

Problem 2408: Design SQL

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

You are given two string arrays,

names

and

columns

, both of size

n

. The

i

th

table is represented by the name

names[i]

and contains

columns[i]

number of columns.

You need to implement a class that supports the following

operations

:

Insert

a row in a specific table with an id assigned using an

auto-increment

method, where the id of the first inserted row is 1, and the id of each

new

row inserted into the same table is

one greater

than the id of the

last inserted

row, even if the last row was

removed

.

Remove

a row from a specific table. Removing a row

does not

affect the id of the next inserted row.

Select

a specific cell from any table and return its value.

Export

all rows from any table in csv format.

Implement the

SQL

class:

```
SQL(String[] names, int[] columns)
```

Creates the

n

tables.

```
bool ins(String name, String[] row)
```

Inserts

row

into the table

name

and returns

true

If

row.length

does not

match the expected number of columns, or

name

is

not

a valid table, returns

false

without any insertion.

void rmv(String name, int rowId)

Removes the row

rowId

from the table

name

.

If

name

is

not

a valid table or there is no row with id

rowId

, no removal is performed.

`String sel(String name, int rowId, int columnId)`

Returns the value of the cell at the specified

rowId

and

columnId

in the table

name

.

If

name

is

not

a valid table, or the cell

`(rowId, columnId)`

is

invalid

, returns

"<null>"

.

String[] exp(String name)

Returns the rows present in the table

name

.

If name is

not

a valid table, returns an empty array. Each row is represented as a string, with each cell value
(

including

the row's id) separated by a

", "

.

Example 1:

Input:

```
["SQL","ins","sel","ins","exp","rmv","sel","exp"] [[[{"one","two","three"},[2,3,1]],[{"two","first","second","third"}], [{"two",1,3}, {"two","fourth","fifth","sixth"}], [{"two"}, {"two",1}, {"two",2,2}, {"two"}]]
```

Output:

```
[null,true,"third",true,[{"1","first","second","third"}, {"2","fourth","fifth","sixth"}], null, "fifth", [{"2","fourth","fifth","sixth"}]]
```

Explanation:

```
// Creates three tables. SQL sql = new SQL(["one", "two", "three"], [2, 3, 1]);  
  
// Adds a row to the table "two" with id 1. Returns True. sql.ins("two", ["first", "second",  
"third"]);  
  
// Returns the value "third" from the third column // in the row with id 1 of the table "two".  
sql.sel("two", 1, 3);  
  
// Adds another row to the table "two" with id 2. Returns True. sql.ins("two", ["fourth", "fifth",  
"sixth"]);  
  
// Exports the rows of the table "two". // Currently, the table has 2 rows with ids 1 and 2.  
sql.exp("two");  
  
// Removes the first row of the table "two". Note that the second row // will still have the id 2.  
sql.rmv("two", 1);  
  
// Returns the value "fifth" from the second column // in the row with id 2 of the table "two".  
sql.sel("two", 2, 2);  
  
// Exports the rows of the table "two". // Currently, the table has 1 row with id 2. sql.exp("two");
```

Example 2:

Input:

```
["SQL", "ins", "sel", "rmv", "sel", "ins", "ins"] [[[{"one": "two", "three": "three"}, [2, 3, 1]], [{"two": [{"first": "second", "third": "third"}]}, {"two": [{"first": "second", "third": "third"}]}], [{"two": [{"first": "second", "third": "third"}]}], [{"two": [{"first": "second", "third": "third"}]}], [{"two": [{"first": "second", "third": "third"}]}]]]
```

Output:

```
[null, true, "third", null, "<null>", false, true]
```

Explanation:

```
// Creates three tables. SQL sQL = new SQL(["one", "two", "three"], [2, 3, 1]);
```

```
// Adds a row to the table "two" with id 1. Returns True. sQL.ins("two", ["first", "second",  
"third"]);  
  
// Returns the value "third" from the third column // in the row with id 1 of the table "two".  
sQL.sel("two", 1, 3);  
  
// Removes the first row of the table "two". sQL.rmv("two", 1);  
  
// Returns <null> as the cell with id 1 // has been removed from table "two". sQL.sel("two", 1,  
2);  
  
// Returns False as number of columns are not correct. sQL.ins("two", ["fourth", "fifth"]);  
  
// Adds a row to the table "two" with id 2. Returns True. sQL.ins("two", ["fourth", "fifth",  
"sixth"]);
```

Constraints:

$n == \text{names.length} == \text{columns.length}$

$1 \leq n \leq 10$

4

$1 \leq \text{names}[i].length, \text{row}[i].length, \text{name}.length \leq 10$

$\text{names}[i]$

,

$\text{row}[i]$

, and

name

consist only of lowercase English letters.

