TREASURE DATA

Hive Guide

Version 2.0

Hive Guide by Treasure Data

Overview

This document was prepared by **Treasure Data, Inc.**. If you do not want to run your own Hive cluster, **check us out**.

About Apache Hive and the Hive Query Language

The Hive Query Language (HiveQL) is the primary data processing method for Treasure Data. HiveQL is powered by **Apache Hive**. **Treasure Data** is a cloud data platform that allows users to collect, store, and analyze their data on the cloud. Treasure Data manages its own Hadoop cluster, which accepts queries from users and executes them using the Hadoop MapReduce framework. HiveQL is one the languages it supports.

Example Hive Query Catalog

To get you started immediately in Hive, please visit Treasure Data's example query catalog for dozens of HiveQL templates.

HiveQL Example Query Catalog

Examples Based on Industry

- **E-Commerce**
- **Gaming**
- **Web Logs**
- **Point of Sale**

Using this Guide

Sample Dataset

Throughout this guide, we will use the following datasets and corresponding data to help demonstrate queries:

Table 1 - Owners

ID	Name	Age	Car_ID
1	Sam	52	3
2	Bob	35	1
3	Jane	18	5
4	Chris	16	4
5	Ashley	28	2

Table 2 - Cars

ID	Make	Model	Year
1	toyota	corolla	2013
2	ford	focus	2013
3	audi	a4	2015
4	toyota	camry	2007
5	kia	sportage	2002

Introduction to the Hive Query Language

The SELECT Statement

The SELECT Statement is the heart of SQL and consists of a collection of required and optional clauses.

The following clauses are required for proper execution:
- SELECT : Defines the columns and/or functions, `select_expr`, needed
- FROM : Defines the `table_reference`: a table, view, join construct, or subquery, in which the remaining clauses may find the data

The remaining clauses are optional clauses:
- WHERE : Defines the `where_condition` that contains logical operations to filter the data set
- GROUP BY : Contains the `col_list to` group on for aggregation functions
- HAVING : Defines the `having_condition` that contains logical operations to filter the returned results
- ORDER BY : Contains the `col_list` that defines how to sort the results
- LIMIT : Defines the number, `n`, of result rows to return

Syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, select_expr, ….
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having condition]
[ORDER BY col_list]
[LIMIT n]
```

Tip:

To select all of the columns available in a given `table_reference`, use the `*` operator:

```
SELECT *
FROM table_reference;
```

`*` can also be used with the COUNT() function to signify all rows should be included. Please note that in HiveQL that COUNT(*) and COUNT(1) are equivalent. See the section on Functions.

Example:

The below query will total number of rows in the `people` table.

```
SELECT COUNT(*)
FROM owners;

Result: 5
```

Filtering Data with the WHERE Clause

The WHERE clause is used to filter the data coming from the given table_reference by using predicate operators and logical operators. Functions can also be used to compute the conditions.

Syntax:

```
SELECT colA,
       colB
FROM tableA
WHERE colA = ‘example’;
```

Using Predicate Operators

A predicate in HiveQL is an expression statement that will return TRUE, FALSE or NULL. You use predicates in the WHERE clause to filter your data depending on the result of each statement. You can chain multiple predicates in a WHERE clause using Logical Operators (see below).

List of Predicate Operators

Operator	Type	Description
A = **B**	All primitive types	**TRUE** if expression **A** is equal to **B**, otherwise **FALSE**
A <=> **B**	All primitive types	Returns same result with EQUAL(=) operator for non-null operands, but returns **TRUE** if both are **NULL**, **FALSE** if one of the them is **NULL** (as of version 0.9.0)
A == **B**	NONE!	Fails because of invalid syntax. SQL uses =, not ==
A <> **B**	All primitive types	**NULL** if **A** or **B** is **NULL**, **TRUE** if **A** is **NOT** equal to **B**, otherwise **FALSE**
A != **B**	All primitive types	A synonym for the <> operator
A < **B**	All primitive types	**NULL** if **A** or **B** is **NULL**, **TRUE** if **A** is less than **B**, otherwise **FALSE**
A <= **B**	All primitive types	**NULL** if **A** or **B** is **NULL**, **TRUE** if **A** is less than or equal to **B**, otherwise **FALSE**
A > **B**	All primitive types	**NULL** if **A** or **B** is **NULL**, **TRUE** if **A** is greater than **B**, otherwise **FALSE**
A >= **B**	All primitive types	**NULL** if **A** or **B** is **NULL**, **TRUE** if **A** is greater than or equal to **B**, otherwise **FALSE**
A [NOT] BETWEEN **B** AND **C**	All primitive types	**NULL** if **A**, **B**, or **C** are **NULL**, **TRUE** if A is greater than or equal to **B** AND **A** is less than or equal to **C**, otherwise **FALSE**. This can be inverted by using the **NOT** keyword. (as of version 0.9.0)
A IS **NULL**	All types	**TRUE** if A evaluates to **NULL**, otherwise **FALSE**
A IS NOT **NULL**	All types	**FALSE** if A evaluates to **NULL**, otherwise **TRUE**

Operator	Type	Description
A [NOT] LIKE **B**	Strings	**NULL** is **A** or **B** is **NULL**, **TRUE** if string **A** matches the SQL simple regular expression **B**, otherwise **FALSE**. See "Using Regular Expressions in HiveQL" for more information
A [NOT] RLIKE **B**	Strings	**NULL** if **A** or **B** is **NULL**, **TRUE** if any (possibly empty) substring of **A** matches the Java regular expression **B**, otherwise **FALSE**
A REGEXP **B**	Strings	Same as RLIKE

Using Logical Operators

If you need to combine multiple predicates within a WHERE clause, you can use a logical operator.

List of Logical Operators

Operator	Type	Description
A AND **B**	boolean	**TRUE** if both **A** and **B** are **TRUE**, otherwise **FALSE**. **NULL** if **A** or **B** is **NULL**
A && **B**	boolean	A synonym of **A** AND **B**
A OR **B**	boolean	**TRUE** if either **A** or **B** are **TRUE** or if both are **TRUE**, otherwise **FALSE**. If one predicate is **NULL** and the other **FALSE**, then **NULL**
A || **B**	boolean	**A** synonym of **A** OR **B**
NOT **A**	boolean	**TRUE** if **A** is **FALSE**, **NULL** if **A** is **NULL**. Otherwise **FALSE**
! **A**	boolean	A synonym of NOT **A**
A IN (val1, val2, ...)	boolean	**TRUE** if **A** is equal to any of the values listed, otherwise **FALSE**
A NOT IN (val1, val2, ...)	boolean	**TRUE** if **A** is not equal to any of the values listed, otherwise **FALSE**

Filtering Using Partitions

In HDFS (Hadoop Distributed File System), data is sectioned into partitions. Most partitions are based on some sort of time variable, such as by day or hour. You can reference your partition as a column in a query, just like any other factor. Any query that references partitions will allow the query to run more efficiently by only reading data that resides in the partitions referenced.

Please refer to the "Leveraging Time-based Partitioning" section on the **Performance Tuning** page for more information.

Finding Unique Combinations with DISTINCT

Used within the SELECT clause of a SQL statement, the DISTINCT keyword indicates whether or not to return duplicate rows from the result set.

Example

Given the following table:

1	a
1	a
1	b
2	c

Query:

