apache.org/jira/browse/HIVE). To work around such an error, disable vectorization by setting hive.vectorized.execution.enabled to false for the specific query that is failing, to run it in standard mode.

Vectorized support continues to be added for additional functions and expressions. If you have a request for one, please comment on this page, or open a JIRA for it.

## **Seeing whether vectorization is used for a query**

You can verify which parts of your query are being vectorized using the [**explain**](https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Explain) feature. For example, with vectorization enabled and the table **all types are** stored in ORC format, for this query:

|  |
| --- |
| select csmallint  from alltypesorc  where csmallint > 0; |

## Fetch Optimization for Simple Queries

For certain simple queries, running a MapReduce task involves much overhead, which can be avoided by using a FETCH task. The hive.fetch.task.conversion property specifies that queries that meet the appropriate conditions be converted to a single FETCH task, minimizing latency.

set hive.fetch.task.conversion=more;

Currently the query must be single sourced, not have any subqueries, and must not have any aggregations or distincts (which incur RS – ReduceSinkOperator, requiring a MapReduce task), or lateral views and joins. The default is "minimal".

Supported values are "minimal" and "more". They support queries as follows:

* minimal: SELECT \*, FILTER on partition columns (WHERE and HAVING clauses), LIMIT only
* more: SELECT, FILTER, LIMIT only (including TABLESAMPLE, virtual columns)

"more" can take any kind of expressions in the SELECT clause, including UDFs.

# Stream table in Joins

Joins play an important role when you need to get information from multiple tables but when you have 1.5 Billion+ records in one table and joining it with master table, the order of joining table becomes very important.

Lets consider we have a table named **'foo'** with **1.5 Billion+ records** which we are joining with another table called **'bar'** with 100 records like:

**select foo.a,foo.b,bar.c from foo join bar on foo.a=bar.a;**

Since Hive streams right-most table(bar) and buffer(in-memory) other tables(foo) before performing map-side/reduce-side join. Hence, if you buffer 1.5 Billion+ records, your join query will fail as buffering 1.5 Billion records will definitely results in Java-Heap space exception.

So, to overcome this limitation and free the user to remember the order of joining tables based on their record-size, Hive provides a key-word **/\*+ STREAMTABLE(foo) \*/** which tells Hive Analyzer to stream table foo.

**Select /\*+ STREAMTABLE(foo) \*/ foo.a,foo.b,bar.c from foo join bar on foo.a=bar.a;**

Hence, in this way user can be free of remembering the order of joining tables.