$1 \leq \text{columns}[i] \leq 10$

$1 \leq \text{row.length} \leq 10$

All

`names[i]`

are

distinct

At most

2000

calls will be made to

`ins`

and

`rmv`

At most

10

4

calls will be made to

`sel`

At most

500

calls will be made to

exp

.

Follow-up:

Which approach would you choose if the table might become sparse due to many deletions, and why? Consider the impact on memory usage and performance.

Code Snippets

C++:

```
class SQL {
public:
    SQL(vector<string>& names, vector<int>& columns) {

}

bool ins(string name, vector<string> row) {

}

void rmv(string name, int rowId) {

}

string sel(string name, int rowId, int columnId) {

}

vector<string> exp(string name) {

}
```

```
};

/**
 * Your SQL object will be instantiated and called as such:
 * SQL* obj = new SQL(names, columns);
 * bool param_1 = obj->ins(name, row);
 * obj->rmv(name, rowId);
 * string param_3 = obj->sel(name, rowId, columnId);
 * vector<string> param_4 = obj->exp(name);
 */
```

Java:

```
class SQL {

    public SQL(List<String> names, List<Integer> columns) {

    }

    public boolean ins(String name, List<String> row) {

    }

    public void rmv(String name, int rowId) {

    }

    public String sel(String name, int rowId, int columnId) {

    }

    public List<String> exp(String name) {

    }

}

/**
 * Your SQL object will be instantiated and called as such:
 * SQL obj = new SQL(names, columns);
 * boolean param_1 = obj.ins(name, row);
 * obj.rmv(name, rowId);
 * String param_3 = obj.sel(name, rowId, columnId);
 */
```

```
* List<String> param_4 = obj.exp(name);  
*/
```

Python3:

```
class SQL:  
  
    def __init__(self, names: List[str], columns: List[int]):  
  
        def ins(self, name: str, row: List[str]) -> bool:  
  
            def rmv(self, name: str, rowId: int) -> None:  
  
                def sel(self, name: str, rowId: int, columnId: int) -> str:  
  
                    def exp(self, name: str) -> List[str]:  
  
                        # Your SQL object will be instantiated and called as such:  
                        # obj = SQL(names, columns)  
                        # param_1 = obj.ins(name, row)  
                        # obj.rmv(name, rowId)  
                        # param_3 = obj.sel(name, rowId, columnId)  
                        # param_4 = obj.exp(name)
```

Python:

```
class SQL(object):  
  
    def __init__(self, names, columns):  
        """  
        :type names: List[str]  
        :type columns: List[int]  
        """  
  
        def ins(self, name, row):
```

```

"""
:type name: str
:type row: List[str]
:rtype: bool
"""

def rmv(self, name, rowId):
"""
:type name: str
:type rowId: int
:rtype: None
"""

def sel(self, name, rowId, columnId):
"""
:type name: str
:type rowId: int
:type columnId: int
:rtype: str
"""

def exp(self, name):
"""
:type name: str
:rtype: List[str]
"""

# Your SQL object will be instantiated and called as such:
# obj = SQL(names, columns)
# param_1 = obj.ins(name, row)
# obj.rmv(name, rowId)
# param_3 = obj.sel(name, rowId, columnId)
# param_4 = obj.exp(name)

```

JavaScript:

```
/***
 * @param {string[]} names
 * @param {number[]} columns
 */
var SQL = function(names, columns) {

};

/***
 * @param {string} name
 * @param {string[]} row
 * @return {boolean}
 */
SQL.prototype.ins = function(name, row) {

};

/***
 * @param {string} name
 * @param {number} rowId
 * @return {void}
 */
SQL.prototype.rmv = function(name, rowId) {

};

/***
 * @param {string} name
 * @param {number} rowId
 * @param {number} columnId
 * @return {string}
 */
SQL.prototype.sel = function(name, rowId, columnId) {

};

/***
 * @param {string} name
 * @return {string[]}
 */
SQL.prototype.exp = function(name) {

};
```

```

/**
 * Your SQL object will be instantiated and called as such:
 * var obj = new SQL(names, columns)
 * var param_1 = obj.ins(name,row)
 * obj.rmv(name,rowId)
 * var param_3 = obj.sel(name,rowId,columnId)
 * var param_4 = obj.exp(name)
 */