```
SELECT DISTINCT colA, colB from tableA;
```

Result:

1	a
1	b
2	c

Tip:

DISTINCT can also be used with the COUNT() function to count the unique set of values in that column. See the section on Functions.

Example:

Using our `cars` table, we would like to know how many different makes of cars we have in our database. The below query will return the set of unique car makes within `cars`.

```
SELECT COUNT(DISTINCT make)
FROM cars;

Query plan: COUNT([audi, ford, kia, toyota])

Return: 4
```

Using Functions in HiveQL

Functions can be used to manipulate column values in either the SELECT clause, to create new columns, or within the WHERE clause.

Syntax:

The below query would turn the lettering in colA to all uppercase:

```
SELECT upper(colA), colB
FROM tableA;
```

List of Arithmetic Operators

Operator	Type	Description
A + **B**	Numbers	Gives the result of adding **A** and **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands. E.g. since every integer is a float, therefore float is a containing type of integer so the + operator on a float and an integer will result in a float.
A - **B**	Numbers	Gives the result of subtracting **B** from **A**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands.
A * **B**	Numbers	Gives the result of multiplying **A** and **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands. Note: If the result of the multiplications causes overflow, you will have to cast one of the operators to a type higher in the type hierarchy **(see Tip below)**.
A / **B**	Numbers	Gives the result of dividing **A** by **B**. The result is a double type.
A % **B**	Numbers	Gives the remainder resulting from dividing **A** by **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands.
A & **B**	Numbers	Gives the result of bitwise AND of **A** and **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands.
A | **B**	Numbers	Gives the result of bitwise OR of **A** and **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands.
A ^ **B**	Numbers	Gives the result of bitwise XOR of **A** and **B**. The type of the result is the same as the common parent (in the type hierarchy) of the types of the operands.
~ **A**	Numbers	Gives the result of bitwise NOT of **A**. The type of the result is the same as the type of **A**.

Tip:

When using the multiplication operator, you may run into overflow. To prevent or fix this from occuring, you can cast one or both predicates to the next larger data type. For example, cast an INT to BIGINT, a FLOAT to a DOUBLE, etc. The table belows shows the sizes and values associated with each data type.

Data Type	Size	Range
TINYINT	1-byte signed integer	-127 .. 128
SMALLINT	2-byte signed integer	-32,768 .. 32,767
INT	4-byte signed integer	-2,147,483,648 .. 2,147,483,647
BIGINT	8-byte signed integer	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
FLOAT	4-byte single precision floating point number	
DOUBLE	8-byte double precision floating point number	

Example:

Assume colA and colB are both INT types. At some point within the table, the product of these two columns produces a number that falls outside the values of -2,147,483,648 to 2,147,483,647, which will cause an error in the output. To fix this, you will have to cast at least one of them to a larger data type value.