```

TypeScript:

```

class SQL {
constructor(names: string[], columns: number[]) {

}

ins(name: string, row: string[]): boolean {

}

rmv(name: string, rowId: number): void {

}

sel(name: string, rowId: number, columnId: number): string {

}

exp(name: string): string[] {

}

}

/**
 * Your SQL object will be instantiated and called as such:
 * var obj = new SQL(names, columns)
 * var param_1 = obj.ins(name,row)
 * obj.rmv(name,rowId)
 * var param_3 = obj.sel(name,rowId,columnId)
 * var param_4 = obj.exp(name)
*/

```

C#:

```
public class SQL {  
  
    public SQL(IList<string> names, IList<int> columns) {  
  
    }  
  
    public bool Ins(string name, IList<string> row) {  
  
    }  
  
    public void Rmv(string name, int rowId) {  
  
    }  
  
    public string Sel(string name, int rowId, int columnId) {  
  
    }  
  
    public IList<string> Exp(string name) {  
  
    }  
}  
  
/**  
 * Your SQL object will be instantiated and called as such:  
 * SQL obj = new SQL(names, columns);  
 * bool param_1 = obj.Ins(name, row);  
 * obj.Rmv(name, rowId);  
 * string param_3 = obj.Sel(name, rowId, columnId);  
 * IList<string> param_4 = obj.Exp(name);  
 */
```

C:

```
typedef struct {  
  
} SQL;
```

```

SQL* SQLCreate(char** names, int namesSize, int* columns, int columnsSize) {

}

bool SQLIns(SQL* obj, char* name, char** row, int rowSize) {

}

void SQLRmv(SQL* obj, char* name, int rowId) {

}

char* SQLSel(SQL* obj, char* name, int rowId, int columnId) {

}

char** SQLExp(SQL* obj, char* name, int* retSize) {

}

void SQLFree(SQL* obj) {

}

/**
 * Your SQL struct will be instantiated and called as such:
 * SQL* obj = SQLCreate(names, namesSize, columns, columnsSize);
 * bool param_1 = SQLIns(obj, name, row, rowSize);
 *
 * SQLRmv(obj, name, rowId);
 *
 * char* param_3 = SQLSel(obj, name, rowId, columnId);
 *
 * char** param_4 = SQLExp(obj, name, retSize);
 *
 * SQLFree(obj);
 */

```

Go:

```

type SQL struct {

}

func Constructor(names []string, columns []int) SQL {
}

func (this *SQL) Ins(name string, row []string) bool {
}

func (this *SQL) Rmv(name string, rowId int) {
}

func (this *SQL) Sel(name string, rowId int, columnId int) string {
}

func (this *SQL) Exp(name string) []string {
}

/**
 * Your SQL object will be instantiated and called as such:
 * obj := Constructor(names, columns);
 * param_1 := obj.Ins(name, row);
 * obj.Rmv(name, rowId);
 * param_3 := obj.Sel(name, rowId, columnId);
 * param_4 := obj.Exp(name);
 */

```

Kotlin:

```

class SQL(names: List<String>, columns: List<Int>) {

```

```

fun ins(name: String, row: List<String>): Boolean {
}

fun rmv(name: String, rowId: Int) {

}

fun sel(name: String, rowId: Int, columnId: Int): String {
}

fun exp(name: String): List<String> {

}

}

/**
 * Your SQL object will be instantiated and called as such:
 * var obj = SQL(names, columns)
 * var param_1 = obj.ins(name, row)
 * obj.rmv(name, rowId)
 * var param_3 = obj.sel(name, rowId, columnId)
 * var param_4 = obj.exp(name)
 */

```

Swift:

```

class SQL {

init(_ names: [String], _ columns: [Int]) {

}

func ins(_ name: String, _ row: [String]) -> Bool {

}

func rmv(_ name: String, _ rowId: Int) {

```

```

}

func sel(_ name: String, _ rowId: Int, _ columnId: Int) -> String {

}

func exp(_ name: String) -> [String] {

}

/**
* Your SQL object will be instantiated and called as such:
* let obj = SQL(names, columns)
* let ret_1: Bool = obj.ins(name, row)
* obj.rmv(name, rowId)
* let ret_3: String = obj.sel(name, rowId, columnId)
* let ret_4: [String] = obj.exp(name)
*/