```
SELECT colA * CAST(colB AS BIGINT) AS productAB
FROM table1;
```

List of Mathematical Functions

Name (Signature)	Return Type	Description
round(DOUBLE **a**)	DOUBLE	Returns the rounded BIGINT value of **a**.
round(DOUBLE **a**, INT **d**)	DOUBLE	Returns **a** rounded to **d** decimal places.
bround(DOUBLE **a**)	DOUBLE	Returns the rounded BIGINT value of **a** using HALF_EVEN rounding mode (as of **Hive 1.3.0, 2.0.0**). Also known as Gaussian rounding or bankers' rounding. Example: bround(2.5) = 2, bround(3.5) = 4.
bround(DOUBLE **a**, INT **d**)	DOUBLE	Returns **a** rounded to **d** decimal places using HALF_EVEN rounding mode (as of **Hive 1.3.0, 2.0.0**). Example: bround(8.25, 1) = 8.2, bround(8.35, 1) = 8.4.

Name (Signature)	Return Type	Description
floor(DOUBLE **a**)	BIGINT	Returns the maximum BIGINT value that is equal to or less than **a**.
ceil(DOUBLE **a**), ceiling(DOUBLE **a**)	BIGINT	Returns the minimum BIGINT value that is equal to or greater than **a**
rand(), rand(INT **seed**)	DOUBLE	Returns a random number (that changes from row to row) that is distributed uniformly from 0 to 1. Specifying the **seed** will make sure the generated random number sequence is deterministic.
exp(DOUBLE **a**), exp(DECIMAL **a**)	DOUBLE	Returns e^{a} where e is the base of the natural logarithm. Decimal version added in **Hive 0.13.0**
ln(DOUBLE **a**), ln(DECIMAL **a**)	DOUBLE	Returns the natural logarithm of the argument **a**. Decimal version added in **Hive 0.13.0**
log10(DOUBLE **a**), log10(DECIMAL **a**)	DOUBLE	Returns the base-10 logarithm of the argument **a**. Decimal version added in **Hive 0.13.0**
log2(DOUBLE **a**), log2(DECIMAL **a**)	DOUBLE	Returns the base-2 logarithm of the argument **a**. Decimal version added in **Hive 0.13.0**
log(DOUBLE **base**, DOUBLE **a**) log(DECIMAL **base**, DECIMAL **a**)	DOUBLE	Returns the base-base logarithm of the argument **a**. Decimal versions added in **Hive 0.13.0**
pow(DOUBLE **a**, DOUBLE **p**), power(DOUBLE **a**, DOUBLE **p**)	DOUBLE	Returns $\mathbf{a}^{p}$.
sqrt(DOUBLE **a**), sqrt(DECIMAL **a**)	DOUBLE	Returns the square root of **a**. Decimal version added in **Hive 0.13.0**
bin(BIGINT **a**)	STRING	Returns the number **a** in binary format (see**http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_bin**).
hex(BIGINT **a**) hex(STRING **a**) hex(BINARY **a**)	STRING	If the argument is an INT or binary, hex returns the number as a STRING in hexadecimal format. Otherwise if the number is a STRING, it converts each character into its hexadecimal representation and returns the resulting STRING. (See**http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_hex**, BINARY version as of Hive 0.12.0.)
unhex(STRING **a**)	BINARY	Inverse of hex. Interprets each pair of characters as a hexadecimal number and converts to the byte representation of the number. (BINARY version as of Hive 0.12.0, used to return a string.)

Name (Signature)	Return Type	Description
conv(BIGINT **num**, INT **from_base**, INT **to_base**), conv(STRING **num**, INT **from_base**, INT **to_base**)	STRING	Converts a number from a given base to another (see**http://dev.mysql.com/doc/refman/5.0/en/mathematical-functions.html#function_conv**).
abs(DOUBLE **a**)	DOUBLE	Returns the absolute value.
pmod(INT **a**, INT **b**), pmod(DOUBLE **a**, DOUBLE **b**)	INT or DOUBLE	Returns the positive value of **a** mod **b**.
sin(DOUBLE **a**), sin(DECIMAL **a**)	DOUBLE	Returns the sine of **a** (**a** is in radians). Decimal version added in **Hive 0.13.0**
asin(DOUBLE **a**), asin(DECIMAL **a**)	DOUBLE	Returns the arc sin of **a** if -1<=**a**<=1 or NULL otherwise. Decimal version added in **Hive 0.13.0**
cos(DOUBLE **a**), cos(DECIMAL **a**)	DOUBLE	Returns the cosine of **a** (**a** is in radians). Decimal version added in **Hive 0.13.0**
acos(DOUBLE **a**), acos(DECIMAL **a**)	DOUBLE	Returns the arccosine of **a** if -1<=**a**<=1 or NULL otherwise. Decimal version added in **Hive 0.13.0**
tan(DOUBLE **a**), tan(DECIMAL **a**)	DOUBLE	Returns the tangent of **a** (**a** is in radians). Decimal version added in **Hive 0.13.0**
atan(DOUBLE **a**), atan(DECIMAL **a**)	DOUBLE	Returns the arctangent of **a**. Decimal version added in **Hive 0.13.0**.
degrees(DOUBLE **a**), degrees(DECIMAL **a**)	DOUBLE	Converts value of **a** from radians to degrees. Decimal version added in **Hive 0.13.0**
radians(DOUBLE **a**), radians(DOUBLE **a**)	DOUBLE	Converts value of **a** from degrees to radians. Decimal version added in **Hive 0.13.0**
positive(INT **a**), positive(DOUBLE **a**)	INT or DOUBLE	Returns |**a**|.
negative(INT **a**), negative(DOUBLE **a**)	INT or DOUBLE	Returns -**a**.
sign(DOUBLE **a**), sign(DECIMAL **a**)	DOUBLE or INT	Returns the sign of **a** as '1.0' (if **a** is positive) or '-1.0' (if **a** is negative), '0.0' otherwise. The decimal version returns INT instead of DOUBLE. Decimal version added in **Hive 0.13.0**.
e()	DOUBLE	Returns the value of e.
pi()	DOUBLE	Returns the value of pi.
factorial(INT **a**)	BIGINT	Returns the factorial of **a** (as of Hive **1.2.0**). Valid if **a** is between [0..20].

Name (Signature)	Return Type	Description
cbrt(DOUBLE **a**)	DOUBLE	Returns the cube root of a double value (as of Hive 1.2.0).
shiftleft(INT **a**), shiftleft(BIGINT **a**)	INT or BIGINT	Bitwise left shift (as of Hive 1.2.0). Returns int for tinyint, smallint, and int **a**. Returns bigint for bigint **a**.
shiftright(INT **a**), shiftright(BIGINT **a**)	INT or BIGINT	Bitwise right shift (as of Hive 1.2.0). Returns int for tinyint, smallint, and int **a**. Returns bigint for bigint **a**.
shiftrightunsigned(INT **a**), shiftrightunsigned(BIGINT **a**)	INT or BIGINT	Bitwise unsigned right shift (as of Hive 1.2.0). Returns int for tinyint, smallint, and int **a**. Returns bigint for bigint **a**.

List of Collection Functions

Name (Signature)	Return Type	Description
map_keys(**Map<K.V>**)	array<K>	Returns an unordered array containing the keys of the input **Map**.
sort_array(**Array<T>**)	array<t>	Sorts the input **Array** in ascending order according to the natural ordering of the array elements and returns it (as of version 0.9.0).
map_values(**Map<K.V>**)	array<V>	Returns an unordered array containing the values of the input **Map**.
array_contains(**Array<T>**, **value**)	boolean	Returns **TRUE** if the **Array** contains **value**.
size(**Map<K.V>**)	int	Returns the number of elements in the **Map**.
size(**Array<T>**)	int	Returns the number of elements in the **Array**.

List of Date Functions

Name (Signature)	Return Type	Description
from_unixtime(bigint **unixtime**[, string **format**])	string	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the format of "1970-01-01 00:00:00".
unix_timestamp()	bigint	Gets current Unix timestamp in seconds.

Name (Signature)	Return Type	Description
unix_timestamp(string **date**)	bigint	Converts time string in format yyyy-MM-dd HH:mm:ss to Unix timestamp (in seconds), using the default timezone and the default locale, return **0** if fail: unix_timestamp('2009-03-20 11:30:01') = 1237573801
unix_timestamp(string **date**, string **pattern**)	bigint	Convert time string with given pattern (see [**http://docs.oracle.com/javase/tutorial/i18n/format/simpleDateFormat.html**]) to Unix time stamp (in seconds), return **0** if fail: unix_timestamp('2009-03-20', 'yyyy-MM-dd') = 1237532400.
to_date(string **timestamp**)	string	Returns the date part of a timestamp string: to_date("1970-01-01 00:00:00") = "1970-01-01".
year(string **date**)	int	Returns the year part of a **date** or a **timestamp** string: year("1970-01-01 00:00:00") = 1970, year("1970-01-01") = 1970.
quarter(date/timestamp/string **d**)	int	Returns the quarter of the year for a **date**, **timestamp**, or **string** in the range 1 to 4 (as of Hive **1.3.0**). Example: quarter('2015-04-08') = 2.
month(string **date**)	int	Returns the month part of a **date** or a **timestamp** string: month("1970-11-01 00:00:00") = 11, month("1970-11-01") = 11.

day(string **date**)
dayofmonth(string **date**)
	int	Returns the day part of a **date** or a **timestamp** string: day("1970-11-01 00:00:00") = 1, day("1970-11-01") = 1.

hour(string **date**)	int	Returns the hour of the **timestamp**: hour('2009-07-30 12:58:59') = 12, hour('12:58:59') = 12.
minute(string **date**)	int	Returns the minute of the **timestamp**.
second(string **date**)	int	Returns the second of the **timestamp**.
weekofyear(string **date**)	int	Returns the week number of a **timestamp** string: weekofyear("1970-11-01 00:00:00") = 44, weekofyear("1970-11-01") = 44.
datediff(string **enddate**, string **startdate**)	int	Returns the number of days from **startdate** to **enddate**: datediff('2009-03-01', '2009-02-27') = 2.
date_add(string **startdate**, int **days**)	string	Adds a number of **days** to **startdate**: date_add('2008-12-31', 1) = '2009-01-01'.
date_sub(string **startdate**, int **days**)	string	Subtracts a number of **days** to **startdate**: date_sub('2008-12-31', 1) = '2008-12-30'.

Name (Signature)	Return Type	Description
from_utc_timestamp(**timestamp**, string **timezone**)	timestamp	Assumes given **timestamp** is UTC and converts to given **timezone** (as of Hive 0.8.0). For example, from_utc_timestamp('1970-01-01 08:00:00','PST') returns 1970-01-01 00:00:00.
to_utc_timestamp(**timestamp**, string **timezone**)	timestamp	Assumes given **timestamp** is in given **timezone** and converts to UTC (as of Hive 0.8.0). For example, to_utc_timestamp('1970-01-01 00:00:00','PST') returns 1970-01-01 08:00:00.
current_date	date	Returns the current date at the start of query evaluation (as of Hive 1.2.0). All calls of current_date within the same query return the same value.
current_timestamp	timestamp	Returns the current timestamp at the start of query evaluation (as of Hive1.2.0). All calls of current_timestamp within the same query return the same value.
add_months(string **start_date**, int **num_months**)	string	Returns the date that is **num_months** after **start_date** (as of Hive 1.1.0). **start_date** is a string, date or timestamp. **num_months** is an integer. The time part of **start_date** is ignored. If **start_date** is the last day of the month or if the resulting month has fewer days than the day component of **start_date**, then the result is the last day of the resulting month. Otherwise, the result has the same day component as **start_date**.
last_day(string **date**)	string	Returns the last day of the month which the **date** belongs to (as of Hive1.1.0). **date** is a string in the format 'yyyy-MM-dd HH:mm:ss' or 'yyyy-MM-dd'. The time part of **date** is ignored.
next_day(string **start_date**, string **day_of_week**)	string	Returns the first date which is later than **start_date** and named as **day_of_week** (as of Hive 1.2.0). **start_date** is a string/date/timestamp. **day_of_week** is 2 letters, 3 letters or full name of the day of the week (e.g. Mo, tue, FRIDAY). The time part of **start_date** is ignored. Example: next_day('2015-01-14', 'TU') = 2015-01-20.
trunc(string **date**, string **format**)	string	Returns **date** truncated to the unit specified by the **format** (as of Hive 1.2.0). Supported formats: MONTH/MON/MM, YEAR/YYYY/YY. Example: trunc('2015-03-17', 'MM') = 2015-03-01.

Name (Signature)	Return Type	Description
months_between(**date1**, **date2**)	double	Returns number of months between dates **date1** and **date2** (as of Hive 1.2.0). If **date1** is later than **date2**, then the result is positive. If **date1** is earlier than **date2**, then the result is negative. If **date1** and **date2** are either the same days of the month or both last days of months, then the result is always an integer. Otherwise the UDF calculates the fractional portion of the result based on a 31-day month and considers the difference in time components **date1** and **date2**. **date1** and **date2** type can be date, timestamp or string in the format 'yyyy-MM-dd' or 'yyyy-MM-dd HH:mm:ss'. The result is rounded to 8 decimal places. Example: months_between('1997-02-28 10:30:00', '1996-10-30') = 3.94959677
date_format(date/timestamp/string **ts**, string **fmt**)	string	Converts a date/timestamp/string to a value of string in the format specified by the date format fmt (as of Hive 1.2.0). Supported formats are Java SimpleDateFormat formats –https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html. The second argument **fmt** should be constant. Example: date_format('2015-04-08', 'y') = '2015'. date_format can be used to implement other UDFs, e.g.: • dayname(date) is date_format(date, 'EEEE') • dayofyear(date) is date_format(date, 'D')

List of String Functions

Name (Signature)	Return Type	Description
ascii(string **str**)	int	Returns the numeric value of the first character of **str**.
base64(binary **bin**)	string	Converts the argument from binary to a base 64 string (as of Hive 0.12.0).
concat(string|binary **A**, string|binary **B**...)	string	Returns the string or bytes resulting from concatenating the strings or bytes passed in as parameters in order. For example, concat('foo', 'bar') results in 'foobar'. Note that this function can take any number of input strings.
context_ngrams(array**<array<string>>**, array<string> **context**, int **K**, int **pf**)	array<struct<string,double>>	Returns the top-**K** contextual N-grams from a set of tokenized sentences, given a string of "context". SeeStatisticsAndDataMining for more information.
concat_ws(string **SEP**, string **A**, string **B**...)	string	Like concat() above, but with custom separator **SEP**.

Name (Signature)	Return Type	Description
concat_ws(string **SEP**, array<string> **strings**)	string	Like concat_ws() above, but taking an array of strings. (as of Hive 0.9.0)
decode(binary **bin**, string **charset**)	string	Decodes the first argument into a String using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is **null**, the result will also be **null**. (As of Hive 0.12.0.)
encode(string **src**, string **charset**)	binary	Encodes the first argument into a BINARY using the provided character set (one of 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'). If either argument is **null**, the result will also be **null**. (As of Hive 0.12.0.)
find_in_set(string **str**, string **strList**)	int	Returns the first occurrance of **str** in **strList** where **strList** is a comma-delimited string. Returns **null** if either argument is **null**. Returns **0** if the first argument contains any commas. For example, find_in_set('ab', 'abc,b,ab,c,def') returns 3.
format_number(number **x**, int **d**)	string	Formats the number **x** to a format like '#,###,###.##', rounded to **d** decimal places, and returns the result as a string. If **d** is **0**, the result has no decimal point or fractional part. (As of Hive 0.10.0; bug with float types fixed in **Hive 0.14.0**, decimal type support added in**Hive 0.14.0**)
get_json_object(string **json_string**, string **path**)	string	Extracts json object from a **json_string** based on json **path** specified, and returns json string of the extracted json object. It will return **null** if the input json string is invalid. NOTE: The json **path** can only have the characters [0-9a-z_], i.e., no upper-case or special characters. Also, the keys *cannot start with numbers.* This is due to restrictions on Hive column names.
in_file(string **str**, string **filename**)	boolean	Returns **true** if the string **str** appears as an entire line in filename.
instr(string **str**, string **substr**)	int	Returns the position of the first occurrence of **substr** in **str**. Returns **null** if either of the arguments are **null** and returns **0** if **substr** could not be found in **str**. BE AWARE THAT THIS IS NOT ZERO BASED. The first character in **str** has index 1.
length(string **A**)	int	Returns the length of the string **A**

Name (Signature)	Return Type	Description
locate(string **substr**, string **str**[, int **pos**])	int	Returns the position of the first occurrence of **substr** in **str** after position **pos**.
lower(string **A**) lcase(string **A**)	string	Returns the string resulting from converting all characters of **A** to lower case. For example, lower('fOoBaR') results in 'foobar'.
lpad(string **str**, int **len**, string **pad**)	string	Returns **str**, left-padded with **pad** to a length of **len**.
ltrim(string **A**)	string	Returns the string resulting from trimming spaces from the beginning (left hand side) of **A**. For example, ltrim(' foobar ') results in 'foobar '.
vddngrams(array**<array<string>>**, int **N**, int **K**, int **pf**)	array <struct<string double>>	Returns the top-**K** N-grams from a set of tokenized sentences, such as those returned by the sentences() UDAF. See StatisticsAndDataMining for more information.
parse_url(string **urlString**, string **partToExtract** [, string **keyToExtract**])	string	Returns the specified part from the URL. Valid values for **partToExtract** include HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO. For example, parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST') returns 'facebook.com'. Also a value of a particular key in QUERY can be extracted by providing the key as the third argument, for example, parse_url('http://facebook.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1') returns 'v1'.