```

Rust:

```

struct SQL {

}

/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl SQL {

fn new(names: Vec<String>, columns: Vec<i32>) -> Self {

}

fn ins(&self, name: String, row: Vec<String>) -> bool {

}

fn rmv(&self, name: String, row_id: i32) {

```

```

}

fn sel(&self, name: String, row_id: i32, column_id: i32) -> String {

}

fn exp(&self, name: String) -> Vec<String> {

}

/**
 * Your SQL object will be instantiated and called as such:
 * let obj = SQL::new(names, columns);
 * let ret_1: bool = obj.ins(name, row);
 * obj.rmv(name, rowId);
 * let ret_3: String = obj.sel(name, rowId, columnId);
 * let ret_4: Vec<String> = obj.exp(name);
 */

```

Ruby:

```

class SQL

=begin
:type names: String[]
:type columns: Integer[]
=end
def initialize(names, columns)

end

=begin
:type name: String
:type row: String[]
:rtype: Boolean
=end
def ins(name, row)

end

```

```
=begin
:type name: String
:type row_id: Integer
:rtype: Void
=end
def rmv(name, row_id)

end

=begin
:type name: String
:type row_id: Integer
:type column_id: Integer
:rtype: String
=end
def sel(name, row_id, column_id)

end

=begin
:type name: String
:rtype: String[]
=end
def exp(name)

end

end

# Your SQL object will be instantiated and called as such:
# obj = SQL.new(names, columns)
# param_1 = obj.ins(name, row)
# obj.rmv(name, row_id)
# param_3 = obj.sel(name, row_id, column_id)
# param_4 = obj.exp(name)
```

PHP:

```
class SQL {  
    /**  
     * @param String[] $names  
     * @param Integer[] $columns  
     */  
    function __construct($names, $columns) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param String[] $row  
     * @return Boolean  
     */  
    function ins($name, $row) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param Integer $rowId  
     * @return NULL  
     */  
    function rmv($name, $rowId) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param Integer $rowId  
     * @param Integer $columnId  
     * @return String  
     */  
    function sel($name, $rowId, $columnId) {  
  
    }  
  
    /**  
     * @param String $name  
     * @return String[]  
     */
```

```

function exp($name) {

}

}

/***
* Your SQL object will be instantiated and called as such:
* $obj = SQL($names, $columns);
* $ret_1 = $obj->ins($name, $row);
* $obj->rmv($name, $rowId);
* $ret_3 = $obj->sel($name, $rowId, $columnId);
* $ret_4 = $obj->exp($name);
*/

```

Dart:

```

class SQL {

SQL(List<String> names, List<int> columns) {

}

bool ins(String name, List<String> row) {

}

void rmv(String name, int rowId) {

}

String sel(String name, int rowId, int columnId) {

}

List<String> exp(String name) {

}

}

/***
* Your SQL object will be instantiated and called as such:
* SQL obj = SQL(names, columns);
*/

```

```

* bool param1 = obj.ins(name, row);
* obj.rmv(name, rowId);
* String param3 = obj.sel(name, rowId, columnId);
* List<String> param4 = obj.exp(name);
*/

```

Scala:

```

class SQL(_names: List[String], _columns: List[Int]) {

    def ins(name: String, row: List[String]): Boolean = {

    }

    def rmv(name: String, rowId: Int): Unit = {

    }

    def sel(name: String, rowId: Int, columnId: Int): String = {

    }

    def exp(name: String): List[String] = {

    }

}

/***
 * Your SQL object will be instantiated and called as such:
 * val obj = new SQL(names, columns)
 * val param_1 = obj.ins(name, row)
 * obj.rmv(name, rowId)
 * val param_3 = obj.sel(name, rowId, columnId)
 * val param_4 = obj.exp(name)
*/

```

Elixir:

```

defmodule SQL do
@spec init_(names :: [String.t], columns :: [integer]) :: any
def init_(names, columns) do

```

```

end

@spec ins(name :: String.t, row :: [String.t]) :: boolean
def ins(name, row) do

end

@spec rmv(name :: String.t, row_id :: integer) :: any
def rmv(name, row_id) do

end

@spec sel(name :: String.t, row_id :: integer, column_id :: integer) :: String.t
def sel(name, row_id, column_id) do

end

@spec exp(name :: String.t) :: [String.t]
def exp(name) do

end

# Your functions will be called as such:
# SQL.init_(names, columns)
# param_1 = SQL.ins(name, row)
# SQL.rmv(name, row_id)
# param_3 = SQL.sel(name, row_id, column_id)
# param_4 = SQL.exp(name)

# SQL.init_ will be called before every test case, in which you can do some
necessary initializations.