printf(String **format**, Obj... **args**)	string	Returns the input formatted according do printf-style **format** strings (as of Hive 0.9.0).
regexp_extract(string **subject**, string **pattern**, int **index**)	string	Returns the string extracted using the **pattern**. For example, regexp_extract('foothebar', 'foo(.*?)(bar)', 2) returns 'bar.' Note that some care is necessary in using predefined character classes: using '\s' as the second argument will match the letter s; '\\s' is necessary to match whitespace, etc. The '**index**' parameter is the Java regex Matcher group() method index. See docs/api/java/util/regex/Matcher.html for more information on the '**index**' or Java regex group() method.

Name (Signature)	Return Type	Description
regexp_replace(string **INITIAL_STRING**, string **PATTERN**, string **REPLACEMENT**)	string	Returns the string resulting from replacing all substrings in **INITIAL_STRING** that match the java regular expression syntax defined in **PATTERN** with instances of **REPLACEMENT**. For example, regexp_replace("foobar", "oo|ar", "") returns 'fb.' Note that some care is necessary in using predefined character classes: using '\s' as the second argument will match the letter s; '\\s' is necessary to match whitespace, etc.
repeat(string **str**, int **n**)	string	Repeats **str n** times.
reverse(string **A**)	string	Returns the reversed string.
rpad(string **str**, int **len**, string **pad**)	string	Returns **str**, right-padded with **pad** to a length of **len**.
rtrim(string **A**)	string	Returns the string resulting from trimming spaces from the end (right hand side) of **A**. For example, rtrim(' foobar ') results in ' foobar'.
sentences(string **str**, string **lang**, string **locale**)	array<array<string>>	Tokenizes a string of natural language text into words and sentences, where each sentence is broken at the appropriate sentence boundary and returned as an array of words. The '**lang**' and '**locale**' are optional arguments. For example, sentences('Hello there! How are you?') returns (("Hello", "there"), ("How", "are", "you")).
space(int **n**)	string	Returns a string of **n** spaces.
split(string **str**, string **pat**)	array	Splits **str** around **pat** (pat is a regular expression).
str_to_map(text[, **delimiter1**, **delimiter2**])	map<string, string>	Splits text into key-value pairs using two delimiters. **Delimiter1** separates text into K-V pairs, and **Delimiter2** splits each K-V pair. Default delimiters are ',' for **delimiter1** and '=' for **delimiter2**.

substr(string|binary **A**, int **start**)
substring(string|binary **A**, int **start**)
	string	Returns the substring or slice of the byte array of **A** starting from **start** position till the end of string **A**. For example, substr('foobar', 4) results in 'bar' (see [http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_substr]).

Name (Signature)	Return Type	Description
substr(string|binary **A**, int **start**, int **len**) substring(string|binary **A**, int **start**, int **len**)	string	Returns the substring or slice of the byte array of **A** starting from **start** position with length **len**. For example, substr('foobar', 4, 1) results in 'b' (see [**http://dev.mysql.com/doc/refman/5.0/en/string-functions.html#function_substr**]).
substring_index(string **A**, string **delim**, int **count**)	string	Returns the substring from string **A** before **count** occurrences of the delimiter **delim** (as of Hive 1.3.0). If **count** is positive, everything to the left of the final delimiter (counting from the left) is returned. If **count** is negative, everything to the right of the final delimiter (counting from the right) is returned. Substring_index performs a case-sensitive match when searching for **delim**. Example: substring_index('www.apache.org', '.', 2) = 'www.apache'.
translate(string|char|varchar **input**, string|char|varchar **from**, string|char|varchar **to**)	string	Translates the **input** string by replacing the characters present in the **from** string with the corresponding characters in the **to** string. This is similar to the translate function in **PostgreSQL**. If any of the parameters to this UDF are **NULL**, the result is **NULL** as well. (Available as of Hive 0.10.0, for string types) Char/varchar support added as of **Hive 0.14.0**.
trim(string **A**)	string	Returns the string resulting from trimming spaces from both ends of **A**. For example, trim(' foobar ') results in 'foobar'
unbase64(string **str**)	binary	Converts the argument from a base 64 string to BINARY. (As of Hive 0.12.0.)
upper(string **A**) ucase(string **A**)	string	Returns the string resulting from converting all characters of **A** to upper case. For example, upper('fOoBaR') results in 'FOOBAR'.
initcap(string **A**)	string	Returns string, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by whitespace. (As of Hive 1.1.0.)
levenshtein(string **A**, string **B**)	int	Returns the Levenshtein distance between two strings(as of Hive 1.2.0). For example, levenshtein('kitten', 'sitting') results in 3.
soundex(string **A**)	string	Returns soundex code of the string (as of Hive 1.2.0). For example, soundex('Miller') results in M460.

List of Conversion Functions

Name (Signature)	Return Type	Description
binary(string|binary **param**)	binary	Casts the **param** into a binary.
cast(**expr** as **<type>**)	Expected "=" to follow "type"	Converts the results of the expression **expr** to **<type>**. For example, cast('1' as BIGINT) will convert the string '1' to its integral representation. A **null** is returned if the conversion does not succeed. If cast(expr as boolean) Hive returns **true** for a non-empty string.

List of Collection Functions

Name (Signature)	Return Type	Description
size(**Map<K.V>**)	int	Returns the number of elements in the **Map** type.
size(**Array<T>**)	int	Returns the number of elements in the **Array** type.
map_keys(**Map<K.V>**)	array<K>	Returns an unordered array containing the keys of the input **Map**.
map_values(**Map<K.V>**)	array<V>	Returns an unordered array containing the values of the input **Map**.
array_contains(**Array<T>**, **value**)	boolean	Returns **TRUE** if the **Array** contains **value**.
sort_array(**Array<T>**)	array<t>	Sorts the input **Array** in ascending order according to the natural ordering of the array elements and returns it (as of version 0.9.0).

List of Conditional Functions

Name (Signature)	Return Type	Description
CASE **a** WHEN **b** THEN **c** [WHEN **d** THEN **e**]* [ELSE **f**] END	T	When **a** = **b**, returns **c**; when **a** = **d**, returns **e**; else returns **f**.
CASE WHEN **a** THEN **b** [WHEN **c** THEN **d**]* [ELSE **e**] END	T	When **a** = **true**, returns **b**; when **c** = **true**, returns **d**; else returns **e**.
COALESCE(T **v1**, T **v2**, ...)	T	Returns the first **v** that is not **NULL**, or **NULL** if all **v**'s are **NULL**.
greatest(T **v1**, T **v2**, ...)	T	Returns the greatest **v** of the list of **v**'s (as of Hive 1.1.0).
if(boolean **testCondition**, T **valueTrue**, T **valueFalseOrNull**)	T	Returns **valueTrue** when **testCondition** is **true**, returns **valueFalseOrNull** otherwise.
isnotnull(**a**)	boolean	Returns **true** if **a** is not **NULL** and **false** otherwise.

Name (Signature)	Return Type	Description
isnull(**a**)	boolean	Returns **true** if **a** is **NULL** and **false** otherwise.
least(T **v1**, T **v2**, ...)	T	Returns the least **v** of the list of **v**'s (as of Hive 1.1.0).
nvl(T **value**, T **default_value**)	T	Returns **default_value** if **value** is **null** else returns **value** (as of HIve 0.11).

Basic Analysis with GROUP BY and Aggregation Functions

Usually before any advanced analytics or statistics are performed, you will want to perform some basic descriptive statistics. This exploratory phase can give you general insights about your data and can be done completely within Hive. It will also allow you to consolidate your data to prepare it for other business intelligence and visualization tools.

The GROUP BY clause allows you group data based on a particular column. Then you can use aggregation functions over each group to gain insight about that group.

Syntax:

Returns the average of `colB` within each group of `colA`.

```
SELECT colA, avg(colB)
FROM tableA
GROUP BY colA;
```

Example:

From our car table, we want to find

```
SELECT make, count(distinct model)
FROM cars
GROUP BY make;
```

Result:

Make	Count (distinct model)
toyota	2
ford	1
audi	1
kia	1

Aggregation Functions available in Hive

Name (Signature)	Description	Return Type
count(*), count(expr), count(DISTINCT expr[, expr...])	count(*) - Returns the total number of rows, including NULL values. count(expr) - Returns the number of rows for which the supplied expression is non-NULL. count(DISTINCT expr[, expr]) - Returns the number of rows for which the supplied expression(s) are unique and non-NULL.	BIGINT
sum(col), sum(DISTINCT col)	Returns the sum of the elements in the group or the sum of the distinct values of the column in the group.	DOUBLE
avg(col), avg(DISTINCT col)	Returns the average of the elements in the group or the average of the distinct values of the column in the group.	DOUBLE
min(col)	Returns the minimum of the column in the group.	DOUBLE
max(col)	Returns the maximum value of the column in the group.	DOUBLE
variance(col), var_pop(col)	Returns the variance of a numeric column in the group.	DOUBLE
var_samp(col)	Returns the unbiased sample variance of a numeric column in the group.	DOUBLE
stddev_pop(col)	Returns the standard deviation of a numeric column in the group.	DOUBLE
stddev_samp(col)	Returns the unbiased sample standard deviation of a numeric column in the group.	DOUBLE
covar_pop(col1, col2)	Returns the population covariance of a pair of numeric columns in the group.	DOUBLE
covar_samp(col1, col2)	Returns the sample covariance of a pair of a numeric columns in the group.	DOUBLE
corr(col1, col2	Returns the Pearson coefficient of correlation of a pair of a numeric columns in the group.	DOUBLE
percentile(BIGINT col, **p**)	Returns the exact **p**th percentile of a column in the group (does not work with floating point types). **p** must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_ APPROX if your input is non-integral.	DOUBLE

Name (Signature)	Description	Return Type
percentile(BIGINT col, array($\mathbf{p}_1$ [, $\mathbf{p}_2$]...))	Returns the exact percentiles $\mathbf{p}_1$, $\mathbf{p}_2$, ... of a column in the group (does not work with floating point types). $\mathbf{p}_i$ must be between 0 and 1. NOTE: A true percentile can only be computed for integer values. Use PERCENTILE_APPROX if your input is non-integral.	array<double>
percentile_approx(DOUBLE col, **p** [, **B**])	Returns an approximate $\mathbf{p}^{th}$ percentile of a numeric column (including floating point types) in the group. The **B** parameter controls approximation accuracy at the cost of memory. Higher values yield better approximations, and the default is 10,000. When the number of distinct values in col is smaller than **B**, this gives an exact percentile value.	DOUBLE
percentile_approx(DOUBLE col, array(p_1 [, p_2]...) [, B])	Same as above, but accepts and returns an array of percentile values instead of a single one.	array<double>
histogram_numeric(col, **b**)	Computes a histogram of a numeric column in the group using **b** non-uniformly spaced bins. The output is an array of size **b** of double-valued (x,y) coordinates that represent the bin centers and heights.	array<struct {'x','y'}>
collect_set(col)	Returns a set of objects with duplicate elements eliminated.	array
collect_list(col)	Returns a list of objects with duplicates. (As of Hive 0.13.0.)	array
ntile(INTEGER **x**)	Divides an ordered partition into **x** groups called buckets and assigns a bucket number to each row in the partition. This allows easy calculation of tertiles, quartiles, deciles, percentiles and other common summary statistics. (As of Hive 0.11.0.)	INTEGER

Filtering Aggregate Results with the HAVING Clause

Like the WHERE clause that allows you to filter the data, you can filter your aggregate results to only show results that fit a particular filter.

Syntax:

Returns the result set where the average of `colB` is strictly less than the value `x` within each group of `colA`.