```

Erlang:

```

-spec sql_init_(Names :: [unicode:unicode_binary()], Columns :: [integer()]) -> any().
sql_init_(Names, Columns) ->
.
```

```

-spec sql_ins(Name :: unicode:unicode_binary(), Row :: [unicode:unicode_binary()]) -> boolean().
sql_ins(Name, Row) ->
.

-spec sql_rmv(Name :: unicode:unicode_binary(), RowId :: integer()) -> any().
sql_rmv(Name, RowId) ->
.

-spec sql_sel(Name :: unicode:unicode_binary(), RowId :: integer(), ColumnId :: integer()) -> unicode:unicode_binary().
sql_sel(Name, RowId, ColumnId) ->
.

-spec sql_exp(Name :: unicode:unicode_binary()) -> [unicode:unicode_binary()].
sql_exp(Name) ->
.

%% Your functions will be called as such:
%% sql_init_(Names, Columns),
%% Param_1 = sql_ins(Name, Row),
%% sql_rmv(Name, RowId),
%% Param_3 = sql_sel(Name, RowId, ColumnId),
%% Param_4 = sql_exp(Name),

%% sql_init_ will be called before every test case, in which you can do some
necessary initializations.

```

Racket:

```

(define sql%
  (class object%
    (super-new)

    ; names : (listof string?)
    ; columns : (listof exact-integer?)
    (init-field
      names
      columns)

```

```

; ins : string? (listof string?) -> boolean?
(define/public (ins name row)
)

; rmv : string? exact-integer? -> void?
(define/public (rmv name row-id)
)

; sel : string? exact-integer? exact-integer? -> string?
(define/public (sel name row-id column-id)
)

; exp : string? -> (listof string?)
(define/public (exp name)
))

;; Your sql% object will be instantiated and called as such:
;; (define obj (new sql% [names names] [columns columns]))
;; (define param_1 (send obj ins name row))
;; (send obj rmv name row-id)
;; (define param_3 (send obj sel name row-id column-id))
;; (define param_4 (send obj exp name))

```

Solutions

C++ Solution:

```

/*
 * Problem: Design SQL
 * Difficulty: Medium
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class SQL {
public:
SQL(vector<string>& names, vector<int>& columns) {

}

```

```

bool ins(string name, vector<string> row) {

}

void rmv(string name, int rowId) {

}

string sel(string name, int rowId, int columnId) {

}

vector<string> exp(string name) {

}

};

/***
 * Your SQL object will be instantiated and called as such:
 * SQL* obj = new SQL(names, columns);
 * bool param_1 = obj->ins(name, row);
 * obj->rmv(name, rowId);
 * string param_3 = obj->sel(name, rowId, columnId);
 * vector<string> param_4 = obj->exp(name);
 */

```

Java Solution:

```

/**
 * Problem: Design SQL
 * Difficulty: Medium
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class SQL {

public SQL(List<String> names, List<Integer> columns) {

```

```

}

public boolean ins(String name, List<String> row) {

}

public void rmv(String name, int rowId) {

}

public String sel(String name, int rowId, int columnId) {

}

public List<String> exp(String name) {

}

}

/***
 * Your SQL object will be instantiated and called as such:
 * SQL obj = new SQL(names, columns);
 * boolean param_1 = obj.ins(name, row);
 * obj.rmv(name, rowId);
 * String param_3 = obj.sel(name, rowId, columnId);
 * List<String> param_4 = obj.exp(name);
 */

```

Python3 Solution:

```

"""
Problem: Design SQL
Difficulty: Medium
Tags: array, string, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

```

```

class SQL:

    def __init__(self, names: List[str], columns: List[int]):

        def ins(self, name: str, row: List[str]) -> bool:
            # TODO: Implement optimized solution
            pass

```

Python Solution:

```

class SQL(object):

    def __init__(self, names, columns):
        """
        :type names: List[str]
        :type columns: List[int]
        """

    def ins(self, name, row):
        """
        :type name: str
        :type row: List[str]
        :rtype: bool
        """

    def rmv(self, name, rowId):
        """
        :type name: str
        :type rowId: int
        :rtype: None
        """

    def sel(self, name, rowId, columnId):
        """
        :type name: str
        :type rowId: int
        :type columnId: int
        """

```

```

:rtype: str
"""

def exp(self, name):
    """
:type name: str
:rtype: List[str]
"""

# Your SQL object will be instantiated and called as such:
# obj = SQL(names, columns)
# param_1 = obj.ins(name, row)
# obj.rmv(name, rowId)
# param_3 = obj.sel(name, rowId, columnId)
# param_4 = obj.exp(name)

```

JavaScript Solution:

```

/**
 * Problem: Design SQL
 * Difficulty: Medium
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

var SQL = function(names, columns) {

};

/**
 * @param {string[]} names
 * @param {number[]} columns
 */

```

```

* @param {string[]} row
* @return {boolean}
*/
SQL.prototype.ins = function(name, row) {

};

/***
* @param {string} name
* @param {number} rowId
* @return {void}
*/
SQL.prototype.rmv = function(name, rowId) {

};

/***
* @param {string} name
* @param {number} rowId
* @param {number} columnId
* @return {string}
*/
SQL.prototype.sel = function(name, rowId, columnId) {

};

/***
* @param {string} name
* @return {string[]}
*/
SQL.prototype.exp = function(name) {

};

/***
* Your SQL object will be instantiated and called as such:
* var obj = new SQL(names, columns)
* var param_1 = obj.ins(name, row)
* obj.rmv(name, rowId)
* var param_3 = obj.sel(name, rowId, columnId)
* var param_4 = obj.exp(name)
*/

```

TypeScript Solution:

```
/**  
 * Problem: Design SQL  
 * Difficulty: Medium  
 * Tags: array, string, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
class SQL {  
    constructor(names: string[], columns: number[]) {  
  
    }  
  
    ins(name: string, row: string[]): boolean {  
  
    }  
  
    rmv(name: string, rowId: number): void {  
  
    }  
  
    sel(name: string, rowId: number, columnId: number): string {  
  
    }  
  
    exp(name: string): string[] {  
  
    }  
}  
  
/**  
 * Your SQL object will be instantiated and called as such:  
 * var obj = new SQL(names, columns)  
 * var param_1 = obj.ins(name, row)  
 * obj.rmv(name, rowId)  
 * var param_3 = obj.sel(name, rowId, columnId)  
 * var param_4 = obj.exp(name)  
 */
```

C# Solution:

```
/*
 * Problem: Design SQL
 * Difficulty: Medium
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

public class SQL {

    public SQL(IList<string> names, IList<int> columns) {

    }

    public bool Ins(string name, IList<string> row) {

    }

    public void Rmv(string name, int rowId) {

    }

    public string Sel(string name, int rowId, int columnId) {

    }

    public IList<string> Exp(string name) {

    }
}

/**
 * Your SQL object will be instantiated and called as such:
 * SQL obj = new SQL(names, columns);
 * bool param_1 = obj.Ins(name, row);
 * obj.Rmv(name, rowId);
 * string param_3 = obj.Sel(name, rowId, columnId);
 * IList<string> param_4 = obj.Exp(name);

```

```
 */
```

C Solution:

```
/*
 * Problem: Design SQL
 * Difficulty: Medium
 * Tags: array, string, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

typedef struct {

} SQL;

SQL* SQLCreate(char** names, int namesSize, int* columns, int columnsSize) {

}

bool sQLIns(SQL* obj, char* name, char** row, int rowSize) {

}

void sQLRmv(SQL* obj, char* name, int rowId) {

}

char* sQLSel(SQL* obj, char* name, int rowId, int columnId) {

}

char** sQLExp(SQL* obj, char* name, int* retSize) {

}
```

```

void sQLFree(SQL* obj) {

}

/***
* Your SQL struct will be instantiated and called as such:
* SQL* obj = sQLCreate(names, namesSize, columns, columnsSize);
* bool param_1 = sQLIns(obj, name, row, rowSize);

* sQLRmv(obj, name, rowId);

* char* param_3 = sQLSel(obj, name, rowId, columnId);

* char** param_4 = sQLExp(obj, name, retSize);

* sQLFree(obj);
*/

```

Go Solution:

```

// Problem: Design SQL
// Difficulty: Medium
// Tags: array, string, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type SQL struct {

}

func Constructor(names []string, columns []int) SQL {

}

func (this *SQL) Ins(name string, row []string) bool {