```
SELECT colA, avg(colB)
FROM tableA
GROUP BY cola
HAVING avg(colB) < x;
```

Example:

Using the same query from the aggregate example, suppose we only want to see the car makers that have more than 1 model in our table.

```
SELECT make, count(distinct model)
FROM cars
GROUP BY make
HAVING count(distinct model) > 1;
```

Result:

Make	Count (distinct model)
toyota	2

LIMITing Results

Sometimes it is useful to restrict the number of rows returned from a result set. You can do this using the LIMIT clause. Placed at the end of a query statement, this clause will limit the number of results returned back to the number specified. Please note, that this clause does not in any way affect performance of your query.

Syntax:

Returns n number of results

```
SELECT colA, colB
FROM tableA
LIMIT n;
```

Example:

From our car table, we want to find

```
SELECT make, model
FROM cars
LIMIT 2;
```

Result:

Make	Model
toyota	corolla
ford	focus

Sorting Results Using ORDER BY

To sort the result set, you can add the ORDER BY clause. Multiple columns can be used for sorting and sorts can be ascending or descending.

ORDER BY in HiveQL can have some limitations. If hive.mapred.mode=strict, the ORDER BY clause must be followed by a LIMIT clause. If you do not have the strict mode turned on, you do not need a LIMIT clause but the results may take a long time to finish.

Syntax:

The below query will return columns colA and colB with colA sorted in ascending order and any common groupings in colB sub-sorted in descending order.

```
SELECT colA, colB
FROM tableA
ORDER BY colA ASC, colB DESC;
```

Example:

From our car table, we want to find

```
SELECT make, model, year
FROM cars
ORDER BY year DESC, model ASC;
```

Result:

Make	Model	Year
audi	a4	2015
toyota	corolla	2013
ford	focus	2013
toyota	camry	2007
kia	sportage	2002

What about SORT BY?

It can be easy to confuse ORDER BY and SORT BY. In other SQL implementations, SORT BY does not exist. SORT BY sorts data per reducer, which means that ORDER BY will guarantee total order in the result set while SORT BY will only give partially sorted results.

However, SORT BY is useful if you know your results will be split appropriately between reducers. For example, if you run a group by and you want the results within the group by to be sorted in way, then you can use SORT BY to sort the results within groups. However, the groups themselves will not be sorted.

Combining Similar Tables with UNION ALL

The UNION ALL command allows you to concatenate two tables with the same schema together into one table. For example, if you have Table 1 with 100 rows and columns <a, b, c> and Table 2 with 50 rows and columns <a, b, c> then using UNION ALL you can create a new data set with 150 rows with columns <a, b, c>.

Version information:

Prior to Hive 1.2.0, only UNION ALL is supported.

As of Hive 0.13.0, unions can be used as part of a top level query.

In Hive 0.12.0 and prior, unions can only be used within a subquery in the FROM statement.

Syntax:

Using UNION ALL as a subquery for Hive versions 0.12.0 and below:

```
SELECT *
FROM (
     SELECT <columns> from <table> ...

     UNION ALL

  SELECT <columns> from <table> ...
) unionResult;
```

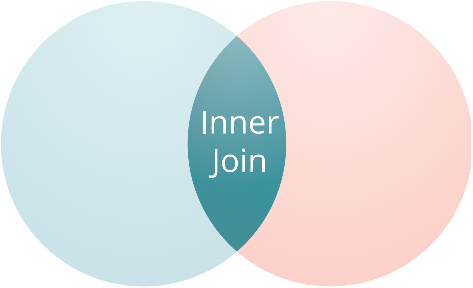
Using UNION ALL within a top level query, which can be used from Hive 0.13.0 and onward:

```
SELECT <columns> from <table> ...

UNION ALL

SELECT <columns> from <table> …

UNION ALL ...
```

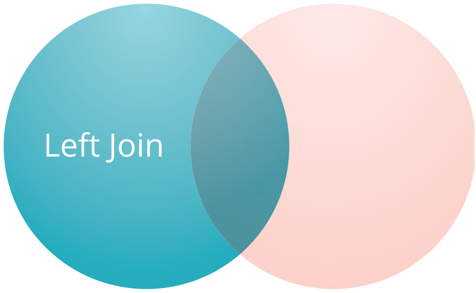
Merging Tables with the JOIN Statement

Joins are used when you have a need to combine two separate tables. The most common use cases include joining two tables so the result set contains columns from both tables, however there are times where you may want to join tables for other outcomes.

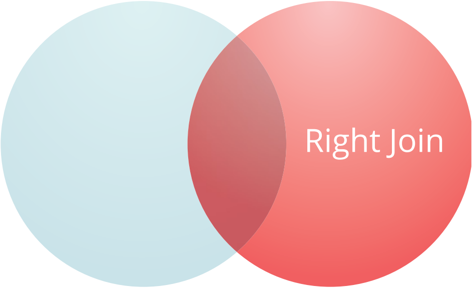
Types of Joins



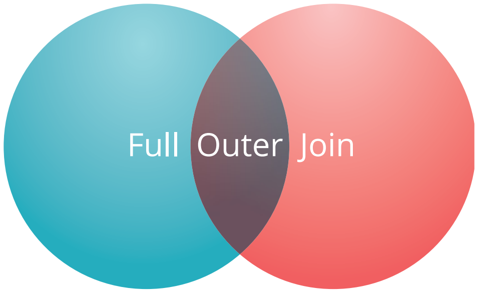
```
SELECT a.coll, ... a.coln, b.coll, ... b.coln
FROM a
JOIN b ON (a.id = b.id);
```