```

```

}

func (this *SQL) Rmv(name string, rowId int) {

}

func (this *SQL) Sel(name string, rowId int, columnId int) string {

}

func (this *SQL) Exp(name string) []string {

}

/**
 * Your SQL object will be instantiated and called as such:
 * obj := Constructor(names, columns);
 * param_1 := obj.Ins(name, row);
 * obj.Rmv(name, rowId);
 * param_3 := obj.Sel(name, rowId, columnId);
 * param_4 := obj.Exp(name);
 */

```

Kotlin Solution:

```

class SQL(names: List<String>, columns: List<Int>) {

    fun ins(name: String, row: List<String>): Boolean {

    }

    fun rmv(name: String, rowId: Int) {

    }

    fun sel(name: String, rowId: Int, columnId: Int): String {

```

```

}

fun exp(name: String): List<String> {

}

/**

* Your SQL object will be instantiated and called as such:
* var obj = SQL(names, columns)
* var param_1 = obj.ins(name, row)
* obj.rmv(name, rowId)
* var param_3 = obj.sel(name, rowId, columnId)
* var param_4 = obj.exp(name)
*/

```

Swift Solution:

```

class SQL {

init(_ names: [String], _ columns: [Int]) {

}

func ins(_ name: String, _ row: [String]) -> Bool {

}

func rmv(_ name: String, _ rowId: Int) {

}

func sel(_ name: String, _ rowId: Int, _ columnId: Int) -> String {

}

func exp(_ name: String) -> [String] {

}

```

```

}

/**
* Your SQL object will be instantiated and called as such:
* let obj = SQL(names, columns)
* let ret_1: Bool = obj.ins(name, row)
* obj.rmv(name, rowId)
* let ret_3: String = obj.sel(name, rowId, columnId)
* let ret_4: [String] = obj.exp(name)
*/

```

Rust Solution:

```

// Problem: Design SQL
// Difficulty: Medium
// Tags: array, string, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct SQL {

}

/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl SQL {

fn new(names: Vec<String>, columns: Vec<i32>) -> Self {
}

fn ins(&self, name: String, row: Vec<String>) -> bool {
}

fn rmv(&self, name: String, row_id: i32) {
}

```

```

}

fn sel(&self, name: String, row_id: i32, column_id: i32) -> String {

}

fn exp(&self, name: String) -> Vec<String> {

}

/**
 * Your SQL object will be instantiated and called as such:
 * let obj = SQL::new(names, columns);
 * let ret_1: bool = obj.ins(name, row);
 * obj.rmv(name, rowId);
 * let ret_3: String = obj.sel(name, rowId, columnId);
 * let ret_4: Vec<String> = obj.exp(name);
 */

```

Ruby Solution:

```

class SQL

=begin
:type names: String[]
:type columns: Integer[]
=end

def initialize(names, columns)

end

=begin
:type name: String
:type row: String[]
:rtype: Boolean
=end

def ins(name, row)

```

```
end

=begin
:type name: String
:type row_id: Integer
:rtype: Void
=end
def rmv(name, row_id)

end

=begin
:type name: String
:type row_id: Integer
:type column_id: Integer
:rtype: String
=end
def sel(name, row_id, column_id)

end

=begin
:type name: String
:rtype: String[ ]
=end
def exp(name)

end

end

# Your SQL object will be instantiated and called as such:
# obj = SQL.new(names, columns)
# param_1 = obj.ins(name, row)
# obj.rmv(name, row_id)
# param_3 = obj.sel(name, row_id, column_id)
# param_4 = obj.exp(name)
```

PHP Solution:

```
class SQL {  
    /**  
     * @param String[] $names  
     * @param Integer[] $columns  
     */  
    function __construct($names, $columns) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param String[] $row  
     * @return Boolean  
     */  
    function ins($name, $row) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param Integer $rowId  
     * @return NULL  
     */  
    function rmv($name, $rowId) {  
  
    }  
  
    /**  
     * @param String $name  
     * @param Integer $rowId  
     * @param Integer $columnId  
     * @return String  
     */  
    function sel($name, $rowId, $columnId) {  
  
    }  
  
    /**  
     * @param String $name  
     * @return String[]  
     */
```

```

*/
function exp($name) {

}

/**
* Your SQL object will be instantiated and called as such:
* $obj = SQL($names, $columns);
* $ret_1 = $obj->ins($name, $row);
* $obj->rmv($name, $rowId);
* $ret_3 = $obj->sel($name, $rowId, $columnId);
* $ret_4 = $obj->exp($name);
*/

```

Dart Solution:

```

class SQL {

SQL(List<String> names, List<int> columns) {

}

bool ins(String name, List<String> row) {

}

void rmv(String name, int rowId) {

}

String sel(String name, int rowId, int columnId) {

}

List<String> exp(String name) {

}

/*

```

```

* Your SQL object will be instantiated and called as such:
* SQL obj = SQL(names, columns);
* bool param1 = obj.ins(name, row);
* obj.rmv(name, rowId);
* String param3 = obj.sel(name, rowId, columnId);
* List<String> param4 = obj.exp(name);
*/

```

Scala Solution:

```

class SQL(_names: List[String], _columns: List[Int]) {

    def ins(name: String, row: List[String]): Boolean = {

    }

    def rmv(name: String, rowId: Int): Unit = {

    }

    def sel(name: String, rowId: Int, columnId: Int): String = {

    }

    def exp(name: String): List[String] = {

    }

}

/***
 * Your SQL object will be instantiated and called as such:
 * val obj = new SQL(names, columns)
 * val param_1 = obj.ins(name, row)
 * obj.rmv(name, rowId)
 * val param_3 = obj.sel(name, rowId, columnId)
 * val param_4 = obj.exp(name)
 */

```

Elixir Solution:

```

defmodule SQL do
  @spec init_(names :: [String.t], columns :: [integer]) :: any
  def init_(names, columns) do
    end

    @spec ins(name :: String.t, row :: [String.t]) :: boolean
    def ins(name, row) do
      end

      @spec rmv(name :: String.t, row_id :: integer) :: any
      def rmv(name, row_id) do
        end

        @spec sel(name :: String.t, row_id :: integer, column_id :: integer) :: String.t
        def sel(name, row_id, column_id) do
          end

          @spec exp(name :: String.t) :: [String.t]
          def exp(name) do
            end
            end

# Your functions will be called as such:
# SQL.init_(names, columns)
# param_1 = SQL.ins(name, row)
# SQL.rmv(name, row_id)
# param_3 = SQL.sel(name, row_id, column_id)
# param_4 = SQL.exp(name)

# SQL.init_ will be called before every test case, in which you can do some
necessary initializations.

```

Erlang Solution:

```

-spec sql_init_(Names :: [unicode:unicode_binary()]), Columns :: [integer()])
-> any().

```

```

sql_init_(Names, Columns) ->
.

-spec sql_ins(Name :: unicode:unicode_binary(), Row :: [unicode:unicode_binary()]) -> boolean().
sql_ins(Name, Row) ->
.

-spec sql_rmv(Name :: unicode:unicode_binary(), RowId :: integer()) -> any().
sql_rmv(Name, RowId) ->
.

-spec sql_sel(Name :: unicode:unicode_binary(), RowId :: integer(), ColumnId :: integer()) -> unicode:unicode_binary().
sql_sel(Name, RowId, ColumnId) ->
.

-spec sql_exp(Name :: unicode:unicode_binary()) -> [unicode:unicode_binary()].
sql_exp(Name) ->
.

%% Your functions will be called as such:
%% sql_init_(Names, Columns),
%% Param_1 = sql_ins(Name, Row),
%% sql_rmv(Name, RowId),
%% Param_3 = sql_sel(Name, RowId, ColumnId),
%% Param_4 = sql_exp(Name),

%% sql_init_ will be called before every test case, in which you can do some
necessary initializations.

```

Racket Solution:

```

(define sql%
  (class object%
    (super-new)

    ; names : (listof string?)
    ; columns : (listof exact-integer?))

```

```
(init-field
names
columns)

; ins : string? (listof string?) -> boolean?
(define/public (ins name row)
)

; rmv : string? exact-integer? -> void?
(define/public (rmv name row-id)
)

; sel : string? exact-integer? exact-integer? -> string?
(define/public (sel name row-id column-id)
)

; exp : string? -> (listof string?)
(define/public (exp name)
)))

;; Your sql% object will be instantiated and called as such:
;; (define obj (new sql% [names names] [columns columns]))
;; (define param_1 (send obj ins name row))
;; (send obj rmv name row-id)
;; (define param_3 (send obj sel name row-id column-id))
;; (define param_4 (send obj exp name))
```