```
SELECT a.coll, ... a.coln, b.coll, ... b.coln
FROM a
LEFT OUTER JOIN b ON (a.id = b.id);
```



```
SELECT a.coll, ... a.coln, b.coll, ... b.coln
FROM a
RIGHT OUTER JOIN b ON (a.id = b.id);
```



```
SELECT a.coll, ... a.coln, b.coll, ... b.coln
FROM a
FULL OUTER JOIN b ON (a.id = b.id);
```

The above figures show the four basic types of joins available in Hive. An inner join returns the direct overlap between the two tables. Single sided joins, like a left join and a right join, will return the result set with all the rows of the designated side with NULL values used to fill in any rows that do not have a corresponding row in the other table.

Example:

We want to associate owners names with each of our cars. To do this, we select the columns we wish to view and perform an inner join relating cars.id to the car_id field in the owners table.

```
SELECT owners.name, cars.make, cars.model, cars.year
FROM cars
JOIN owners ON (cars.id = owners.car_id);
```

Result:

owners.names	cars.make	cars.model	cars.year
Bob	toyota	corolla	2013
Ashley	ford	focus	2013
Sam	audi	a4	2015
Chris	toyota	camry	2007
Jane	kia	sportage	2002

Example:

Let's say we wanted to produce a full join between both tables and look at all the data together. To do this, we simply do an outer join, again relating the two common id fields in both tables.

```
SELECT *
FROM cars
FULL OUTER JOIN owners ON (cars.id = owners.car_id);
```

Result:

cars.id	cars.make	cars.model	cars.year	owners.id	owners.names	owners.age	owners.car_id
1	toyota	corolla	2013	2	Bob	35	1
2	ford	focus	2013	5	Ashley	28	2
3	audi	a4	2015	1	Sam	52	3
4	toyota	camry	2007	4	Chris	16	4
5	kia	sportage	2002	3	Jane	18	5

Case Study Example - Point of Sale Analysis

Let's pretend we're running an office supply store and want to do some analysis on our business. In this real world example, we have data from our in-store transaction logs (Point of Sale system) and our customer rewards program. In this case, the only way we have to link customer information with our transaction logs is the loyalty_id associated with the rewards cards we issue. For the sake of this example, we'll assume all of our customers have enrolled in the loyalty program and use their rewards card with every transaction. Our raw data is as follows:

Orders

ID	date	loyalty_ID
86093	9/23/15	544-2391303
94118	9/26/15	210-2931024
81555	9/26/15	544-2391303
22963	10/1/15	323-4293021
27019	10/3/15	210-2931024
27399	10/7/15	213-1235543
36302	10/7/15	213-3342030
63379	10/12/15	213-3342030
69077	10/13/15	323-4293021
10236	10/15/15	210-2931024

Transactions

order_ID	product_ID		order_ID	product_ID
86093	234212		27399	239423
94118	201433		36302	234212
81555	239423		63379	201433
81555	518293		63379	234212
81555	232532		63379	518293
22963	564723		69077	239423
22963	518293		10236	518293
27019	518293		10236	232532

Products

ID	name	wholesale_price	sale_price
239423	PENS	2.00	2.99
518293	COPIER PAPER	4.00	5.99
234212	JETINK	20.00	29.99
236123	MECHANICAL PENCILS	1.50	3.99
564723	INDEX CARDS	2.00	2.99
201433	AAA BATTERIES	10.00	12.99
232532	PAPER CLIPS	1.00	5.99

Customers

ID	first_name	last_name
213-1235543	JANE	WHITE
213-3342030	JOHN	COOK
323-4293021	EMILY	MCDONALD
544-2391303	KAREN	SMITH
210-2931024	ROBERT	BROWN

Question 1 - Imagine you have a customer complaint. All you have to look up relevant information about that customer is their loyalty card number. Find all customer information for the customer with the card number "213-3342030".

Query

```
SELECT
  *
FROM
  customers
WHERE
  customers.ID = “213-3342030”;
```

Result

id	first_name	last_name
213-3342030	JOHN	COOK

Question 2 - What was our total revenue, cost and profit margin across this dataset.

Query

```
SELECT

  COUNT(DISTINCT orders.ID) AS num_orders,
  SUM(products.sale_price) AS total_revenue,
  SUM(products.wholesale_price) AS total_cost,
  (
    SUM(products.sale_price)- SUM(products.wholesale_price)
  )/ SUM(products.sale_price) AS percentage_profit
FROM
  orders
JOIN
  transactions
  ON (
    orders.ID = transactions.order
  )
JOIN
  products
  ON (
    transactions.product = products.ID
  );
```

Result

num_orders	total_revenue	total_cost	percentage_profit
10	169.84	110	0.352331606217616

Question 3 - Which customers made more than one order and, of those customers, what was both the average spend and maximum one time spend per account?

Query

```
SELECT
  CONCAT(customers.first_name,
    “ “,
    customers.last_name) AS name,
  COUNT(DISTINCT orders.ID) AS num_orders,
  AVG(products.sale_price) AS avg_sale_price,
  MAX(products.sale_price) AS max_sale_price
FROM
  customers
JOIN
  orders
  ON (
    customers.ID = orders.loyalty_ID
  )
JOIN
  transactions
  ON (
    orders.ID = transactions.order
  )
JOIN
  products
  ON (
    transactions.product = products.ID
  )
GROUP BY
  customers.first_name,
  customers.last_name
HAVING
  COUNT(orders.id)> 1
ORDER BY
  num_orders DESC
;
```

Result

name	num_orders	avg_sale_price	max_sale_price
ROBERT BROWN	3	7.74	12.99
KAREN SMITH	2	11.24	29.99
JOHN COOK	2	19.74	29.99
EMILY MCDONALD	2	3.99	5.99